PATTERN RECOGNITION AND MACHINE LEARNING

CHRISTOPHER M. BISHOP

Information Science and Statistics

Series Editors:
M. Jordan
J. Kleinberg
B. Schölkopf

Information Science and Statistics

Akaike and Kitagawa: The Practice of Time Series Analysis.
Bishop: Pattern Recognition and Machine Learning.
Cowell, Dawid, Lauritzen, and Spiegelhalter: Probabilistic Networks and Expert Systems.
Doucet, de Freitas, and Gordon: Sequential Monte Carlo Methods in Practice.
Fine: Feedforward Neural Network Methodology.
Hawkins and Olwell: Cumulative Sum Charts and Charting for Quality Improvement.
Jensen: Bayesian Networks and Decision Graphs.
Marchette: Computer Intrusion Detection and Network Monitoring: A Statistical Viewpoint.
Rubinstein and Kroese: The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte Carlo Simulation, and Machine Learning.
Studený: Probabilistic Conditional Independence Structures.
Vapnik: The Nature of Statistical Learning Theory, Second Edition.
Wallace: Statistical and Inductive Inference by Minimum Massage Length.

Christopher M. Bishop

Pattern Recognition and Machine Learning

Springer

Christopher M. Bishop F.R.Eng.
Assistant Director
Microsoft Research Ltd
Cambridge CB3 0FB, U.K.
cmbishop@microsoft.com
http://research.microsoft.com/~cmbishop

Series Editors
Michael Jordan
Department of Computer Science and Department of Statistics
University of California, Berkeley
Berkeley, CA 94720
USA

Professor Jon Kleinberg
Department of Computer Science
Cornell University
Ithaca, NY 14853
USA

Bernhard Schölkopf
Max Planck Institute for Biological Cybernetics
Spemannstrasse 38
72076 Tübingen
Germany

Library of Congress Control Number: 2006922522

ISBN-10: 0-387-31073-8
ISBN-13: 978-0387-31073-2

Printed on acid-free paper.

© 2006 Springer Science+Business Media, LLC
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed in Singapore. (KYO)

9 8 7 6 5 4 3 2 1

springer.com

This book is dedicated to my family:

Jenna, Mark, and Hugh

Total eclipse of the sun, Antalya, Turkey, 29 March 2006.

Preface

Pattern recognition has its origins in engineering, whereas machine learning grew out of computer science. However, these activities can be viewed as two facets of the same field, and together they have undergone substantial development over the past ten years. In particular, Bayesian methods have grown from a specialist niche to become mainstream, while graphical models have emerged as a general framework for describing and applying probabilistic models. Also, the practical applicability of Bayesian methods has been greatly enhanced through the development of a range of approximate inference algorithms such as variational Bayes and expectation propagation. Similarly, new models based on kernels have had significant impact on both algorithms and applications.

This new textbook reflects these recent developments while providing a comprehensive introduction to the fields of pattern recognition and machine learning. It is aimed at advanced undergraduates or first year PhD students, as well as researchers and practitioners, and assumes no previous knowledge of pattern recognition or machine learning concepts. Knowledge of multivariate calculus and basic linear algebra is required, and some familiarity with probabilities would be helpful though not essential as the book includes a self-contained introduction to basic probability theory.

Because this book has broad scope, it is impossible to provide a complete list of references, and in particular no attempt has been made to provide accurate historical attribution of ideas. Instead, the aim has been to give references that offer greater detail than is possible here and that hopefully provide entry points into what, in some cases, is a very extensive literature. For this reason, the references are often to more recent textbooks and review articles rather than to original sources.

The book is supported by a great deal of additional material, including lecture slides as well as the complete set of figures used in the book, and the reader is encouraged to visit the book web site for the latest information:

http://research.microsoft.com/∼cmbishop/PRML

Exercises

The exercises that appear at the end of every chapter form an important component of the book. Each exercise has been carefully chosen to reinforce concepts explained in the text or to develop and generalize them in significant ways, and each is graded according to difficulty ranging from ($\star$), which denotes a simple exercise taking a few minutes to complete, through to ($\star\star\star$), which denotes a significantly more complex exercise.

It has been difficult to know to what extent these solutions should be made widely available. Those engaged in self study will find worked solutions very beneficial, whereas many course tutors request that solutions be available only via the publisher so that the exercises may be used in class. In order to try to meet these conflicting requirements, those exercises that help amplify key points in the text, or that fill in important details, have solutions that are available as a PDF file from the book web site. Such exercises are denoted by **www**. Solutions for the remaining exercises are available to course tutors by contacting the publisher (contact details are given on the book web site). Readers are strongly encouraged to work through the exercises unaided, and to turn to the solutions only as required.

Although this book focuses on concepts and principles, in a taught course the students should ideally have the opportunity to experiment with some of the key algorithms using appropriate data sets. A companion volume (Bishop and Nabney, 2008) will deal with practical aspects of pattern recognition and machine learning, and will be accompanied by Matlab software implementing most of the algorithms discussed in this book.

Acknowledgements

First of all I would like to express my sincere thanks to Markus Svensén who has provided immense help with preparation of figures and with the typesetting of the book in LaTeX. His assistance has been invaluable.

I am very grateful to Microsoft Research for providing a highly stimulating research environment and for giving me the freedom to write this book (the views and opinions expressed in this book, however, are my own and are therefore not necessarily the same as those of Microsoft or its affiliates).

Springer has provided excellent support throughout the final stages of preparation of this book, and I would like to thank my commissioning editor John Kimmel for his support and professionalism, as well as Joseph Piliero for his help in designing the cover and the text format and MaryAnn Brickner for her numerous contributions during the production phase. The inspiration for the cover design came from a discussion with Antonio Criminisi.

I also wish to thank Oxford University Press for permission to reproduce excerpts from an earlier textbook, *Neural Networks for Pattern Recognition* (Bishop, 1995a). The images of the Mark 1 perceptron and of Frank Rosenblatt are reproduced with the permission of Arvin Calspan Advanced Technology Center. I would also like to thank Asela Gunawardana for plotting the spectrogram in Figure 13.1, and Bernhard Schölkopf for permission to use his kernel PCA code to plot Figure 12.17.

Many people have helped by proofreading draft material and providing comments and suggestions, including Shivani Agarwal, Cédric Archambeau, Arik Azran, Andrew Blake, Hakan Cevikalp, Michael Fourman, Brendan Frey, Zoubin Ghahramani, Thore Graepel, Katherine Heller, Ralf Herbrich, Geoffrey Hinton, Adam Johansen, Matthew Johnson, Michael Jordan, Eva Kalyvianaki, Anitha Kannan, Julia Lasserre, David Liu, Tom Minka, Ian Nabney, Tonatiuh Pena, Yuan Qi, Sam Roweis, Balaji Sanjiya, Toby Sharp, Ana Costa e Silva, David Spiegelhalter, Jay Stokes, Tara Symeonides, Martin Szummer, Marshall Tappen, Ilkay Ulusoy, Chris Williams, John Winn, and Andrew Zisserman.

Finally, I would like to thank my wife Jenna who has been hugely supportive throughout the several years it has taken to write this book.

Chris Bishop
Cambridge
February 2006

Mathematical notation

I have tried to keep the mathematical content of the book to the minimum necessary to achieve a proper understanding of the field. However, this minimum level is nonzero, and it should be emphasized that a good grasp of calculus, linear algebra, and probability theory is essential for a clear understanding of modern pattern recognition and machine learning techniques. Nevertheless, the emphasis in this book is on conveying the underlying concepts rather than on mathematical rigour.

I have tried to use a consistent notation throughout the book, although at times this means departing from some of the conventions used in the corresponding research literature. Vectors are denoted by lower case bold Roman letters such as $\mathbf{x}$, and all vectors are assumed to be column vectors. A superscript T denotes the transpose of a matrix or vector, so that $\mathbf{x}^{\mathrm{T}}$ will be a row vector. Uppercase bold roman letters, such as $\mathbf{M}$, denote matrices. The notation $(w_1, \ldots, w_M)$ denotes a row vector with M elements, while the corresponding column vector is written as $\mathbf{w} = (w_1, \ldots, w_M)^{\mathrm{T}}$.

The notation $[a, b]$ is used to denote the *closed* interval from a to b, that is the interval including the values a and b themselves, while (a, b) denotes the corresponding *open* interval, that is the interval excluding a and b. Similarly, $[a, b)$ denotes an interval that includes a but excludes b. For the most part, however, there will be little need to dwell on such refinements as whether the end points of an interval are included or not.

The $M \times M$ identity matrix (also known as the unit matrix) is denoted $\mathbf{I}_M$, which will be abbreviated to $\mathbf{I}$ where there is no ambiguity about it dimensionality. It has elements I_{ij} that equal 1 if $i = j$ and 0 if $i \neq j$.

A functional is denoted $f[y]$ where $y(x)$ is some function. The concept of a functional is discussed in Appendix D.

The notation $g(x) = O(f(x))$ denotes that $|f(x)/g(x)|$ is bounded as $x \to \infty$. For instance if $g(x) = 3x^2 + 2$, then $g(x) = O(x^2)$.

The expectation of a function $f(x, y)$ with respect to a random variable x is denoted by $\mathbb{E}_x[f(x, y)]$. In situations where there is no ambiguity as to which variable is being averaged over, this will be simplified by omitting the suffix, for instance

$\mathbb{E}[x]$. If the distribution of x is conditioned on another variable z, then the corresponding conditional expectation will be written $\mathbb{E}_x[f(x)|z]$. Similarly, the variance is denoted $\mathrm{var}[f(x)]$, and for vector variables the covariance is written $\mathrm{cov}[\mathbf{x}, \mathbf{y}]$. We shall also use $\mathrm{cov}[\mathbf{x}]$ as a shorthand notation for $\mathrm{cov}[\mathbf{x}, \mathbf{x}]$. The concepts of expectations and covariances are introduced in Section 1.2.2.

If we have N values $\mathbf{x}_1, \ldots, \mathbf{x}_N$ of a D-dimensional vector $\mathbf{x} = (x_1, \ldots, x_D)^{\mathrm{T}}$, we can combine the observations into a data matrix $\mathbf{X}$ in which the nth row of $\mathbf{X}$ corresponds to the row vector $\mathbf{x}_n^{\mathrm{T}}$. Thus the n, i element of $\mathbf{X}$ corresponds to the ith element of the nth observation $\mathbf{x}_n$. For the case of one-dimensional variables we shall denote such a matrix by $\mathbf{x}$, which is a column vector whose n^{th} element is x_n. Note that $\mathbf{x}$ (which has dimensionality N) uses a different typeface to distinguish it from $\mathbf{x}$ (which has dimensionality D).

Contents

Preface vii

Mathematical notation xi

Contents xiii

1 Introduction 1
- 1.1 Example: Polynomial Curve Fitting . . . 4
- 1.2 Probability Theory . . . 12
  - 1.2.1 Probability densities . . . 17
  - 1.2.2 Expectations and covariances . . . 19
  - 1.2.3 Bayesian probabilities . . . 21
  - 1.2.4 The Gaussian distribution . . . 24
  - 1.2.5 Curve fitting re-visited . . . 28
  - 1.2.6 Bayesian curve fitting . . . 30
- 1.3 Model Selection . . . 32
- 1.4 The Curse of Dimensionality . . . 33
- 1.5 Decision Theory . . . 38
  - 1.5.1 Minimizing the misclassification rate . . . 39
  - 1.5.2 Minimizing the expected loss . . . 41
  - 1.5.3 The reject option . . . 42
  - 1.5.4 Inference and decision . . . 42
  - 1.5.5 Loss functions for regression . . . 46
- 1.6 Information Theory . . . 48
  - 1.6.1 Relative entropy and mutual information . . . 55
- Exercises . . . 58

2 Probability Distributions **67**
2.1 Binary Variables . . . 68
2.1.1 The beta distribution . . . 71
2.2 Multinomial Variables . . . 74
2.2.1 The Dirichlet distribution . . . 76
2.3 The Gaussian Distribution . . . 78
2.3.1 Conditional Gaussian distributions . . . 85
2.3.2 Marginal Gaussian distributions . . . 88
2.3.3 Bayes' theorem for Gaussian variables . . . 90
2.3.4 Maximum likelihood for the Gaussian . . . 93
2.3.5 Sequential estimation . . . 94
2.3.6 Bayesian inference for the Gaussian . . . 97
2.3.7 Student's t-distribution . . . 102
2.3.8 Periodic variables . . . 105
2.3.9 Mixtures of Gaussians . . . 110
2.4 The Exponential Family . . . 113
2.4.1 Maximum likelihood and sufficient statistics . . . 116
2.4.2 Conjugate priors . . . 117
2.4.3 Noninformative priors . . . 117
2.5 Nonparametric Methods . . . 120
2.5.1 Kernel density estimators . . . 122
2.5.2 Nearest-neighbour methods . . . 124
Exercises . . . 127

3 Linear Models for Regression **137**
3.1 Linear Basis Function Models . . . 138
3.1.1 Maximum likelihood and least squares . . . 140
3.1.2 Geometry of least squares . . . 143
3.1.3 Sequential learning . . . 143
3.1.4 Regularized least squares . . . 144
3.1.5 Multiple outputs . . . 146
3.2 The Bias-Variance Decomposition . . . 147
3.3 Bayesian Linear Regression . . . 152
3.3.1 Parameter distribution . . . 152
3.3.2 Predictive distribution . . . 156
3.3.3 Equivalent kernel . . . 159
3.4 Bayesian Model Comparison . . . 161
3.5 The Evidence Approximation . . . 165
3.5.1 Evaluation of the evidence function . . . 166
3.5.2 Maximizing the evidence function . . . 168
3.5.3 Effective number of parameters . . . 170
3.6 Limitations of Fixed Basis Functions . . . 172
Exercises . . . 173

4 Linear Models for Classification 179
4.1 Discriminant Functions . . . 181
4.1.1 Two classes . . . 181
4.1.2 Multiple classes . . . 182
4.1.3 Least squares for classification . . . 184
4.1.4 Fisher's linear discriminant . . . 186
4.1.5 Relation to least squares . . . 189
4.1.6 Fisher's discriminant for multiple classes . . . 191
4.1.7 The perceptron algorithm . . . 192
4.2 Probabilistic Generative Models . . . 196
4.2.1 Continuous inputs . . . 198
4.2.2 Maximum likelihood solution . . . 200
4.2.3 Discrete features . . . 202
4.2.4 Exponential family . . . 202
4.3 Probabilistic Discriminative Models . . . 203
4.3.1 Fixed basis functions . . . 204
4.3.2 Logistic regression . . . 205
4.3.3 Iterative reweighted least squares . . . 207
4.3.4 Multiclass logistic regression . . . 209
4.3.5 Probit regression . . . 210
4.3.6 Canonical link functions . . . 212
4.4 The Laplace Approximation . . . 213
4.4.1 Model comparison and BIC . . . 216
4.5 Bayesian Logistic Regression . . . 217
4.5.1 Laplace approximation . . . 217
4.5.2 Predictive distribution . . . 218
Exercises . . . 220

5 Neural Networks 225
5.1 Feed-forward Network Functions . . . 227
5.1.1 Weight-space symmetries . . . 231
5.2 Network Training . . . 232
5.2.1 Parameter optimization . . . 236
5.2.2 Local quadratic approximation . . . 237
5.2.3 Use of gradient information . . . 239
5.2.4 Gradient descent optimization . . . 240
5.3 Error Backpropagation . . . 241
5.3.1 Evaluation of error-function derivatives . . . 242
5.3.2 A simple example . . . 245
5.3.3 Efficiency of backpropagation . . . 246
5.3.4 The Jacobian matrix . . . 247
5.4 The Hessian Matrix . . . 249
5.4.1 Diagonal approximation . . . 250
5.4.2 Outer product approximation . . . 251
5.4.3 Inverse Hessian . . . 252

5.4.4 Finite differences . . . 252
5.4.5 Exact evaluation of the Hessian . . . 253
5.4.6 Fast multiplication by the Hessian . . . 254
5.5 Regularization in Neural Networks . . . 256
5.5.1 Consistent Gaussian priors . . . 257
5.5.2 Early stopping . . . 259
5.5.3 Invariances . . . 261
5.5.4 Tangent propagation . . . 263
5.5.5 Training with transformed data . . . 265
5.5.6 Convolutional networks . . . 267
5.5.7 Soft weight sharing . . . 269
5.6 Mixture Density Networks . . . 272
5.7 Bayesian Neural Networks . . . 277
5.7.1 Posterior parameter distribution . . . 278
5.7.2 Hyperparameter optimization . . . 280
5.7.3 Bayesian neural networks for classification . . . 281
Exercises . . . 284

6 Kernel Methods 291
6.1 Dual Representations . . . 293
6.2 Constructing Kernels . . . 294
6.3 Radial Basis Function Networks . . . 299
6.3.1 Nadaraya-Watson model . . . 301
6.4 Gaussian Processes . . . 303
6.4.1 Linear regression revisited . . . 304
6.4.2 Gaussian processes for regression . . . 306
6.4.3 Learning the hyperparameters . . . 311
6.4.4 Automatic relevance determination . . . 312
6.4.5 Gaussian processes for classification . . . 313
6.4.6 Laplace approximation . . . 315
6.4.7 Connection to neural networks . . . 319
Exercises . . . 320

7 Sparse Kernel Machines 325
7.1 Maximum Margin Classifiers . . . 326
7.1.1 Overlapping class distributions . . . 331
7.1.2 Relation to logistic regression . . . 336
7.1.3 Multiclass SVMs . . . 338
7.1.4 SVMs for regression . . . 339
7.1.5 Computational learning theory . . . 344
7.2 Relevance Vector Machines . . . 345
7.2.1 RVM for regression . . . 345
7.2.2 Analysis of sparsity . . . 349
7.2.3 RVM for classification . . . 353
Exercises . . . 357

8 Graphical Models **359**
8.1 Bayesian Networks . . . 360
8.1.1 Example: Polynomial regression . . . 362
8.1.2 Generative models . . . 365
8.1.3 Discrete variables . . . 366
8.1.4 Linear-Gaussian models . . . 370
8.2 Conditional Independence . . . 372
8.2.1 Three example graphs . . . 373
8.2.2 D-separation . . . 378
8.3 Markov Random Fields . . . 383
8.3.1 Conditional independence properties . . . 383
8.3.2 Factorization properties . . . 384
8.3.3 Illustration: Image de-noising . . . 387
8.3.4 Relation to directed graphs . . . 390
8.4 Inference in Graphical Models . . . 393
8.4.1 Inference on a chain . . . 394
8.4.2 Trees . . . 398
8.4.3 Factor graphs . . . 399
8.4.4 The sum-product algorithm . . . 402
8.4.5 The max-sum algorithm . . . 411
8.4.6 Exact inference in general graphs . . . 416
8.4.7 Loopy belief propagation . . . 417
8.4.8 Learning the graph structure . . . 418
Exercises . . . 418

9 Mixture Models and EM **423**
9.1 K-means Clustering . . . 424
9.1.1 Image segmentation and compression . . . 428
9.2 Mixtures of Gaussians . . . 430
9.2.1 Maximum likelihood . . . 432
9.2.2 EM for Gaussian mixtures . . . 435
9.3 An Alternative View of EM . . . 439
9.3.1 Gaussian mixtures revisited . . . 441
9.3.2 Relation to K-means . . . 443
9.3.3 Mixtures of Bernoulli distributions . . . 444
9.3.4 EM for Bayesian linear regression . . . 448
9.4 The EM Algorithm in General . . . 450
Exercises . . . 455

10 Approximate Inference **461**
10.1 Variational Inference . . . 462
10.1.1 Factorized distributions . . . 464
10.1.2 Properties of factorized approximations . . . 466
10.1.3 Example: The univariate Gaussian . . . 470
10.1.4 Model comparison . . . 473
10.2 Illustration: Variational Mixture of Gaussians . . . 474

10.2.1 Variational distribution 475
10.2.2 Variational lower bound 481
10.2.3 Predictive density 482
10.2.4 Determining the number of components 483
10.2.5 Induced factorizations 485
10.3 Variational Linear Regression 486
10.3.1 Variational distribution 486
10.3.2 Predictive distribution 488
10.3.3 Lower bound 489
10.4 Exponential Family Distributions 490
10.4.1 Variational message passing 491
10.5 Local Variational Methods 493
10.6 Variational Logistic Regression 498
10.6.1 Variational posterior distribution 498
10.6.2 Optimizing the variational parameters 500
10.6.3 Inference of hyperparameters 502
10.7 Expectation Propagation 505
10.7.1 Example: The clutter problem 511
10.7.2 Expectation propagation on graphs 513
Exercises 517

11 Sampling Methods 523
11.1 Basic Sampling Algorithms 526
11.1.1 Standard distributions 526
11.1.2 Rejection sampling 528
11.1.3 Adaptive rejection sampling 530
11.1.4 Importance sampling 532
11.1.5 Sampling-importance-resampling 534
11.1.6 Sampling and the EM algorithm 536
11.2 Markov Chain Monte Carlo 537
11.2.1 Markov chains 539
11.2.2 The Metropolis-Hastings algorithm 541
11.3 Gibbs Sampling 542
11.4 Slice Sampling 546
11.5 The Hybrid Monte Carlo Algorithm 548
11.5.1 Dynamical systems 548
11.5.2 Hybrid Monte Carlo 552
11.6 Estimating the Partition Function 554
Exercises 556

12 Continuous Latent Variables 559
12.1 Principal Component Analysis 561
12.1.1 Maximum variance formulation 561
12.1.2 Minimum-error formulation 563
12.1.3 Applications of PCA 565
12.1.4 PCA for high-dimensional data 569

12.2 Probabilistic PCA 570
12.2.1 Maximum likelihood PCA 574
12.2.2 EM algorithm for PCA 577
12.2.3 Bayesian PCA 580
12.2.4 Factor analysis 583
12.3 Kernel PCA 586
12.4 Nonlinear Latent Variable Models 591
12.4.1 Independent component analysis 591
12.4.2 Autoassociative neural networks 592
12.4.3 Modelling nonlinear manifolds 595
Exercises 599

13 Sequential Data 605
13.1 Markov Models 607
13.2 Hidden Markov Models 610
13.2.1 Maximum likelihood for the HMM 615
13.2.2 The forward-backward algorithm 618
13.2.3 The sum-product algorithm for the HMM 625
13.2.4 Scaling factors 627
13.2.5 The Viterbi algorithm 629
13.2.6 Extensions of the hidden Markov model 631
13.3 Linear Dynamical Systems 635
13.3.1 Inference in LDS 638
13.3.2 Learning in LDS 642
13.3.3 Extensions of LDS 644
13.3.4 Particle filters 645
Exercises 646

14 Combining Models 653
14.1 Bayesian Model Averaging 654
14.2 Committees 655
14.3 Boosting 657
14.3.1 Minimizing exponential error 659
14.3.2 Error functions for boosting 661
14.4 Tree-based Models 663
14.5 Conditional Mixture Models 666
14.5.1 Mixtures of linear regression models 667
14.5.2 Mixtures of logistic models 670
14.5.3 Mixtures of experts 672
Exercises 674

Appendix A Data Sets 677

Appendix B Probability Distributions 685

Appendix C Properties of Matrices 695

Appendix D Calculus of Variations **703**

Appendix E Lagrange Multipliers **707**

References **711**

Index **729**

1 Introduction

The problem of searching for patterns in data is a fundamental one and has a long and successful history. For instance, the extensive astronomical observations of Tycho Brahe in the 16th century allowed Johannes Kepler to discover the empirical laws of planetary motion, which in turn provided a springboard for the development of classical mechanics. Similarly, the discovery of regularities in atomic spectra played a key role in the development and verification of quantum physics in the early twentieth century. The field of pattern recognition is concerned with the automatic discovery of regularities in data through the use of computer algorithms and with the use of these regularities to take actions such as classifying the data into different categories.

Consider the example of recognizing handwritten digits, illustrated in Figure 1.1. Each digit corresponds to a 28×28 pixel image and so can be represented by a vector $\mathbf{x}$ comprising 784 real numbers. The goal is to build a machine that will take such a vector $\mathbf{x}$ as input and that will produce the identity of the digit $0, \ldots, 9$ as the output. This is a nontrivial problem due to the wide variability of handwriting. It could be

Figure 1.1 Examples of hand-written digits taken from US zip codes.

tackled using handcrafted rules or heuristics for distinguishing the digits based on the shapes of the strokes, but in practice such an approach leads to a proliferation of rules and of exceptions to the rules and so on, and invariably gives poor results.

Far better results can be obtained by adopting a machine learning approach in which a large set of N digits $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ called a *training set* is used to tune the parameters of an adaptive model. The categories of the digits in the training set are known in advance, typically by inspecting them individually and hand-labelling them. We can express the category of a digit using *target vector* $\mathbf{t}$, which represents the identity of the corresponding digit. Suitable techniques for representing categories in terms of vectors will be discussed later. Note that there is one such target vector $\mathbf{t}$ for each digit image $\mathbf{x}$.

The result of running the machine learning algorithm can be expressed as a function $\mathbf{y}(\mathbf{x})$ which takes a new digit image $\mathbf{x}$ as input and that generates an output vector $\mathbf{y}$, encoded in the same way as the target vectors. The precise form of the function $\mathbf{y}(\mathbf{x})$ is determined during the *training* phase, also known as the *learning* phase, on the basis of the training data. Once the model is trained it can then determine the identity of new digit images, which are said to comprise a *test set*. The ability to categorize correctly new examples that differ from those used for training is known as *generalization*. In practical applications, the variability of the input vectors will be such that the training data can comprise only a tiny fraction of all possible input vectors, and so generalization is a central goal in pattern recognition.

For most practical applications, the original input variables are typically *preprocessed* to transform them into some new space of variables where, it is hoped, the pattern recognition problem will be easier to solve. For instance, in the digit recognition problem, the images of the digits are typically translated and scaled so that each digit is contained within a box of a fixed size. This greatly reduces the variability within each digit class, because the location and scale of all the digits are now the same, which makes it much easier for a subsequent pattern recognition algorithm to distinguish between the different classes. This pre-processing stage is sometimes also called *feature extraction*. Note that new test data must be pre-processed using the same steps as the training data.

Pre-processing might also be performed in order to speed up computation. For example, if the goal is real-time face detection in a high-resolution video stream, the computer must handle huge numbers of pixels per second, and presenting these directly to a complex pattern recognition algorithm may be computationally infeasible. Instead, the aim is to find useful features that are fast to compute, and yet that

also preserve useful discriminatory information enabling faces to be distinguished from non-faces. These features are then used as the inputs to the pattern recognition algorithm. For instance, the average value of the image intensity over a rectangular subregion can be evaluated extremely efficiently (Viola and Jones, 2004), and a set of such features can prove very effective in fast face detection. Because the number of such features is smaller than the number of pixels, this kind of pre-processing represents a form of dimensionality reduction. Care must be taken during pre-processing because often information is discarded, and if this information is important to the solution of the problem then the overall accuracy of the system can suffer.

Applications in which the training data comprises examples of the input vectors along with their corresponding target vectors are known as *supervised learning* problems. Cases such as the digit recognition example, in which the aim is to assign each input vector to one of a finite number of discrete categories, are called *classification* problems. If the desired output consists of one or more continuous variables, then the task is called *regression*. An example of a regression problem would be the prediction of the yield in a chemical manufacturing process in which the inputs consist of the concentrations of reactants, the temperature, and the pressure.

In other pattern recognition problems, the training data consists of a set of input vectors $\mathbf{x}$ without any corresponding target values. The goal in such *unsupervised learning* problems may be to discover groups of similar examples within the data, where it is called *clustering*, or to determine the distribution of data within the input space, known as *density estimation*, or to project the data from a high-dimensional space down to two or three dimensions for the purpose of *visualization*.

Finally, the technique of *reinforcement learning* (Sutton and Barto, 1998) is concerned with the problem of finding suitable actions to take in a given situation in order to maximize a reward. Here the learning algorithm is not given examples of optimal outputs, in contrast to supervised learning, but must instead discover them by a process of trial and error. Typically there is a sequence of states and actions in which the learning algorithm is interacting with its environment. In many cases, the current action not only affects the immediate reward but also has an impact on the reward at all subsequent time steps. For example, by using appropriate reinforcement learning techniques a neural network can learn to play the game of backgammon to a high standard (Tesauro, 1994). Here the network must learn to take a board position as input, along with the result of a dice throw, and produce a strong move as the output. This is done by having the network play against a copy of itself for perhaps a million games. A major challenge is that a game of backgammon can involve dozens of moves, and yet it is only at the end of the game that the reward, in the form of victory, is achieved. The reward must then be attributed appropriately to all of the moves that led to it, even though some moves will have been good ones and others less so. This is an example of a *credit assignment* problem. A general feature of reinforcement learning is the trade-off between *exploration*, in which the system tries out new kinds of actions to see how effective they are, and *exploitation*, in which the system makes use of actions that are known to yield a high reward. Too strong a focus on either exploration or exploitation will yield poor results. Reinforcement learning continues to be an active area of machine learning research. However, a

Figure 1.2 Plot of a training data set of $N = 10$ points, shown as blue circles, each comprising an observation of the input variable x along with the corresponding target variable t. The green curve shows the function $\sin(2\pi x)$ used to generate the data. Our goal is to predict the value of t for some new value of x, without knowledge of the green curve.

detailed treatment lies beyond the scope of this book.

Although each of these tasks needs its own tools and techniques, many of the key ideas that underpin them are common to all such problems. One of the main goals of this chapter is to introduce, in a relatively informal way, several of the most important of these concepts and to illustrate them using simple examples. Later in the book we shall see these same ideas re-emerge in the context of more sophisticated models that are applicable to real-world pattern recognition applications. This chapter also provides a self-contained introduction to three important tools that will be used throughout the book, namely probability theory, decision theory, and information theory. Although these might sound like daunting topics, they are in fact straightforward, and a clear understanding of them is essential if machine learning techniques are to be used to best effect in practical applications.

1.1. Example: Polynomial Curve Fitting

We begin by introducing a simple regression problem, which we shall use as a running example throughout this chapter to motivate a number of key concepts. Suppose we observe a real-valued input variable x and we wish to use this observation to predict the value of a real-valued target variable t. For the present purposes, it is instructive to consider an artificial example using synthetically generated data because we then know the precise process that generated the data for comparison against any learned model. The data for this example is generated from the function $\sin(2\pi x)$ with random noise included in the target values, as described in detail in Appendix A.

Now suppose that we are given a training set comprising N observations of x, written $\mathbf{x} \equiv (x_1, \ldots, x_N)^{\mathrm{T}}$, together with corresponding observations of the values of t, denoted $\mathbf{t} \equiv (t_1, \ldots, t_N)^{\mathrm{T}}$. Figure 1.2 shows a plot of a training set comprising $N = 10$ data points. The input data set $\mathbf{x}$ in Figure 1.2 was generated by choosing values of x_n, for $n = 1, \ldots, N$, spaced uniformly in range $[0, 1]$, and the target data set $\mathbf{t}$ was obtained by first computing the corresponding values of the function

$\sin(2\pi x)$ and then adding a small level of random noise having a Gaussian distribution (the Gaussian distribution is discussed in Section 1.2.4) to each such point in order to obtain the corresponding value t_n. By generating data in this way, we are capturing a property of many real data sets, namely that they possess an underlying regularity, which we wish to learn, but that individual observations are corrupted by random noise. This noise might arise from intrinsically stochastic (i.e. random) processes such as radioactive decay but more typically is due to there being sources of variability that are themselves unobserved.

Our goal is to exploit this training set in order to make predictions of the value $\widehat{t}$ of the target variable for some new value $\widehat{x}$ of the input variable. As we shall see later, this involves implicitly trying to discover the underlying function $\sin(2\pi x)$. This is intrinsically a difficult problem as we have to generalize from a finite data set. Furthermore the observed data are corrupted with noise, and so for a given $\widehat{x}$ there is uncertainty as to the appropriate value for $\widehat{t}$. Probability theory, discussed in Section 1.2, provides a framework for expressing such uncertainty in a precise and quantitative manner, and decision theory, discussed in Section 1.5, allows us to exploit this probabilistic representation in order to make predictions that are optimal according to appropriate criteria.

For the moment, however, we shall proceed rather informally and consider a simple approach based on curve fitting. In particular, we shall fit the data using a polynomial function of the form

$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j \tag{1.1}$$

where M is the *order* of the polynomial, and x^j denotes x raised to the power of j. The polynomial coefficients $w_0, \ldots, w_M$ are collectively denoted by the vector $\mathbf{w}$. Note that, although the polynomial function $y(x, \mathbf{w})$ is a nonlinear function of x, it is a linear function of the coefficients $\mathbf{w}$. Functions, such as the polynomial, which are linear in the unknown parameters have important properties and are called *linear models* and will be discussed extensively in Chapters 3 and 4.

The values of the coefficients will be determined by fitting the polynomial to the training data. This can be done by minimizing an *error function* that measures the misfit between the function $y(x, \mathbf{w})$, for any given value of $\mathbf{w}$, and the training set data points. One simple choice of error function, which is widely used, is given by the sum of the squares of the errors between the predictions $y(x_n, \mathbf{w})$ for each data point x_n and the corresponding target values t_n, so that we minimize

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \{y(x_n, \mathbf{w}) - t_n\}^2 \tag{1.2}$$

where the factor of $1/2$ is included for later convenience. We shall discuss the motivation for this choice of error function later in this chapter. For the moment we simply note that it is a nonnegative quantity that would be zero if, and only if, the

Figure 1.3 The error function (1.2) corresponds to (one half of) the sum of the squares of the displacements (shown by the vertical green bars) of each data point from the function $y(x, \mathbf{w})$.

function $y(x, \mathbf{w})$ were to pass exactly through each training data point. The geometrical interpretation of the sum-of-squares error function is illustrated in Figure 1.3.

We can solve the curve fitting problem by choosing the value of $\mathbf{w}$ for which $E(\mathbf{w})$ is as small as possible. Because the error function is a quadratic function of the coefficients $\mathbf{w}$, its derivatives with respect to the coefficients will be linear in the elements of $\mathbf{w}$, and so the minimization of the error function has a unique solution, denoted by $\mathbf{w}^\star$, which can be found in closed form. The resulting polynomial is given by the function $y(x, \mathbf{w}^\star)$.

Exercise 1.1

There remains the problem of choosing the order M of the polynomial, and as we shall see this will turn out to be an example of an important concept called *model comparison* or *model selection*. In Figure 1.4, we show four examples of the results of fitting polynomials having orders $M = 0, 1, 3$, and 9 to the data set shown in Figure 1.2.

We notice that the constant ($M = 0$) and first order ($M = 1$) polynomials give rather poor fits to the data and consequently rather poor representations of the function $\sin(2\pi x)$. The third order ($M = 3$) polynomial seems to give the best fit to the function $\sin(2\pi x)$ of the examples shown in Figure 1.4. When we go to a much higher order polynomial ($M = 9$), we obtain an excellent fit to the training data. In fact, the polynomial passes exactly through each data point and $E(\mathbf{w}^\star) = 0$. However, the fitted curve oscillates wildly and gives a very poor representation of the function $\sin(2\pi x)$. This latter behaviour is known as *over-fitting*.

As we have noted earlier, the goal is to achieve good generalization by making accurate predictions for new data. We can obtain some quantitative insight into the dependence of the generalization performance on M by considering a separate test set comprising 100 data points generated using exactly the same procedure used to generate the training set points but with new choices for the random noise values included in the target values. For each choice of M, we can then evaluate the residual value of $E(\mathbf{w}^\star)$ given by (1.2) for the training data, and we can also evaluate $E(\mathbf{w}^\star)$ for the test data set. It is sometimes more convenient to use the root-mean-square

Figure 1.4 Plots of polynomials having various orders M, shown as red curves, fitted to the data set shown in Figure 1.2.

(RMS) error defined by

$$E_{\mathrm{RMS}} = \sqrt{2E(\mathbf{w}^\star)/N} \tag{1.3}$$

in which the division by N allows us to compare different sizes of data sets on an equal footing, and the square root ensures that E_{RMS} is measured on the same scale (and in the same units) as the target variable t. Graphs of the training and test set RMS errors are shown, for various values of M, in Figure 1.5. The test set error is a measure of how well we are doing in predicting the values of t for new data observations of x. We note from Figure 1.5 that small values of M give relatively large values of the test set error, and this can be attributed to the fact that the corresponding polynomials are rather inflexible and are incapable of capturing the oscillations in the function $\sin(2\pi x)$. Values of M in the range $3 \leqslant M \leqslant 8$ give small values for the test set error, and these also give reasonable representations of the generating function $\sin(2\pi x)$, as can be seen, for the case of $M = 3$, from Figure 1.4.

Figure 1.5 Graphs of the root-mean-square error, defined by (1.3), evaluated on the training set and on an independent test set for various values of M.

For $M = 9$, the training set error goes to zero, as we might expect because this polynomial contains 10 degrees of freedom corresponding to the 10 coefficients $w_0, \ldots, w_9$, and so can be tuned exactly to the 10 data points in the training set. However, the test set error has become very large and, as we saw in Figure 1.4, the corresponding function $y(x, \mathbf{w}^\star)$ exhibits wild oscillations.

This may seem paradoxical because a polynomial of given order contains all lower order polynomials as special cases. The $M = 9$ polynomial is therefore capable of generating results at least as good as the $M = 3$ polynomial. Furthermore, we might suppose that the best predictor of new data would be the function $\sin(2\pi x)$ from which the data was generated (and we shall see later that this is indeed the case). We know that a power series expansion of the function $\sin(2\pi x)$ contains terms of all orders, so we might expect that results should improve monotonically as we increase M.

We can gain some insight into the problem by examining the values of the coefficients $\mathbf{w}^\star$ obtained from polynomials of various order, as shown in Table 1.1. We see that, as M increases, the magnitude of the coefficients typically gets larger. In particular for the $M = 9$ polynomial, the coefficients have become finely tuned to the data by developing large positive and negative values so that the correspond-

Table 1.1 Table of the coefficients $\mathbf{w}^\star$ for polynomials of various order. Observe how the typical magnitude of the coefficients increases dramatically as the order of the polynomial increases.

	$M = 0$	$M = 1$	$M = 6$	$M = 9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

Figure 1.6 Plots of the solutions obtained by minimizing the sum-of-squares error function using the $M = 9$ polynomial for $N = 15$ data points (left plot) and $N = 100$ data points (right plot). We see that increasing the size of the data set reduces the over-fitting problem.

ing polynomial function matches each of the data points exactly, but between data points (particularly near the ends of the range) the function exhibits the large oscillations observed in Figure 1.4. Intuitively, what is happening is that the more flexible polynomials with larger values of M are becoming increasingly tuned to the random noise on the target values.

It is also interesting to examine the behaviour of a given model as the size of the data set is varied, as shown in Figure 1.6. We see that, for a given model complexity, the over-fitting problem become less severe as the size of the data set increases. Another way to say this is that the larger the data set, the more complex (in other words more flexible) the model that we can afford to fit to the data. One rough heuristic that is sometimes advocated is that the number of data points should be no less than some multiple (say 5 or 10) of the number of adaptive parameters in the model. However, as we shall see in Chapter 3, the number of parameters is not necessarily the most appropriate measure of model complexity.

Also, there is something rather unsatisfying about having to limit the number of parameters in a model according to the size of the available training set. It would seem more reasonable to choose the complexity of the model according to the complexity of the problem being solved. We shall see that the least squares approach to finding the model parameters represents a specific case of *maximum likelihood* (discussed in Section 1.2.5), and that the over-fitting problem can be understood as a general property of maximum likelihood. By adopting a *Bayesian* approach, the over-fitting problem can be avoided. We shall see that there is no difficulty from a Bayesian perspective in employing models for which the number of parameters greatly exceeds the number of data points. Indeed, in a Bayesian model the *effective* number of parameters adapts automatically to the size of the data set.

Section 3.4

For the moment, however, it is instructive to continue with the current approach and to consider how in practice we can apply it to data sets of limited size where we

Figure 1.7 Plots of $M = 9$ polynomials fitted to the data set shown in Figure 1.2 using the regularized error function (1.4) for two values of the regularization parameter λ corresponding to $\ln \lambda = -18$ and $\ln \lambda = 0$. The case of no regularizer, i.e., $\lambda = 0$, corresponding to $\ln \lambda = -\infty$, is shown at the bottom right of Figure 1.4.

may wish to use relatively complex and flexible models. One technique that is often used to control the over-fitting phenomenon in such cases is that of *regularization*, which involves adding a penalty term to the error function (1.2) in order to discourage the coefficients from reaching large values. The simplest such penalty term takes the form of a sum of squares of all of the coefficients, leading to a modified error function of the form

$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}) - t_n\}^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2 \tag{1.4}$$

where $\|\mathbf{w}\|^2 \equiv \mathbf{w}^{\mathrm{T}}\mathbf{w} = w_0^2 + w_1^2 + \ldots + w_M^2$, and the coefficient λ governs the relative importance of the regularization term compared with the sum-of-squares error term. Note that often the coefficient w_0 is omitted from the regularizer because its inclusion causes the results to depend on the choice of origin for the target variable (Hastie *et al.*, 2001), or it may be included but with its own regularization coefficient (we shall discuss this topic in more detail in Section 5.5.1). Again, the error function in (1.4) can be minimized exactly in closed form. Techniques such as this are known in the statistics literature as *shrinkage* methods because they reduce the value of the coefficients. The particular case of a quadratic regularizer is called *ridge regression* (Hoerl and Kennard, 1970). In the context of neural networks, this approach is known as *weight decay*.

Exercise 1.2

Figure 1.7 shows the results of fitting the polynomial of order $M = 9$ to the same data set as before but now using the regularized error function given by (1.4). We see that, for a value of $\ln \lambda = -18$, the over-fitting has been suppressed and we now obtain a much closer representation of the underlying function $\sin(2\pi x)$. If, however, we use too large a value for λ then we again obtain a poor fit, as shown in Figure 1.7 for $\ln \lambda = 0$. The corresponding coefficients from the fitted polynomials are given in Table 1.2, showing that regularization has the desired effect of reducing

Table 1.2 Table of the coefficients $\mathbf{w}^\star$ for $M = 9$ polynomials with various values for the regularization parameter λ. Note that $\ln \lambda = -\infty$ corresponds to a model with no regularization, i.e., to the graph at the bottom right in Figure 1.4. We see that, as the value of λ increases, the typical magnitude of the coefficients gets smaller.

	$\ln \lambda = -\infty$	$\ln \lambda = -18$	$\ln \lambda = 0$
$w_0^\star$	0.35	0.35	0.13
$w_1^\star$	232.37	4.74	-0.05
$w_2^\star$	-5321.83	-0.77	-0.06
$w_3^\star$	48568.31	-31.97	-0.05
$w_4^\star$	-231639.30	-3.89	-0.03
$w_5^\star$	640042.26	55.28	-0.02
$w_6^\star$	-1061800.52	41.32	-0.01
$w_7^\star$	1042400.18	-45.95	-0.00
$w_8^\star$	-557682.99	-91.53	0.00
$w_9^\star$	125201.43	72.68	0.01

the magnitude of the coefficients.

The impact of the regularization term on the generalization error can be seen by plotting the value of the RMS error (1.3) for both training and test sets against $\ln \lambda$, as shown in Figure 1.8. We see that in effect λ now controls the effective complexity of the model and hence determines the degree of over-fitting.

The issue of model complexity is an important one and will be discussed at length in Section 1.3. Here we simply note that, if we were trying to solve a practical application using this approach of minimizing an error function, we would have to find a way to determine a suitable value for the model complexity. The results above suggest a simple way of achieving this, namely by taking the available data and partitioning it into a training set, used to determine the coefficients $\mathbf{w}$, and a separate *validation* set, also called a *hold-out* set, used to optimize the model complexity (either M or λ). In many cases, however, this will prove to be too wasteful of valuable training data, and we have to seek more sophisticated approaches.

Section 1.3

So far our discussion of polynomial curve fitting has appealed largely to intuition. We now seek a more principled approach to solving problems in pattern recognition by turning to a discussion of probability theory. As well as providing the foundation for nearly all of the subsequent developments in this book, it will also

Figure 1.8 Graph of the root-mean-square error (1.3) versus $\ln \lambda$ for the $M = 9$ polynomial.

give us some important insights into the concepts we have introduced in the context of polynomial curve fitting and will allow us to extend these to more complex situations.

1.2. Probability Theory

A key concept in the field of pattern recognition is that of uncertainty. It arises both through noise on measurements, as well as through the finite size of data sets. Probability theory provides a consistent framework for the quantification and manipulation of uncertainty and forms one of the central foundations for pattern recognition. When combined with decision theory, discussed in Section 1.5, it allows us to make optimal predictions given all the information available to us, even though that information may be incomplete or ambiguous.

We will introduce the basic concepts of probability theory by considering a simple example. Imagine we have two boxes, one red and one blue, and in the red box we have 2 apples and 6 oranges, and in the blue box we have 3 apples and 1 orange. This is illustrated in Figure 1.9. Now suppose we randomly pick one of the boxes and from that box we randomly select an item of fruit, and having observed which sort of fruit it is we replace it in the box from which it came. We could imagine repeating this process many times. Let us suppose that in so doing we pick the red box 40% of the time and we pick the blue box 60% of the time, and that when we remove an item of fruit from a box we are equally likely to select any of the pieces of fruit in the box.

In this example, the identity of the box that will be chosen is a random variable, which we shall denote by B. This random variable can take one of two possible values, namely r (corresponding to the red box) or b (corresponding to the blue box). Similarly, the identity of the fruit is also a random variable and will be denoted by F. It can take either of the values a (for apple) or o (for orange).

To begin with, we shall define the probability of an event to be the fraction of times that event occurs out of the total number of trials, in the limit that the total number of trials goes to infinity. Thus the probability of selecting the red box is $4/10$

Figure 1.9 We use a simple example of two coloured boxes each containing fruit (apples shown in green and oranges shown in orange) to introduce the basic ideas of probability.

Figure 1.10 We can derive the sum and product rules of probability by considering two random variables, X, which takes the values $\{x_i\}$ where $i = 1, \ldots, M$, and Y, which takes the values $\{y_j\}$ where $j = 1, \ldots, L$. In this illustration we have $M = 5$ and $L = 3$. If we consider a total number N of instances of these variables, then we denote the number of instances where $X = x_i$ and $Y = y_j$ by n_{ij}, which is the number of points in the corresponding cell of the array. The number of points in column i, corresponding to $X = x_i$, is denoted by c_i, and the number of points in row j, corresponding to $Y = y_j$, is denoted by r_j.

and the probability of selecting the blue box is $6/10$. We write these probabilities as $p(B = r) = 4/10$ and $p(B = b) = 6/10$. Note that, by definition, probabilities must lie in the interval $[0, 1]$. Also, if the events are mutually exclusive and if they include all possible outcomes (for instance, in this example the box must be either red or blue), then we see that the probabilities for those events must sum to one.

We can now ask questions such as: "what is the overall probability that the selection procedure will pick an apple?", or "given that we have chosen an orange, what is the probability that the box we chose was the blue one?". We can answer questions such as these, and indeed much more complex questions associated with problems in pattern recognition, once we have equipped ourselves with the two elementary rules of probability, known as the *sum rule* and the *product rule*. Having obtained these rules, we shall then return to our boxes of fruit example.

In order to derive the rules of probability, consider the slightly more general example shown in Figure 1.10 involving two random variables X and Y (which could for instance be the Box and Fruit variables considered above). We shall suppose that X can take any of the values x_i where $i = 1, \ldots, M$, and Y can take the values y_j where $j = 1, \ldots, L$. Consider a total of N trials in which we sample both of the variables X and Y, and let the number of such trials in which $X = x_i$ and $Y = y_j$ be n_{ij}. Also, let the number of trials in which X takes the value x_i (irrespective of the value that Y takes) be denoted by c_i, and similarly let the number of trials in which Y takes the value y_j be denoted by r_j.

The probability that X will take the value x_i and Y will take the value y_j is written $p(X = x_i, Y = y_j)$ and is called the *joint* probability of $X = x_i$ and $Y = y_j$. It is given by the number of points falling in the cell i,j as a fraction of the total number of points, and hence

$$p(X = x_i, Y = y_j) = \frac{n_{ij}}{N}. \tag{1.5}$$

Here we are implicitly considering the limit $N \to \infty$. Similarly, the probability that X takes the value x_i irrespective of the value of Y is written as $p(X = x_i)$ and is given by the fraction of the total number of points that fall in column i, so that

$$p(X = x_i) = \frac{c_i}{N}. \tag{1.6}$$

Because the number of instances in column i in Figure 1.10 is just the sum of the number of instances in each cell of that column, we have $c_i = \sum_j n_{ij}$ and therefore,

from (1.5) and (1.6), we have

$$p(X = x_i) = \sum_{j=1}^{L} p(X = x_i, Y = y_j) \tag{1.7}$$

which is the *sum rule* of probability. Note that $p(X = x_i)$ is sometimes called the *marginal* probability, because it is obtained by marginalizing, or summing out, the other variables (in this case Y).

If we consider only those instances for which $X = x_i$, then the fraction of such instances for which $Y = y_j$ is written $p(Y = y_j|X = x_i)$ and is called the *conditional* probability of $Y = y_j$ given $X = x_i$. It is obtained by finding the fraction of those points in column i that fall in cell i,j and hence is given by

$$p(Y = y_j|X = x_i) = \frac{n_{ij}}{c_i}. \tag{1.8}$$

From (1.5), (1.6), and (1.8), we can then derive the following relationship

$$\begin{aligned} p(X = x_i, Y = y_j) &= \frac{n_{ij}}{N} = \frac{n_{ij}}{c_i} \cdot \frac{c_i}{N} \\ &= p(Y = y_j|X = x_i)p(X = x_i) \end{aligned} \tag{1.9}$$

which is the *product rule* of probability.

So far we have been quite careful to make a distinction between a random variable, such as the box B in the fruit example, and the values that the random variable can take, for example r if the box were the red one. Thus the probability that B takes the value r is denoted $p(B = r)$. Although this helps to avoid ambiguity, it leads to a rather cumbersome notation, and in many cases there will be no need for such pedantry. Instead, we may simply write $p(B)$ to denote a distribution over the random variable B, or $p(r)$ to denote the distribution evaluated for the particular value r, provided that the interpretation is clear from the context.

With this more compact notation, we can write the two fundamental rules of probability theory in the following form.

The Rules of Probability

sum rule $$p(X) = \sum_{Y} p(X, Y) \tag{1.10}$$

product rule $$p(X, Y) = p(Y|X)p(X). \tag{1.11}$$

Here $p(X, Y)$ is a joint probability and is verbalized as "the probability of X *and* Y". Similarly, the quantity $p(Y|X)$ is a conditional probability and is verbalized as "the probability of Y *given* X", whereas the quantity $p(X)$ is a marginal probability

and is simply "the probability of X". These two simple rules form the basis for all of the probabilistic machinery that we use throughout this book.

From the product rule, together with the symmetry property $p(X, Y) = p(Y, X)$, we immediately obtain the following relationship between conditional probabilities

$$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)} \tag{1.12}$$

which is called *Bayes' theorem* and which plays a central role in pattern recognition and machine learning. Using the sum rule, the denominator in Bayes' theorem can be expressed in terms of the quantities appearing in the numerator

$$p(X) = \sum_Y p(X|Y)p(Y). \tag{1.13}$$

We can view the denominator in Bayes' theorem as being the normalization constant required to ensure that the sum of the conditional probability on the left-hand side of (1.12) over all values of Y equals one.

In Figure 1.11, we show a simple example involving a joint distribution over two variables to illustrate the concept of marginal and conditional distributions. Here a finite sample of $N = 60$ data points has been drawn from the joint distribution and is shown in the top left. In the top right is a histogram of the fractions of data points having each of the two values of Y. From the definition of probability, these fractions would equal the corresponding probabilities $p(Y)$ in the limit $N \to \infty$. We can view the histogram as a simple way to model a probability distribution given only a finite number of points drawn from that distribution. Modelling distributions from data lies at the heart of statistical pattern recognition and will be explored in great detail in this book. The remaining two plots in Figure 1.11 show the corresponding histogram estimates of $p(X)$ and $p(X|Y = 1)$.

Let us now return to our example involving boxes of fruit. For the moment, we shall once again be explicit about distinguishing between the random variables and their instantiations. We have seen that the probabilities of selecting either the red or the blue boxes are given by

$$p(B = r) = 4/10 \tag{1.14}$$
$$p(B = b) = 6/10 \tag{1.15}$$

respectively. Note that these satisfy $p(B = r) + p(B = b) = 1$.

Now suppose that we pick a box at random, and it turns out to be the blue box. Then the probability of selecting an apple is just the fraction of apples in the blue box which is $3/4$, and so $p(F = a|B = b) = 3/4$. In fact, we can write out all four conditional probabilities for the type of fruit, given the selected box

$$p(F = a|B = r) = 1/4 \tag{1.16}$$
$$p(F = o|B = r) = 3/4 \tag{1.17}$$
$$p(F = a|B = b) = 3/4 \tag{1.18}$$
$$p(F = o|B = b) = 1/4. \tag{1.19}$$

Figure 1.11 An illustration of a distribution over two variables, X, which takes 9 possible values, and Y, which takes two possible values. The top left figure shows a sample of 60 points drawn from a joint probability distribution over these variables. The remaining figures show histogram estimates of the marginal distributions $p(X)$ and $p(Y)$, as well as the conditional distribution $p(X|Y = 1)$ corresponding to the bottom row in the top left figure.

Again, note that these probabilities are normalized so that

$$p(F = a|B = r) + p(F = o|B = r) = 1 \tag{1.20}$$

and similarly

$$p(F = a|B = b) + p(F = o|B = b) = 1. \tag{1.21}$$

We can now use the sum and product rules of probability to evaluate the overall probability of choosing an apple

$$\begin{aligned} p(F = a) &= p(F = a|B = r)p(B = r) + p(F = a|B = b)p(B = b) \\ &= \frac{1}{4} \times \frac{4}{10} + \frac{3}{4} \times \frac{6}{10} = \frac{11}{20} \end{aligned} \tag{1.22}$$

from which it follows, using the sum rule, that $p(F = o) = 1 - 11/20 = 9/20$.

Suppose instead we are told that a piece of fruit has been selected and it is an orange, and we would like to know which box it came from. This requires that we evaluate the probability distribution over boxes conditioned on the identity of the fruit, whereas the probabilities in (1.16)–(1.19) give the probability distribution over the fruit conditioned on the identity of the box. We can solve the problem of reversing the conditional probability by using Bayes' theorem to give

$$p(B = r|F = o) = \frac{p(F = o|B = r)p(B = r)}{p(F = o)} = \frac{3}{4} \times \frac{4}{10} \times \frac{20}{9} = \frac{2}{3}. \tag{1.23}$$

From the sum rule, it then follows that $p(B = b|F = o) = 1 - 2/3 = 1/3$.

We can provide an important interpretation of Bayes' theorem as follows. If we had been asked which box had been chosen before being told the identity of the selected item of fruit, then the most complete information we have available is provided by the probability $p(B)$. We call this the *prior probability* because it is the probability available *before* we observe the identity of the fruit. Once we are told that the fruit is an orange, we can then use Bayes' theorem to compute the probability $p(B|F)$, which we shall call the *posterior probability* because it is the probability obtained *after* we have observed F. Note that in this example, the prior probability of selecting the red box was $4/10$, so that we were more likely to select the blue box than the red one. However, once we have observed that the piece of selected fruit is an orange, we find that the posterior probability of the red box is now $2/3$, so that it is now more likely that the box we selected was in fact the red one. This result accords with our intuition, as the proportion of oranges is much higher in the red box than it is in the blue box, and so the observation that the fruit was an orange provides significant evidence favouring the red box. In fact, the evidence is sufficiently strong that it outweighs the prior and makes it more likely that the red box was chosen rather than the blue one.

Finally, we note that if the joint distribution of two variables factorizes into the product of the marginals, so that $p(X, Y) = p(X)p(Y)$, then X and Y are said to be *independent*. From the product rule, we see that $p(Y|X) = p(Y)$, and so the conditional distribution of Y given X is indeed independent of the value of X. For instance, in our boxes of fruit example, if each box contained the same fraction of apples and oranges, then $p(F|B) = P(F)$, so that the probability of selecting, say, an apple is independent of which box is chosen.

1.2.1 Probability densities

As well as considering probabilities defined over discrete sets of events, we also wish to consider probabilities with respect to continuous variables. We shall limit ourselves to a relatively informal discussion. If the probability of a real-valued variable x falling in the interval $(x, x + \delta x)$ is given by $p(x)\delta x$ for $\delta x \to 0$, then $p(x)$ is called the *probability density* over x. This is illustrated in Figure 1.12. The probability that x will lie in an interval (a, b) is then given by

$$p(x \in (a, b)) = \int_a^b p(x)\,\mathrm{d}x. \tag{1.24}$$

Figure 1.12 The concept of probability for discrete variables can be extended to that of a probability density $p(x)$ over a continuous variable x and is such that the probability of x lying in the interval $(x, x+\delta x)$ is given by $p(x)\delta x$ for $\delta x \to 0$. The probability density can be expressed as the derivative of a cumulative distribution function $P(x)$.

Because probabilities are nonnegative, and because the value of x must lie somewhere on the real axis, the probability density $p(x)$ must satisfy the two conditions

$$p(x) \geqslant 0 \tag{1.25}$$

$$\int_{-\infty}^{\infty} p(x)\,\mathrm{d}x = 1. \tag{1.26}$$

Under a nonlinear change of variable, a probability density transforms differently from a simple function, due to the Jacobian factor. For instance, if we consider a change of variables $x = g(y)$, then a function $f(x)$ becomes $\widetilde{f}(y) = f(g(y))$. Now consider a probability density $p_x(x)$ that corresponds to a density $p_y(y)$ with respect to the new variable y, where the suffices denote the fact that $p_x(x)$ and $p_y(y)$ are different densities. Observations falling in the range $(x, x+\delta x)$ will, for small values of δx, be transformed into the range $(y, y+\delta y)$ where $p_x(x)\delta x \simeq p_y(y)\delta y$, and hence

$$\begin{aligned} p_y(y) &= p_x(x)\left|\frac{\mathrm{d}x}{\mathrm{d}y}\right| \\ &= p_x(g(y))\,|g'(y)|\,. \end{aligned} \tag{1.27}$$

One consequence of this property is that the concept of the maximum of a probability density is dependent on the choice of variable.

Exercise 1.4

The probability that x lies in the interval $(-\infty, z)$ is given by the *cumulative distribution function* defined by

$$P(z) = \int_{-\infty}^{z} p(x)\,\mathrm{d}x \tag{1.28}$$

which satisfies $P'(x) = p(x)$, as shown in Figure 1.12.

If we have several continuous variables $x_1, \ldots, x_D$, denoted collectively by the vector $\mathbf{x}$, then we can define a joint probability density $p(\mathbf{x}) = p(x_1, \ldots, x_D)$ such

that the probability of $\mathbf{x}$ falling in an infinitesimal volume $\delta\mathbf{x}$ containing the point $\mathbf{x}$ is given by $p(\mathbf{x})\delta\mathbf{x}$. This multivariate probability density must satisfy

$$p(\mathbf{x}) \geqslant 0 \tag{1.29}$$

$$\int p(\mathbf{x})\,\mathrm{d}\mathbf{x} = 1 \tag{1.30}$$

in which the integral is taken over the whole of $\mathbf{x}$ space. We can also consider joint probability distributions over a combination of discrete and continuous variables.

Note that if x is a discrete variable, then $p(x)$ is sometimes called a *probability mass function* because it can be regarded as a set of 'probability masses' concentrated at the allowed values of x.

The sum and product rules of probability, as well as Bayes' theorem, apply equally to the case of probability densities, or to combinations of discrete and continuous variables. For instance, if x and y are two real variables, then the sum and product rules take the form

$$p(x) = \int p(x,y)\,\mathrm{d}y \tag{1.31}$$

$$p(x,y) = p(y|x)p(x). \tag{1.32}$$

A formal justification of the sum and product rules for continuous variables (Feller, 1966) requires a branch of mathematics called *measure theory* and lies outside the scope of this book. Its validity can be seen informally, however, by dividing each real variable into intervals of width Δ and considering the discrete probability distribution over these intervals. Taking the limit $\Delta \to 0$ then turns sums into integrals and gives the desired result.

1.2.2 Expectations and covariances

One of the most important operations involving probabilities is that of finding weighted averages of functions. The average value of some function $f(x)$ under a probability distribution $p(x)$ is called the *expectation* of $f(x)$ and will be denoted by $\mathbb{E}[f]$. For a discrete distribution, it is given by

$$\mathbb{E}[f] = \sum_x p(x)f(x) \tag{1.33}$$

so that the average is weighted by the relative probabilities of the different values of x. In the case of continuous variables, expectations are expressed in terms of an integration with respect to the corresponding probability density

$$\mathbb{E}[f] = \int p(x)f(x)\,\mathrm{d}x. \tag{1.34}$$

In either case, if we are given a finite number N of points drawn from the probability distribution or probability density, then the expectation can be approximated as a

finite sum over these points

$$\mathbb{E}[f] \simeq \frac{1}{N}\sum_{n=1}^{N} f(x_n). \tag{1.35}$$

We shall make extensive use of this result when we discuss sampling methods in Chapter 11. The approximation in (1.35) becomes exact in the limit $N \to \infty$.

Sometimes we will be considering expectations of functions of several variables, in which case we can use a subscript to indicate which variable is being averaged over, so that for instance

$$\mathbb{E}_x[f(x, y)] \tag{1.36}$$

denotes the average of the function $f(x, y)$ with respect to the distribution of x. Note that $\mathbb{E}_x[f(x, y)]$ will be a function of y.

We can also consider a *conditional expectation* with respect to a conditional distribution, so that

$$\mathbb{E}_x[f|y] = \sum_x p(x|y)f(x) \tag{1.37}$$

with an analogous definition for continuous variables.

The *variance* of $f(x)$ is defined by

$$\mathrm{var}[f] = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])^2\right] \tag{1.38}$$

and provides a measure of how much variability there is in $f(x)$ around its mean value $\mathbb{E}[f(x)]$. Expanding out the square, we see that the variance can also be written in terms of the expectations of $f(x)$ and $f(x)^2$

Exercise 1.5

$$\mathrm{var}[f] = \mathbb{E}[f(x)^2] - \mathbb{E}[f(x)]^2. \tag{1.39}$$

In particular, we can consider the variance of the variable x itself, which is given by

$$\mathrm{var}[x] = \mathbb{E}[x^2] - \mathbb{E}[x]^2. \tag{1.40}$$

For two random variables x and y, the *covariance* is defined by

$$\begin{aligned}\mathrm{cov}[x, y] &= \mathbb{E}_{x,y}\left[\{x - \mathbb{E}[x]\}\{y - \mathbb{E}[y]\}\right] \\ &= \mathbb{E}_{x,y}[xy] - \mathbb{E}[x]\mathbb{E}[y]\end{aligned} \tag{1.41}$$

which expresses the extent to which x and y vary together. If x and y are independent, then their covariance vanishes.

Exercise 1.6

In the case of two vectors of random variables $\mathbf{x}$ and $\mathbf{y}$, the covariance is a matrix

$$\begin{aligned}\mathrm{cov}[\mathbf{x}, \mathbf{y}] &= \mathbb{E}_{\mathbf{x},\mathbf{y}}\left[\{\mathbf{x} - \mathbb{E}[\mathbf{x}]\}\{\mathbf{y}^{\mathrm{T}} - \mathbb{E}[\mathbf{y}^{\mathrm{T}}]\}\right] \\ &= \mathbb{E}_{\mathbf{x},\mathbf{y}}[\mathbf{x}\mathbf{y}^{\mathrm{T}}] - \mathbb{E}[\mathbf{x}]\mathbb{E}[\mathbf{y}^{\mathrm{T}}].\end{aligned} \tag{1.42}$$

If we consider the covariance of the components of a vector $\mathbf{x}$ with each other, then we use a slightly simpler notation $\mathrm{cov}[\mathbf{x}] \equiv \mathrm{cov}[\mathbf{x}, \mathbf{x}]$.

1.2.3 Bayesian probabilities

So far in this chapter, we have viewed probabilities in terms of the frequencies of random, repeatable events. We shall refer to this as the *classical* or *frequentist* interpretation of probability. Now we turn to the more general *Bayesian* view, in which probabilities provide a quantification of uncertainty.

Consider an uncertain event, for example whether the moon was once in its own orbit around the sun, or whether the Arctic ice cap will have disappeared by the end of the century. These are not events that can be repeated numerous times in order to define a notion of probability as we did earlier in the context of boxes of fruit. Nevertheless, we will generally have some idea, for example, of how quickly we think the polar ice is melting. If we now obtain fresh evidence, for instance from a new Earth observation satellite gathering novel forms of diagnostic information, we may revise our opinion on the rate of ice loss. Our assessment of such matters will affect the actions we take, for instance the extent to which we endeavour to reduce the emission of greenhouse gasses. In such circumstances, we would like to be able to quantify our expression of uncertainty and make precise revisions of uncertainty in the light of new evidence, as well as subsequently to be able to take optimal actions or decisions as a consequence. This can all be achieved through the elegant, and very general, Bayesian interpretation of probability.

The use of probability to represent uncertainty, however, is not an ad-hoc choice, but is inevitable if we are to respect common sense while making rational coherent inferences. For instance, Cox (1946) showed that if numerical values are used to represent degrees of belief, then a simple set of axioms encoding common sense properties of such beliefs leads uniquely to a set of rules for manipulating degrees of belief that are equivalent to the sum and product rules of probability. This provided the first rigorous proof that probability theory could be regarded as an extension of Boolean logic to situations involving uncertainty (Jaynes, 2003). Numerous other authors have proposed different sets of properties or axioms that such measures of uncertainty should satisfy (Ramsey, 1931; Good, 1950; Savage, 1961; deFinetti, 1970; Lindley, 1982). In each case, the resulting numerical quantities behave precisely according to the rules of probability. It is therefore natural to refer to these quantities as (Bayesian) probabilities.

In the field of pattern recognition, too, it is helpful to have a more general no-

Thomas Bayes
1701–1761

Thomas Bayes was born in Tunbridge Wells and was a clergyman as well as an amateur scientist and a mathematician. He studied logic and theology at Edinburgh University and was elected Fellow of the Royal Society in 1742. During the 18th century, issues regarding probability arose in connection with gambling and with the new concept of insurance. One particularly important problem concerned so-called inverse probability. A solution was proposed by Thomas Bayes in his paper 'Essay towards solving a problem in the doctrine of chances', which was published in 1764, some three years after his death, in the *Philosophical Transactions of the Royal Society*. In fact, Bayes only formulated his theory for the case of a uniform prior, and it was Pierre-Simon Laplace who independently rediscovered the theory in general form and who demonstrated its broad applicability.

tion of probability. Consider the example of polynomial curve fitting discussed in Section 1.1. It seems reasonable to apply the frequentist notion of probability to the random values of the observed variables t_n. However, we would like to address and quantify the uncertainty that surrounds the appropriate choice for the model parameters $\mathbf{w}$. We shall see that, from a Bayesian perspective, we can use the machinery of probability theory to describe the uncertainty in model parameters such as $\mathbf{w}$, or indeed in the choice of model itself.

Bayes' theorem now acquires a new significance. Recall that in the boxes of fruit example, the observation of the identity of the fruit provided relevant information that altered the probability that the chosen box was the red one. In that example, Bayes' theorem was used to convert a prior probability into a posterior probability by incorporating the evidence provided by the observed data. As we shall see in detail later, we can adopt a similar approach when making inferences about quantities such as the parameters $\mathbf{w}$ in the polynomial curve fitting example. We capture our assumptions about $\mathbf{w}$, before observing the data, in the form of a prior probability distribution $p(\mathbf{w})$. The effect of the observed data $\mathcal{D} = \{t_1, \ldots, t_N\}$ is expressed through the conditional probability $p(\mathcal{D}|\mathbf{w})$, and we shall see later, in Section 1.2.5, how this can be represented explicitly. Bayes' theorem, which takes the form

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathbf{w})p(\mathbf{w})}{p(\mathcal{D})} \tag{1.43}$$

then allows us to evaluate the uncertainty in $\mathbf{w}$ *after* we have observed $\mathcal{D}$ in the form of the posterior probability $p(\mathbf{w}|\mathcal{D})$.

The quantity $p(\mathcal{D}|\mathbf{w})$ on the right-hand side of Bayes' theorem is evaluated for the observed data set $\mathcal{D}$ and can be viewed as a function of the parameter vector $\mathbf{w}$, in which case it is called the *likelihood function*. It expresses how probable the observed data set is for different settings of the parameter vector $\mathbf{w}$. Note that the likelihood is not a probability distribution over $\mathbf{w}$, and its integral with respect to $\mathbf{w}$ does not (necessarily) equal one.

Given this definition of likelihood, we can state Bayes' theorem in words

$$\text{posterior} \propto \text{likelihood} \times \text{prior} \tag{1.44}$$

where all of these quantities are viewed as functions of $\mathbf{w}$. The denominator in (1.43) is the normalization constant, which ensures that the posterior distribution on the left-hand side is a valid probability density and integrates to one. Indeed, integrating both sides of (1.43) with respect to $\mathbf{w}$, we can express the denominator in Bayes' theorem in terms of the prior distribution and the likelihood function

$$p(\mathcal{D}) = \int p(\mathcal{D}|\mathbf{w})p(\mathbf{w})\,\mathrm{d}\mathbf{w}. \tag{1.45}$$

In both the Bayesian and frequentist paradigms, the likelihood function $p(\mathcal{D}|\mathbf{w})$ plays a central role. However, the manner in which it is used is fundamentally different in the two approaches. In a frequentist setting, $\mathbf{w}$ is considered to be a fixed parameter, whose value is determined by some form of 'estimator', and error bars

on this estimate are obtained by considering the distribution of possible data sets $\mathcal{D}$. By contrast, from the Bayesian viewpoint there is only a single data set $\mathcal{D}$ (namely the one that is actually observed), and the uncertainty in the parameters is expressed through a probability distribution over $\mathbf{w}$.

A widely used frequentist estimator is *maximum likelihood*, in which $\mathbf{w}$ is set to the value that maximizes the likelihood function $p(\mathcal{D}|\mathbf{w})$. This corresponds to choosing the value of $\mathbf{w}$ for which the probability of the observed data set is maximized. In the machine learning literature, the negative log of the likelihood function is called an *error function*. Because the negative logarithm is a monotonically decreasing function, maximizing the likelihood is equivalent to minimizing the error.

One approach to determining frequentist error bars is the *bootstrap* (Efron, 1979; Hastie *et al.*, 2001), in which multiple data sets are created as follows. Suppose our original data set consists of N data points $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$. We can create a new data set $\mathbf{X}_\mathrm{B}$ by drawing N points at random from $\mathbf{X}$, with replacement, so that some points in $\mathbf{X}$ may be replicated in $\mathbf{X}_\mathrm{B}$, whereas other points in $\mathbf{X}$ may be absent from $\mathbf{X}_\mathrm{B}$. This process can be repeated L times to generate L data sets each of size N and each obtained by sampling from the original data set $\mathbf{X}$. The statistical accuracy of parameter estimates can then be evaluated by looking at the variability of predictions between the different bootstrap data sets.

One advantage of the Bayesian viewpoint is that the inclusion of prior knowledge arises naturally. Suppose, for instance, that a fair-looking coin is tossed three times and lands heads each time. A classical maximum likelihood estimate of the probability of landing heads would give 1, implying that all future tosses will land heads! By contrast, a Bayesian approach with any reasonable prior will lead to a much less extreme conclusion.

Section 2.1

There has been much controversy and debate associated with the relative merits of the frequentist and Bayesian paradigms, which have not been helped by the fact that there is no unique frequentist, or even Bayesian, viewpoint. For instance, one common criticism of the Bayesian approach is that the prior distribution is often selected on the basis of mathematical convenience rather than as a reflection of any prior beliefs. Even the subjective nature of the conclusions through their dependence on the choice of prior is seen by some as a source of difficulty. Reducing the dependence on the prior is one motivation for so-called *noninformative* priors. However, these lead to difficulties when comparing different models, and indeed Bayesian methods based on poor choices of prior can give poor results with high confidence. Frequentist evaluation methods offer some protection from such problems, and techniques such as cross-validation remain useful in areas such as model comparison.

Section 2.4.3

Section 1.3

This book places a strong emphasis on the Bayesian viewpoint, reflecting the huge growth in the practical importance of Bayesian methods in the past few years, while also discussing useful frequentist concepts as required.

Although the Bayesian framework has its origins in the 18th century, the practical application of Bayesian methods was for a long time severely limited by the difficulties in carrying through the full Bayesian procedure, particularly the need to marginalize (sum or integrate) over the whole of parameter space, which, as we shall

see, is required in order to make predictions or to compare different models. The development of sampling methods, such as Markov chain Monte Carlo (discussed in Chapter 11) along with dramatic improvements in the speed and memory capacity of computers, opened the door to the practical use of Bayesian techniques in an impressive range of problem domains. Monte Carlo methods are very flexible and can be applied to a wide range of models. However, they are computationally intensive and have mainly been used for small-scale problems.

More recently, highly efficient deterministic approximation schemes such as variational Bayes and expectation propagation (discussed in Chapter 10) have been developed. These offer a complementary alternative to sampling methods and have allowed Bayesian techniques to be used in large-scale applications (Blei *et al.*, 2003).

1.2.4 The Gaussian distribution

We shall devote the whole of Chapter 2 to a study of various probability distributions and their key properties. It is convenient, however, to introduce here one of the most important probability distributions for continuous variables, called the *normal* or *Gaussian* distribution. We shall make extensive use of this distribution in the remainder of this chapter and indeed throughout much of the book.

For the case of a single real-valued variable x, the Gaussian distribution is defined by

$$\mathcal{N}\left(x|\mu,\sigma^2\right) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\} \tag{1.46}$$

which is governed by two parameters: μ, called the *mean*, and σ^2, called the *variance*. The square root of the variance, given by σ, is called the *standard deviation*, and the reciprocal of the variance, written as $\beta = 1/\sigma^2$, is called the *precision*. We shall see the motivation for these terms shortly. Figure 1.13 shows a plot of the Gaussian distribution.

From the form of (1.46) we see that the Gaussian distribution satisfies

$$\mathcal{N}(x|\mu,\sigma^2) > 0. \tag{1.47}$$

Exercise 1.7 Also it is straightforward to show that the Gaussian is normalized, so that

Pierre-Simon Laplace
1749–1827

It is said that Laplace was seriously lacking in modesty and at one point declared himself to be the best mathematician in France at the time, a claim that was arguably true. As well as being prolific in mathematics, he also made numerous contributions to astronomy, including the nebular hypothesis by which the earth is thought to have formed from the condensation and cooling of a large rotating disk of gas and dust. In 1812 he published the first edition of *Théorie Analytique des Probabilités*, in which Laplace states that "probability theory is nothing but common sense reduced to calculation". This work included a discussion of the inverse probability calculation (later termed Bayes' theorem by Poincaré), which he used to solve problems in life expectancy, jurisprudence, planetary masses, triangulation, and error estimation.

Figure 1.13 Plot of the univariate Gaussian showing the mean μ and the standard deviation σ.

$$\int_{-\infty}^{\infty} \mathcal{N}\left(x|\mu,\sigma^2\right)\,\mathrm{d}x = 1. \tag{1.48}$$

Thus (1.46) satisfies the two requirements for a valid probability density.

We can readily find expectations of functions of x under the Gaussian distribution. In particular, the average value of x is given by

Exercise 1.8

$$\mathbb{E}[x] = \int_{-\infty}^{\infty} \mathcal{N}\left(x|\mu,\sigma^2\right) x\,\mathrm{d}x = \mu. \tag{1.49}$$

Because the parameter μ represents the average value of x under the distribution, it is referred to as the mean. Similarly, for the second order moment

$$\mathbb{E}[x^2] = \int_{-\infty}^{\infty} \mathcal{N}\left(x|\mu,\sigma^2\right) x^2\,\mathrm{d}x = \mu^2 + \sigma^2. \tag{1.50}$$

From (1.49) and (1.50), it follows that the variance of x is given by

$$\mathrm{var}[x] = \mathbb{E}[x^2] - \mathbb{E}[x]^2 = \sigma^2 \tag{1.51}$$

and hence σ^2 is referred to as the variance parameter. The maximum of a distribution is known as its mode. For a Gaussian, the mode coincides with the mean.

Exercise 1.9

We are also interested in the Gaussian distribution defined over a D-dimensional vector $\mathbf{x}$ of continuous variables, which is given by

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}}\frac{1}{|\boldsymbol{\Sigma}|^{1/2}}\exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right\} \tag{1.52}$$

where the D-dimensional vector $\boldsymbol{\mu}$ is called the mean, the $D \times D$ matrix $\boldsymbol{\Sigma}$ is called the covariance, and $|\boldsymbol{\Sigma}|$ denotes the determinant of $\boldsymbol{\Sigma}$. We shall make use of the multivariate Gaussian distribution briefly in this chapter, although its properties will be studied in detail in Section 2.3.

Figure 1.14 Illustration of the likelihood function for a Gaussian distribution, shown by the red curve. Here the black points denote a data set of values $\{x_n\}$, and the likelihood function given by (1.53) corresponds to the product of the blue values. Maximizing the likelihood involves adjusting the mean and variance of the Gaussian so as to maximize this product.

Now suppose that we have a data set of observations $\mathbf{x} = (x_1, \ldots, x_N)^{\mathrm{T}}$, representing N observations of the scalar variable x. Note that we are using the typeface $\mathbf{x}$ to distinguish this from a single observation of the vector-valued variable $(x_1, \ldots, x_D)^{\mathrm{T}}$, which we denote by $\mathbf{x}$. We shall suppose that the observations are drawn independently from a Gaussian distribution whose mean μ and variance σ^2 are unknown, and we would like to determine these parameters from the data set. Data points that are drawn independently from the same distribution are said to be *independent and identically distributed*, which is often abbreviated to i.i.d. We have seen that the joint probability of two independent events is given by the product of the marginal probabilities for each event separately. Because our data set $\mathbf{x}$ is i.i.d., we can therefore write the probability of the data set, given μ and σ^2, in the form

$$p(\mathbf{x}|\mu, \sigma^2) = \prod_{n=1}^{N} \mathcal{N}\left(x_n|\mu, \sigma^2\right). \tag{1.53}$$

When viewed as a function of μ and σ^2, this is the likelihood function for the Gaussian and is interpreted diagrammatically in Figure 1.14.

One common criterion for determining the parameters in a probability distribution using an observed data set is to find the parameter values that maximize the likelihood function. This might seem like a strange criterion because, from our foregoing discussion of probability theory, it would seem more natural to maximize the probability of the parameters given the data, not the probability of the data given the parameters. In fact, these two criteria are related, as we shall discuss in the context of curve fitting.

Section 1.2.5

For the moment, however, we shall determine values for the unknown parameters μ and σ^2 in the Gaussian by maximizing the likelihood function (1.53). In practice, it is more convenient to maximize the log of the likelihood function. Because the logarithm is a monotonically increasing function of its argument, maximization of the log of a function is equivalent to maximization of the function itself. Taking the log not only simplifies the subsequent mathematical analysis, but it also helps numerically because the product of a large number of small probabilities can easily underflow the numerical precision of the computer, and this is resolved by computing instead the sum of the log probabilities. From (1.46) and (1.53), the log likelihood

function can be written in the form

$$\ln p\left(\mathbf{x}|\mu,\sigma^2\right) = -\frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n-\mu)^2 - \frac{N}{2}\ln\sigma^2 - \frac{N}{2}\ln(2\pi). \tag{1.54}$$

Maximizing (1.54) with respect to μ, we obtain the maximum likelihood solution given by

Exercise 1.11

$$\mu_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}x_n \tag{1.55}$$

which is the *sample mean*, i.e., the mean of the observed values $\{x_n\}$. Similarly, maximizing (1.54) with respect to σ^2, we obtain the maximum likelihood solution for the variance in the form

$$\sigma^2_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}(x_n-\mu_{\mathrm{ML}})^2 \tag{1.56}$$

which is the *sample variance* measured with respect to the sample mean μ_{ML}. Note that we are performing a joint maximization of (1.54) with respect to μ and σ^2, but in the case of the Gaussian distribution the solution for μ decouples from that for σ^2 so that we can first evaluate (1.55) and then subsequently use this result to evaluate (1.56).

Later in this chapter, and also in subsequent chapters, we shall highlight the significant limitations of the maximum likelihood approach. Here we give an indication of the problem in the context of our solutions for the maximum likelihood parameter settings for the univariate Gaussian distribution. In particular, we shall show that the maximum likelihood approach systematically underestimates the variance of the distribution. This is an example of a phenomenon called *bias* and is related to the problem of over-fitting encountered in the context of polynomial curve fitting. We first note that the maximum likelihood solutions μ_{ML} and σ^2_{ML} are functions of the data set values $x_1,\ldots,x_N$. Consider the expectations of these quantities with respect to the data set values, which themselves come from a Gaussian distribution with parameters μ and σ^2. It is straightforward to show that

Section 1.1

Exercise 1.12

$$\mathbb{E}[\mu_{\mathrm{ML}}] = \mu \tag{1.57}$$

$$\mathbb{E}[\sigma^2_{\mathrm{ML}}] = \left(\frac{N-1}{N}\right)\sigma^2 \tag{1.58}$$

so that on average the maximum likelihood estimate will obtain the correct mean but will underestimate the true variance by a factor $(N-1)/N$. The intuition behind this result is given by Figure 1.15.

From (1.58) it follows that the following estimate for the variance parameter is unbiased

$$\widetilde{\sigma}^2 = \frac{N}{N-1}\sigma^2_{\mathrm{ML}} = \frac{1}{N-1}\sum_{n=1}^{N}(x_n-\mu_{\mathrm{ML}})^2. \tag{1.59}$$

Figure 1.15 Illustration of how bias arises in using maximum likelihood to determine the variance of a Gaussian. The green curve shows the true Gaussian distribution from which data is generated, and the three red curves show the Gaussian distributions obtained by fitting to three data sets, each consisting of two data points shown in blue, using the maximum likelihood results (1.55) and (1.56). Averaged across the three data sets, the mean is correct, but the variance is systematically under-estimated because it is measured relative to the sample mean and not relative to the true mean.

In Section 10.1.3, we shall see how this result arises automatically when we adopt a Bayesian approach.

Note that the bias of the maximum likelihood solution becomes less significant as the number N of data points increases, and in the limit $N \to \infty$ the maximum likelihood solution for the variance equals the true variance of the distribution that generated the data. In practice, for anything other than small N, this bias will not prove to be a serious problem. However, throughout this book we shall be interested in more complex models with many parameters, for which the bias problems associated with maximum likelihood will be much more severe. In fact, as we shall see, the issue of bias in maximum likelihood lies at the root of the over-fitting problem that we encountered earlier in the context of polynomial curve fitting.

1.2.5 Curve fitting re-visited

Section 1.1

We have seen how the problem of polynomial curve fitting can be expressed in terms of error minimization. Here we return to the curve fitting example and view it from a probabilistic perspective, thereby gaining some insights into error functions and regularization, as well as taking us towards a full Bayesian treatment.

The goal in the curve fitting problem is to be able to make predictions for the target variable t given some new value of the input variable x on the basis of a set of training data comprising N input values $\mathbf{x} = (x_1, \ldots, x_N)^{\mathrm{T}}$ and their corresponding target values $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$. We can express our uncertainty over the value of the target variable using a probability distribution. For this purpose, we shall assume that, given the value of x, the corresponding value of t has a Gaussian distribution with a mean equal to the value $y(x, \mathbf{w})$ of the polynomial curve given by (1.1). Thus we have

$$p(t|x, \mathbf{w}, \beta) = \mathcal{N}\left(t|y(x, \mathbf{w}), \beta^{-1}\right) \tag{1.60}$$

where, for consistency with the notation in later chapters, we have defined a precision parameter β corresponding to the inverse variance of the distribution. This is illustrated schematically in Figure 1.16.

Figure 1.16 Schematic illustration of a Gaussian conditional distribution for t given x given by (1.60), in which the mean is given by the polynomial function $y(x, \mathbf{w})$, and the precision is given by the parameter β, which is related to the variance by $\beta^{-1} = \sigma^2$.

We now use the training data $\{\mathbf{x}, \mathbf{t}\}$ to determine the values of the unknown parameters $\mathbf{w}$ and β by maximum likelihood. If the data are assumed to be drawn independently from the distribution (1.60), then the likelihood function is given by

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}\left(t_n | y(x_n, \mathbf{w}), \beta^{-1}\right). \tag{1.61}$$

As we did in the case of the simple Gaussian distribution earlier, it is convenient to maximize the logarithm of the likelihood function. Substituting for the form of the Gaussian distribution, given by (1.46), we obtain the log likelihood function in the form

$$\ln p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \beta) = -\frac{\beta}{2} \sum_{n=1}^{N} \{y(x_n, \mathbf{w}) - t_n\}^2 + \frac{N}{2} \ln \beta - \frac{N}{2} \ln(2\pi). \tag{1.62}$$

Consider first the determination of the maximum likelihood solution for the polynomial coefficients, which will be denoted by $\mathbf{w}_{\mathrm{ML}}$. These are determined by maximizing (1.62) with respect to $\mathbf{w}$. For this purpose, we can omit the last two terms on the right-hand side of (1.62) because they do not depend on $\mathbf{w}$. Also, we note that scaling the log likelihood by a positive constant coefficient does not alter the location of the maximum with respect to $\mathbf{w}$, and so we can replace the coefficient $\beta/2$ with $1/2$. Finally, instead of maximizing the log likelihood, we can equivalently minimize the negative log likelihood. We therefore see that maximizing likelihood is equivalent, so far as determining $\mathbf{w}$ is concerned, to minimizing the *sum-of-squares error function* defined by (1.2). Thus the sum-of-squares error function has arisen as a consequence of maximizing likelihood under the assumption of a Gaussian noise distribution.

We can also use maximum likelihood to determine the precision parameter β of the Gaussian conditional distribution. Maximizing (1.62) with respect to β gives

$$\frac{1}{\beta_{\mathrm{ML}}} = \frac{1}{N} \sum_{n=1}^{N} \{y(x_n, \mathbf{w}_{\mathrm{ML}}) - t_n\}^2. \tag{1.63}$$

Again we can first determine the parameter vector $\mathbf{w}_{\mathrm{ML}}$ governing the mean and subsequently use this to find the precision β_{ML} as was the case for the simple Gaussian distribution.

Section 1.2.4

Having determined the parameters $\mathbf{w}$ and β, we can now make predictions for new values of x. Because we now have a probabilistic model, these are expressed in terms of the *predictive distribution* that gives the probability distribution over t, rather than simply a point estimate, and is obtained by substituting the maximum likelihood parameters into (1.60) to give

$$p(t|x, \mathbf{w}_{\mathrm{ML}}, \beta_{\mathrm{ML}}) = \mathcal{N}\left(t|y(x, \mathbf{w}_{\mathrm{ML}}), \beta_{\mathrm{ML}}^{-1}\right). \tag{1.64}$$

Now let us take a step towards a more Bayesian approach and introduce a prior distribution over the polynomial coefficients $\mathbf{w}$. For simplicity, let us consider a Gaussian distribution of the form

$$p(\mathbf{w}|\alpha) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I}) = \left(\frac{\alpha}{2\pi}\right)^{(M+1)/2} \exp\left\{-\frac{\alpha}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}\right\} \tag{1.65}$$

where α is the precision of the distribution, and $M+1$ is the total number of elements in the vector $\mathbf{w}$ for an Mth order polynomial. Variables such as α, which control the distribution of model parameters, are called *hyperparameters*. Using Bayes' theorem, the posterior distribution for $\mathbf{w}$ is proportional to the product of the prior distribution and the likelihood function

$$p(\mathbf{w}|\mathbf{x}, \mathbf{t}, \alpha, \beta) \propto p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \beta)p(\mathbf{w}|\alpha). \tag{1.66}$$

We can now determine $\mathbf{w}$ by finding the most probable value of $\mathbf{w}$ given the data, in other words by maximizing the posterior distribution. This technique is called *maximum posterior*, or simply *MAP*. Taking the negative logarithm of (1.66) and combining with (1.62) and (1.65), we find that the maximum of the posterior is given by the minimum of

$$\frac{\beta}{2}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}) - t_n\}^2 + \frac{\alpha}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}. \tag{1.67}$$

Thus we see that maximizing the posterior distribution is equivalent to minimizing the regularized sum-of-squares error function encountered earlier in the form (1.4), with a regularization parameter given by $\lambda = \alpha/\beta$.

1.2.6 Bayesian curve fitting

Although we have included a prior distribution $p(\mathbf{w}|\alpha)$, we are so far still making a point estimate of $\mathbf{w}$ and so this does not yet amount to a Bayesian treatment. In a fully Bayesian approach, we should consistently apply the sum and product rules of probability, which requires, as we shall see shortly, that we integrate over all values of $\mathbf{w}$. Such marginalizations lie at the heart of Bayesian methods for pattern recognition.

In the curve fitting problem, we are given the training data $\mathbf{x}$ and $\mathbf{t}$, along with a new test point x, and our goal is to predict the value of t. We therefore wish to evaluate the predictive distribution $p(t|x, \mathbf{x}, \mathbf{t})$. Here we shall assume that the parameters α and β are fixed and known in advance (in later chapters we shall discuss how such parameters can be inferred from data in a Bayesian setting).

A Bayesian treatment simply corresponds to a consistent application of the sum and product rules of probability, which allow the predictive distribution to be written in the form

$$p(t|x, \mathbf{x}, \mathbf{t}) = \int p(t|x, \mathbf{w}) p(\mathbf{w}|\mathbf{x}, \mathbf{t}) \, \mathrm{d}\mathbf{w}. \tag{1.68}$$

Here $p(t|x, \mathbf{w})$ is given by (1.60), and we have omitted the dependence on α and β to simplify the notation. Here $p(\mathbf{w}|\mathbf{x}, \mathbf{t})$ is the posterior distribution over parameters, and can be found by normalizing the right-hand side of (1.66). We shall see in Section 3.3 that, for problems such as the curve-fitting example, this posterior distribution is a Gaussian and can be evaluated analytically. Similarly, the integration in (1.68) can also be performed analytically with the result that the predictive distribution is given by a Gaussian of the form

$$p(t|x, \mathbf{x}, \mathbf{t}) = \mathcal{N}\left(t|m(x), s^2(x)\right) \tag{1.69}$$

where the mean and variance are given by

$$m(x) = \beta \boldsymbol{\phi}(x)^{\mathrm{T}} \mathbf{S} \sum_{n=1}^{N} \boldsymbol{\phi}(x_n) t_n \tag{1.70}$$

$$s^2(x) = \beta^{-1} + \boldsymbol{\phi}(x)^{\mathrm{T}} \mathbf{S} \boldsymbol{\phi}(x). \tag{1.71}$$

Here the matrix $\mathbf{S}$ is given by

$$\mathbf{S}^{-1} = \alpha \mathbf{I} + \beta \sum_{n=1}^{N} \boldsymbol{\phi}(x_n) \boldsymbol{\phi}(x)^{\mathrm{T}} \tag{1.72}$$

where $\mathbf{I}$ is the unit matrix, and we have defined the vector $\boldsymbol{\phi}(x)$ with elements $\phi_i(x) = x^i$ for $i = 0, \ldots, M$.

We see that the variance, as well as the mean, of the predictive distribution in (1.69) is dependent on x. The first term in (1.71) represents the uncertainty in the predicted value of t due to the noise on the target variables and was expressed already in the maximum likelihood predictive distribution (1.64) through β_{ML}^{-1}. However, the second term arises from the uncertainty in the parameters $\mathbf{w}$ and is a consequence of the Bayesian treatment. The predictive distribution for the synthetic sinusoidal regression problem is illustrated in Figure 1.17.

Figure 1.17 The predictive distribution resulting from a Bayesian treatment of polynomial curve fitting using an $M = 9$ polynomial, with the fixed parameters $\alpha = 5 \times 10^{-3}$ and $\beta = 11.1$ (corresponding to the known noise variance), in which the red curve denotes the mean of the predictive distribution and the red region corresponds to ± 1 standard deviation around the mean.

1.3. Model Selection

In our example of polynomial curve fitting using least squares, we saw that there was an optimal order of polynomial that gave the best generalization. The order of the polynomial controls the number of free parameters in the model and thereby governs the model complexity. With regularized least squares, the regularization coefficient λ also controls the effective complexity of the model, whereas for more complex models, such as mixture distributions or neural networks there may be multiple parameters governing complexity. In a practical application, we need to determine the values of such parameters, and the principal objective in doing so is usually to achieve the best predictive performance on new data. Furthermore, as well as finding the appropriate values for complexity parameters within a given model, we may wish to consider a range of different types of model in order to find the best one for our particular application.

We have already seen that, in the maximum likelihood approach, the performance on the training set is not a good indicator of predictive performance on unseen data due to the problem of over-fitting. If data is plentiful, then one approach is simply to use some of the available data to train a range of models, or a given model with a range of values for its complexity parameters, and then to compare them on independent data, sometimes called a *validation set*, and select the one having the best predictive performance. If the model design is iterated many times using a limited size data set, then some over-fitting to the validation data can occur and so it may be necessary to keep aside a third *test set* on which the performance of the selected model is finally evaluated.

In many applications, however, the supply of data for training and testing will be limited, and in order to build good models, we wish to use as much of the available data as possible for training. However, if the validation set is small, it will give a relatively noisy estimate of predictive performance. One solution to this dilemma is to use *cross-validation*, which is illustrated in Figure 1.18. This allows a proportion $(S-1)/S$ of the available data to be used for training while making use of all of the

Figure 1.18 The technique of S-fold cross-validation, illustrated here for the case of $S = 4$, involves taking the available data and partitioning it into S groups (in the simplest case these are of equal size). Then $S - 1$ of the groups are used to train a set of models that are then evaluated on the remaining group. This procedure is then repeated for all S possible choices for the held-out group, indicated here by the red blocks, and the performance scores from the S runs are then averaged.

data to assess performance. When data is particularly scarce, it may be appropriate to consider the case $S = N$, where N is the total number of data points, which gives the *leave-one-out* technique.

One major drawback of cross-validation is that the number of training runs that must be performed is increased by a factor of S, and this can prove problematic for models in which the training is itself computationally expensive. A further problem with techniques such as cross-validation that use separate data to assess performance is that we might have multiple complexity parameters for a single model (for instance, there might be several regularization parameters). Exploring combinations of settings for such parameters could, in the worst case, require a number of training runs that is exponential in the number of parameters. Clearly, we need a better approach. Ideally, this should rely only on the training data and should allow multiple hyperparameters and model types to be compared in a single training run. We therefore need to find a measure of performance which depends only on the training data and which does not suffer from bias due to over-fitting.

Historically various 'information criteria' have been proposed that attempt to correct for the bias of maximum likelihood by the addition of a penalty term to compensate for the over-fitting of more complex models. For example, the *Akaike information criterion*, or AIC (Akaike, 1974), chooses the model for which the quantity

$$\ln p(\mathcal{D}|\mathbf{w}_{\mathrm{ML}}) - M \tag{1.73}$$

is largest. Here $p(\mathcal{D}|\mathbf{w}_{\mathrm{ML}})$ is the best-fit log likelihood, and M is the number of adjustable parameters in the model. A variant of this quantity, called the *Bayesian information criterion*, or *BIC*, will be discussed in Section 4.4.1. Such criteria do not take account of the uncertainty in the model parameters, however, and in practice they tend to favour overly simple models. We therefore turn in Section 3.4 to a fully Bayesian approach where we shall see how complexity penalties arise in a natural and principled way.

1.4. The Curse of Dimensionality

In the polynomial curve fitting example we had just one input variable x. For practical applications of pattern recognition, however, we will have to deal with spaces

Figure 1.19 Scatter plot of the oil flow data for input variables x_6 and x_7, in which red denotes the 'homogenous' class, green denotes the 'annular' class, and blue denotes the 'laminar' class. Our goal is to classify the new test point denoted by '$\times$'.

of high dimensionality comprising many input variables. As we now discuss, this poses some serious challenges and is an important factor influencing the design of pattern recognition techniques.

In order to illustrate the problem we consider a synthetically generated data set representing measurements taken from a pipeline containing a mixture of oil, water, and gas (Bishop and James, 1993). These three materials can be present in one of three different geometrical configurations known as 'homogenous', 'annular', and 'laminar', and the fractions of the three materials can also vary. Each data point comprises a 12-dimensional input vector consisting of measurements taken with gamma ray densitometers that measure the attenuation of gamma rays passing along narrow beams through the pipe. This data set is described in detail in Appendix A. Figure 1.19 shows 100 points from this data set on a plot showing two of the measurements x_6 and x_7 (the remaining ten input values are ignored for the purposes of this illustration). Each data point is labelled according to which of the three geometrical classes it belongs to, and our goal is to use this data as a training set in order to be able to classify a new observation (x_6, x_7), such as the one denoted by the cross in Figure 1.19. We observe that the cross is surrounded by numerous red points, and so we might suppose that it belongs to the red class. However, there are also plenty of green points nearby, so we might think that it could instead belong to the green class. It seems unlikely that it belongs to the blue class. The intuition here is that the identity of the cross should be determined more strongly by nearby points from the training set and less strongly by more distant points. In fact, this intuition turns out to be reasonable and will be discussed more fully in later chapters.

How can we turn this intuition into a learning algorithm? One very simple approach would be to divide the input space into regular cells, as indicated in Figure 1.20. When we are given a test point and we wish to predict its class, we first decide which cell it belongs to, and we then find all of the training data points that

Figure 1.20 Illustration of a simple approach to the solution of a classification problem in which the input space is divided into cells and any new test point is assigned to the class that has a majority number of representatives in the same cell as the test point. As we shall see shortly, this simplistic approach has some severe shortcomings.

fall in the same cell. The identity of the test point is predicted as being the same as the class having the largest number of training points in the same cell as the test point (with ties being broken at random).

There are numerous problems with this naive approach, but one of the most severe becomes apparent when we consider its extension to problems having larger numbers of input variables, corresponding to input spaces of higher dimensionality. The origin of the problem is illustrated in Figure 1.21, which shows that, if we divide a region of a space into regular cells, then the number of such cells grows exponentially with the dimensionality of the space. The problem with an exponentially large number of cells is that we would need an exponentially large quantity of training data in order to ensure that the cells are not empty. Clearly, we have no hope of applying such a technique in a space of more than a few variables, and so we need to find a more sophisticated approach.

We can gain further insight into the problems of high-dimensional spaces by returning to the example of polynomial curve fitting and considering how we would

Section 1.1

Figure 1.21 Illustration of the curse of dimensionality, showing how the number of regions of a regular grid grows exponentially with the dimensionality D of the space. For clarity, only a subset of the cubical regions are shown for $D = 3$.

extend this approach to deal with input spaces having several variables. If we have D input variables, then a general polynomial with coefficients up to order 3 would take the form

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{i=1}^{D} w_i x_i + \sum_{i=1}^{D}\sum_{j=1}^{D} w_{ij} x_i x_j + \sum_{i=1}^{D}\sum_{j=1}^{D}\sum_{k=1}^{D} w_{ijk} x_i x_j x_k. \quad (1.74)$$

As D increases, so the number of independent coefficients (not all of the coefficients are independent due to interchange symmetries amongst the x variables) grows proportionally to D^3. In practice, to capture complex dependencies in the data, we may need to use a higher-order polynomial. For a polynomial of order M, the growth in the number of coefficients is like D^M. Although this is now a power law growth, rather than an exponential growth, it still points to the method becoming rapidly unwieldy and of limited practical utility.

Exercise 1.16

Our geometrical intuitions, formed through a life spent in a space of three dimensions, can fail badly when we consider spaces of higher dimensionality. As a simple example, consider a sphere of radius $r = 1$ in a space of D dimensions, and ask what is the fraction of the volume of the sphere that lies between radius $r = 1-\epsilon$ and $r = 1$. We can evaluate this fraction by noting that the volume of a sphere of radius r in D dimensions must scale as r^D, and so we write

$$V_D(r) = K_D r^D \quad (1.75)$$

Exercise 1.18

where the constant K_D depends only on D. Thus the required fraction is given by

$$\frac{V_D(1) - V_D(1-\epsilon)}{V_D(1)} = 1 - (1-\epsilon)^D \quad (1.76)$$

which is plotted as a function of ϵ for various values of D in Figure 1.22. We see that, for large D, this fraction tends to 1 even for small values of ϵ. Thus, in spaces of high dimensionality, most of the volume of a sphere is concentrated in a thin shell near the surface!

As a further example, of direct relevance to pattern recognition, consider the behaviour of a Gaussian distribution in a high-dimensional space. If we transform from Cartesian to polar coordinates, and then integrate out the directional variables, we obtain an expression for the density $p(r)$ as a function of radius r from the origin. Thus $p(r)\delta r$ is the probability mass inside a thin shell of thickness δr located at radius r. This distribution is plotted, for various values of D, in Figure 1.23, and we see that for large D the probability mass of the Gaussian is concentrated in a thin shell.

Exercise 1.20

The severe difficulty that can arise in spaces of many dimensions is sometimes called the *curse of dimensionality* (Bellman, 1961). In this book, we shall make extensive use of illustrative examples involving input spaces of one or two dimensions, because this makes it particularly easy to illustrate the techniques graphically. The reader should be warned, however, that not all intuitions developed in spaces of low dimensionality will generalize to spaces of many dimensions.

Figure 1.22 Plot of the fraction of the volume of a sphere lying in the range $r = 1-\epsilon$ to $r = 1$ for various values of the dimensionality D.

Although the curse of dimensionality certainly raises important issues for pattern recognition applications, it does not prevent us from finding effective techniques applicable to high-dimensional spaces. The reasons for this are twofold. First, real data will often be confined to a region of the space having lower effective dimensionality, and in particular the directions over which important variations in the target variables occur may be so confined. Second, real data will typically exhibit some smoothness properties (at least locally) so that for the most part small changes in the input variables will produce small changes in the target variables, and so we can exploit local interpolation-like techniques to allow us to make predictions of the target variables for new values of the input variables. Successful pattern recognition techniques exploit one or both of these properties. Consider, for example, an application in manufacturing in which images are captured of identical planar objects on a conveyor belt, in which the goal is to determine their orientation. Each image is a point

Figure 1.23 Plot of the probability density with respect to radius r of a Gaussian distribution for various values of the dimensionality D. In a high-dimensional space, most of the probability mass of a Gaussian is located within a thin shell at a specific radius.

in a high-dimensional space whose dimensionality is determined by the number of pixels. Because the objects can occur at different positions within the image and in different orientations, there are three degrees of freedom of variability between images, and a set of images will live on a three dimensional *manifold* embedded within the high-dimensional space. Due to the complex relationships between the object position or orientation and the pixel intensities, this manifold will be highly nonlinear. If the goal is to learn a model that can take an input image and output the orientation of the object irrespective of its position, then there is only one degree of freedom of variability within the manifold that is significant.

1.5. Decision Theory

We have seen in Section 1.2 how probability theory provides us with a consistent mathematical framework for quantifying and manipulating uncertainty. Here we turn to a discussion of decision theory that, when combined with probability theory, allows us to make optimal decisions in situations involving uncertainty such as those encountered in pattern recognition.

Suppose we have an input vector $\mathbf{x}$ together with a corresponding vector $\mathbf{t}$ of target variables, and our goal is to predict $\mathbf{t}$ given a new value for $\mathbf{x}$. For regression problems, $\mathbf{t}$ will comprise continuous variables, whereas for classification problems $\mathbf{t}$ will represent class labels. The joint probability distribution $p(\mathbf{x}, \mathbf{t})$ provides a complete summary of the uncertainty associated with these variables. Determination of $p(\mathbf{x}, \mathbf{t})$ from a set of training data is an example of *inference* and is typically a very difficult problem whose solution forms the subject of much of this book. In a practical application, however, we must often make a specific prediction for the value of $\mathbf{t}$, or more generally take a specific action based on our understanding of the values $\mathbf{t}$ is likely to take, and this aspect is the subject of decision theory.

Consider, for example, a medical diagnosis problem in which we have taken an X-ray image of a patient, and we wish to determine whether the patient has cancer or not. In this case, the input vector $\mathbf{x}$ is the set of pixel intensities in the image, and output variable t will represent the presence of cancer, which we denote by the class $\mathcal{C}_1$, or the absence of cancer, which we denote by the class $\mathcal{C}_2$. We might, for instance, choose t to be a binary variable such that $t = 0$ corresponds to class $\mathcal{C}_1$ and $t = 1$ corresponds to class $\mathcal{C}_2$. We shall see later that this choice of label values is particularly convenient for probabilistic models. The general inference problem then involves determining the joint distribution $p(\mathbf{x}, \mathcal{C}_k)$, or equivalently $p(\mathbf{x}, t)$, which gives us the most complete probabilistic description of the situation. Although this can be a very useful and informative quantity, in the end we must decide either to give treatment to the patient or not, and we would like this choice to be optimal in some appropriate sense (Duda and Hart, 1973). This is the *decision* step, and it is the subject of decision theory to tell us how to make optimal decisions given the appropriate probabilities. We shall see that the decision stage is generally very simple, even trivial, once we have solved the inference problem.

Here we give an introduction to the key ideas of decision theory as required for

the rest of the book. Further background, as well as more detailed accounts, can be found in Berger (1985) and Bather (2000).

Before giving a more detailed analysis, let us first consider informally how we might expect probabilities to play a role in making decisions. When we obtain the X-ray image $\mathbf{x}$ for a new patient, our goal is to decide which of the two classes to assign to the image. We are interested in the probabilities of the two classes given the image, which are given by $p(\mathcal{C}_k|\mathbf{x})$. Using Bayes' theorem, these probabilities can be expressed in the form

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})}. \tag{1.77}$$

Note that any of the quantities appearing in Bayes' theorem can be obtained from the joint distribution $p(\mathbf{x}, \mathcal{C}_k)$ by either marginalizing or conditioning with respect to the appropriate variables. We can now interpret $p(\mathcal{C}_k)$ as the prior probability for the class $\mathcal{C}_k$, and $p(\mathcal{C}_k|\mathbf{x})$ as the corresponding posterior probability. Thus $p(\mathcal{C}_1)$ represents the probability that a person has cancer, before we take the X-ray measurement. Similarly, $p(\mathcal{C}_1|\mathbf{x})$ is the corresponding probability, revised using Bayes' theorem in light of the information contained in the X-ray. If our aim is to minimize the chance of assigning $\mathbf{x}$ to the wrong class, then intuitively we would choose the class having the higher posterior probability. We now show that this intuition is correct, and we also discuss more general criteria for making decisions.

1.5.1 Minimizing the misclassification rate

Suppose that our goal is simply to make as few misclassifications as possible. We need a rule that assigns each value of $\mathbf{x}$ to one of the available classes. Such a rule will divide the input space into regions $\mathcal{R}_k$ called *decision regions*, one for each class, such that all points in $\mathcal{R}_k$ are assigned to class $\mathcal{C}_k$. The boundaries between decision regions are called *decision boundaries* or *decision surfaces*. Note that each decision region need not be contiguous but could comprise some number of disjoint regions. We shall encounter examples of decision boundaries and decision regions in later chapters. In order to find the optimal decision rule, consider first of all the case of two classes, as in the cancer problem for instance. A mistake occurs when an input vector belonging to class $\mathcal{C}_1$ is assigned to class $\mathcal{C}_2$ or vice versa. The probability of this occurring is given by

$$\begin{aligned} p(\text{mistake}) &= p(\mathbf{x} \in \mathcal{R}_1, \mathcal{C}_2) + p(\mathbf{x} \in \mathcal{R}_2, \mathcal{C}_1) \\ &= \int_{\mathcal{R}_1} p(\mathbf{x}, \mathcal{C}_2)\,\mathrm{d}\mathbf{x} + \int_{\mathcal{R}_2} p(\mathbf{x}, \mathcal{C}_1)\,\mathrm{d}\mathbf{x}. \end{aligned} \tag{1.78}$$

We are free to choose the decision rule that assigns each point $\mathbf{x}$ to one of the two classes. Clearly to minimize $p(\text{mistake})$ we should arrange that each $\mathbf{x}$ is assigned to whichever class has the smaller value of the integrand in (1.78). Thus, if $p(\mathbf{x}, \mathcal{C}_1) > p(\mathbf{x}, \mathcal{C}_2)$ for a given value of $\mathbf{x}$, then we should assign that $\mathbf{x}$ to class $\mathcal{C}_1$. From the product rule of probability we have $p(\mathbf{x}, \mathcal{C}_k) = p(\mathcal{C}_k|\mathbf{x})p(\mathbf{x})$. Because the factor $p(\mathbf{x})$ is common to both terms, we can restate this result as saying that the minimum

Figure 1.24 Schematic illustration of the joint probabilities $p(x, \mathcal{C}_k)$ for each of two classes plotted against x, together with the decision boundary $x = \widehat{x}$. Values of $x \geqslant \widehat{x}$ are classified as class $\mathcal{C}_2$ and hence belong to decision region $\mathcal{R}_2$, whereas points $x < \widehat{x}$ are classified as $\mathcal{C}_1$ and belong to $\mathcal{R}_1$. Errors arise from the blue, green, and red regions, so that for $x < \widehat{x}$ the errors are due to points from class $\mathcal{C}_2$ being misclassified as $\mathcal{C}_1$ (represented by the sum of the red and green regions), and conversely for points in the region $x \geqslant \widehat{x}$ the errors are due to points from class $\mathcal{C}_1$ being misclassified as $\mathcal{C}_2$ (represented by the blue region). As we vary the location $\widehat{x}$ of the decision boundary, the combined areas of the blue and green regions remains constant, whereas the size of the red region varies. The optimal choice for $\widehat{x}$ is where the curves for $p(x, \mathcal{C}_1)$ and $p(x, \mathcal{C}_2)$ cross, corresponding to $\widehat{x} = x_0$, because in this case the red region disappears. This is equivalent to the minimum misclassification rate decision rule, which assigns each value of x to the class having the higher posterior probability $p(\mathcal{C}_k|x)$.

probability of making a mistake is obtained if each value of $\mathbf{x}$ is assigned to the class for which the posterior probability $p(\mathcal{C}_k|\mathbf{x})$ is largest. This result is illustrated for two classes, and a single input variable x, in Figure 1.24.

For the more general case of K classes, it is slightly easier to maximize the probability of being correct, which is given by

$$\begin{aligned} p(\text{correct}) &= \sum_{k=1}^{K} p(\mathbf{x} \in \mathcal{R}_k, \mathcal{C}_k) \\ &= \sum_{k=1}^{K} \int_{\mathcal{R}_k} p(\mathbf{x}, \mathcal{C}_k)\, \mathrm{d}\mathbf{x} \end{aligned} \tag{1.79}$$

which is maximized when the regions $\mathcal{R}_k$ are chosen such that each $\mathbf{x}$ is assigned to the class for which $p(\mathbf{x}, \mathcal{C}_k)$ is largest. Again, using the product rule $p(\mathbf{x}, \mathcal{C}_k) = p(\mathcal{C}_k|\mathbf{x})p(\mathbf{x})$, and noting that the factor of $p(\mathbf{x})$ is common to all terms, we see that each $\mathbf{x}$ should be assigned to the class having the largest posterior probability $p(\mathcal{C}_k|\mathbf{x})$.

Figure 1.25 An example of a loss matrix with elements L_{kj} for the cancer treatment problem. The rows correspond to the true class, whereas the columns correspond to the assignment of class made by our decision criterion.

$$\begin{array}{cc} & \begin{array}{cc} \text{cancer} & \text{normal} \end{array} \\ \begin{array}{c} \text{cancer} \\ \text{normal} \end{array} & \begin{pmatrix} 0 & 1000 \\ 1 & 0 \end{pmatrix} \end{array}$$

1.5.2 Minimizing the expected loss

For many applications, our objective will be more complex than simply minimizing the number of misclassifications. Let us consider again the medical diagnosis problem. We note that, if a patient who does not have cancer is incorrectly diagnosed as having cancer, the consequences may be some patient distress plus the need for further investigations. Conversely, if a patient with cancer is diagnosed as healthy, the result may be premature death due to lack of treatment. Thus the consequences of these two types of mistake can be dramatically different. It would clearly be better to make fewer mistakes of the second kind, even if this was at the expense of making more mistakes of the first kind.

We can formalize such issues through the introduction of a *loss function*, also called a *cost function*, which is a single, overall measure of loss incurred in taking any of the available decisions or actions. Our goal is then to minimize the total loss incurred. Note that some authors consider instead a *utility function*, whose value they aim to maximize. These are equivalent concepts if we take the utility to be simply the negative of the loss, and throughout this text we shall use the loss function convention. Suppose that, for a new value of $\mathbf{x}$, the true class is $\mathcal{C}_k$ and that we assign $\mathbf{x}$ to class $\mathcal{C}_j$ (where j may or may not be equal to k). In so doing, we incur some level of loss that we denote by L_{kj}, which we can view as the k, j element of a *loss matrix*. For instance, in our cancer example, we might have a loss matrix of the form shown in Figure 1.25. This particular loss matrix says that there is no loss incurred if the correct decision is made, there is a loss of 1 if a healthy patient is diagnosed as having cancer, whereas there is a loss of 1000 if a patient having cancer is diagnosed as healthy.

The optimal solution is the one which minimizes the loss function. However, the loss function depends on the true class, which is unknown. For a given input vector $\mathbf{x}$, our uncertainty in the true class is expressed through the joint probability distribution $p(\mathbf{x}, \mathcal{C}_k)$ and so we seek instead to minimize the average loss, where the average is computed with respect to this distribution, which is given by

$$\mathbb{E}[L] = \sum_k \sum_j \int_{\mathcal{R}_j} L_{kj} p(\mathbf{x}, \mathcal{C}_k) \, \mathrm{d}\mathbf{x}. \tag{1.80}$$

Each $\mathbf{x}$ can be assigned independently to one of the decision regions $\mathcal{R}_j$. Our goal is to choose the regions $\mathcal{R}_j$ in order to minimize the expected loss (1.80), which implies that for each $\mathbf{x}$ we should minimize $\sum_k L_{kj} p(\mathbf{x}, \mathcal{C}_k)$. As before, we can use the product rule $p(\mathbf{x}, \mathcal{C}_k) = p(\mathcal{C}_k|\mathbf{x})p(\mathbf{x})$ to eliminate the common factor of $p(\mathbf{x})$. Thus the decision rule that minimizes the expected loss is the one that assigns each

Figure 1.26 Illustration of the reject option. Inputs x such that the larger of the two posterior probabilities is less than or equal to some threshold θ will be rejected.

new $\mathbf{x}$ to the class j for which the quantity

$$\sum_k L_{kj} p(\mathcal{C}_k|\mathbf{x}) \tag{1.81}$$

is a minimum. This is clearly trivial to do, once we know the posterior class probabilities $p(\mathcal{C}_k|\mathbf{x})$.

1.5.3 The reject option

We have seen that classification errors arise from the regions of input space where the largest of the posterior probabilities $p(\mathcal{C}_k|\mathbf{x})$ is significantly less than unity, or equivalently where the joint distributions $p(\mathbf{x}, \mathcal{C}_k)$ have comparable values. These are the regions where we are relatively uncertain about class membership. In some applications, it will be appropriate to avoid making decisions on the difficult cases in anticipation of a lower error rate on those examples for which a classification decision is made. This is known as the *reject option*. For example, in our hypothetical medical illustration, it may be appropriate to use an automatic system to classify those X-ray images for which there is little doubt as to the correct class, while leaving a human expert to classify the more ambiguous cases. We can achieve this by introducing a threshold θ and rejecting those inputs $\mathbf{x}$ for which the largest of the posterior probabilities $p(\mathcal{C}_k|\mathbf{x})$ is less than or equal to θ. This is illustrated for the case of two classes, and a single continuous input variable x, in Figure 1.26. Note that setting $\theta = 1$ will ensure that all examples are rejected, whereas if there are K classes then setting $\theta < 1/K$ will ensure that no examples are rejected. Thus the fraction of examples that get rejected is controlled by the value of θ.

We can easily extend the reject criterion to minimize the expected loss, when a loss matrix is given, taking account of the loss incurred when a reject decision is made.

Exercise 1.24

1.5.4 Inference and decision

We have broken the classification problem down into two separate stages, the *inference stage* in which we use training data to learn a model for $p(\mathcal{C}_k|\mathbf{x})$, and the

subsequent *decision* stage in which we use these posterior probabilities to make optimal class assignments. An alternative possibility would be to solve both problems together and simply learn a function that maps inputs $\mathbf{x}$ directly into decisions. Such a function is called a *discriminant function*.

In fact, we can identify three distinct approaches to solving decision problems, all of which have been used in practical applications. These are given, in decreasing order of complexity, by:

(a) First solve the inference problem of determining the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$ for each class $\mathcal{C}_k$ individually. Also separately infer the prior class probabilities $p(\mathcal{C}_k)$. Then use Bayes' theorem in the form

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})} \tag{1.82}$$

to find the posterior class probabilities $p(\mathcal{C}_k|\mathbf{x})$. As usual, the denominator in Bayes' theorem can be found in terms of the quantities appearing in the numerator, because

$$p(\mathbf{x}) = \sum_k p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k). \tag{1.83}$$

Equivalently, we can model the joint distribution $p(\mathbf{x}, \mathcal{C}_k)$ directly and then normalize to obtain the posterior probabilities. Having found the posterior probabilities, we use decision theory to determine class membership for each new input $\mathbf{x}$. Approaches that explicitly or implicitly model the distribution of inputs as well as outputs are known as *generative models*, because by sampling from them it is possible to generate synthetic data points in the input space.

(b) First solve the inference problem of determining the posterior class probabilities $p(\mathcal{C}_k|\mathbf{x})$, and then subsequently use decision theory to assign each new $\mathbf{x}$ to one of the classes. Approaches that model the posterior probabilities directly are called *discriminative models*.

(c) Find a function $f(\mathbf{x})$, called a discriminant function, which maps each input $\mathbf{x}$ directly onto a class label. For instance, in the case of two-class problems, $f(\cdot)$ might be binary valued and such that $f = 0$ represents class $\mathcal{C}_1$ and $f = 1$ represents class $\mathcal{C}_2$. In this case, probabilities play no role.

Let us consider the relative merits of these three alternatives. Approach (a) is the most demanding because it involves finding the joint distribution over both $\mathbf{x}$ and $\mathcal{C}_k$. For many applications, $\mathbf{x}$ will have high dimensionality, and consequently we may need a large training set in order to be able to determine the class-conditional densities to reasonable accuracy. Note that the class priors $p(\mathcal{C}_k)$ can often be estimated simply from the fractions of the training set data points in each of the classes. One advantage of approach (a), however, is that it also allows the marginal density of data $p(\mathbf{x})$ to be determined from (1.83). This can be useful for detecting new data points that have low probability under the model and for which the predictions may

Figure 1.27 Example of the class-conditional densities for two classes having a single input variable x (left plot) together with the corresponding posterior probabilities (right plot). Note that the left-hand mode of the class-conditional density $p(\mathbf{x}|\mathcal{C}_1)$, shown in blue on the left plot, has no effect on the posterior probabilities. The vertical green line in the right plot shows the decision boundary in x that gives the minimum misclassification rate.

be of low accuracy, which is known as *outlier detection* or *novelty detection* (Bishop, 1994; Tarassenko, 1995).

However, if we only wish to make classification decisions, then it can be wasteful of computational resources, and excessively demanding of data, to find the joint distribution $p(\mathbf{x}, \mathcal{C}_k)$ when in fact we only really need the posterior probabilities $p(\mathcal{C}_k|\mathbf{x})$, which can be obtained directly through approach (b). Indeed, the class-conditional densities may contain a lot of structure that has little effect on the posterior probabilities, as illustrated in Figure 1.27. There has been much interest in exploring the relative merits of generative and discriminative approaches to machine learning, and in finding ways to combine them (Jebara, 2004; Lasserre *et al.*, 2006).

An even simpler approach is (c) in which we use the training data to find a discriminant function $f(\mathbf{x})$ that maps each $\mathbf{x}$ directly onto a class label, thereby combining the inference and decision stages into a single learning problem. In the example of Figure 1.27, this would correspond to finding the value of x shown by the vertical green line, because this is the decision boundary giving the minimum probability of misclassification.

With option (c), however, we no longer have access to the posterior probabilities $p(\mathcal{C}_k|\mathbf{x})$. There are many powerful reasons for wanting to compute the posterior probabilities, even if we subsequently use them to make decisions. These include:

Minimizing risk. Consider a problem in which the elements of the loss matrix are subjected to revision from time to time (such as might occur in a financial

application). If we know the posterior probabilities, we can trivially revise the minimum risk decision criterion by modifying (1.81) appropriately. If we have only a discriminant function, then any change to the loss matrix would require that we return to the training data and solve the classification problem afresh.

Reject option. Posterior probabilities allow us to determine a rejection criterion that will minimize the misclassification rate, or more generally the expected loss, for a given fraction of rejected data points.

Compensating for class priors. Consider our medical X-ray problem again, and suppose that we have collected a large number of X-ray images from the general population for use as training data in order to build an automated screening system. Because cancer is rare amongst the general population, we might find that, say, only 1 in every 1,000 examples corresponds to the presence of cancer. If we used such a data set to train an adaptive model, we could run into severe difficulties due to the small proportion of the cancer class. For instance, a classifier that assigned every point to the normal class would already achieve 99.9% accuracy and it would be difficult to avoid this trivial solution. Also, even a large data set will contain very few examples of X-ray images corresponding to cancer, and so the learning algorithm will not be exposed to a broad range of examples of such images and hence is not likely to generalize well. A balanced data set in which we have selected equal numbers of examples from each of the classes would allow us to find a more accurate model. However, we then have to compensate for the effects of our modifications to the training data. Suppose we have used such a modified data set and found models for the posterior probabilities. From Bayes' theorem (1.82), we see that the posterior probabilities are proportional to the prior probabilities, which we can interpret as the fractions of points in each class. We can therefore simply take the posterior probabilities obtained from our artificially balanced data set and first divide by the class fractions in that data set and then multiply by the class fractions in the population to which we wish to apply the model. Finally, we need to normalize to ensure that the new posterior probabilities sum to one. Note that this procedure cannot be applied if we have learned a discriminant function directly instead of determining posterior probabilities.

Combining models. For complex applications, we may wish to break the problem into a number of smaller subproblems each of which can be tackled by a separate module. For example, in our hypothetical medical diagnosis problem, we may have information available from, say, blood tests as well as X-ray images. Rather than combine all of this heterogeneous information into one huge input space, it may be more effective to build one system to interpret the X-ray images and a different one to interpret the blood data. As long as each of the two models gives posterior probabilities for the classes, we can combine the outputs systematically using the rules of probability. One simple way to do this is to assume that, for each class separately, the distributions of inputs for the X-ray images, denoted by $\mathbf{x}_{\mathrm{I}}$, and the blood data, denoted by $\mathbf{x}_{\mathrm{B}}$, are

independent, so that

$$p(\mathbf{x}_{\mathrm{I}}, \mathbf{x}_{\mathrm{B}}|\mathcal{C}_k) = p(\mathbf{x}_{\mathrm{I}}|\mathcal{C}_k)p(\mathbf{x}_{\mathrm{B}}|\mathcal{C}_k). \tag{1.84}$$

Section 8.2

This is an example of *conditional independence* property, because the independence holds when the distribution is conditioned on the class $\mathcal{C}_k$. The posterior probability, given both the X-ray and blood data, is then given by

$$\begin{aligned} p(\mathcal{C}_k|\mathbf{x}_{\mathrm{I}}, \mathbf{x}_{\mathrm{B}}) &\propto p(\mathbf{x}_{\mathrm{I}}, \mathbf{x}_{\mathrm{B}}|\mathcal{C}_k)p(\mathcal{C}_k) \\ &\propto p(\mathbf{x}_{\mathrm{I}}|\mathcal{C}_k)p(\mathbf{x}_{\mathrm{B}}|\mathcal{C}_k)p(\mathcal{C}_k) \\ &\propto \frac{p(\mathcal{C}_k|\mathbf{x}_{\mathrm{I}})p(\mathcal{C}_k|\mathbf{x}_{\mathrm{B}})}{p(\mathcal{C}_k)} \end{aligned} \tag{1.85}$$

Thus we need the class prior probabilities $p(\mathcal{C}_k)$, which we can easily estimate from the fractions of data points in each class, and then we need to normalize the resulting posterior probabilities so they sum to one. The particular conditional independence assumption (1.84) is an example of the *naive Bayes model*. Note that the joint marginal distribution $p(\mathbf{x}_{\mathrm{I}}, \mathbf{x}_{\mathrm{B}})$ will typically not factorize under this model. We shall see in later chapters how to construct models for combining data that do not require the conditional independence assumption (1.84).

Section 8.2.2

1.5.5 Loss functions for regression

So far, we have discussed decision theory in the context of classification problems. We now turn to the case of regression problems, such as the curve fitting example discussed earlier. The decision stage consists of choosing a specific estimate $y(\mathbf{x})$ of the value of t for each input $\mathbf{x}$. Suppose that in doing so, we incur a loss $L(t, y(\mathbf{x}))$. The average, or expected, loss is then given by

Section 1.1

$$\mathbb{E}[L] = \iint L(t, y(\mathbf{x}))p(\mathbf{x}, t)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t. \tag{1.86}$$

A common choice of loss function in regression problems is the squared loss given by $L(t, y(\mathbf{x})) = \{y(\mathbf{x}) - t\}^2$. In this case, the expected loss can be written

$$\mathbb{E}[L] = \iint \{y(\mathbf{x}) - t\}^2 p(\mathbf{x}, t)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t. \tag{1.87}$$

Our goal is to choose $y(\mathbf{x})$ so as to minimize $\mathbb{E}[L]$. If we assume a completely flexible function $y(\mathbf{x})$, we can do this formally using the calculus of variations to give

Appendix D

$$\frac{\delta\mathbb{E}[L]}{\delta y(\mathbf{x})} = 2\int \{y(\mathbf{x}) - t\}p(\mathbf{x}, t)\,\mathrm{d}t = 0. \tag{1.88}$$

Solving for $y(\mathbf{x})$, and using the sum and product rules of probability, we obtain

$$y(\mathbf{x}) = \frac{\int tp(\mathbf{x}, t)\,\mathrm{d}t}{p(\mathbf{x})} = \int tp(t|\mathbf{x})\,\mathrm{d}t = \mathbb{E}_t[t|\mathbf{x}] \tag{1.89}$$

Figure 1.28 The regression function $y(x)$, which minimizes the expected squared loss, is given by the mean of the conditional distribution $p(t|x)$.

which is the conditional average of t conditioned on $\mathbf{x}$ and is known as the *regression function*. This result is illustrated in Figure 1.28. It can readily be extended to multiple target variables represented by the vector $\mathbf{t}$, in which case the optimal solution is the conditional average $\mathbf{y}(\mathbf{x}) = \mathbb{E}_t[\mathbf{t}|\mathbf{x}]$.

Exercise 1.25

We can also derive this result in a slightly different way, which will also shed light on the nature of the regression problem. Armed with the knowledge that the optimal solution is the conditional expectation, we can expand the square term as follows

$$\begin{aligned}\{y(\mathbf{x}) - t\}^2 &= \{y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}] + \mathbb{E}[t|\mathbf{x}] - t\}^2 \\ &= \{y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}]\}^2 + 2\{y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}]\}\{\mathbb{E}[t|\mathbf{x}] - t\} + \{\mathbb{E}[t|\mathbf{x}] - t\}^2\end{aligned}$$

where, to keep the notation uncluttered, we use $\mathbb{E}[t|\mathbf{x}]$ to denote $\mathbb{E}_t[t|\mathbf{x}]$. Substituting into the loss function and performing the integral over t, we see that the cross-term vanishes and we obtain an expression for the loss function in the form

$$\mathbb{E}[L] = \int \{y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}]\}^2 p(\mathbf{x})\,\mathrm{d}\mathbf{x} + \int \{\mathbb{E}[t|\mathbf{x}] - t\}^2 p(\mathbf{x})\,\mathrm{d}\mathbf{x}. \tag{1.90}$$

The function $y(\mathbf{x})$ we seek to determine enters only in the first term, which will be minimized when $y(\mathbf{x})$ is equal to $\mathbb{E}[t|\mathbf{x}]$, in which case this term will vanish. This is simply the result that we derived previously and that shows that the optimal least squares predictor is given by the conditional mean. The second term is the variance of the distribution of t, averaged over $\mathbf{x}$. It represents the intrinsic variability of the target data and can be regarded as noise. Because it is independent of $y(\mathbf{x})$, it represents the irreducible minimum value of the loss function.

As with the classification problem, we can either determine the appropriate probabilities and then use these to make optimal decisions, or we can build models that make decisions directly. Indeed, we can identify three distinct approaches to solving regression problems given, in order of decreasing complexity, by:

(a) First solve the inference problem of determining the joint density $p(\mathbf{x}, t)$. Then normalize to find the conditional density $p(t|\mathbf{x})$, and finally marginalize to find the conditional mean given by (1.89).

(b) First solve the inference problem of determining the conditional density $p(t|\mathbf{x})$, and then subsequently marginalize to find the conditional mean given by (1.89).

(c) Find a regression function $y(\mathbf{x})$ directly from the training data.

The relative merits of these three approaches follow the same lines as for classification problems above.

The squared loss is not the only possible choice of loss function for regression. Indeed, there are situations in which squared loss can lead to very poor results and where we need to develop more sophisticated approaches. An important example concerns situations in which the conditional distribution $p(t|\mathbf{x})$ is multimodal, as often arises in the solution of inverse problems. Here we consider briefly one simple generalization of the squared loss, called the *Minkowski* loss, whose expectation is given by

Section 5.6

$$\mathbb{E}[L_q] = \iint |y(\mathbf{x}) - t|^q p(\mathbf{x}, t)\, \mathrm{d}\mathbf{x}\, \mathrm{d}t \tag{1.91}$$

which reduces to the expected squared loss for $q = 2$. The function $|y - t|^q$ is plotted against $y - t$ for various values of q in Figure 1.29. The minimum of $\mathbb{E}[L_q]$ is given by the conditional mean for $q = 2$, the conditional median for $q = 1$, and the conditional mode for $q \to 0$.

Exercise 1.27

1.6. Information Theory

In this chapter, we have discussed a variety of concepts from probability theory and decision theory that will form the foundations for much of the subsequent discussion in this book. We close this chapter by introducing some additional concepts from the field of information theory, which will also prove useful in our development of pattern recognition and machine learning techniques. Again, we shall focus only on the key concepts, and we refer the reader elsewhere for more detailed discussions (Viterbi and Omura, 1979; Cover and Thomas, 1991; MacKay, 2003) .

We begin by considering a discrete random variable x and we ask how much information is received when we observe a specific value for this variable. The amount of information can be viewed as the 'degree of surprise' on learning the value of x. If we are told that a highly improbable event has just occurred, we will have received more information than if we were told that some very likely event has just occurred, and if we knew that the event was certain to happen we would receive no information. Our measure of information content will therefore depend on the probability distribution $p(x)$, and we therefore look for a quantity $h(x)$ that is a monotonic function of the probability $p(x)$ and that expresses the information content. The form of $h(\cdot)$ can be found by noting that if we have two events x and y that are unrelated, then the information gain from observing both of them should be the sum of the information gained from each of them separately, so that $h(x, y) = h(x) + h(y)$. Two unrelated events will be statistically independent and so $p(x, y) = p(x)p(y)$. From these two relationships, it is easily shown that $h(x)$ must be given by the logarithm of $p(x)$ and so we have

Exercise 1.28

Figure 1.29 Plots of the quantity $L_q = |y - t|^q$ for various values of q.

$$h(x) = -\log_2 p(x) \tag{1.92}$$

where the negative sign ensures that information is positive or zero. Note that low probability events x correspond to high information content. The choice of basis for the logarithm is arbitrary, and for the moment we shall adopt the convention prevalent in information theory of using logarithms to the base of 2. In this case, as we shall see shortly, the units of $h(x)$ are bits ('binary digits').

Now suppose that a sender wishes to transmit the value of a random variable to a receiver. The average amount of information that they transmit in the process is obtained by taking the expectation of (1.92) with respect to the distribution $p(x)$ and is given by

$$\mathrm{H}[x] = -\sum_x p(x) \log_2 p(x). \tag{1.93}$$

This important quantity is called the *entropy* of the random variable x. Note that $\lim_{p\to 0} p \ln p = 0$ and so we shall take $p(x) \ln p(x) = 0$ whenever we encounter a value for x such that $p(x) = 0$.

So far we have given a rather heuristic motivation for the definition of informa-

tion (1.92) and the corresponding entropy (1.93). We now show that these definitions indeed possess useful properties. Consider a random variable x having 8 possible states, each of which is equally likely. In order to communicate the value of x to a receiver, we would need to transmit a message of length 3 bits. Notice that the entropy of this variable is given by

$$\mathrm{H}[x] = -8 \times \frac{1}{8} \log_2 \frac{1}{8} = 3 \text{ bits}.$$

Now consider an example (Cover and Thomas, 1991) of a variable having 8 possible states $\{a, b, c, d, e, f, g, h\}$ for which the respective probabilities are given by $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64})$. The entropy in this case is given by

$$\mathrm{H}[x] = -\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{4} \log_2 \frac{1}{4} - \frac{1}{8} \log_2 \frac{1}{8} - \frac{1}{16} \log_2 \frac{1}{16} - \frac{4}{64} \log_2 \frac{1}{64} = 2 \text{ bits}.$$

We see that the nonuniform distribution has a smaller entropy than the uniform one, and we shall gain some insight into this shortly when we discuss the interpretation of entropy in terms of disorder. For the moment, let us consider how we would transmit the identity of the variable's state to a receiver. We could do this, as before, using a 3-bit number. However, we can take advantage of the nonuniform distribution by using shorter codes for the more probable events, at the expense of longer codes for the less probable events, in the hope of getting a shorter average code length. This can be done by representing the states $\{a, b, c, d, e, f, g, h\}$ using, for instance, the following set of code strings: 0, 10, 110, 1110, 111100, 111101, 111110, 111111. The average length of the code that has to be transmitted is then

$$\text{average code length} = \frac{1}{2} \times 1 + \frac{1}{4} \times 2 + \frac{1}{8} \times 3 + \frac{1}{16} \times 4 + 4 \times \frac{1}{64} \times 6 = 2 \text{ bits}$$

which again is the same as the entropy of the random variable. Note that shorter code strings cannot be used because it must be possible to disambiguate a concatenation of such strings into its component parts. For instance, 11001110 decodes uniquely into the state sequence c, a, d.

This relation between entropy and shortest coding length is a general one. The *noiseless coding theorem* (Shannon, 1948) states that the entropy is a lower bound on the number of bits needed to transmit the state of a random variable.

From now on, we shall switch to the use of natural logarithms in defining entropy, as this will provide a more convenient link with ideas elsewhere in this book. In this case, the entropy is measured in units of 'nats' instead of bits, which differ simply by a factor of $\ln 2$.

We have introduced the concept of entropy in terms of the average amount of information needed to specify the state of a random variable. In fact, the concept of entropy has much earlier origins in physics where it was introduced in the context of equilibrium thermodynamics and later given a deeper interpretation as a measure of disorder through developments in statistical mechanics. We can understand this alternative view of entropy by considering a set of N identical objects that are to be divided amongst a set of bins, such that there are n_i objects in the ith bin. Consider

the number of different ways of allocating the objects to the bins. There are N ways to choose the first object, $(N-1)$ ways to choose the second object, and so on, leading to a total of $N!$ ways to allocate all N objects to the bins, where $N!$ (pronounced 'factorial N') denotes the product $N \times (N-1) \times \cdots \times 2 \times 1$. However, we don't wish to distinguish between rearrangements of objects within each bin. In the ith bin there are $n_i!$ ways of reordering the objects, and so the total number of ways of allocating the N objects to the bins is given by

$$W = \frac{N!}{\prod_i n_i!} \tag{1.94}$$

which is called the *multiplicity*. The entropy is then defined as the logarithm of the multiplicity scaled by an appropriate constant

$$\mathrm{H} = \frac{1}{N} \ln W = \frac{1}{N} \ln N! - \frac{1}{N} \sum_i \ln n_i!. \tag{1.95}$$

We now consider the limit $N \to \infty$, in which the fractions n_i/N are held fixed, and apply Stirling's approximation

$$\ln N! \simeq N \ln N - N \tag{1.96}$$

which gives

$$\mathrm{H} = -\lim_{N\to\infty} \sum_i \left(\frac{n_i}{N}\right) \ln\left(\frac{n_i}{N}\right) = -\sum_i p_i \ln p_i \tag{1.97}$$

where we have used $\sum_i n_i = N$. Here $p_i = \lim_{N\to\infty}(n_i/N)$ is the probability of an object being assigned to the ith bin. In physics terminology, the specific arrangements of objects in the bins is called a *microstate*, and the overall distribution of occupation numbers, expressed through the ratios n_i/N, is called a *macrostate*. The multiplicity W is also known as the *weight* of the macrostate.

We can interpret the bins as the states x_i of a discrete random variable X, where $p(X = x_i) = p_i$. The entropy of the random variable X is then

$$\mathrm{H}[p] = -\sum_i p(x_i) \ln p(x_i). \tag{1.98}$$

Distributions $p(x_i)$ that are sharply peaked around a few values will have a relatively low entropy, whereas those that are spread more evenly across many values will have higher entropy, as illustrated in Figure 1.30. Because $0 \leqslant p_i \leqslant 1$, the entropy is nonnegative, and it will equal its minimum value of 0 when one of the $p_i = 1$ and all other $p_{j\neq i} = 0$. The maximum entropy configuration can be found by maximizing H using a Lagrange multiplier to enforce the normalization constraint on the probabilities. Thus we maximize

Appendix E

$$\widetilde{\mathrm{H}} = -\sum_i p(x_i) \ln p(x_i) + \lambda \left(\sum_i p(x_i) - 1\right) \tag{1.99}$$

Figure 1.30 Histograms of two probability distributions over 30 bins illustrating the higher value of the entropy H for the broader distribution. The largest entropy would arise from a uniform distribution that would give $\mathrm{H} = -\ln(1/30) = 3.40$.

from which we find that all of the $p(x_i)$ are equal and are given by $p(x_i) = 1/M$ where M is the total number of states x_i. The corresponding value of the entropy is then $\mathrm{H} = \ln M$. This result can also be derived from Jensen's inequality (to be discussed shortly). To verify that the stationary point is indeed a maximum, we can evaluate the second derivative of the entropy, which gives

Exercise 1.29

$$\frac{\partial \widetilde{\mathrm{H}}}{\partial p(x_i)\partial p(x_j)} = -I_{ij}\frac{1}{p_i} \tag{1.100}$$

where I_{ij} are the elements of the identity matrix.

We can extend the definition of entropy to include distributions $p(x)$ over continuous variables x as follows. First divide x into bins of width Δ. Then, assuming $p(x)$ is continuous, the *mean value theorem* (Weisstein, 1999) tells us that, for each such bin, there must exist a value x_i such that

$$\int_{i\Delta}^{(i+1)\Delta} p(x)\,\mathrm{d}x = p(x_i)\Delta. \tag{1.101}$$

We can now quantize the continuous variable x by assigning any value x to the value x_i whenever x falls in the ith bin. The probability of observing the value x_i is then $p(x_i)\Delta$. This gives a discrete distribution for which the entropy takes the form

$$\mathrm{H}_\Delta = -\sum_i p(x_i)\Delta \ln\left(p(x_i)\Delta\right) = -\sum_i p(x_i)\Delta \ln p(x_i) - \ln\Delta \tag{1.102}$$

where we have used $\sum_i p(x_i)\Delta = 1$, which follows from (1.101). We now omit the second term $-\ln\Delta$ on the right-hand side of (1.102) and then consider the limit

$\Delta \to 0$. The first term on the right-hand side of (1.102) will approach the integral of $p(x) \ln p(x)$ in this limit so that

$$\lim_{\Delta \to 0} \left\{ \sum_i p(x_i) \Delta \ln p(x_i) \right\} = -\int p(x) \ln p(x) \, \mathrm{d}x \tag{1.103}$$

where the quantity on the right-hand side is called the *differential entropy*. We see that the discrete and continuous forms of the entropy differ by a quantity $\ln \Delta$, which diverges in the limit $\Delta \to 0$. This reflects the fact that to specify a continuous variable very precisely requires a large number of bits. For a density defined over multiple continuous variables, denoted collectively by the vector $\mathbf{x}$, the differential entropy is given by

$$\mathrm{H}[\mathbf{x}] = -\int p(\mathbf{x}) \ln p(\mathbf{x}) \, \mathrm{d}\mathbf{x}. \tag{1.104}$$

In the case of discrete distributions, we saw that the maximum entropy configuration corresponded to an equal distribution of probabilities across the possible states of the variable. Let us now consider the maximum entropy configuration for a continuous variable. In order for this maximum to be well defined, it will be necessary to constrain the first and second moments of $p(x)$ as well as preserving the normalization constraint. We therefore maximize the differential entropy with the

Ludwig Boltzmann
1844–1906

Ludwig Eduard Boltzmann was an Austrian physicist who created the field of statistical mechanics. Prior to Boltzmann, the concept of entropy was already known from classical thermodynamics where it quantifies the fact that when we take energy from a system, not all of that energy is typically available to do useful work. Boltzmann showed that the thermodynamic entropy S, a macroscopic quantity, could be related to the statistical properties at the microscopic level. This is expressed through the famous equation $S = k \ln W$ in which W represents the number of possible microstates in a macrostate, and $k \simeq 1.38 \times 10^{-23}$ (in units of Joules per Kelvin) is known as Boltzmann's constant. Boltzmann's ideas were disputed by many scientists of they day. One difficulty they saw arose from the second law of thermodynamics, which states that the entropy of a closed system tends to increase with time. By contrast, at the microscopic level the classical Newtonian equations of physics are reversible, and so they found it difficult to see how the latter could explain the former. They didn't fully appreciate Boltzmann's arguments, which were statistical in nature and which concluded not that entropy could never decrease over time but simply that with overwhelming probability it would generally increase. Boltzmann even had a long-running dispute with the editor of the leading German physics journal who refused to let him refer to atoms and molecules as anything other than convenient theoretical constructs. The continued attacks on his work lead to bouts of depression, and eventually he committed suicide. Shortly after Boltzmann's death, new experiments by Perrin on colloidal suspensions verified his theories and confirmed the value of the Boltzmann constant. The equation $S = k \ln W$ is carved on Boltzmann's tombstone.

three constraints

$$\int_{-\infty}^{\infty} p(x)\,\mathrm{d}x = 1 \tag{1.105}$$

$$\int_{-\infty}^{\infty} xp(x)\,\mathrm{d}x = \mu \tag{1.106}$$

$$\int_{-\infty}^{\infty} (x-\mu)^2 p(x)\,\mathrm{d}x = \sigma^2. \tag{1.107}$$

Appendix E The constrained maximization can be performed using Lagrange multipliers so that we maximize the following functional with respect to $p(x)$

$$-\int_{-\infty}^{\infty} p(x)\ln p(x)\,\mathrm{d}x + \lambda_1\left(\int_{-\infty}^{\infty} p(x)\,\mathrm{d}x - 1\right)$$
$$+\lambda_2\left(\int_{-\infty}^{\infty} xp(x)\,\mathrm{d}x - \mu\right) + \lambda_3\left(\int_{-\infty}^{\infty} (x-\mu)^2 p(x)\,\mathrm{d}x - \sigma^2\right).$$

Appendix D Using the calculus of variations, we set the derivative of this functional to zero giving

$$p(x) = \exp\left\{-1 + \lambda_1 + \lambda_2 x + \lambda_3 (x-\mu)^2\right\}. \tag{1.108}$$

The Lagrange multipliers can be found by back substitution of this result into the three constraint equations, leading finally to the result *Exercise 1.34*

$$p(x) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\} \tag{1.109}$$

and so the distribution that maximizes the differential entropy is the Gaussian. Note that we did not constrain the distribution to be nonnegative when we maximized the entropy. However, because the resulting distribution is indeed nonnegative, we see with hindsight that such a constraint is not necessary.

Exercise 1.35 If we evaluate the differential entropy of the Gaussian, we obtain

$$\mathrm{H}[x] = \frac{1}{2}\left\{1 + \ln(2\pi\sigma^2)\right\}. \tag{1.110}$$

Thus we see again that the entropy increases as the distribution becomes broader, i.e., as σ^2 increases. This result also shows that the differential entropy, unlike the discrete entropy, can be negative, because $\mathrm{H}(x) < 0$ in (1.110) for $\sigma^2 < 1/(2\pi e)$.

Suppose we have a joint distribution $p(\mathbf{x}, \mathbf{y})$ from which we draw pairs of values of $\mathbf{x}$ and $\mathbf{y}$. If a value of $\mathbf{x}$ is already known, then the additional information needed to specify the corresponding value of $\mathbf{y}$ is given by $-\ln p(\mathbf{y}|\mathbf{x})$. Thus the average additional information needed to specify $\mathbf{y}$ can be written as

$$\mathrm{H}[\mathbf{y}|\mathbf{x}] = -\iint p(\mathbf{y}, \mathbf{x}) \ln p(\mathbf{y}|\mathbf{x})\,\mathrm{d}\mathbf{y}\,\mathrm{d}\mathbf{x} \tag{1.111}$$

Exercise 1.37

which is called the *conditional entropy* of $\mathbf{y}$ given $\mathbf{x}$. It is easily seen, using the product rule, that the conditional entropy satisfies the relation

$$\mathrm{H}[\mathbf{x}, \mathbf{y}] = \mathrm{H}[\mathbf{y}|\mathbf{x}] + \mathrm{H}[\mathbf{x}] \tag{1.112}$$

where $\mathrm{H}[\mathbf{x}, \mathbf{y}]$ is the differential entropy of $p(\mathbf{x}, \mathbf{y})$ and $\mathrm{H}[\mathbf{x}]$ is the differential entropy of the marginal distribution $p(\mathbf{x})$. Thus the information needed to describe $\mathbf{x}$ and $\mathbf{y}$ is given by the sum of the information needed to describe $\mathbf{x}$ alone plus the additional information required to specify $\mathbf{y}$ given $\mathbf{x}$.

1.6.1 Relative entropy and mutual information

So far in this section, we have introduced a number of concepts from information theory, including the key notion of entropy. We now start to relate these ideas to pattern recognition. Consider some unknown distribution $p(\mathbf{x})$, and suppose that we have modelled this using an approximating distribution $q(\mathbf{x})$. If we use $q(\mathbf{x})$ to construct a coding scheme for the purpose of transmitting values of $\mathbf{x}$ to a receiver, then the average *additional* amount of information (in nats) required to specify the value of $\mathbf{x}$ (assuming we choose an efficient coding scheme) as a result of using $q(\mathbf{x})$ instead of the true distribution $p(\mathbf{x})$ is given by

$$\begin{aligned} \mathrm{KL}(p\|q) &= -\int p(\mathbf{x}) \ln q(\mathbf{x})\,\mathrm{d}\mathbf{x} - \left(-\int p(\mathbf{x}) \ln p(\mathbf{x})\,\mathrm{d}\mathbf{x}\right) \\ &= -\int p(\mathbf{x}) \ln \left\{\frac{q(\mathbf{x})}{p(\mathbf{x})}\right\} \mathrm{d}\mathbf{x}. \end{aligned} \tag{1.113}$$

This is known as the *relative entropy* or *Kullback-Leibler divergence*, or *KL divergence* (Kullback and Leibler, 1951), between the distributions $p(\mathbf{x})$ and $q(\mathbf{x})$. Note that it is not a symmetrical quantity, that is to say $\mathrm{KL}(p\|q) \not\equiv \mathrm{KL}(q\|p)$.

We now show that the Kullback-Leibler divergence satisfies $\mathrm{KL}(p\|q) \geqslant 0$ with equality if, and only if, $p(\mathbf{x}) = q(\mathbf{x})$. To do this we first introduce the concept of *convex* functions. A function $f(x)$ is said to be convex if it has the property that every chord lies on or above the function, as shown in Figure 1.31. Any value of x in the interval from $x = a$ to $x = b$ can be written in the form $\lambda a + (1 - \lambda)b$ where $0 \leqslant \lambda \leqslant 1$. The corresponding point on the chord is given by $\lambda f(a) + (1 - \lambda)f(b)$,

Claude Shannon
1916–2001

After graduating from Michigan and MIT, Shannon joined the AT&T Bell Telephone laboratories in 1941. His paper 'A Mathematical Theory of Communication' published in the *Bell System Technical Journal* in 1948 laid the foundations for modern information theory. This paper introduced the word 'bit', and his concept that information could be sent as a stream of 1s and 0s paved the way for the communications revolution. It is said that von Neumann recommended to Shannon that he use the term entropy, not only because of its similarity to the quantity used in physics, but also because "nobody knows what entropy really is, so in any discussion you will always have an advantage".

Figure 1.31 A convex function $f(x)$ is one for which every chord (shown in blue) lies on or above the function (shown in red).

and the corresponding value of the function is $f(\lambda a + (1-\lambda)b)$. Convexity then implies

$$f(\lambda a + (1-\lambda)b) \leqslant \lambda f(a) + (1-\lambda) f(b). \tag{1.114}$$

This is equivalent to the requirement that the second derivative of the function be everywhere positive. Examples of convex functions are $x \ln x$ (for $x > 0$) and x^2. A function is called *strictly convex* if the equality is satisfied only for $\lambda = 0$ and $\lambda = 1$. If a function has the opposite property, namely that every chord lies on or below the function, it is called *concave*, with a corresponding definition for *strictly concave*. If a function $f(x)$ is convex, then $-f(x)$ will be concave.

Exercise 1.36

Exercise 1.38

Using the technique of proof by induction, we can show from (1.114) that a convex function $f(x)$ satisfies

$$f\left(\sum_{i=1}^{M} \lambda_i x_i\right) \leqslant \sum_{i=1}^{M} \lambda_i f(x_i) \tag{1.115}$$

where $\lambda_i \geqslant 0$ and $\sum_i \lambda_i = 1$, for any set of points $\{x_i\}$. The result (1.115) is known as *Jensen's inequality*. If we interpret the λ_i as the probability distribution over a discrete variable x taking the values $\{x_i\}$, then (1.115) can be written

$$f\left(\mathbb{E}[x]\right) \leqslant \mathbb{E}[f(x)] \tag{1.116}$$

where $\mathbb{E}[\cdot]$ denotes the expectation. For continuous variables, Jensen's inequality takes the form

$$f\left(\int \mathbf{x} p(\mathbf{x})\, \mathrm{d}\mathbf{x}\right) \leqslant \int f(\mathbf{x}) p(\mathbf{x})\, \mathrm{d}\mathbf{x}. \tag{1.117}$$

We can apply Jensen's inequality in the form (1.117) to the Kullback-Leibler divergence (1.113) to give

$$\mathrm{KL}(p\|q) = -\int p(\mathbf{x}) \ln\left\{\frac{q(\mathbf{x})}{p(\mathbf{x})}\right\} \mathrm{d}\mathbf{x} \geqslant -\ln \int q(\mathbf{x})\, \mathrm{d}\mathbf{x} = 0 \tag{1.118}$$

where we have used the fact that $-\ln x$ is a convex function, together with the normalization condition $\int q(\mathbf{x})\,\mathrm{d}\mathbf{x} = 1$. In fact, $-\ln x$ is a strictly convex function, so the equality will hold if, and only if, $q(\mathbf{x}) = p(\mathbf{x})$ for all $\mathbf{x}$. Thus we can interpret the Kullback-Leibler divergence as a measure of the dissimilarity of the two distributions $p(\mathbf{x})$ and $q(\mathbf{x})$.

We see that there is an intimate relationship between data compression and density estimation (i.e., the problem of modelling an unknown probability distribution) because the most efficient compression is achieved when we know the true distribution. If we use a distribution that is different from the true one, then we must necessarily have a less efficient coding, and on average the additional information that must be transmitted is (at least) equal to the Kullback-Leibler divergence between the two distributions.

Suppose that data is being generated from an unknown distribution $p(\mathbf{x})$ that we wish to model. We can try to approximate this distribution using some parametric distribution $q(\mathbf{x}|\boldsymbol{\theta})$, governed by a set of adjustable parameters $\boldsymbol{\theta}$, for example a multivariate Gaussian. One way to determine $\boldsymbol{\theta}$ is to minimize the Kullback-Leibler divergence between $p(\mathbf{x})$ and $q(\mathbf{x}|\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$. We cannot do this directly because we don't know $p(\mathbf{x})$. Suppose, however, that we have observed a finite set of training points $\mathbf{x}_n$, for $n = 1, \ldots, N$, drawn from $p(\mathbf{x})$. Then the expectation with respect to $p(\mathbf{x})$ can be approximated by a finite sum over these points, using (1.35), so that

$$\mathrm{KL}(p\|q) \simeq \sum_{n=1}^{N} \left\{-\ln q(\mathbf{x}_n|\boldsymbol{\theta}) + \ln p(\mathbf{x}_n)\right\}. \tag{1.119}$$

The second term on the right-hand side of (1.119) is independent of $\boldsymbol{\theta}$, and the first term is the negative log likelihood function for $\boldsymbol{\theta}$ under the distribution $q(\mathbf{x}|\boldsymbol{\theta})$ evaluated using the training set. Thus we see that minimizing this Kullback-Leibler divergence is equivalent to maximizing the likelihood function.

Now consider the joint distribution between two sets of variables $\mathbf{x}$ and $\mathbf{y}$ given by $p(\mathbf{x}, \mathbf{y})$. If the sets of variables are independent, then their joint distribution will factorize into the product of their marginals $p(\mathbf{x}, \mathbf{y}) = p(\mathbf{x})p(\mathbf{y})$. If the variables are not independent, we can gain some idea of whether they are 'close' to being independent by considering the Kullback-Leibler divergence between the joint distribution and the product of the marginals, given by

$$\begin{aligned} \mathrm{I}[\mathbf{x}, \mathbf{y}] &\equiv \mathrm{KL}(p(\mathbf{x}, \mathbf{y})\|p(\mathbf{x})p(\mathbf{y})) \\ &= -\iint p(\mathbf{x}, \mathbf{y}) \ln\left(\frac{p(\mathbf{x})p(\mathbf{y})}{p(\mathbf{x}, \mathbf{y})}\right) \mathrm{d}\mathbf{x}\,\mathrm{d}\mathbf{y} \end{aligned} \tag{1.120}$$

which is called the *mutual information* between the variables $\mathbf{x}$ and $\mathbf{y}$. From the properties of the Kullback-Leibler divergence, we see that $I(\mathbf{x}, \mathbf{y}) \geqslant 0$ with equality if, and only if, $\mathbf{x}$ and $\mathbf{y}$ are independent. Using the sum and product rules of probability, we see that the mutual information is related to the conditional entropy through

Exercise 1.41

$$\mathrm{I}[\mathbf{x}, \mathbf{y}] = \mathrm{H}[\mathbf{x}] - \mathrm{H}[\mathbf{x}|\mathbf{y}] = \mathrm{H}[\mathbf{y}] - \mathrm{H}[\mathbf{y}|\mathbf{x}]. \tag{1.121}$$

Thus we can view the mutual information as the reduction in the uncertainty about $\mathbf{x}$ by virtue of being told the value of $\mathbf{y}$ (or vice versa). From a Bayesian perspective, we can view $p(\mathbf{x})$ as the prior distribution for $\mathbf{x}$ and $p(\mathbf{x}|\mathbf{y})$ as the posterior distribution after we have observed new data $\mathbf{y}$. The mutual information therefore represents the reduction in uncertainty about $\mathbf{x}$ as a consequence of the new observation $\mathbf{y}$.

Exercises

1.1 ($\star$) **www** Consider the sum-of-squares error function given by (1.2) in which the function $y(x, \mathbf{w})$ is given by the polynomial (1.1). Show that the coefficients $\mathbf{w} = \{w_i\}$ that minimize this error function are given by the solution to the following set of linear equations

$$\sum_{j=0}^{M} A_{ij} w_j = T_i \tag{1.122}$$

where

$$A_{ij} = \sum_{n=1}^{N} (x_n)^{i+j}, \qquad T_i = \sum_{n=1}^{N} (x_n)^i t_n. \tag{1.123}$$

Here a suffix i or j denotes the index of a component, whereas $(x)^i$ denotes x raised to the power of i.

1.2 ($\star$) Write down the set of coupled linear equations, analogous to (1.122), satisfied by the coefficients w_i which minimize the regularized sum-of-squares error function given by (1.4).

1.3 ($\star\star$) Suppose that we have three coloured boxes r (red), b (blue), and g (green). Box r contains 3 apples, 4 oranges, and 3 limes, box b contains 1 apple, 1 orange, and 0 limes, and box g contains 3 apples, 3 oranges, and 4 limes. If a box is chosen at random with probabilities $p(r) = 0.2$, $p(b) = 0.2$, $p(g) = 0.6$, and a piece of fruit is removed from the box (with equal probability of selecting any of the items in the box), then what is the probability of selecting an apple? If we observe that the selected fruit is in fact an orange, what is the probability that it came from the green box?

1.4 ($\star\star$) **www** Consider a probability density $p_x(x)$ defined over a continuous variable x, and suppose that we make a nonlinear change of variable using $x = g(y)$, so that the density transforms according to (1.27). By differentiating (1.27), show that the location $\widehat{y}$ of the maximum of the density in y is not in general related to the location $\widehat{x}$ of the maximum of the density over x by the simple functional relation $\widehat{x} = g(\widehat{y})$ as a consequence of the Jacobian factor. This shows that the maximum of a probability density (in contrast to a simple function) is dependent on the choice of variable. Verify that, in the case of a linear transformation, the location of the maximum transforms in the same way as the variable itself.

1.5 ($\star$) Using the definition (1.38) show that $\mathrm{var}[f(x)]$ satisfies (1.39).

1.6 ($\star$) Show that if two variables x and y are independent, then their covariance is zero.

1.7 ($\star\star$) **www** In this exercise, we prove the normalization condition (1.48) for the univariate Gaussian. To do this consider, the integral

$$I = \int_{-\infty}^{\infty} \exp\left(-\frac{1}{2\sigma^2}x^2\right) \mathrm{d}x \tag{1.124}$$

which we can evaluate by first writing its square in the form

$$I^2 = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \exp\left(-\frac{1}{2\sigma^2}x^2 - \frac{1}{2\sigma^2}y^2\right) \mathrm{d}x\,\mathrm{d}y. \tag{1.125}$$

Now make the transformation from Cartesian coordinates (x, y) to polar coordinates (r, θ) and then substitute $u = r^2$. Show that, by performing the integrals over θ and u, and then taking the square root of both sides, we obtain

$$I = \left(2\pi\sigma^2\right)^{1/2}. \tag{1.126}$$

Finally, use this result to show that the Gaussian distribution $\mathcal{N}(x|\mu, \sigma^2)$ is normalized.

1.8 ($\star\star$) **www** By using a change of variables, verify that the univariate Gaussian distribution given by (1.46) satisfies (1.49). Next, by differentiating both sides of the normalization condition

$$\int_{-\infty}^{\infty} \mathcal{N}\left(x|\mu, \sigma^2\right) \mathrm{d}x = 1 \tag{1.127}$$

with respect to σ^2, verify that the Gaussian satisfies (1.50). Finally, show that (1.51) holds.

1.9 ($\star$) **www** Show that the mode (i.e. the maximum) of the Gaussian distribution (1.46) is given by μ. Similarly, show that the mode of the multivariate Gaussian (1.52) is given by $\boldsymbol{\mu}$.

1.10 ($\star$) **www** Suppose that the two variables x and z are statistically independent. Show that the mean and variance of their sum satisfies

$$\mathbb{E}[x + z] = \mathbb{E}[x] + \mathbb{E}[z] \tag{1.128}$$

$$\mathrm{var}[x + z] = \mathrm{var}[x] + \mathrm{var}[z]. \tag{1.129}$$

1.11 ($\star$) By setting the derivatives of the log likelihood function (1.54) with respect to μ and σ^2 equal to zero, verify the results (1.55) and (1.56).

1.12 ($\star\star$) **www** Using the results (1.49) and (1.50), show that

$$\mathbb{E}[x_n x_m] = \mu^2 + I_{nm}\sigma^2 \tag{1.130}$$

where x_n and x_m denote data points sampled from a Gaussian distribution with mean μ and variance σ^2, and I_{nm} satisfies $I_{nm} = 1$ if $n = m$ and $I_{nm} = 0$ otherwise. Hence prove the results (1.57) and (1.58).

1.13 ($\star$) Suppose that the variance of a Gaussian is estimated using the result (1.56) but with the maximum likelihood estimate μ_{ML} replaced with the true value μ of the mean. Show that this estimator has the property that its expectation is given by the true variance σ^2.

1.14 ($\star\star$) Show that an arbitrary square matrix with elements w_{ij} can be written in the form $w_{ij} = w_{ij}^{\mathrm{S}} + w_{ij}^{\mathrm{A}}$ where w_{ij}^{S} and w_{ij}^{A} are symmetric and anti-symmetric matrices, respectively, satisfying $w_{ij}^{\mathrm{S}} = w_{ji}^{\mathrm{S}}$ and $w_{ij}^{\mathrm{A}} = -w_{ji}^{\mathrm{A}}$ for all i and j. Now consider the second order term in a higher order polynomial in D dimensions, given by

$$\sum_{i=1}^{D}\sum_{j=1}^{D} w_{ij} x_i x_j. \tag{1.131}$$

Show that

$$\sum_{i=1}^{D}\sum_{j=1}^{D} w_{ij} x_i x_j = \sum_{i=1}^{D}\sum_{j=1}^{D} w_{ij}^{\mathrm{S}} x_i x_j \tag{1.132}$$

so that the contribution from the anti-symmetric matrix vanishes. We therefore see that, without loss of generality, the matrix of coefficients w_{ij} can be chosen to be symmetric, and so not all of the D^2 elements of this matrix can be chosen independently. Show that the number of independent parameters in the matrix w_{ij}^{S} is given by $D(D+1)/2$.

1.15 ($\star\star\star$) **www** In this exercise and the next, we explore how the number of independent parameters in a polynomial grows with the order M of the polynomial and with the dimensionality D of the input space. We start by writing down the Mth order term for a polynomial in D dimensions in the form

$$\sum_{i_1=1}^{D}\sum_{i_2=1}^{D}\cdots\sum_{i_M=1}^{D} w_{i_1 i_2 \cdots i_M} x_{i_1} x_{i_2} \cdots x_{i_M}. \tag{1.133}$$

The coefficients $w_{i_1 i_2 \cdots i_M}$ comprise D^M elements, but the number of independent parameters is significantly fewer due to the many interchange symmetries of the factor $x_{i_1} x_{i_2} \cdots x_{i_M}$. Begin by showing that the redundancy in the coefficients can be removed by rewriting this M^{th} order term in the form

$$\sum_{i_1=1}^{D}\sum_{i_2=1}^{i_1}\cdots\sum_{i_M=1}^{i_{M-1}} \widetilde{w}_{i_1 i_2 \cdots i_M} x_{i_1} x_{i_2} \cdots x_{i_M}. \tag{1.134}$$

Note that the precise relationship between the $\widetilde{w}$ coefficients and w coefficients need not be made explicit. Use this result to show that the number of *independent* parameters $n(D, M)$, which appear at order M, satisfies the following recursion relation

$$n(D, M) = \sum_{i=1}^{D} n(i, M-1). \tag{1.135}$$

Next use proof by induction to show that the following result holds

$$\sum_{i=1}^{D} \frac{(i+M-2)!}{(i-1)!\,(M-1)!} = \frac{(D+M-1)!}{(D-1)!\,M!} \tag{1.136}$$

which can be done by first proving the result for $D = 1$ and arbitrary M by making use of the result $0! = 1$, then assuming it is correct for dimension D and verifying that it is correct for dimension $D + 1$. Finally, use the two previous results, together with proof by induction, to show

$$n(D, M) = \frac{(D+M-1)!}{(D-1)!\,M!}. \tag{1.137}$$

To do this, first show that the result is true for $M = 2$, and any value of $D \geqslant 1$, by comparison with the result of Exercise 1.14. Then make use of (1.135), together with (1.136), to show that, if the result holds at order $M - 1$, then it will also hold at order M

1.16 ($\star\star\star$) In Exercise 1.15, we proved the result (1.135) for the number of independent parameters in the Mth order term of a D-dimensional polynomial. We now find an expression for the total number $N(D, M)$ of independent parameters in all of the terms up to and including the $M6\text{th}$ order. First show that $N(D, M)$ satisfies

$$N(D, M) = \sum_{m=0}^{M} n(D, m) \tag{1.138}$$

where $n(D, m)$ is the number of independent parameters in the term of order m. Now make use of the result (1.137), together with proof by induction, to show that

$$N(d, M) = \frac{(D+M)!}{D!\,M!}. \tag{1.139}$$

This can be done by first proving that the result holds for $M = 0$ and arbitrary $D \geqslant 1$, then assuming that it holds at order M, and hence showing that it holds at order $M + 1$. Finally, make use of Stirling's approximation in the form

$$n! \simeq n^n e^{-n} \tag{1.140}$$

for large n to show that, for $D \gg M$, the quantity $N(D, M)$ grows like D^M, and for $M \gg D$ it grows like M^D. Consider a cubic ($M = 3$) polynomial in D dimensions, and evaluate numerically the total number of independent parameters for (i) $D = 10$ and (ii) $D = 100$, which correspond to typical small-scale and medium-scale machine learning applications.

1.17 ($\star\star$) **www** The gamma function is defined by

$$\Gamma(x) \equiv \int_0^\infty u^{x-1} e^{-u}\, \mathrm{d}u. \tag{1.141}$$

Using integration by parts, prove the relation $\Gamma(x+1) = x\Gamma(x)$. Show also that $\Gamma(1) = 1$ and hence that $\Gamma(x+1) = x!$ when x is an integer.

1.18 ($\star\star$) **www** We can use the result (1.126) to derive an expression for the surface area S_D, and the volume V_D, of a sphere of unit radius in D dimensions. To do this, consider the following result, which is obtained by transforming from Cartesian to polar coordinates

$$\prod_{i=1}^{D} \int_{-\infty}^{\infty} e^{-x_i^2}\, \mathrm{d}x_i = S_D \int_0^\infty e^{-r^2} r^{D-1}\, \mathrm{d}r. \tag{1.142}$$

Using the definition (1.141) of the Gamma function, together with (1.126), evaluate both sides of this equation, and hence show that

$$S_D = \frac{2\pi^{D/2}}{\Gamma(D/2)}. \tag{1.143}$$

Next, by integrating with respect to radius from 0 to 1, show that the volume of the unit sphere in D dimensions is given by

$$V_D = \frac{S_D}{D}. \tag{1.144}$$

Finally, use the results $\Gamma(1) = 1$ and $\Gamma(3/2) = \sqrt{\pi}/2$ to show that (1.143) and (1.144) reduce to the usual expressions for $D = 2$ and $D = 3$.

1.19 ($\star\star$) Consider a sphere of radius a in D-dimensions together with the concentric hypercube of side $2a$, so that the sphere touches the hypercube at the centres of each of its sides. By using the results of Exercise 1.18, show that the ratio of the volume of the sphere to the volume of the cube is given by

$$\frac{\text{volume of sphere}}{\text{volume of cube}} = \frac{\pi^{D/2}}{D 2^{D-1}\Gamma(D/2)}. \tag{1.145}$$

Now make use of Stirling's formula in the form

$$\Gamma(x+1) \simeq (2\pi)^{1/2} e^{-x} x^{x+1/2} \tag{1.146}$$

which is valid for $x \gg 1$, to show that, as $D \to \infty$, the ratio (1.145) goes to zero. Show also that the ratio of the distance from the centre of the hypercube to one of the corners, divided by the perpendicular distance to one of the sides, is $\sqrt{D}$, which therefore goes to ∞ as $D \to \infty$. From these results we see that, in a space of high dimensionality, most of the volume of a cube is concentrated in the large number of corners, which themselves become very long 'spikes'!

1.20 ($\star\star$) **WWW** In this exercise, we explore the behaviour of the Gaussian distribution in high-dimensional spaces. Consider a Gaussian distribution in D dimensions given by

$$p(\mathbf{x}) = \frac{1}{(2\pi\sigma^2)^{D/2}} \exp\left(-\frac{\|\mathbf{x}\|^2}{2\sigma^2}\right). \tag{1.147}$$

We wish to find the density with respect to radius in polar coordinates in which the direction variables have been integrated out. To do this, show that the integral of the probability density over a thin shell of radius r and thickness ϵ, where $\epsilon \ll 1$, is given by $p(r)\epsilon$ where

$$p(r) = \frac{S_D r^{D-1}}{(2\pi\sigma^2)^{D/2}} \exp\left(-\frac{r^2}{2\sigma^2}\right) \tag{1.148}$$

where S_D is the surface area of a unit sphere in D dimensions. Show that the function $p(r)$ has a single stationary point located, for large D, at $\widehat{r} \simeq \sqrt{D}\sigma$. By considering $p(\widehat{r} + \epsilon)$ where $\epsilon \ll \widehat{r}$, show that for large D,

$$p(\widehat{r} + \epsilon) = p(\widehat{r}) \exp\left(-\frac{3\epsilon^2}{2\sigma^2}\right) \tag{1.149}$$

which shows that $\widehat{r}$ is a maximum of the radial probability density and also that $p(r)$ decays exponentially away from its maximum at $\widehat{r}$ with length scale σ. We have already seen that $\sigma \ll \widehat{r}$ for large D, and so we see that most of the probability mass is concentrated in a thin shell at large radius. Finally, show that the probability density $p(\mathbf{x})$ is larger at the origin than at the radius $\widehat{r}$ by a factor of $\exp(D/2)$. We therefore see that most of the probability mass in a high-dimensional Gaussian distribution is located at a different radius from the region of high probability density. This property of distributions in spaces of high dimensionality will have important consequences when we consider Bayesian inference of model parameters in later chapters.

1.21 ($\star\star$) Consider two nonnegative numbers a and b, and show that, if $a \leqslant b$, then $a \leqslant (ab)^{1/2}$. Use this result to show that, if the decision regions of a two-class classification problem are chosen to minimize the probability of misclassification, this probability will satisfy

$$p(\text{mistake}) \leqslant \int \{p(\mathbf{x}, \mathcal{C}_1)p(\mathbf{x}, \mathcal{C}_2)\}^{1/2}\, \mathrm{d}\mathbf{x}. \tag{1.150}$$

1.22 ($\star$) **WWW** Given a loss matrix with elements L_{kj}, the expected risk is minimized if, for each $\mathbf{x}$, we choose the class that minimizes (1.81). Verify that, when the loss matrix is given by $L_{kj} = 1 - I_{kj}$, where I_{kj} are the elements of the identity matrix, this reduces to the criterion of choosing the class having the largest posterior probability. What is the interpretation of this form of loss matrix?

1.23 ($\star$) Derive the criterion for minimizing the expected loss when there is a general loss matrix and general prior probabilities for the classes.

1.24 ($\star\star$) **www** Consider a classification problem in which the loss incurred when an input vector from class $\mathcal{C}_k$ is classified as belonging to class $\mathcal{C}_j$ is given by the loss matrix L_{kj}, and for which the loss incurred in selecting the reject option is λ. Find the decision criterion that will give the minimum expected loss. Verify that this reduces to the reject criterion discussed in Section 1.5.3 when the loss matrix is given by $L_{kj} = 1 - I_{kj}$. What is the relationship between λ and the rejection threshold θ?

1.25 ($\star$) **www** Consider the generalization of the squared loss function (1.87) for a single target variable t to the case of multiple target variables described by the vector $\mathbf{t}$ given by

$$\mathbb{E}[L(\mathbf{t}, \mathbf{y}(\mathbf{x}))] = \iint \|\mathbf{y}(\mathbf{x}) - \mathbf{t}\|^2 p(\mathbf{x}, \mathbf{t}) \, \mathrm{d}\mathbf{x} \, \mathrm{d}\mathbf{t}. \tag{1.151}$$

Using the calculus of variations, show that the function $\mathbf{y}(\mathbf{x})$ for which this expected loss is minimized is given by $\mathbf{y}(\mathbf{x}) = \mathbb{E}_{\mathbf{t}}[\mathbf{t}|\mathbf{x}]$. Show that this result reduces to (1.89) for the case of a single target variable t.

1.26 ($\star$) By expansion of the square in (1.151), derive a result analogous to (1.90) and hence show that the function $\mathbf{y}(\mathbf{x})$ that minimizes the expected squared loss for the case of a vector $\mathbf{t}$ of target variables is again given by the conditional expectation of $\mathbf{t}$.

1.27 ($\star\star$) **www** Consider the expected loss for regression problems under the L_q loss function given by (1.91). Write down the condition that $y(\mathbf{x})$ must satisfy in order to minimize $\mathbb{E}[L_q]$. Show that, for $q = 1$, this solution represents the conditional median, i.e., the function $y(\mathbf{x})$ such that the probability mass for $t < y(\mathbf{x})$ is the same as for $t \geqslant y(\mathbf{x})$. Also show that the minimum expected L_q loss for $q \to 0$ is given by the conditional mode, i.e., by the function $y(\mathbf{x})$ equal to the value of t that maximizes $p(t|\mathbf{x})$ for each $\mathbf{x}$.

1.28 ($\star$) In Section 1.6, we introduced the idea of entropy $h(x)$ as the information gained on observing the value of a random variable x having distribution $p(x)$. We saw that, for independent variables x and y for which $p(x, y) = p(x)p(y)$, the entropy functions are additive, so that $h(x, y) = h(x) + h(y)$. In this exercise, we derive the relation between h and p in the form of a function $h(p)$. First show that $h(p^2) = 2h(p)$, and hence by induction that $h(p^n) = nh(p)$ where n is a positive integer. Hence show that $h(p^{n/m}) = (n/m)h(p)$ where m is also a positive integer. This implies that $h(p^x) = xh(p)$ where x is a positive rational number, and hence by continuity when it is a positive real number. Finally, show that this implies $h(p)$ must take the form $h(p) \propto \ln p$.

1.29 ($\star$) **www** Consider an M-state discrete random variable x, and use Jensen's inequality in the form (1.115) to show that the entropy of its distribution $p(x)$ satisfies $\mathrm{H}[x] \leqslant \ln M$.

1.30 ($\star\star$) Evaluate the Kullback-Leibler divergence (1.113) between two Gaussians $p(x) = \mathcal{N}(x|\mu, \sigma^2)$ and $q(x) = \mathcal{N}(x|m, s^2)$.

Table 1.3 The joint distribution $p(x, y)$ for two binary variables x and y used in Exercise 1.39.

		y = 0	y = 1
x	0	1/3	1/3
	1	0	1/3

1.31 ($\star\star$) **www** Consider two variables $\mathbf{x}$ and $\mathbf{y}$ having joint distribution $p(\mathbf{x}, \mathbf{y})$. Show that the differential entropy of this pair of variables satisfies

$$\mathrm{H}[\mathbf{x}, \mathbf{y}] \leqslant \mathrm{H}[\mathbf{x}] + \mathrm{H}[\mathbf{y}] \tag{1.152}$$

with equality if, and only if, $\mathbf{x}$ and $\mathbf{y}$ are statistically independent.

1.32 ($\star$) Consider a vector $\mathbf{x}$ of continuous variables with distribution $p(\mathbf{x})$ and corresponding entropy $\mathrm{H}[\mathbf{x}]$. Suppose that we make a nonsingular linear transformation of $\mathbf{x}$ to obtain a new variable $\mathbf{y} = \mathbf{A}\mathbf{x}$. Show that the corresponding entropy is given by $\mathrm{H}[\mathbf{y}] = \mathrm{H}[\mathbf{x}] + \ln |\mathbf{A}|$ where $|\mathbf{A}|$ denotes the determinant of $\mathbf{A}$.

1.33 ($\star\star$) Suppose that the conditional entropy $\mathrm{H}[y|x]$ between two discrete random variables x and y is zero. Show that, for all values of x such that $p(x) > 0$, the variable y must be a function of x, in other words for each x there is only one value of y such that $p(y|x) \neq 0$.

1.34 ($\star\star$) **www** Use the calculus of variations to show that the stationary point of the functional (1.108) is given by (1.108). Then use the constraints (1.105), (1.106), and (1.107) to eliminate the Lagrange multipliers and hence show that the maximum entropy solution is given by the Gaussian (1.109).

1.35 ($\star$) **www** Use the results (1.106) and (1.107) to show that the entropy of the univariate Gaussian (1.109) is given by (1.110).

1.36 ($\star$) A strictly convex function is defined as one for which every chord lies above the function. Show that this is equivalent to the condition that the second derivative of the function be positive.

1.37 ($\star$) Using the definition (1.111) together with the product rule of probability, prove the result (1.112).

1.38 ($\star\star$) **www** Using proof by induction, show that the inequality (1.114) for convex functions implies the result (1.115).

1.39 ($\star\star\star$) Consider two binary variables x and y having the joint distribution given in Table 1.3.

Evaluate the following quantities

(a) $\mathrm{H}[x]$ **(c)** $\mathrm{H}[y|x]$ **(e)** $\mathrm{H}[x, y]$
(b) $\mathrm{H}[y]$ **(d)** $\mathrm{H}[x|y]$ **(f)** $\mathrm{I}[x, y]$.

Draw a diagram to show the relationship between these various quantities.

1.40 ($\star$) By applying Jensen's inequality (1.115) with $f(x) = \ln x$, show that the arithmetic mean of a set of real numbers is never less than their geometrical mean.

1.41 ($\star$) **www** Using the sum and product rules of probability, show that the mutual information $I(\mathbf{x}, \mathbf{y})$ satisfies the relation (1.121).

2

Probability Distributions

In Chapter 1, we emphasized the central role played by probability theory in the solution of pattern recognition problems. We turn now to an exploration of some particular examples of probability distributions and their properties. As well as being of great interest in their own right, these distributions can form building blocks for more complex models and will be used extensively throughout the book. The distributions introduced in this chapter will also serve another important purpose, namely to provide us with the opportunity to discuss some key statistical concepts, such as Bayesian inference, in the context of simple models before we encounter them in more complex situations in later chapters.

One role for the distributions discussed in this chapter is to model the probability distribution $p(\mathbf{x})$ of a random variable $\mathbf{x}$, given a finite set $\mathbf{x}_1, \ldots, \mathbf{x}_N$ of observations. This problem is known as *density estimation*. For the purposes of this chapter, we shall assume that the data points are independent and identically distributed. It should be emphasized that the problem of density estimation is fun-

damentally ill-posed, because there are infinitely many probability distributions that could have given rise to the observed finite data set. Indeed, any distribution $p(\mathbf{x})$ that is nonzero at each of the data points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ is a potential candidate. The issue of choosing an appropriate distribution relates to the problem of model selection that has already been encountered in the context of polynomial curve fitting in Chapter 1 and that is a central issue in pattern recognition.

We begin by considering the binomial and multinomial distributions for discrete random variables and the Gaussian distribution for continuous random variables. These are specific examples of *parametric* distributions, so-called because they are governed by a small number of adaptive parameters, such as the mean and variance in the case of a Gaussian for example. To apply such models to the problem of density estimation, we need a procedure for determining suitable values for the parameters, given an observed data set. In a frequentist treatment, we choose specific values for the parameters by optimizing some criterion, such as the likelihood function. By contrast, in a Bayesian treatment we introduce prior distributions over the parameters and then use Bayes' theorem to compute the corresponding posterior distribution given the observed data.

We shall see that an important role is played by *conjugate* priors, that lead to posterior distributions having the same functional form as the prior, and that therefore lead to a greatly simplified Bayesian analysis. For example, the conjugate prior for the parameters of the multinomial distribution is called the *Dirichlet* distribution, while the conjugate prior for the mean of a Gaussian is another Gaussian. All of these distributions are examples of the *exponential family* of distributions, which possess a number of important properties, and which will be discussed in some detail.

One limitation of the parametric approach is that it assumes a specific functional form for the distribution, which may turn out to be inappropriate for a particular application. An alternative approach is given by *nonparametric* density estimation methods in which the form of the distribution typically depends on the size of the data set. Such models still contain parameters, but these control the model complexity rather than the form of the distribution. We end this chapter by considering three nonparametric methods based respectively on histograms, nearest-neighbours, and kernels.

2.1. Binary Variables

We begin by considering a single binary random variable $x \in \{0, 1\}$. For example, x might describe the outcome of flipping a coin, with $x = 1$ representing 'heads', and $x = 0$ representing 'tails'. We can imagine that this is a damaged coin so that the probability of landing heads is not necessarily the same as that of landing tails. The probability of $x = 1$ will be denoted by the parameter μ so that

$$p(x = 1|\mu) = \mu \tag{2.1}$$

where $0 \leqslant \mu \leqslant 1$, from which it follows that $p(x = 0|\mu) = 1 - \mu$. The probability distribution over x can therefore be written in the form

$$\mathrm{Bern}(x|\mu) = \mu^x (1 - \mu)^{1-x} \tag{2.2}$$

Exercise 2.1

which is known as the *Bernoulli* distribution. It is easily verified that this distribution is normalized and that it has mean and variance given by

$$\mathbb{E}[x] = \mu \tag{2.3}$$

$$\mathrm{var}[x] = \mu(1 - \mu). \tag{2.4}$$

Now suppose we have a data set $\mathcal{D} = \{x_1, \ldots, x_N\}$ of observed values of x. We can construct the likelihood function, which is a function of μ, on the assumption that the observations are drawn independently from $p(x|\mu)$, so that

$$p(\mathcal{D}|\mu) = \prod_{n=1}^{N} p(x_n|\mu) = \prod_{n=1}^{N} \mu^{x_n} (1 - \mu)^{1-x_n}. \tag{2.5}$$

In a frequentist setting, we can estimate a value for μ by maximizing the likelihood function, or equivalently by maximizing the logarithm of the likelihood. In the case of the Bernoulli distribution, the log likelihood function is given by

$$\ln p(\mathcal{D}|\mu) = \sum_{n=1}^{N} \ln p(x_n|\mu) = \sum_{n=1}^{N} \{x_n \ln \mu + (1 - x_n) \ln(1 - \mu)\}. \tag{2.6}$$

At this point, it is worth noting that the log likelihood function depends on the N observations x_n only through their sum $\sum_n x_n$. This sum provides an example of a *sufficient statistic* for the data under this distribution, and we shall study the important role of sufficient statistics in some detail. If we set the derivative of $\ln p(\mathcal{D}|\mu)$ with respect to μ equal to zero, we obtain the maximum likelihood estimator

Section 2.4

$$\mu_{\mathrm{ML}} = \frac{1}{N} \sum_{n=1}^{N} x_n \tag{2.7}$$

Jacob Bernoulli

1654–1705

Jacob Bernoulli, also known as Jacques or James Bernoulli, was a Swiss mathematician and was the first of many in the Bernoulli family to pursue a career in science and mathematics. Although compelled to study philosophy and theology against his will by his parents, he travelled extensively after graduating in order to meet with many of the leading scientists of his time, including Boyle and Hooke in England. When he returned to Switzerland, he taught mechanics and became Professor of Mathematics at Basel in 1687. Unfortunately, rivalry between Jacob and his younger brother Johann turned an initially productive collaboration into a bitter and public dispute. Jacob's most significant contributions to mathematics appeared in *The Art of Conjecture* published in 1713, eight years after his death, which deals with topics in probability theory including what has become known as the Bernoulli distribution.

Figure 2.1 Histogram plot of the binomial distribution (2.9) as a function of m for $N = 10$ and $\mu = 0.25$.

which is also known as the *sample mean*. If we denote the number of observations of $x = 1$ (heads) within this data set by m, then we can write (2.7) in the form

$$\mu_{\mathrm{ML}} = \frac{m}{N} \tag{2.8}$$

so that the probability of landing heads is given, in this maximum likelihood framework, by the fraction of observations of heads in the data set.

Now suppose we flip a coin, say, 3 times and happen to observe 3 heads. Then $N = m = 3$ and $\mu_{\mathrm{ML}} = 1$. In this case, the maximum likelihood result would predict that all future observations should give heads. Common sense tells us that this is unreasonable, and in fact this is an extreme example of the over-fitting associated with maximum likelihood. We shall see shortly how to arrive at more sensible conclusions through the introduction of a prior distribution over μ.

We can also work out the distribution of the number m of observations of $x = 1$, given that the data set has size N. This is called the *binomial* distribution, and from (2.5) we see that it is proportional to $\mu^m(1-\mu)^{N-m}$. In order to obtain the normalization coefficient we note that out of N coin flips, we have to add up all of the possible ways of obtaining m heads, so that the binomial distribution can be written

$$\mathrm{Bin}(m|N,\mu) = \binom{N}{m}\mu^m(1-\mu)^{N-m} \tag{2.9}$$

where

$$\binom{N}{m} \equiv \frac{N!}{(N-m)!m!} \tag{2.10}$$

Exercise 2.3

is the number of ways of choosing m objects out of a total of N identical objects. Figure 2.1 shows a plot of the binomial distribution for $N = 10$ and $\mu = 0.25$.

The mean and variance of the binomial distribution can be found by using the result of Exercise 1.10, which shows that for independent events the mean of the sum is the sum of the means, and the variance of the sum is the sum of the variances. Because $m = x_1 + \ldots + x_N$, and for each observation the mean and variance are

given by (2.3) and (2.4), respectively, we have

$$\mathbb{E}[m] \equiv \sum_{m=0}^{N} m\mathrm{Bin}(m|N,\mu) = N\mu \tag{2.11}$$

$$\mathrm{var}[m] \equiv \sum_{m=0}^{N} (m-\mathbb{E}[m])^2\,\mathrm{Bin}(m|N,\mu) = N\mu(1-\mu). \tag{2.12}$$

Exercise 2.4 These results can also be proved directly using calculus.

2.1.1 The beta distribution

We have seen in (2.8) that the maximum likelihood setting for the parameter μ in the Bernoulli distribution, and hence in the binomial distribution, is given by the fraction of the observations in the data set having $x = 1$. As we have already noted, this can give severely over-fitted results for small data sets. In order to develop a Bayesian treatment for this problem, we need to introduce a prior distribution $p(\mu)$ over the parameter μ. Here we consider a form of prior distribution that has a simple interpretation as well as some useful analytical properties. To motivate this prior, we note that the likelihood function takes the form of the product of factors of the form $\mu^x(1-\mu)^{1-x}$. If we choose a prior to be proportional to powers of μ and $(1-\mu)$, then the posterior distribution, which is proportional to the product of the prior and the likelihood function, will have the same functional form as the prior. This property is called *conjugacy* and we will see several examples of it later in this chapter. We therefore choose a prior, called the *beta* distribution, given by

$$\mathrm{Beta}(\mu|a,b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\mu^{a-1}(1-\mu)^{b-1} \tag{2.13}$$

where $\Gamma(x)$ is the gamma function defined by (1.141), and the coefficient in (2.13) ensures that the beta distribution is normalized, so that

Exercise 2.5

$$\int_0^1 \mathrm{Beta}(\mu|a,b)\,\mathrm{d}\mu = 1. \tag{2.14}$$

Exercise 2.6 The mean and variance of the beta distribution are given by

$$\mathbb{E}[\mu] = \frac{a}{a+b} \tag{2.15}$$

$$\mathrm{var}[\mu] = \frac{ab}{(a+b)^2(a+b+1)}. \tag{2.16}$$

The parameters a and b are often called *hyperparameters* because they control the distribution of the parameter μ. Figure 2.2 shows plots of the beta distribution for various values of the hyperparameters.

The posterior distribution of μ is now obtained by multiplying the beta prior (2.13) by the binomial likelihood function (2.9) and normalizing. Keeping only the factors that depend on μ, we see that this posterior distribution has the form

$$p(\mu|m,l,a,b) \propto \mu^{m+a-1}(1-\mu)^{l+b-1} \tag{2.17}$$

Figure 2.2 Plots of the beta distribution $\mathrm{Beta}(\mu|a, b)$ given by (2.13) as a function of μ for various values of the hyperparameters a and b.

where $l = N - m$, and therefore corresponds to the number of 'tails' in the coin example. We see that (2.17) has the same functional dependence on μ as the prior distribution, reflecting the conjugacy properties of the prior with respect to the likelihood function. Indeed, it is simply another beta distribution, and its normalization coefficient can therefore be obtained by comparison with (2.13) to give

$$p(\mu|m, l, a, b) = \frac{\Gamma(m + a + l + b)}{\Gamma(m + a)\Gamma(l + b)} \mu^{m+a-1}(1 - \mu)^{l+b-1}. \tag{2.18}$$

We see that the effect of observing a data set of m observations of $x = 1$ and l observations of $x = 0$ has been to increase the value of a by m, and the value of b by l, in going from the prior distribution to the posterior distribution. This allows us to provide a simple interpretation of the hyperparameters a and b in the prior as an *effective number of observations* of $x = 1$ and $x = 0$, respectively. Note that a and b need not be integers. Furthermore, the posterior distribution can act as the prior if we subsequently observe additional data. To see this, we can imagine taking observations one at a time and after each observation updating the current posterior

Figure 2.3 Illustration of one step of sequential Bayesian inference. The prior is given by a beta distribution with parameters $a = 2$, $b = 2$, and the likelihood function, given by (2.9) with $N = m = 1$, corresponds to a single observation of $x = 1$, so that the posterior is given by a beta distribution with parameters $a = 3$, $b = 2$.

distribution by multiplying by the likelihood function for the new observation and then normalizing to obtain the new, revised posterior distribution. At each stage, the posterior is a beta distribution with some total number of (prior and actual) observed values for $x = 1$ and $x = 0$ given by the parameters a and b. Incorporation of an additional observation of $x = 1$ simply corresponds to incrementing the value of a by 1, whereas for an observation of $x = 0$ we increment b by 1. Figure 2.3 illustrates one step in this process.

We see that this *sequential* approach to learning arises naturally when we adopt a Bayesian viewpoint. It is independent of the choice of prior and of the likelihood function and depends only on the assumption of i.i.d. data. Sequential methods make use of observations one at a time, or in small batches, and then discard them before the next observations are used. They can be used, for example, in real-time learning scenarios where a steady stream of data is arriving, and predictions must be made before all of the data is seen. Because they do not require the whole data set to be stored or loaded into memory, sequential methods are also useful for large data sets. Maximum likelihood methods can also be cast into a sequential framework.

Section 2.3.5

If our goal is to predict, as best we can, the outcome of the next trial, then we must evaluate the predictive distribution of x, given the observed data set $\mathcal{D}$. From the sum and product rules of probability, this takes the form

$$p(x = 1|\mathcal{D}) = \int_0^1 p(x = 1|\mu)p(\mu|\mathcal{D})\,\mathrm{d}\mu = \int_0^1 \mu p(\mu|\mathcal{D})\,\mathrm{d}\mu = \mathbb{E}[\mu|\mathcal{D}]. \tag{2.19}$$

Using the result (2.18) for the posterior distribution $p(\mu|\mathcal{D})$, together with the result (2.15) for the mean of the beta distribution, we obtain

$$p(x = 1|\mathcal{D}) = \frac{m + a}{m + a + l + b} \tag{2.20}$$

which has a simple interpretation as the total fraction of observations (both real observations and fictitious prior observations) that correspond to $x = 1$. Note that in the limit of an infinitely large data set $m, l \to \infty$ the result (2.20) reduces to the maximum likelihood result (2.8). As we shall see, it is a very general property that the Bayesian and maximum likelihood results will agree in the limit of an infinitely

large data set. For a finite data set, the posterior mean for μ always lies between the prior mean and the maximum likelihood estimate for μ corresponding to the relative frequencies of events given by (2.7).

Exercise 2.7

From Figure 2.2, we see that as the number of observations increases, so the posterior distribution becomes more sharply peaked. This can also be seen from the result (2.16) for the variance of the beta distribution, in which we see that the variance goes to zero for $a \to \infty$ or $b \to \infty$. In fact, we might wonder whether it is a general property of Bayesian learning that, as we observe more and more data, the uncertainty represented by the posterior distribution will steadily decrease.

To address this, we can take a frequentist view of Bayesian learning and show that, on average, such a property does indeed hold. Consider a general Bayesian inference problem for a parameter $\boldsymbol{\theta}$ for which we have observed a data set $\mathcal{D}$, described by the joint distribution $p(\boldsymbol{\theta}, \mathcal{D})$. The following result

Exercise 2.8

$$\mathbb{E}_{\boldsymbol{\theta}}[\boldsymbol{\theta}] = \mathbb{E}_{\mathcal{D}}\left[\mathbb{E}_{\boldsymbol{\theta}}[\boldsymbol{\theta}|\mathcal{D}]\right] \tag{2.21}$$

where

$$\mathbb{E}_{\boldsymbol{\theta}}[\boldsymbol{\theta}] \equiv \int p(\boldsymbol{\theta})\boldsymbol{\theta}\,\mathrm{d}\boldsymbol{\theta} \tag{2.22}$$

$$\mathbb{E}_{\mathcal{D}}[\mathbb{E}_{\boldsymbol{\theta}}[\boldsymbol{\theta}|\mathcal{D}]] \equiv \int \left\{ \int \boldsymbol{\theta} p(\boldsymbol{\theta}|\mathcal{D})\,\mathrm{d}\boldsymbol{\theta} \right\} p(\mathcal{D})\,\mathrm{d}\mathcal{D} \tag{2.23}$$

says that the posterior mean of $\boldsymbol{\theta}$, averaged over the distribution generating the data, is equal to the prior mean of $\boldsymbol{\theta}$. Similarly, we can show that

$$\mathrm{var}_{\boldsymbol{\theta}}[\boldsymbol{\theta}] = \mathbb{E}_{\mathcal{D}}\left[\mathrm{var}_{\boldsymbol{\theta}}[\boldsymbol{\theta}|\mathcal{D}]\right] + \mathrm{var}_{\mathcal{D}}\left[\mathbb{E}_{\boldsymbol{\theta}}[\boldsymbol{\theta}|\mathcal{D}]\right]. \tag{2.24}$$

The term on the left-hand side of (2.24) is the prior variance of $\boldsymbol{\theta}$. On the right-hand side, the first term is the average posterior variance of $\boldsymbol{\theta}$, and the second term measures the variance in the posterior mean of $\boldsymbol{\theta}$. Because this variance is a positive quantity, this result shows that, on average, the posterior variance of $\boldsymbol{\theta}$ is smaller than the prior variance. The reduction in variance is greater if the variance in the posterior mean is greater. Note, however, that this result only holds on average, and that for a particular observed data set it is possible for the posterior variance to be larger than the prior variance.

2.2. Multinomial Variables

Binary variables can be used to describe quantities that can take one of two possible values. Often, however, we encounter discrete variables that can take on one of K possible mutually exclusive states. Although there are various alternative ways to express such variables, we shall see shortly that a particularly convenient representation is the 1-of-K scheme in which the variable is represented by a K-dimensional vector $\mathbf{x}$ in which one of the elements x_k equals 1, and all remaining elements equal

0. So, for instance if we have a variable that can take $K = 6$ states and a particular observation of the variable happens to correspond to the state where $x_3 = 1$, then $\mathbf{x}$ will be represented by

$$\mathbf{x} = (0, 0, 1, 0, 0, 0)^{\mathrm{T}}. \tag{2.25}$$

Note that such vectors satisfy $\sum_{k=1}^{K} x_k = 1$. If we denote the probability of $x_k = 1$ by the parameter μ_k, then the distribution of $\mathbf{x}$ is given

$$p(\mathbf{x}|\boldsymbol{\mu}) = \prod_{k=1}^{K} \mu_k^{x_k} \tag{2.26}$$

where $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_K)^{\mathrm{T}}$, and the parameters μ_k are constrained to satisfy $\mu_k \geqslant 0$ and $\sum_k \mu_k = 1$, because they represent probabilities. The distribution (2.26) can be regarded as a generalization of the Bernoulli distribution to more than two outcomes. It is easily seen that the distribution is normalized

$$\sum_{\mathbf{x}} p(\mathbf{x}|\boldsymbol{\mu}) = \sum_{k=1}^{K} \mu_k = 1 \tag{2.27}$$

and that

$$\mathbb{E}[\mathbf{x}|\boldsymbol{\mu}] = \sum_{\mathbf{x}} p(\mathbf{x}|\boldsymbol{\mu})\mathbf{x} = (\mu_1, \ldots, \mu_M)^{\mathrm{T}} = \boldsymbol{\mu}. \tag{2.28}$$

Now consider a data set $\mathcal{D}$ of N independent observations $\mathbf{x}_1, \ldots, \mathbf{x}_N$. The corresponding likelihood function takes the form

$$p(\mathcal{D}|\boldsymbol{\mu}) = \prod_{n=1}^{N} \prod_{k=1}^{K} \mu_k^{x_{nk}} = \prod_{k=1}^{K} \mu_k^{\left(\sum_n x_{nk}\right)} = \prod_{k=1}^{K} \mu_k^{m_k}. \tag{2.29}$$

We see that the likelihood function depends on the N data points only through the K quantities

$$m_k = \sum_n x_{nk} \tag{2.30}$$

which represent the number of observations of $x_k = 1$. These are called the *sufficient statistics* for this distribution.

Section 2.4

In order to find the maximum likelihood solution for $\boldsymbol{\mu}$, we need to maximize $\ln p(\mathcal{D}|\boldsymbol{\mu})$ with respect to μ_k taking account of the constraint that the μ_k must sum to one. This can be achieved using a Lagrange multiplier λ and maximizing

Appendix E

$$\sum_{k=1}^{K} m_k \ln \mu_k + \lambda \left(\sum_{k=1}^{K} \mu_k - 1 \right). \tag{2.31}$$

Setting the derivative of (2.31) with respect to μ_k to zero, we obtain

$$\mu_k = -m_k/\lambda. \tag{2.32}$$

We can solve for the Lagrange multiplier λ by substituting (2.32) into the constraint $\sum_k \mu_k = 1$ to give $\lambda = -N$. Thus we obtain the maximum likelihood solution in the form

$$\mu_k^{\mathrm{ML}} = \frac{m_k}{N} \tag{2.33}$$

which is the fraction of the N observations for which $x_k = 1$.

We can consider the joint distribution of the quantities $m_1, \ldots, m_K$, conditioned on the parameters $\boldsymbol{\mu}$ and on the total number N of observations. From (2.29) this takes the form

$$\mathrm{Mult}(m_1, m_2, \ldots, m_K | \boldsymbol{\mu}, N) = \binom{N}{m_1 m_2 \ldots m_K} \prod_{k=1}^{K} \mu_k^{m_k} \tag{2.34}$$

which is known as the *multinomial* distribution. The normalization coefficient is the number of ways of partitioning N objects into K groups of size $m_1, \ldots, m_K$ and is given by

$$\binom{N}{m_1 m_2 \ldots m_K} = \frac{N!}{m_1! m_2! \ldots m_K!}. \tag{2.35}$$

Note that the variables m_k are subject to the constraint

$$\sum_{k=1}^{K} m_k = N. \tag{2.36}$$

2.2.1 The Dirichlet distribution

We now introduce a family of prior distributions for the parameters $\{\mu_k\}$ of the multinomial distribution (2.34). By inspection of the form of the multinomial distribution, we see that the conjugate prior is given by

$$p(\boldsymbol{\mu}|\boldsymbol{\alpha}) \propto \prod_{k=1}^{K} \mu_k^{\alpha_k - 1} \tag{2.37}$$

where $0 \leqslant \mu_k \leqslant 1$ and $\sum_k \mu_k = 1$. Here $\alpha_1, \ldots, \alpha_K$ are the parameters of the distribution, and $\boldsymbol{\alpha}$ denotes $(\alpha_1, \ldots, \alpha_K)^{\mathrm{T}}$. Note that, because of the summation constraint, the distribution over the space of the $\{\mu_k\}$ is confined to a *simplex* of dimensionality $K - 1$, as illustrated for $K = 3$ in Figure 2.4.

Exercise 2.9 The normalized form for this distribution is by

$$\mathrm{Dir}(\boldsymbol{\mu}|\boldsymbol{\alpha}) = \frac{\Gamma(\alpha_0)}{\Gamma(\alpha_1) \cdots \Gamma(\alpha_K)} \prod_{k=1}^{K} \mu_k^{\alpha_k - 1} \tag{2.38}$$

which is called the *Dirichlet* distribution. Here $\Gamma(x)$ is the gamma function defined by (1.141) while

$$\alpha_0 = \sum_{k=1}^{K} \alpha_k. \tag{2.39}$$

Figure 2.4 The Dirichlet distribution over three variables μ_1, μ_2, μ_3 is confined to a simplex (a bounded linear manifold) of the form shown, as a consequence of the constraints $0 \leqslant \mu_k \leqslant 1$ and $\sum_k \mu_k = 1$.

Plots of the Dirichlet distribution over the simplex, for various settings of the parameters α_k, are shown in Figure 2.5.

Multiplying the prior (2.38) by the likelihood function (2.34), we obtain the posterior distribution for the parameters $\{\mu_k\}$ in the form

$$p(\boldsymbol{\mu}|\mathcal{D}, \boldsymbol{\alpha}) \propto p(\mathcal{D}|\boldsymbol{\mu})p(\boldsymbol{\mu}|\boldsymbol{\alpha}) \propto \prod_{k=1}^{K} \mu_k^{\alpha_k + m_k - 1}. \tag{2.40}$$

We see that the posterior distribution again takes the form of a Dirichlet distribution, confirming that the Dirichlet is indeed a conjugate prior for the multinomial. This allows us to determine the normalization coefficient by comparison with (2.38) so that

$$\begin{aligned} p(\boldsymbol{\mu}|\mathcal{D}, \boldsymbol{\alpha}) &= \mathrm{Dir}(\boldsymbol{\mu}|\boldsymbol{\alpha} + \mathbf{m}) \\ &= \frac{\Gamma(\alpha_0 + N)}{\Gamma(\alpha_1 + m_1) \cdots \Gamma(\alpha_K + m_K)} \prod_{k=1}^{K} \mu_k^{\alpha_k + m_k - 1} \end{aligned} \tag{2.41}$$

where we have denoted $\mathbf{m} = (m_1, \ldots, m_K)^{\mathrm{T}}$. As for the case of the binomial distribution with its beta prior, we can interpret the parameters α_k of the Dirichlet prior as an effective number of observations of $x_k = 1$.

Note that two-state quantities can either be represented as binary variables and

Lejeune Dirichlet
1805–1859

Johann Peter Gustav Lejeune Dirichlet was a modest and reserved mathematician who made contributions in number theory, mechanics, and astronomy, and who gave the first rigorous analysis of Fourier series. His family originated from Richelet in Belgium, and the name Lejeune Dirichlet comes from 'le jeune de Richelet' (the young person from Richelet). Dirichlet's first paper, which was published in 1825, brought him instant fame. It concerned Fermat's last theorem, which claims that there are no positive integer solutions to $x^n + y^n = z^n$ for $n > 2$. Dirichlet gave a partial proof for the case $n = 5$, which was sent to Legendre for review and who in turn completed the proof. Later, Dirichlet gave a complete proof for $n = 14$, although a full proof of Fermat's last theorem for arbitrary n had to wait until the work of Andrew Wiles in the closing years of the 20th century.

Figure 2.5 Plots of the Dirichlet distribution over three variables, where the two horizontal axes are coordinates in the plane of the simplex and the vertical axis corresponds to the value of the density. Here $\{\alpha_k\} = 0.1$ on the left plot, $\{\alpha_k\} = 1$ in the centre plot, and $\{\alpha_k\} = 10$ in the right plot.

modelled using the binomial distribution (2.9) or as 1-of-2 variables and modelled using the multinomial distribution (2.34) with $K = 2$.

2.3. The Gaussian Distribution

The Gaussian, also known as the normal distribution, is a widely used model for the distribution of continuous variables. In the case of a single variable x, the Gaussian distribution can be written in the form

$$\mathcal{N}(x|\mu, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\} \tag{2.42}$$

where μ is the mean and σ^2 is the variance. For a D-dimensional vector $\mathbf{x}$, the multivariate Gaussian distribution takes the form

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} \exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right\} \tag{2.43}$$

where $\boldsymbol{\mu}$ is a D-dimensional mean vector, $\boldsymbol{\Sigma}$ is a $D \times D$ covariance matrix, and $|\boldsymbol{\Sigma}|$ denotes the determinant of $\boldsymbol{\Sigma}$.

The Gaussian distribution arises in many different contexts and can be motivated from a variety of different perspectives. For example, we have already seen that for a single real variable, the distribution that maximizes the entropy is the Gaussian. This property applies also to the multivariate Gaussian.

Section 1.6

Exercise 2.14

Another situation in which the Gaussian distribution arises is when we consider the sum of multiple random variables. The *central limit theorem* (due to Laplace) tells us that, subject to certain mild conditions, the sum of a set of random variables, which is of course itself a random variable, has a distribution that becomes increasingly Gaussian as the number of terms in the sum increases (Walker, 1969). We can

Figure 2.6 Histogram plots of the mean of N uniformly distributed numbers for various values of N. We observe that as N increases, the distribution tends towards a Gaussian.

illustrate this by considering N variables $x_1, \ldots, x_N$ each of which has a uniform distribution over the interval $[0, 1]$ and then considering the distribution of the mean $(x_1 + \cdots + x_N)/N$. For large N, this distribution tends to a Gaussian, as illustrated in Figure 2.6. In practice, the convergence to a Gaussian as N increases can be very rapid. One consequence of this result is that the binomial distribution (2.9), which is a distribution over m defined by the sum of N observations of the random binary variable x, will tend to a Gaussian as $N \to \infty$ (see Figure 2.1 for the case of $N = 10$).

The Gaussian distribution has many important analytical properties, and we shall consider several of these in detail. As a result, this section will be rather more technically involved than some of the earlier sections, and will require familiarity with various matrix identities. However, we strongly encourage the reader to become proficient in manipulating Gaussian distributions using the techniques presented here as this will prove invaluable in understanding the more complex models presented in later chapters.

Appendix C

We begin by considering the geometrical form of the Gaussian distribution. The

Carl Friedrich Gauss
1777–1855

It is said that when Gauss went to elementary school at age 7, his teacher Büttner, trying to keep the class occupied, asked the pupils to sum the integers from 1 to 100. To the teacher's amazement, Gauss arrived at the answer in a matter of moments by noting that the sum can be represented as 50 pairs ($1 + 100$, $2+99$, etc.) each of which added to 101, giving the answer 5,050. It is now believed that the problem which was actually set was of the same form but somewhat harder in that the sequence had a larger starting value and a larger increment. Gauss was a German mathematician and scientist with a reputation for being a hard-working perfectionist. One of his many contributions was to show that least squares can be derived under the assumption of normally distributed errors. He also created an early formulation of non-Euclidean geometry (a self-consistent geometrical theory that violates the axioms of Euclid) but was reluctant to discuss it openly for fear that his reputation might suffer if it were seen that he believed in such a geometry. At one point, Gauss was asked to conduct a geodetic survey of the state of Hanover, which led to his formulation of the normal distribution, now also known as the Gaussian. After his death, a study of his diaries revealed that he had discovered several important mathematical results years or even decades before they were published by others.

functional dependence of the Gaussian on $\mathbf{x}$ is through the quadratic form

$$\Delta^2 = (\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \tag{2.44}$$

which appears in the exponent. The quantity Δ is called the *Mahalanobis distance* from $\boldsymbol{\mu}$ to $\mathbf{x}$ and reduces to the Euclidean distance when $\boldsymbol{\Sigma}$ is the identity matrix. The Gaussian distribution will be constant on surfaces in $\mathbf{x}$-space for which this quadratic form is constant.

First of all, we note that the matrix $\boldsymbol{\Sigma}$ can be taken to be symmetric, without loss of generality, because any antisymmetric component would disappear from the exponent. Now consider the eigenvector equation for the covariance matrix

Exercise 2.17

$$\boldsymbol{\Sigma}\mathbf{u}_i = \lambda_i \mathbf{u}_i \tag{2.45}$$

where $i = 1, \ldots, D$. Because $\boldsymbol{\Sigma}$ is a real, symmetric matrix its eigenvalues will be real, and its eigenvectors can be chosen to form an orthonormal set, so that

Exercise 2.18

$$\mathbf{u}_i^{\mathrm{T}} \mathbf{u}_j = I_{ij} \tag{2.46}$$

where I_{ij} is the i, j element of the identity matrix and satisfies

$$I_{ij} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise.} \end{cases} \tag{2.47}$$

The covariance matrix $\boldsymbol{\Sigma}$ can be expressed as an expansion in terms of its eigenvectors in the form

Exercise 2.19

$$\boldsymbol{\Sigma} = \sum_{i=1}^{D} \lambda_i \mathbf{u}_i \mathbf{u}_i^{\mathrm{T}} \tag{2.48}$$

and similarly the inverse covariance matrix $\boldsymbol{\Sigma}^{-1}$ can be expressed as

$$\boldsymbol{\Sigma}^{-1} = \sum_{i=1}^{D} \frac{1}{\lambda_i} \mathbf{u}_i \mathbf{u}_i^{\mathrm{T}}. \tag{2.49}$$

Substituting (2.49) into (2.44), the quadratic form becomes

$$\Delta^2 = \sum_{i=1}^{D} \frac{y_i^2}{\lambda_i} \tag{2.50}$$

where we have defined

$$y_i = \mathbf{u}_i^{\mathrm{T}} (\mathbf{x} - \boldsymbol{\mu}). \tag{2.51}$$

We can interpret $\{y_i\}$ as a new coordinate system defined by the orthonormal vectors $\mathbf{u}_i$ that are shifted and rotated with respect to the original x_i coordinates. Forming the vector $\mathbf{y} = (y_1, \ldots, y_D)^{\mathrm{T}}$, we have

$$\mathbf{y} = \mathbf{U}(\mathbf{x} - \boldsymbol{\mu}) \tag{2.52}$$

Figure 2.7 The red curve shows the elliptical surface of constant probability density for a Gaussian in a two-dimensional space $\mathbf{x} = (x_1, x_2)$ on which the density is $\exp(-1/2)$ of its value at $\mathbf{x} = \boldsymbol{\mu}$. The major axes of the ellipse are defined by the eigenvectors $\mathbf{u}_i$ of the covariance matrix, with corresponding eigenvalues λ_i.

where $\mathbf{U}$ is a matrix whose rows are given by $\mathbf{u}_i^{\mathrm{T}}$. From (2.46) it follows that $\mathbf{U}$ is an *orthogonal* matrix, i.e., it satisfies $\mathbf{U}\mathbf{U}^{\mathrm{T}} = \mathbf{I}$, and hence also $\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix.

Appendix C

The quadratic form, and hence the Gaussian density, will be constant on surfaces for which (2.51) is constant. If all of the eigenvalues λ_i are positive, then these surfaces represent ellipsoids, with their centres at $\boldsymbol{\mu}$ and their axes oriented along $\mathbf{u}_i$, and with scaling factors in the directions of the axes given by $\lambda_i^{1/2}$, as illustrated in Figure 2.7.

For the Gaussian distribution to be well defined, it is necessary for all of the eigenvalues λ_i of the covariance matrix to be strictly positive, otherwise the distribution cannot be properly normalized. A matrix whose eigenvalues are strictly positive is said to be *positive definite*. In Chapter 12, we will encounter Gaussian distributions for which one or more of the eigenvalues are zero, in which case the distribution is singular and is confined to a subspace of lower dimensionality. If all of the eigenvalues are nonnegative, then the covariance matrix is said to be *positive semidefinite*.

Now consider the form of the Gaussian distribution in the new coordinate system defined by the y_i. In going from the $\mathbf{x}$ to the $\mathbf{y}$ coordinate system, we have a Jacobian matrix $\mathbf{J}$ with elements given by

$$J_{ij} = \frac{\partial x_i}{\partial y_j} = U_{ji} \tag{2.53}$$

where U_{ji} are the elements of the matrix $\mathbf{U}^{\mathrm{T}}$. Using the orthonormality property of the matrix $\mathbf{U}$, we see that the square of the determinant of the Jacobian matrix is

$$|\mathbf{J}|^2 = \left|\mathbf{U}^{\mathrm{T}}\right|^2 = \left|\mathbf{U}^{\mathrm{T}}\right| |\mathbf{U}| = \left|\mathbf{U}^{\mathrm{T}}\mathbf{U}\right| = |\mathbf{I}| = 1 \tag{2.54}$$

and hence $|\mathbf{J}| = 1$. Also, the determinant $|\boldsymbol{\Sigma}|$ of the covariance matrix can be written

as the product of its eigenvalues, and hence

$$|\boldsymbol{\Sigma}|^{1/2} = \prod_{j=1}^{D} \lambda_j^{1/2}. \tag{2.55}$$

Thus in the y_j coordinate system, the Gaussian distribution takes the form

$$p(\mathbf{y}) = p(\mathbf{x})|\mathbf{J}| = \prod_{j=1}^{D} \frac{1}{(2\pi\lambda_j)^{1/2}} \exp\left\{-\frac{y_j^2}{2\lambda_j}\right\} \tag{2.56}$$

which is the product of D independent univariate Gaussian distributions. The eigenvectors therefore define a new set of shifted and rotated coordinates with respect to which the joint probability distribution factorizes into a product of independent distributions. The integral of the distribution in the $\mathbf{y}$ coordinate system is then

$$\int p(\mathbf{y})\,\mathrm{d}\mathbf{y} = \prod_{j=1}^{D} \int_{-\infty}^{\infty} \frac{1}{(2\pi\lambda_j)^{1/2}} \exp\left\{-\frac{y_j^2}{2\lambda_j}\right\} \mathrm{d}y_j = 1 \tag{2.57}$$

where we have used the result (1.48) for the normalization of the univariate Gaussian. This confirms that the multivariate Gaussian (2.43) is indeed normalized.

We now look at the moments of the Gaussian distribution and thereby provide an interpretation of the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. The expectation of $\mathbf{x}$ under the Gaussian distribution is given by

$$\begin{aligned}\mathbb{E}[\mathbf{x}] &= \frac{1}{(2\pi)^{D/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} \int \exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right\} \mathbf{x}\,\mathrm{d}\mathbf{x} \\ &= \frac{1}{(2\pi)^{D/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} \int \exp\left\{-\frac{1}{2}\mathbf{z}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\mathbf{z}\right\} (\mathbf{z}+\boldsymbol{\mu})\,\mathrm{d}\mathbf{z} \end{aligned} \tag{2.58}$$

where we have changed variables using $\mathbf{z} = \mathbf{x} - \boldsymbol{\mu}$. We now note that the exponent is an even function of the components of $\mathbf{z}$ and, because the integrals over these are taken over the range $(-\infty, \infty)$, the term in $\mathbf{z}$ in the factor $(\mathbf{z} + \boldsymbol{\mu})$ will vanish by symmetry. Thus

$$\mathbb{E}[\mathbf{x}] = \boldsymbol{\mu} \tag{2.59}$$

and so we refer to $\boldsymbol{\mu}$ as the mean of the Gaussian distribution.

We now consider second order moments of the Gaussian. In the univariate case, we considered the second order moment given by $\mathbb{E}[x^2]$. For the multivariate Gaussian, there are D^2 second order moments given by $\mathbb{E}[x_i x_j]$, which we can group together to form the matrix $\mathbb{E}[\mathbf{x}\mathbf{x}^{\mathrm{T}}]$. This matrix can be written as

$$\begin{aligned}\mathbb{E}[\mathbf{x}\mathbf{x}^{\mathrm{T}}] &= \frac{1}{(2\pi)^{D/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} \int \exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right\} \mathbf{x}\mathbf{x}^{\mathrm{T}}\,\mathrm{d}\mathbf{x} \\ &= \frac{1}{(2\pi)^{D/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} \int \exp\left\{-\frac{1}{2}\mathbf{z}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\mathbf{z}\right\} (\mathbf{z}+\boldsymbol{\mu})(\mathbf{z}+\boldsymbol{\mu})^{\mathrm{T}}\,\mathrm{d}\mathbf{z} \end{aligned}$$

where again we have changed variables using $\mathbf{z} = \mathbf{x} - \boldsymbol{\mu}$. Note that the cross-terms involving $\boldsymbol{\mu}\mathbf{z}^{\mathrm{T}}$ and $\boldsymbol{\mu}^{\mathrm{T}}\mathbf{z}$ will again vanish by symmetry. The term $\boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}}$ is constant and can be taken outside the integral, which itself is unity because the Gaussian distribution is normalized. Consider the term involving $\mathbf{z}\mathbf{z}^{\mathrm{T}}$. Again, we can make use of the eigenvector expansion of the covariance matrix given by (2.45), together with the completeness of the set of eigenvectors, to write

$$\mathbf{z} = \sum_{j=1}^{D} y_j \mathbf{u}_j \tag{2.60}$$

where $y_j = \mathbf{u}_j^{\mathrm{T}}\mathbf{z}$, which gives

$$\begin{aligned}
&\frac{1}{(2\pi)^{D/2}}\frac{1}{|\boldsymbol{\Sigma}|^{1/2}}\int \exp\left\{-\frac{1}{2}\mathbf{z}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\mathbf{z}\right\}\mathbf{z}\mathbf{z}^{\mathrm{T}}\,\mathrm{d}\mathbf{z} \\
&= \frac{1}{(2\pi)^{D/2}}\frac{1}{|\boldsymbol{\Sigma}|^{1/2}}\sum_{i=1}^{D}\sum_{j=1}^{D}\mathbf{u}_i\mathbf{u}_j^{\mathrm{T}}\int \exp\left\{-\sum_{k=1}^{D}\frac{y_k^2}{2\lambda_k}\right\} y_i y_j\,\mathrm{d}\mathbf{y} \\
&= \sum_{i=1}^{D}\mathbf{u}_i\mathbf{u}_i^{\mathrm{T}}\lambda_i = \boldsymbol{\Sigma}
\end{aligned} \tag{2.61}$$

where we have made use of the eigenvector equation (2.45), together with the fact that the integral on the right-hand side of the middle line vanishes by symmetry unless $i = j$, and in the final line we have made use of the results (1.50) and (2.55), together with (2.48). Thus we have

$$\mathbb{E}[\mathbf{x}\mathbf{x}^{\mathrm{T}}] = \boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}} + \boldsymbol{\Sigma}. \tag{2.62}$$

For single random variables, we subtracted the mean before taking second moments in order to define a variance. Similarly, in the multivariate case it is again convenient to subtract off the mean, giving rise to the *covariance* of a random vector $\mathbf{x}$ defined by

$$\mathrm{cov}[\mathbf{x}] = \mathbb{E}\left[(\mathbf{x} - \mathbb{E}[\mathbf{x}])(\mathbf{x} - \mathbb{E}[\mathbf{x}])^{\mathrm{T}}\right]. \tag{2.63}$$

For the specific case of a Gaussian distribution, we can make use of $\mathbb{E}[\mathbf{x}] = \boldsymbol{\mu}$, together with the result (2.62), to give

$$\mathrm{cov}[\mathbf{x}] = \boldsymbol{\Sigma}. \tag{2.64}$$

Because the parameter matrix $\boldsymbol{\Sigma}$ governs the covariance of $\mathbf{x}$ under the Gaussian distribution, it is called the covariance matrix.

Although the Gaussian distribution (2.43) is widely used as a density model, it suffers from some significant limitations. Consider the number of free parameters in the distribution. A general symmetric covariance matrix $\boldsymbol{\Sigma}$ will have $D(D+1)/2$ independent parameters, and there are another D independent parameters in $\boldsymbol{\mu}$, giving $D(D+3)/2$ parameters in total. For large D, the total number of parameters

Exercise 2.21

Figure 2.8 Contours of constant probability density for a Gaussian distribution in two dimensions in which the covariance matrix is (a) of general form, (b) diagonal, in which the elliptical contours are aligned with the coordinate axes, and (c) proportional to the identity matrix, in which the contours are concentric circles.

therefore grows quadratically with D, and the computational task of manipulating and inverting large matrices can become prohibitive. One way to address this problem is to use restricted forms of the covariance matrix. If we consider covariance matrices that are *diagonal*, so that $\mathbf{\Sigma} = \mathrm{diag}(\sigma_i^2)$, we then have a total of $2D$ independent parameters in the density model. The corresponding contours of constant density are given by axis-aligned ellipsoids. We could further restrict the covariance matrix to be proportional to the identity matrix, $\mathbf{\Sigma} = \sigma^2\mathbf{I}$, known as an *isotropic* covariance, giving $D + 1$ independent parameters in the model and spherical surfaces of constant density. The three possibilities of general, diagonal, and isotropic covariance matrices are illustrated in Figure 2.8. Unfortunately, whereas such approaches limit the number of degrees of freedom in the distribution and make inversion of the covariance matrix a much faster operation, they also greatly restrict the form of the probability density and limit its ability to capture interesting correlations in the data.

A further limitation of the Gaussian distribution is that it is intrinsically unimodal (i.e., has a single maximum) and so is unable to provide a good approximation to multimodal distributions. Thus the Gaussian distribution can be both too flexible, in the sense of having too many parameters, while also being too limited in the range of distributions that it can adequately represent. We will see later that the introduction of *latent* variables, also called *hidden* variables or *unobserved* variables, allows both of these problems to be addressed. In particular, a rich family of multimodal distributions is obtained by introducing discrete latent variables leading to mixtures of Gaussians, as discussed in Section 2.3.9. Similarly, the introduction of continuous latent variables, as described in Chapter 12, leads to models in which the number of free parameters can be controlled independently of the dimensionality D of the data space while still allowing the model to capture the dominant correlations in the data set. Indeed, these two approaches can be combined and further extended to derive a very rich set of hierarchical models that can be adapted to a broad range of practical applications. For instance, the Gaussian version of the *Markov random field*, which is widely used as a probabilistic model of images, is a Gaussian distribution over the joint space of pixel intensities but rendered tractable through the imposition of considerable structure reflecting the spatial organization of the pixels. Similarly, the *linear dynamical system*, used to model time series data for applications such as tracking, is also a joint Gaussian distribution over a potentially large number of observed and latent variables and again is tractable due to the structure imposed on the distribution. A powerful framework for expressing the form and properties of

Section 8.3

Section 13.3

such complex distributions is that of probabilistic graphical models, which will form the subject of Chapter 8.

2.3.1 Conditional Gaussian distributions

An important property of the multivariate Gaussian distribution is that if two sets of variables are jointly Gaussian, then the conditional distribution of one set conditioned on the other is again Gaussian. Similarly, the marginal distribution of either set is also Gaussian.

Consider first the case of conditional distributions. Suppose $\mathbf{x}$ is a D-dimensional vector with Gaussian distribution $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ and that we partition $\mathbf{x}$ into two disjoint subsets $\mathbf{x}_a$ and $\mathbf{x}_b$. Without loss of generality, we can take $\mathbf{x}_a$ to form the first M components of $\mathbf{x}$, with $\mathbf{x}_b$ comprising the remaining $D - M$ components, so that

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{pmatrix}. \tag{2.65}$$

We also define corresponding partitions of the mean vector $\boldsymbol{\mu}$ given by

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{pmatrix} \tag{2.66}$$

and of the covariance matrix $\boldsymbol{\Sigma}$ given by

$$\boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} \end{pmatrix}. \tag{2.67}$$

Note that the symmetry $\boldsymbol{\Sigma}^{\mathrm{T}} = \boldsymbol{\Sigma}$ of the covariance matrix implies that $\boldsymbol{\Sigma}_{aa}$ and $\boldsymbol{\Sigma}_{bb}$ are symmetric, while $\boldsymbol{\Sigma}_{ba} = \boldsymbol{\Sigma}_{ab}^{\mathrm{T}}$.

In many situations, it will be convenient to work with the inverse of the covariance matrix

$$\boldsymbol{\Lambda} \equiv \boldsymbol{\Sigma}^{-1} \tag{2.68}$$

which is known as the *precision matrix*. In fact, we shall see that some properties of Gaussian distributions are most naturally expressed in terms of the covariance, whereas others take a simpler form when viewed in terms of the precision. We therefore also introduce the partitioned form of the precision matrix

$$\boldsymbol{\Lambda} = \begin{pmatrix} \boldsymbol{\Lambda}_{aa} & \boldsymbol{\Lambda}_{ab} \\ \boldsymbol{\Lambda}_{ba} & \boldsymbol{\Lambda}_{bb} \end{pmatrix} \tag{2.69}$$

corresponding to the partitioning (2.65) of the vector $\mathbf{x}$. Because the inverse of a symmetric matrix is also symmetric, we see that $\boldsymbol{\Lambda}_{aa}$ and $\boldsymbol{\Lambda}_{bb}$ are symmetric, while $\boldsymbol{\Lambda}_{ab}^{\mathrm{T}} = \boldsymbol{\Lambda}_{ba}$. It should be stressed at this point that, for instance, $\boldsymbol{\Lambda}_{aa}$ is not simply given by the inverse of $\boldsymbol{\Sigma}_{aa}$. In fact, we shall shortly examine the relation between the inverse of a partitioned matrix and the inverses of its partitions.

Exercise 2.22

Let us begin by finding an expression for the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$. From the product rule of probability, we see that this conditional distribution can be

evaluated from the joint distribution $p(\mathbf{x}) = p(\mathbf{x}_a, \mathbf{x}_b)$ simply by fixing $\mathbf{x}_b$ to the observed value and normalizing the resulting expression to obtain a valid probability distribution over $\mathbf{x}_a$. Instead of performing this normalization explicitly, we can obtain the solution more efficiently by considering the quadratic form in the exponent of the Gaussian distribution given by (2.44) and then reinstating the normalization coefficient at the end of the calculation. If we make use of the partitioning (2.65), (2.66), and (2.69), we obtain

$$\begin{aligned}
-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) = \\
-\frac{1}{2}(\mathbf{x}_a-\boldsymbol{\mu}_a)^{\mathrm{T}}\boldsymbol{\Lambda}_{aa}(\mathbf{x}_a-\boldsymbol{\mu}_a) - \frac{1}{2}(\mathbf{x}_a-\boldsymbol{\mu}_a)^{\mathrm{T}}\boldsymbol{\Lambda}_{ab}(\mathbf{x}_b-\boldsymbol{\mu}_b) \\
-\frac{1}{2}(\mathbf{x}_b-\boldsymbol{\mu}_b)^{\mathrm{T}}\boldsymbol{\Lambda}_{ba}(\mathbf{x}_a-\boldsymbol{\mu}_a) - \frac{1}{2}(\mathbf{x}_b-\boldsymbol{\mu}_b)^{\mathrm{T}}\boldsymbol{\Lambda}_{bb}(\mathbf{x}_b-\boldsymbol{\mu}_b). \qquad (2.70)
\end{aligned}$$

We see that as a function of $\mathbf{x}_a$, this is again a quadratic form, and hence the corresponding conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ will be Gaussian. Because this distribution is completely characterized by its mean and its covariance, our goal will be to identify expressions for the mean and covariance of $p(\mathbf{x}_a|\mathbf{x}_b)$ by inspection of (2.70).

This is an example of a rather common operation associated with Gaussian distributions, sometimes called 'completing the square', in which we are given a quadratic form defining the exponent terms in a Gaussian distribution, and we need to determine the corresponding mean and covariance. Such problems can be solved straightforwardly by noting that the exponent in a general Gaussian distribution $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ can be written

$$-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) = -\frac{1}{2}\mathbf{x}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\mathbf{x} + \mathbf{x}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu} + \mathrm{const} \qquad (2.71)$$

where 'const' denotes terms which are independent of $\mathbf{x}$, and we have made use of the symmetry of $\boldsymbol{\Sigma}$. Thus if we take our general quadratic form and express it in the form given by the right-hand side of (2.71), then we can immediately equate the matrix of coefficients entering the second order term in $\mathbf{x}$ to the inverse covariance matrix $\boldsymbol{\Sigma}^{-1}$ and the coefficient of the linear term in $\mathbf{x}$ to $\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}$, from which we can obtain $\boldsymbol{\mu}$.

Now let us apply this procedure to the conditional Gaussian distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ for which the quadratic form in the exponent is given by (2.70). We will denote the mean and covariance of this distribution by $\boldsymbol{\mu}_{a|b}$ and $\boldsymbol{\Sigma}_{a|b}$, respectively. Consider the functional dependence of (2.70) on $\mathbf{x}_a$ in which $\mathbf{x}_b$ is regarded as a constant. If we pick out all terms that are second order in $\mathbf{x}_a$, we have

$$-\frac{1}{2}\mathbf{x}_a^{\mathrm{T}}\boldsymbol{\Lambda}_{aa}\mathbf{x}_a \qquad (2.72)$$

from which we can immediately conclude that the covariance (inverse precision) of $p(\mathbf{x}_a|\mathbf{x}_b)$ is given by

$$\boldsymbol{\Sigma}_{a|b} = \boldsymbol{\Lambda}_{aa}^{-1}. \qquad (2.73)$$

Now consider all of the terms in (2.70) that are linear in $\mathbf{x}_a$

$$\mathbf{x}_a^{\mathrm{T}} \left\{ \boldsymbol{\Lambda}_{aa}\boldsymbol{\mu}_a - \boldsymbol{\Lambda}_{ab}(\mathbf{x}_b - \boldsymbol{\mu}_b) \right\} \tag{2.74}$$

where we have used $\boldsymbol{\Lambda}_{ba}^{\mathrm{T}} = \boldsymbol{\Lambda}_{ab}$. From our discussion of the general form (2.71), the coefficient of $\mathbf{x}_a$ in this expression must equal $\boldsymbol{\Sigma}_{a|b}^{-1}\boldsymbol{\mu}_{a|b}$ and hence

$$\begin{aligned} \boldsymbol{\mu}_{a|b} &= \boldsymbol{\Sigma}_{a|b} \left\{ \boldsymbol{\Lambda}_{aa}\boldsymbol{\mu}_a - \boldsymbol{\Lambda}_{ab}(\mathbf{x}_b - \boldsymbol{\mu}_b) \right\} \\ &= \boldsymbol{\mu}_a - \boldsymbol{\Lambda}_{aa}^{-1}\boldsymbol{\Lambda}_{ab}(\mathbf{x}_b - \boldsymbol{\mu}_b) \end{aligned} \tag{2.75}$$

where we have made use of (2.73).

The results (2.73) and (2.75) are expressed in terms of the partitioned precision matrix of the original joint distribution $p(\mathbf{x}_a, \mathbf{x}_b)$. We can also express these results in terms of the corresponding partitioned covariance matrix. To do this, we make use of the following identity for the inverse of a partitioned matrix

Exercise 2.24

$$\begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{pmatrix}^{-1} = \begin{pmatrix} \mathbf{M} & -\mathbf{M}\mathbf{B}\mathbf{D}^{-1} \\ -\mathbf{D}^{-1}\mathbf{C}\mathbf{M} & \mathbf{D}^{-1} + \mathbf{D}^{-1}\mathbf{C}\mathbf{M}\mathbf{B}\mathbf{D}^{-1} \end{pmatrix} \tag{2.76}$$

where we have defined

$$\mathbf{M} = (\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1}. \tag{2.77}$$

The quantity $\mathbf{M}^{-1}$ is known as the *Schur complement* of the matrix on the left-hand side of (2.76) with respect to the submatrix $\mathbf{D}$. Using the definition

$$\begin{pmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} \end{pmatrix}^{-1} = \begin{pmatrix} \boldsymbol{\Lambda}_{aa} & \boldsymbol{\Lambda}_{ab} \\ \boldsymbol{\Lambda}_{ba} & \boldsymbol{\Lambda}_{bb} \end{pmatrix} \tag{2.78}$$

and making use of (2.76), we have

$$\boldsymbol{\Lambda}_{aa} = (\boldsymbol{\Sigma}_{aa} - \boldsymbol{\Sigma}_{ab}\boldsymbol{\Sigma}_{bb}^{-1}\boldsymbol{\Sigma}_{ba})^{-1} \tag{2.79}$$

$$\boldsymbol{\Lambda}_{ab} = -(\boldsymbol{\Sigma}_{aa} - \boldsymbol{\Sigma}_{ab}\boldsymbol{\Sigma}_{bb}^{-1}\boldsymbol{\Sigma}_{ba})^{-1}\boldsymbol{\Sigma}_{ab}\boldsymbol{\Sigma}_{bb}^{-1}. \tag{2.80}$$

From these we obtain the following expressions for the mean and covariance of the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$

$$\boldsymbol{\mu}_{a|b} = \boldsymbol{\mu}_a + \boldsymbol{\Sigma}_{ab}\boldsymbol{\Sigma}_{bb}^{-1}(\mathbf{x}_b - \boldsymbol{\mu}_b) \tag{2.81}$$

$$\boldsymbol{\Sigma}_{a|b} = \boldsymbol{\Sigma}_{aa} - \boldsymbol{\Sigma}_{ab}\boldsymbol{\Sigma}_{bb}^{-1}\boldsymbol{\Sigma}_{ba}. \tag{2.82}$$

Comparing (2.73) and (2.82), we see that the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ takes a simpler form when expressed in terms of the partitioned precision matrix than when it is expressed in terms of the partitioned covariance matrix. Note that the mean of the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$, given by (2.81), is a linear function of $\mathbf{x}_b$ and that the covariance, given by (2.82), is independent of $\mathbf{x}_a$. This represents an example of a *linear-Gaussian* model.

Section 8.1.4

2.3.2 Marginal Gaussian distributions

We have seen that if a joint distribution $p(\mathbf{x}_a, \mathbf{x}_b)$ is Gaussian, then the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ will again be Gaussian. Now we turn to a discussion of the marginal distribution given by

$$p(\mathbf{x}_a) = \int p(\mathbf{x}_a, \mathbf{x}_b)\,\mathrm{d}\mathbf{x}_b \tag{2.83}$$

which, as we shall see, is also Gaussian. Once again, our strategy for evaluating this distribution efficiently will be to focus on the quadratic form in the exponent of the joint distribution and thereby to identify the mean and covariance of the marginal distribution $p(\mathbf{x}_a)$.

The quadratic form for the joint distribution can be expressed, using the partitioned precision matrix, in the form (2.70). Because our goal is to integrate out $\mathbf{x}_b$, this is most easily achieved by first considering the terms involving $\mathbf{x}_b$ and then completing the square in order to facilitate integration. Picking out just those terms that involve $\mathbf{x}_b$, we have

$$-\frac{1}{2}\mathbf{x}_b^{\mathrm{T}}\boldsymbol{\Lambda}_{bb}\mathbf{x}_b+\mathbf{x}_b^{T}\mathbf{m} = -\frac{1}{2}(\mathbf{x}_b-\boldsymbol{\Lambda}_{bb}^{-1}\mathbf{m})^{\mathrm{T}}\boldsymbol{\Lambda}_{bb}(\mathbf{x}_b-\boldsymbol{\Lambda}_{bb}^{-1}\mathbf{m})+\frac{1}{2}\mathbf{m}^{\mathrm{T}}\boldsymbol{\Lambda}_{bb}^{-1}\mathbf{m} \tag{2.84}$$

where we have defined

$$\mathbf{m} = \boldsymbol{\Lambda}_{bb}\boldsymbol{\mu}_b - \boldsymbol{\Lambda}_{ba}(\mathbf{x}_a - \boldsymbol{\mu}_a). \tag{2.85}$$

We see that the dependence on $\mathbf{x}_b$ has been cast into the standard quadratic form of a Gaussian distribution corresponding to the first term on the right-hand side of (2.84), plus a term that does not depend on $\mathbf{x}_b$ (but that does depend on $\mathbf{x}_a$). Thus, when we take the exponential of this quadratic form, we see that the integration over $\mathbf{x}_b$ required by (2.83) will take the form

$$\int \exp\left\{-\frac{1}{2}(\mathbf{x}_b - \boldsymbol{\Lambda}_{bb}^{-1}\mathbf{m})^{\mathrm{T}}\boldsymbol{\Lambda}_{bb}(\mathbf{x}_b - \boldsymbol{\Lambda}_{bb}^{-1}\mathbf{m})\right\}\,\mathrm{d}\mathbf{x}_b. \tag{2.86}$$

This integration is easily performed by noting that it is the integral over an unnormalized Gaussian, and so the result will be the reciprocal of the normalization coefficient. We know from the form of the normalized Gaussian given by (2.43), that this coefficient is independent of the mean and depends only on the determinant of the covariance matrix. Thus, by completing the square with respect to $\mathbf{x}_b$, we can integrate out $\mathbf{x}_b$ and the only term remaining from the contributions on the left-hand side of (2.84) that depends on $\mathbf{x}_a$ is the last term on the right-hand side of (2.84) in which $\mathbf{m}$ is given by (2.85). Combining this term with the remaining terms from

(2.70) that depend on $\mathbf{x}_a$, we obtain

$$\begin{aligned}
&\frac{1}{2}\left[\boldsymbol{\Lambda}_{bb}\boldsymbol{\mu}_b - \boldsymbol{\Lambda}_{ba}(\mathbf{x}_a - \boldsymbol{\mu}_a)\right]^{\mathrm{T}} \boldsymbol{\Lambda}_{bb}^{-1}\left[\boldsymbol{\Lambda}_{bb}\boldsymbol{\mu}_b - \boldsymbol{\Lambda}_{ba}(\mathbf{x}_a - \boldsymbol{\mu}_a)\right] \\
&\qquad -\frac{1}{2}\mathbf{x}_a^{\mathrm{T}}\boldsymbol{\Lambda}_{aa}\mathbf{x}_a + \mathbf{x}_a^{\mathrm{T}}(\boldsymbol{\Lambda}_{aa}\boldsymbol{\mu}_a + \boldsymbol{\Lambda}_{ab}\boldsymbol{\mu}_b) + \text{const} \\
&= -\frac{1}{2}\mathbf{x}_a^{\mathrm{T}}(\boldsymbol{\Lambda}_{aa} - \boldsymbol{\Lambda}_{ab}\boldsymbol{\Lambda}_{bb}^{-1}\boldsymbol{\Lambda}_{ba})\mathbf{x}_a \\
&\qquad +\mathbf{x}_a^{\mathrm{T}}(\boldsymbol{\Lambda}_{aa} - \boldsymbol{\Lambda}_{ab}\boldsymbol{\Lambda}_{bb}^{-1}\boldsymbol{\Lambda}_{ba})^{-1}\boldsymbol{\mu}_a + \text{const} \qquad (2.87)
\end{aligned}$$

where 'const' denotes quantities independent of $\mathbf{x}_a$. Again, by comparison with (2.71), we see that the covariance of the marginal distribution of $p(\mathbf{x}_a)$ is given by

$$\boldsymbol{\Sigma}_a = (\boldsymbol{\Lambda}_{aa} - \boldsymbol{\Lambda}_{ab}\boldsymbol{\Lambda}_{bb}^{-1}\boldsymbol{\Lambda}_{ba})^{-1}. \qquad (2.88)$$

Similarly, the mean is given by

$$\boldsymbol{\Sigma}_a(\boldsymbol{\Lambda}_{aa} - \boldsymbol{\Lambda}_{ab}\boldsymbol{\Lambda}_{bb}^{-1}\boldsymbol{\Lambda}_{ba})\boldsymbol{\mu}_a = \boldsymbol{\mu}_a \qquad (2.89)$$

where we have used (2.88). The covariance in (2.88) is expressed in terms of the partitioned precision matrix given by (2.69). We can rewrite this in terms of the corresponding partitioning of the covariance matrix given by (2.67), as we did for the conditional distribution. These partitioned matrices are related by

$$\begin{pmatrix} \boldsymbol{\Lambda}_{aa} & \boldsymbol{\Lambda}_{ab} \\ \boldsymbol{\Lambda}_{ba} & \boldsymbol{\Lambda}_{bb} \end{pmatrix}^{-1} = \begin{pmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} \end{pmatrix} \qquad (2.90)$$

Making use of (2.76), we then have

$$\left(\boldsymbol{\Lambda}_{aa} - \boldsymbol{\Lambda}_{ab}\boldsymbol{\Lambda}_{bb}^{-1}\boldsymbol{\Lambda}_{ba}\right)^{-1} = \boldsymbol{\Sigma}_{aa}. \qquad (2.91)$$

Thus we obtain the intuitively satisfying result that the marginal distribution $p(\mathbf{x}_a)$ has mean and covariance given by

$$\begin{aligned}
\mathbb{E}[\mathbf{x}_a] &= \boldsymbol{\mu}_a \qquad (2.92) \\
\mathrm{cov}[\mathbf{x}_a] &= \boldsymbol{\Sigma}_{aa}. \qquad (2.93)
\end{aligned}$$

We see that for a marginal distribution, the mean and covariance are most simply expressed in terms of the partitioned covariance matrix, in contrast to the conditional distribution for which the partitioned precision matrix gives rise to simpler expressions.

Our results for the marginal and conditional distributions of a partitioned Gaussian are summarized below.

Partitioned Gaussians

Given a joint Gaussian distribution $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with $\boldsymbol{\Lambda} \equiv \boldsymbol{\Sigma}^{-1}$ and

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{pmatrix}, \qquad \boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{pmatrix} \qquad (2.94)$$

Figure 2.9 The plot on the left shows the contours of a Gaussian distribution $p(x_a, x_b)$ over two variables, and the plot on the right shows the marginal distribution $p(x_a)$ (blue curve) and the conditional distribution $p(x_a|x_b)$ for $x_b = 0.7$ (red curve).

$$\boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} \end{pmatrix}, \qquad \boldsymbol{\Lambda} = \begin{pmatrix} \boldsymbol{\Lambda}_{aa} & \boldsymbol{\Lambda}_{ab} \\ \boldsymbol{\Lambda}_{ba} & \boldsymbol{\Lambda}_{bb} \end{pmatrix}. \tag{2.95}$$

Conditional distribution:

$$p(\mathbf{x}_a|\mathbf{x}_b) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_{a|b}, \boldsymbol{\Lambda}_{aa}^{-1}) \tag{2.96}$$

$$\boldsymbol{\mu}_{a|b} = \boldsymbol{\mu}_a - \boldsymbol{\Lambda}_{aa}^{-1}\boldsymbol{\Lambda}_{ab}(\mathbf{x}_b - \boldsymbol{\mu}_b). \tag{2.97}$$

Marginal distribution:

$$p(\mathbf{x}_a) = \mathcal{N}(\mathbf{x}_a|\boldsymbol{\mu}_a, \boldsymbol{\Sigma}_{aa}). \tag{2.98}$$

We illustrate the idea of conditional and marginal distributions associated with a multivariate Gaussian using an example involving two variables in Figure 2.9.

2.3.3 Bayes' theorem for Gaussian variables

In Sections 2.3.1 and 2.3.2, we considered a Gaussian $p(\mathbf{x})$ in which we partitioned the vector $\mathbf{x}$ into two subvectors $\mathbf{x} = (\mathbf{x}_a, \mathbf{x}_b)$ and then found expressions for the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ and the marginal distribution $p(\mathbf{x}_a)$. We noted that the mean of the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ was a linear function of $\mathbf{x}_b$. Here we shall suppose that we are given a Gaussian marginal distribution $p(\mathbf{x})$ and a Gaussian conditional distribution $p(\mathbf{y}|\mathbf{x})$ in which $p(\mathbf{y}|\mathbf{x})$ has a mean that is a linear function of $\mathbf{x}$, and a covariance which is independent of $\mathbf{x}$. This is an example of

a *linear Gaussian model* (Roweis and Ghahramani, 1999), which we shall study in greater generality in Section 8.1.4. We wish to find the marginal distribution $p(\mathbf{y})$ and the conditional distribution $p(\mathbf{x}|\mathbf{y})$. This is a problem that will arise frequently in subsequent chapters, and it will prove convenient to derive the general results here.

We shall take the marginal and conditional distributions to be

$$p(\mathbf{x}) = \mathcal{N}\left(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}^{-1}\right) \tag{2.99}$$

$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}\left(\mathbf{y}|\mathbf{A}\mathbf{x} + \mathbf{b}, \mathbf{L}^{-1}\right) \tag{2.100}$$

where $\boldsymbol{\mu}$, $\mathbf{A}$, and $\mathbf{b}$ are parameters governing the means, and $\boldsymbol{\Lambda}$ and $\mathbf{L}$ are precision matrices. If $\mathbf{x}$ has dimensionality M and $\mathbf{y}$ has dimensionality D, then the matrix $\mathbf{A}$ has size $D \times M$.

First we find an expression for the joint distribution over $\mathbf{x}$ and $\mathbf{y}$. To do this, we define

$$\mathbf{z} = \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \tag{2.101}$$

and then consider the log of the joint distribution

$$\begin{aligned} \ln p(\mathbf{z}) &= \ln p(\mathbf{x}) + \ln p(\mathbf{y}|\mathbf{x}) \\ &= -\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Lambda}(\mathbf{x} - \boldsymbol{\mu}) \\ &\quad -\frac{1}{2}(\mathbf{y} - \mathbf{A}\mathbf{x} - \mathbf{b})^{\mathrm{T}}\mathbf{L}(\mathbf{y} - \mathbf{A}\mathbf{x} - \mathbf{b}) + \mathrm{const} \end{aligned} \tag{2.102}$$

where 'const' denotes terms independent of $\mathbf{x}$ and $\mathbf{y}$. As before, we see that this is a quadratic function of the components of $\mathbf{z}$, and hence $p(\mathbf{z})$ is Gaussian distribution. To find the precision of this Gaussian, we consider the second order terms in (2.102), which can be written as

$$\begin{aligned} &-\frac{1}{2}\mathbf{x}^{\mathrm{T}}(\boldsymbol{\Lambda} + \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{A})\mathbf{x} - \frac{1}{2}\mathbf{y}^{\mathrm{T}}\mathbf{L}\mathbf{y} + \frac{1}{2}\mathbf{y}^{\mathrm{T}}\mathbf{L}\mathbf{A}\mathbf{x} + \frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{y} \\ &= -\frac{1}{2}\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix}^{\mathrm{T}} \begin{pmatrix} \boldsymbol{\Lambda} + \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{A} & -\mathbf{A}^{\mathrm{T}}\mathbf{L} \\ -\mathbf{L}\mathbf{A} & \mathbf{L} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = -\frac{1}{2}\mathbf{z}^{\mathrm{T}}\mathbf{R}\mathbf{z} \end{aligned} \tag{2.103}$$

and so the Gaussian distribution over $\mathbf{z}$ has precision (inverse covariance) matrix given by

$$\mathbf{R} = \begin{pmatrix} \boldsymbol{\Lambda} + \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{A} & -\mathbf{A}^{\mathrm{T}}\mathbf{L} \\ -\mathbf{L}\mathbf{A} & \mathbf{L} \end{pmatrix}. \tag{2.104}$$

The covariance matrix is found by taking the inverse of the precision, which can be done using the matrix inversion formula (2.76) to give

Exercise 2.29

$$\mathrm{cov}[\mathbf{z}] = \mathbf{R}^{-1} = \begin{pmatrix} \boldsymbol{\Lambda}^{-1} & \boldsymbol{\Lambda}^{-1}\mathbf{A}^{\mathrm{T}} \\ \mathbf{A}\boldsymbol{\Lambda}^{-1} & \mathbf{L}^{-1} + \mathbf{A}\boldsymbol{\Lambda}^{-1}\mathbf{A}^{\mathrm{T}} \end{pmatrix}. \tag{2.105}$$

Similarly, we can find the mean of the Gaussian distribution over $\mathbf{z}$ by identifying the linear terms in (2.102), which are given by

$$\mathbf{x}^{\mathrm{T}}\boldsymbol{\Lambda}\boldsymbol{\mu} - \mathbf{x}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{b} + \mathbf{y}^{\mathrm{T}}\mathbf{L}\mathbf{b} = \begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix}^{\mathrm{T}} \begin{pmatrix}\boldsymbol{\Lambda}\boldsymbol{\mu} - \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{b}\\ \mathbf{L}\mathbf{b}\end{pmatrix}. \tag{2.106}$$

Using our earlier result (2.71) obtained by completing the square over the quadratic form of a multivariate Gaussian, we find that the mean of $\mathbf{z}$ is given by

$$\mathbb{E}[\mathbf{z}] = \mathbf{R}^{-1}\begin{pmatrix}\boldsymbol{\Lambda}\boldsymbol{\mu} - \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{b}\\ \mathbf{L}\mathbf{b}\end{pmatrix}. \tag{2.107}$$

Exercise 2.30 Making use of (2.105), we then obtain

$$\mathbb{E}[\mathbf{z}] = \begin{pmatrix}\boldsymbol{\mu}\\ \mathbf{A}\boldsymbol{\mu} + \mathbf{b}\end{pmatrix}. \tag{2.108}$$

Next we find an expression for the marginal distribution $p(\mathbf{y})$ in which we have marginalized over $\mathbf{x}$. Recall that the marginal distribution over a subset of the components of a Gaussian random vector takes a particularly simple form when expressed in terms of the partitioned covariance matrix. Specifically, its mean and covariance are given by (2.92) and (2.93), respectively. Making use of (2.105) and (2.108) we see that the mean and covariance of the marginal distribution $p(\mathbf{y})$ are given by

Section 2.3

$$\mathbb{E}[\mathbf{y}] = \mathbf{A}\boldsymbol{\mu} + \mathbf{b} \tag{2.109}$$

$$\mathrm{cov}[\mathbf{y}] = \mathbf{L}^{-1} + \mathbf{A}\boldsymbol{\Lambda}^{-1}\mathbf{A}^{\mathrm{T}}. \tag{2.110}$$

A special case of this result is when $\mathbf{A} = \mathbf{I}$, in which case it reduces to the convolution of two Gaussians, for which we see that the mean of the convolution is the sum of the mean of the two Gaussians, and the covariance of the convolution is the sum of their covariances.

Finally, we seek an expression for the conditional $p(\mathbf{x}|\mathbf{y})$. Recall that the results for the conditional distribution are most easily expressed in terms of the partitioned precision matrix, using (2.73) and (2.75). Applying these results to (2.105) and (2.108) we see that the conditional distribution $p(\mathbf{x}|\mathbf{y})$ has mean and covariance given by

Section 2.3

$$\mathbb{E}[\mathbf{x}|\mathbf{y}] = (\boldsymbol{\Lambda} + \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{A})^{-1}\left\{\mathbf{A}^{\mathrm{T}}\mathbf{L}(\mathbf{y} - \mathbf{b}) + \boldsymbol{\Lambda}\boldsymbol{\mu}\right\} \tag{2.111}$$

$$\mathrm{cov}[\mathbf{x}|\mathbf{y}] = (\boldsymbol{\Lambda} + \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{A})^{-1}. \tag{2.112}$$

The evaluation of this conditional can be seen as an example of Bayes' theorem. We can interpret the distribution $p(\mathbf{x})$ as a prior distribution over $\mathbf{x}$. If the variable $\mathbf{y}$ is observed, then the conditional distribution $p(\mathbf{x}|\mathbf{y})$ represents the corresponding posterior distribution over $\mathbf{x}$. Having found the marginal and conditional distributions, we effectively expressed the joint distribution $p(\mathbf{z}) = p(\mathbf{x})p(\mathbf{y}|\mathbf{x})$ in the form $p(\mathbf{x}|\mathbf{y})p(\mathbf{y})$. These results are summarized below.

Marginal and Conditional Gaussians

Given a marginal Gaussian distribution for $\mathbf{x}$ and a conditional Gaussian distribution for $\mathbf{y}$ given $\mathbf{x}$ in the form

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}^{-1}) \tag{2.113}$$
$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}|\mathbf{A}\mathbf{x} + \mathbf{b}, \mathbf{L}^{-1}) \tag{2.114}$$

the marginal distribution of $\mathbf{y}$ and the conditional distribution of $\mathbf{x}$ given $\mathbf{y}$ are given by

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{y}|\mathbf{A}\boldsymbol{\mu} + \mathbf{b}, \mathbf{L}^{-1} + \mathbf{A}\boldsymbol{\Lambda}^{-1}\mathbf{A}^{\mathrm{T}}) \tag{2.115}$$
$$p(\mathbf{x}|\mathbf{y}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\Sigma}\{\mathbf{A}^{\mathrm{T}}\mathbf{L}(\mathbf{y} - \mathbf{b}) + \boldsymbol{\Lambda}\boldsymbol{\mu}\}, \boldsymbol{\Sigma}) \tag{2.116}$$

where

$$\boldsymbol{\Sigma} = (\boldsymbol{\Lambda} + \mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{A})^{-1}. \tag{2.117}$$

2.3.4 Maximum likelihood for the Gaussian

Given a data set $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)^{\mathrm{T}}$ in which the observations $\{\mathbf{x}_n\}$ are assumed to be drawn independently from a multivariate Gaussian distribution, we can estimate the parameters of the distribution by maximum likelihood. The log likelihood function is given by

$$\ln p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{ND}{2}\ln(2\pi) - \frac{N}{2}\ln|\boldsymbol{\Sigma}| - \frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n - \boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}_n - \boldsymbol{\mu}). \tag{2.118}$$

By simple rearrangement, we see that the likelihood function depends on the data set only through the two quantities

$$\sum_{n=1}^{N}\mathbf{x}_n, \qquad \sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^{\mathrm{T}}. \tag{2.119}$$

These are known as the *sufficient statistics* for the Gaussian distribution. Using (C.19), the derivative of the log likelihood with respect to $\boldsymbol{\mu}$ is given by

Appendix C

$$\frac{\partial}{\partial\boldsymbol{\mu}}\ln p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N}\boldsymbol{\Sigma}^{-1}(\mathbf{x}_n - \boldsymbol{\mu}) \tag{2.120}$$

and setting this derivative to zero, we obtain the solution for the maximum likelihood estimate of the mean given by

$$\boldsymbol{\mu}_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n \tag{2.121}$$

which is the mean of the observed set of data points. The maximization of (2.118) with respect to $\boldsymbol{\Sigma}$ is rather more involved. The simplest approach is to ignore the symmetry constraint and show that the resulting solution is symmetric as required. Alternative derivations of this result, which impose the symmetry and positive definiteness constraints explicitly, can be found in Magnus and Neudecker (1999). The result is as expected and takes the form

Exercise 2.34

$$\boldsymbol{\Sigma}_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})(\mathbf{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})^{\mathrm{T}} \tag{2.122}$$

which involves $\boldsymbol{\mu}_{\mathrm{ML}}$ because this is the result of a joint maximization with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. Note that the solution (2.121) for $\boldsymbol{\mu}_{\mathrm{ML}}$ does not depend on $\boldsymbol{\Sigma}_{\mathrm{ML}}$, and so we can first evaluate $\boldsymbol{\mu}_{\mathrm{ML}}$ and then use this to evaluate $\boldsymbol{\Sigma}_{\mathrm{ML}}$.

If we evaluate the expectations of the maximum likelihood solutions under the true distribution, we obtain the following results

Exercise 2.35

$$\mathbb{E}[\boldsymbol{\mu}_{\mathrm{ML}}] = \boldsymbol{\mu} \tag{2.123}$$

$$\mathbb{E}[\boldsymbol{\Sigma}_{\mathrm{ML}}] = \frac{N-1}{N}\boldsymbol{\Sigma}. \tag{2.124}$$

We see that the expectation of the maximum likelihood estimate for the mean is equal to the true mean. However, the maximum likelihood estimate for the covariance has an expectation that is less than the true value, and hence it is biased. We can correct this bias by defining a different estimator $\widetilde{\boldsymbol{\Sigma}}$ given by

$$\widetilde{\boldsymbol{\Sigma}} = \frac{1}{N-1}\sum_{n=1}^{N}(\mathbf{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})(\mathbf{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})^{\mathrm{T}}. \tag{2.125}$$

Clearly from (2.122) and (2.124), the expectation of $\widetilde{\boldsymbol{\Sigma}}$ is equal to $\boldsymbol{\Sigma}$.

2.3.5 Sequential estimation

Our discussion of the maximum likelihood solution for the parameters of a Gaussian distribution provides a convenient opportunity to give a more general discussion of the topic of sequential estimation for maximum likelihood. Sequential methods allow data points to be processed one at a time and then discarded and are important for on-line applications, and also where large data sets are involved so that batch processing of all data points at once is infeasible.

Consider the result (2.121) for the maximum likelihood estimator of the mean $\boldsymbol{\mu}_{\mathrm{ML}}$, which we will denote by $\boldsymbol{\mu}_{\mathrm{ML}}^{(N)}$ when it is based on N observations. If we

Figure 2.10 A schematic illustration of two correlated random variables z and θ, together with the regression function $f(\theta)$ given by the conditional expectation $\mathbb{E}[z|\theta]$. The Robbins-Monro algorithm provides a general sequential procedure for finding the root $\theta^\star$ of such functions.

dissect out the contribution from the final data point $\mathbf{x}_N$, we obtain

$$\begin{aligned}
\boldsymbol{\mu}_{\mathrm{ML}}^{(N)} &= \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n \\
&= \frac{1}{N}\mathbf{x}_N + \frac{1}{N}\sum_{n=1}^{N-1}\mathbf{x}_n \\
&= \frac{1}{N}\mathbf{x}_N + \frac{N-1}{N}\boldsymbol{\mu}_{\mathrm{ML}}^{(N-1)} \\
&= \boldsymbol{\mu}_{\mathrm{ML}}^{(N-1)} + \frac{1}{N}(\mathbf{x}_N - \boldsymbol{\mu}_{\mathrm{ML}}^{(N-1)}).
\end{aligned} \tag{2.126}$$

This result has a nice interpretation, as follows. After observing $N-1$ data points we have estimated $\boldsymbol{\mu}$ by $\boldsymbol{\mu}_{\mathrm{ML}}^{(N-1)}$. We now observe data point $\mathbf{x}_N$, and we obtain our revised estimate $\boldsymbol{\mu}_{\mathrm{ML}}^{(N)}$ by moving the old estimate a small amount, proportional to $1/N$, in the direction of the 'error signal' $(\mathbf{x}_N - \boldsymbol{\mu}_{\mathrm{ML}}^{(N-1)})$. Note that, as N increases, so the contribution from successive data points gets smaller.

The result (2.126) will clearly give the same answer as the batch result (2.121) because the two formulae are equivalent. However, we will not always be able to derive a sequential algorithm by this route, and so we seek a more general formulation of sequential learning, which leads us to the *Robbins-Monro* algorithm. Consider a pair of random variables θ and z governed by a joint distribution $p(z,\theta)$. The conditional expectation of z given θ defines a deterministic function $f(\theta)$ that is given by

$$f(\theta) \equiv \mathbb{E}[z|\theta] = \int zp(z|\theta)\,\mathrm{d}z \tag{2.127}$$

and is illustrated schematically in Figure 2.10. Functions defined in this way are called *regression functions*.

Our goal is to find the root $\theta^\star$ at which $f(\theta^\star) = 0$. If we had a large data set of observations of z and θ, then we could model the regression function directly and then obtain an estimate of its root. Suppose, however, that we observe values of z one at a time and we wish to find a corresponding sequential estimation scheme for $\theta^\star$. The following general procedure for solving such problems was given by

Robbins and Monro (1951). We shall assume that the conditional variance of z is finite so that

$$\mathbb{E}\left[(z-f)^2 \mid \theta\right] < \infty \tag{2.128}$$

and we shall also, without loss of generality, consider the case where $f(\theta) > 0$ for $\theta > \theta^\star$ and $f(\theta) < 0$ for $\theta < \theta^\star$, as is the case in Figure 2.10. The Robbins-Monro procedure then defines a sequence of successive estimates of the root $\theta^\star$ given by

$$\theta^{(N)} = \theta^{(N-1)} + a_{N-1} z(\theta^{(N-1)}) \tag{2.129}$$

where $z(\theta^{(N)})$ is an observed value of z when θ takes the value $\theta^{(N)}$. The coefficients $\{a_N\}$ represent a sequence of positive numbers that satisfy the conditions

$$\lim_{N\to\infty} a_N = 0 \tag{2.130}$$

$$\sum_{N=1}^{\infty} a_N = \infty \tag{2.131}$$

$$\sum_{N=1}^{\infty} a_N^2 < \infty. \tag{2.132}$$

It can then be shown (Robbins and Monro, 1951; Fukunaga, 1990) that the sequence of estimates given by (2.129) does indeed converge to the root with probability one. Note that the first condition (2.130) ensures that the successive corrections decrease in magnitude so that the process can converge to a limiting value. The second condition (2.131) is required to ensure that the algorithm does not converge short of the root, and the third condition (2.132) is needed to ensure that the accumulated noise has finite variance and hence does not spoil convergence.

Now let us consider how a general maximum likelihood problem can be solved sequentially using the Robbins-Monro algorithm. By definition, the maximum likelihood solution θ_{ML} is a stationary point of the log likelihood function and hence satisfies

$$\frac{\partial}{\partial\theta}\left\{\frac{1}{N}\sum_{n=1}^{N}\ln p(\mathbf{x}_n|\theta)\right\}\Bigg|_{\theta_{\mathrm{ML}}} = 0. \tag{2.133}$$

Exchanging the derivative and the summation, and taking the limit $N \to \infty$ we have

$$\lim_{N\to\infty}\frac{1}{N}\sum_{n=1}^{N}\frac{\partial}{\partial\theta}\ln p(x_n|\theta) = \mathbb{E}_x\left[\frac{\partial}{\partial\theta}\ln p(x|\theta)\right] \tag{2.134}$$

and so we see that finding the maximum likelihood solution corresponds to finding the root of a regression function. We can therefore apply the Robbins-Monro procedure, which now takes the form

$$\theta^{(N)} = \theta^{(N-1)} + a_{N-1}\frac{\partial}{\partial\theta^{(N-1)}}\ln p(x_N|\theta^{(N-1)}). \tag{2.135}$$

Figure 2.11 In the case of a Gaussian distribution, with θ corresponding to the mean μ, the regression function illustrated in Figure 2.10 takes the form of a straight line, as shown in red. In this case, the random variable z corresponds to the derivative of the log likelihood function and is given by $(x - \mu_{\mathrm{ML}})/\sigma^2$, and its expectation that defines the regression function is a straight line given by $(\mu - \mu_{\mathrm{ML}})/\sigma^2$. The root of the regression function corresponds to the maximum likelihood estimator μ_{ML}.

As a specific example, we consider once again the sequential estimation of the mean of a Gaussian distribution, in which case the parameter $\theta^{(N)}$ is the estimate $\mu_{\mathrm{ML}}^{(N)}$ of the mean of the Gaussian, and the random variable z is given by

$$z = \frac{\partial}{\partial \mu_{\mathrm{ML}}} \ln p(x|\mu_{\mathrm{ML}}, \sigma^2) = \frac{1}{\sigma^2}(x - \mu_{\mathrm{ML}}). \tag{2.136}$$

Thus the distribution of z is Gaussian with mean $\mu - \mu_{\mathrm{ML}}$, as illustrated in Figure 2.11. Substituting (2.136) into (2.135), we obtain the univariate form of (2.126), provided we choose the coefficients a_N to have the form $a_N = \sigma^2/N$. Note that although we have focussed on the case of a single variable, the same technique, together with the same restrictions (2.130)–(2.132) on the coefficients a_N, apply equally to the multivariate case (Blum, 1965).

2.3.6 Bayesian inference for the Gaussian

The maximum likelihood framework gave point estimates for the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. Now we develop a Bayesian treatment by introducing prior distributions over these parameters. Let us begin with a simple example in which we consider a single Gaussian random variable x. We shall suppose that the variance σ^2 is known, and we consider the task of inferring the mean μ given a set of N observations $\mathbf{X} = \{x_1, \ldots, x_N\}$. The likelihood function, that is the probability of the observed data given μ, viewed as a function of μ, is given by

$$p(\mathbf{X}|\mu) = \prod_{n=1}^{N} p(x_n|\mu) = \frac{1}{(2\pi\sigma^2)^{N/2}} \exp\left\{-\frac{1}{2\sigma^2}\sum_{n=1}^{N}(x_n - \mu)^2\right\}. \tag{2.137}$$

Again we emphasize that the likelihood function $p(\mathbf{X}|\mu)$ is not a probability distribution over μ and is not normalized.

We see that the likelihood function takes the form of the exponential of a quadratic form in μ. Thus if we choose a prior $p(\mu)$ given by a Gaussian, it will be a

conjugate distribution for this likelihood function because the corresponding posterior will be a product of two exponentials of quadratic functions of μ and hence will also be Gaussian. We therefore take our prior distribution to be

$$p(\mu) = \mathcal{N}\left(\mu|\mu_0, \sigma_0^2\right) \tag{2.138}$$

and the posterior distribution is given by

$$p(\mu|\mathbf{X}) \propto p(\mathbf{X}|\mu)p(\mu). \tag{2.139}$$

Exercise 2.38 Simple manipulation involving completing the square in the exponent shows that the posterior distribution is given by

$$p(\mu|\mathbf{X}) = \mathcal{N}\left(\mu|\mu_N, \sigma_N^2\right) \tag{2.140}$$

where

$$\mu_N = \frac{\sigma^2}{N\sigma_0^2 + \sigma^2}\mu_0 + \frac{N\sigma_0^2}{N\sigma_0^2 + \sigma^2}\mu_{\mathrm{ML}} \tag{2.141}$$

$$\frac{1}{\sigma_N^2} = \frac{1}{\sigma_0^2} + \frac{N}{\sigma^2} \tag{2.142}$$

in which μ_{ML} is the maximum likelihood solution for μ given by the sample mean

$$\mu_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N} x_n. \tag{2.143}$$

It is worth spending a moment studying the form of the posterior mean and variance. First of all, we note that the mean of the posterior distribution given by (2.141) is a compromise between the prior mean μ_0 and the maximum likelihood solution μ_{ML}. If the number of observed data points $N = 0$, then (2.141) reduces to the prior mean as expected. For $N \to \infty$, the posterior mean is given by the maximum likelihood solution. Similarly, consider the result (2.142) for the variance of the posterior distribution. We see that this is most naturally expressed in terms of the inverse variance, which is called the precision. Furthermore, the precisions are additive, so that the precision of the posterior is given by the precision of the prior plus one contribution of the data precision from each of the observed data points. As we increase the number of observed data points, the precision steadily increases, corresponding to a posterior distribution with steadily decreasing variance. With no observed data points, we have the prior variance, whereas if the number of data points $N \to \infty$, the variance σ_N^2 goes to zero and the posterior distribution becomes infinitely peaked around the maximum likelihood solution. We therefore see that the maximum likelihood result of a point estimate for μ given by (2.143) is recovered precisely from the Bayesian formalism in the limit of an infinite number of observations. Note also that for finite N, if we take the limit $\sigma_0^2 \to \infty$ in which the prior has infinite variance then the posterior mean (2.141) reduces to the maximum likelihood result, while from (2.142) the posterior variance is given by $\sigma_N^2 = \sigma^2/N$.

Figure 2.12 Illustration of Bayesian inference for the mean μ of a Gaussian distribution, in which the variance is assumed to be known. The curves show the prior distribution over μ (the curve labelled $N = 0$), which in this case is itself Gaussian, along with the posterior distribution given by (2.140) for increasing numbers N of data points. The data points are generated from a Gaussian of mean 0.8 and variance 0.1, and the prior is chosen to have mean 0. In both the prior and the likelihood function, the variance is set to the true value.

We illustrate our analysis of Bayesian inference for the mean of a Gaussian distribution in Figure 2.12. The generalization of this result to the case of a D-dimensional Gaussian random variable $\mathbf{x}$ with known covariance and unknown mean is straightforward.

Exercise 2.40

We have already seen how the maximum likelihood expression for the mean of a Gaussian can be re-cast as a sequential update formula in which the mean after observing N data points was expressed in terms of the mean after observing $N-1$ data points together with the contribution from data point $\mathbf{x}_N$. In fact, the Bayesian paradigm leads very naturally to a sequential view of the inference problem. To see this in the context of the inference of the mean of a Gaussian, we write the posterior distribution with the contribution from the final data point $\mathbf{x}_N$ separated out so that

Section 2.3.5

$$p(\boldsymbol{\mu}|D) \propto \left[p(\boldsymbol{\mu}) \prod_{n=1}^{N-1} p(\mathbf{x}_n|\boldsymbol{\mu}) \right] p(\mathbf{x}_N|\boldsymbol{\mu}). \tag{2.144}$$

The term in square brackets is (up to a normalization coefficient) just the posterior distribution after observing $N-1$ data points. We see that this can be viewed as a prior distribution, which is combined using Bayes' theorem with the likelihood function associated with data point $\mathbf{x}_N$ to arrive at the posterior distribution after observing N data points. This sequential view of Bayesian inference is very general and applies to any problem in which the observed data are assumed to be independent and identically distributed.

So far, we have assumed that the variance of the Gaussian distribution over the data is known and our goal is to infer the mean. Now let us suppose that the mean is known and we wish to infer the variance. Again, our calculations will be greatly simplified if we choose a conjugate form for the prior distribution. It turns out to be most convenient to work with the precision $\lambda \equiv 1/\sigma^2$. The likelihood function for λ takes the form

$$p(\mathbf{X}|\lambda) = \prod_{n=1}^{N} \mathcal{N}(x_n|\mu, \lambda^{-1}) \propto \lambda^{N/2} \exp\left\{ -\frac{\lambda}{2} \sum_{n=1}^{N} (x_n - \mu)^2 \right\}. \tag{2.145}$$

Figure 2.13 Plot of the gamma distribution $\mathrm{Gam}(\lambda|a, b)$ defined by (2.146) for various values of the parameters a and b.

The corresponding conjugate prior should therefore be proportional to the product of a power of λ and the exponential of a linear function of λ. This corresponds to the *gamma* distribution which is defined by

$$\mathrm{Gam}(\lambda|a, b) = \frac{1}{\Gamma(a)} b^a \lambda^{a-1} \exp(-b\lambda). \tag{2.146}$$

Here $\Gamma(a)$ is the gamma function that is defined by (1.141) and that ensures that (2.146) is correctly normalized. The gamma distribution has a finite integral if $a > 0$, and the distribution itself is finite if $a \geqslant 1$. It is plotted, for various values of a and b, in Figure 2.13. The mean and variance of the gamma distribution are given by

Exercise 2.41

Exercise 2.42

$$\mathbb{E}[\lambda] = \frac{a}{b} \tag{2.147}$$

$$\mathrm{var}[\lambda] = \frac{a}{b^2}. \tag{2.148}$$

Consider a prior distribution $\mathrm{Gam}(\lambda|a_0, b_0)$. If we multiply by the likelihood function (2.145), then we obtain a posterior distribution

$$p(\lambda|\mathbf{X}) \propto \lambda^{a_0-1} \lambda^{N/2} \exp\left\{-b_0\lambda - \frac{\lambda}{2}\sum_{n=1}^{N}(x_n - \mu)^2\right\} \tag{2.149}$$

which we recognize as a gamma distribution of the form $\mathrm{Gam}(\lambda|a_N, b_N)$ where

$$a_N = a_0 + \frac{N}{2} \tag{2.150}$$

$$b_N = b_0 + \frac{1}{2}\sum_{n=1}^{N}(x_n - \mu)^2 = b_0 + \frac{N}{2}\sigma^2_{\mathrm{ML}} \tag{2.151}$$

where σ^2_{ML} is the maximum likelihood estimator of the variance. Note that in (2.149) there is no need to keep track of the normalization constants in the prior and the likelihood function because, if required, the correct coefficient can be found at the end using the normalized form (2.146) for the gamma distribution.

From (2.150), we see that the effect of observing N data points is to increase the value of the coefficient a by $N/2$. Thus we can interpret the parameter a_0 in the prior in terms of $2a_0$ 'effective' prior observations. Similarly, from (2.151) we see that the N data points contribute $N\sigma^2_{\mathrm{ML}}/2$ to the parameter b, where σ^2_{ML} is the variance, and so we can interpret the parameter b_0 in the prior as arising from the $2a_0$ 'effective' prior observations having variance $2b_0/(2a_0) = b_0/a_0$. Recall that we made an analogous interpretation for the Dirichlet prior. These distributions are examples of the exponential family, and we shall see that the interpretation of a conjugate prior in terms of effective fictitious data points is a general one for the exponential family of distributions.

Section 2.2

Instead of working with the precision, we can consider the variance itself. The conjugate prior in this case is called the *inverse gamma* distribution, although we shall not discuss this further because we will find it more convenient to work with the precision.

Now suppose that both the mean and the precision are unknown. To find a conjugate prior, we consider the dependence of the likelihood function on μ and λ

$$\begin{aligned} p(\mathbf{X}|\mu,\lambda) &= \prod_{n=1}^{N}\left(\frac{\lambda}{2\pi}\right)^{1/2}\exp\left\{-\frac{\lambda}{2}(x_n-\mu)^2\right\} \\ &\propto \left[\lambda^{1/2}\exp\left(-\frac{\lambda\mu^2}{2}\right)\right]^N \exp\left\{\lambda\mu\sum_{n=1}^{N}x_n - \frac{\lambda}{2}\sum_{n=1}^{N}x_n^2\right\}. \end{aligned} \tag{2.152}$$

We now wish to identify a prior distribution $p(\mu,\lambda)$ that has the same functional dependence on μ and λ as the likelihood function and that should therefore take the form

$$\begin{aligned} p(\mu,\lambda) &\propto \left[\lambda^{1/2}\exp\left(-\frac{\lambda\mu^2}{2}\right)\right]^{\beta}\exp\{c\lambda\mu - d\lambda\} \\ &= \exp\left\{-\frac{\beta\lambda}{2}(\mu-c/\beta)^2\right\}\lambda^{\beta/2}\exp\left\{-\left(d-\frac{c^2}{2\beta}\right)\lambda\right\} \end{aligned} \tag{2.153}$$

where c, d, and β are constants. Since we can always write $p(\mu,\lambda) = p(\mu|\lambda)p(\lambda)$, we can find $p(\mu|\lambda)$ and $p(\lambda)$ by inspection. In particular, we see that $p(\mu|\lambda)$ is a Gaussian whose precision is a linear function of λ and that $p(\lambda)$ is a gamma distribution, so that the normalized prior takes the form

$$p(\mu,\lambda) = \mathcal{N}(\mu|\mu_0,(\beta\lambda)^{-1})\mathrm{Gam}(\lambda|a,b) \tag{2.154}$$

where we have defined new constants given by $\mu_0 = c/\beta$, $a = 1+\beta/2$, $b = d - c^2/2\beta$. The distribution (2.154) is called the *normal-gamma* or *Gaussian-gamma* distribution and is plotted in Figure 2.14. Note that this is not simply the product of an independent Gaussian prior over μ and a gamma prior over λ, because the precision of μ is a linear function of λ. Even if we chose a prior in which μ and λ were independent, the posterior distribution would exhibit a coupling between the precision of μ and the value of λ.

Figure 2.14 Contour plot of the normal-gamma distribution (2.154) for parameter values $\mu_0 = 0$, $\beta = 2$, $a = 5$ and $b = 6$.

In the case of the multivariate Gaussian distribution $\mathcal{N}\left(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}^{-1}\right)$ for a D-dimensional variable $\mathbf{x}$, the conjugate prior distribution for the mean $\boldsymbol{\mu}$, assuming the precision is known, is again a Gaussian. For known mean and unknown precision matrix $\boldsymbol{\Lambda}$, the conjugate prior is the *Wishart* distribution given by

Exercise 2.45

$$\mathcal{W}(\boldsymbol{\Lambda}|\mathbf{W}, \nu) = B|\boldsymbol{\Lambda}|^{(\nu-D-1)/2} \exp\left(-\frac{1}{2}\mathrm{Tr}(\mathbf{W}^{-1}\boldsymbol{\Lambda})\right) \tag{2.155}$$

where ν is called the number of *degrees of freedom* of the distribution, $\mathbf{W}$ is a $D \times D$ scale matrix, and $\mathrm{Tr}(\cdot)$ denotes the trace. The normalization constant B is given by

$$B(\mathbf{W}, \nu) = |\mathbf{W}|^{-\nu/2} \left(2^{\nu D/2} \pi^{D(D-1)/4} \prod_{i=1}^{D} \Gamma\left(\frac{\nu+1-i}{2}\right)\right)^{-1}. \tag{2.156}$$

Again, it is also possible to define a conjugate prior over the covariance matrix itself, rather than over the precision matrix, which leads to the *inverse Wishart* distribution, although we shall not discuss this further. If both the mean and the precision are unknown, then, following a similar line of reasoning to the univariate case, the conjugate prior is given by

$$p(\boldsymbol{\mu}, \boldsymbol{\Lambda}|\boldsymbol{\mu}_0, \beta, \mathbf{W}, \nu) = \mathcal{N}(\boldsymbol{\mu}|\boldsymbol{\mu}_0, (\beta\boldsymbol{\Lambda})^{-1})\, \mathcal{W}(\boldsymbol{\Lambda}|\mathbf{W}, \nu) \tag{2.157}$$

which is known as the *normal-Wishart* or *Gaussian-Wishart* distribution.

2.3.7 Student's t-distribution

We have seen that the conjugate prior for the precision of a Gaussian is given by a gamma distribution. If we have a univariate Gaussian $\mathcal{N}(x|\mu, \tau^{-1})$ together with a Gamma prior $\mathrm{Gam}(\tau|a, b)$ and we integrate out the precision, we obtain the marginal distribution of x in the form

Section 2.3.6

Exercise 2.46

Figure 2.15 Plot of Student's t-distribution (2.159) for $\mu = 0$ and $\lambda = 1$ for various values of ν. The limit $\nu \to \infty$ corresponds to a Gaussian distribution with mean μ and precision λ.

$$
\begin{aligned}
p(x|\mu, a, b) &= \int_0^\infty \mathcal{N}(x|\mu, \tau^{-1})\mathrm{Gam}(\tau|a, b)\,\mathrm{d}\tau \qquad (2.158)\\
&= \int_0^\infty \frac{b^a e^{(-b\tau)}\tau^{a-1}}{\Gamma(a)} \left(\frac{\tau}{2\pi}\right)^{1/2} \exp\left\{-\frac{\tau}{2}(x-\mu)^2\right\}\,\mathrm{d}\tau\\
&= \frac{b^a}{\Gamma(a)} \left(\frac{1}{2\pi}\right)^{1/2} \left[b + \frac{(x-\mu)^2}{2}\right]^{-a-1/2} \Gamma(a + 1/2)
\end{aligned}
$$

where we have made the change of variable $z = \tau[b + (x - \mu)^2/2]$. By convention we define new parameters given by $\nu = 2a$ and $\lambda = a/b$, in terms of which the distribution $p(x|\mu, a, b)$ takes the form

$$
\mathrm{St}(x|\mu, \lambda, \nu) = \frac{\Gamma(\nu/2 + 1/2)}{\Gamma(\nu/2)} \left(\frac{\lambda}{\pi\nu}\right)^{1/2} \left[1 + \frac{\lambda(x-\mu)^2}{\nu}\right]^{-\nu/2-1/2} \qquad (2.159)
$$

which is known as *Student's t-distribution*. The parameter λ is sometimes called the *precision* of the t-distribution, even though it is not in general equal to the inverse of the variance. The parameter ν is called the *degrees of freedom*, and its effect is illustrated in Figure 2.15. For the particular case of $\nu = 1$, the t-distribution reduces to the *Cauchy* distribution, while in the limit $\nu \to \infty$ the t-distribution $\mathrm{St}(x|\mu, \lambda, \nu)$ becomes a Gaussian $\mathcal{N}(x|\mu, \lambda^{-1})$ with mean μ and precision λ.

Exercise 2.47

From (2.158), we see that Student's t-distribution is obtained by adding up an infinite number of Gaussian distributions having the same mean but different precisions. This can be interpreted as an infinite mixture of Gaussians (Gaussian mixtures will be discussed in detail in Section 2.3.9. The result is a distribution that in general has longer 'tails' than a Gaussian, as was seen in Figure 2.15. This gives the t-distribution an important property called *robustness*, which means that it is much less sensitive than the Gaussian to the presence of a few data points which are *outliers*. The robustness of the t-distribution is illustrated in Figure 2.16, which compares the maximum likelihood solutions for a Gaussian and a t-distribution. Note that the maximum likelihood solution for the t-distribution can be found using the expectation-maximization (EM) algorithm. Here we see that the effect of a small number of

Exercise 12.24

Figure 2.16 Illustration of the robustness of Student's t-distribution compared to a Gaussian. (a) Histogram distribution of 30 data points drawn from a Gaussian distribution, together with the maximum likelihood fit obtained from a t-distribution (red curve) and a Gaussian (green curve, largely hidden by the red curve). Because the t-distribution contains the Gaussian as a special case it gives almost the same solution as the Gaussian. (b) The same data set but with three additional outlying data points showing how the Gaussian (green curve) is strongly distorted by the outliers, whereas the t-distribution (red curve) is relatively unaffected.

outliers is much less significant for the t-distribution than for the Gaussian. Outliers can arise in practical applications either because the process that generates the data corresponds to a distribution having a heavy tail or simply through mislabelled data. Robustness is also an important property for regression problems. Unsurprisingly, the least squares approach to regression does not exhibit robustness, because it corresponds to maximum likelihood under a (conditional) Gaussian distribution. By basing a regression model on a heavy-tailed distribution such as a t-distribution, we obtain a more robust model.

If we go back to (2.158) and substitute the alternative parameters $\nu = 2a$, $\lambda = a/b$, and $\eta = \tau b/a$, we see that the t-distribution can be written in the form

$$\mathrm{St}(x|\mu, \lambda, \nu) = \int_0^\infty \mathcal{N}\left(x|\mu, (\eta\lambda)^{-1}\right) \mathrm{Gam}(\eta|\nu/2, \nu/2)\, \mathrm{d}\eta. \tag{2.160}$$

We can then generalize this to a multivariate Gaussian $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda})$ to obtain the corresponding multivariate Student's t-distribution in the form

$$\mathrm{St}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}, \nu) = \int_0^\infty \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, (\eta\boldsymbol{\Lambda})^{-1})\mathrm{Gam}(\eta|\nu/2, \nu/2)\, \mathrm{d}\eta. \tag{2.161}$$

Exercise 2.48 Using the same technique as for the univariate case, we can evaluate this integral to give

$$\mathrm{St}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda}, \nu) = \frac{\Gamma(D/2+\nu/2)}{\Gamma(\nu/2)} \frac{|\boldsymbol{\Lambda}|^{1/2}}{(\pi\nu)^{D/2}} \left[1 + \frac{\Delta^2}{\nu}\right]^{-D/2-\nu/2} \tag{2.162}$$

where D is the dimensionality of $\mathbf{x}$, and Δ^2 is the squared Mahalanobis distance defined by

$$\Delta^2 = (\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Lambda} (\mathbf{x} - \boldsymbol{\mu}). \tag{2.163}$$

This is the multivariate form of Student's t-distribution and satisfies the following properties

Exercise 2.49

$$\mathbb{E}[\mathbf{x}] = \boldsymbol{\mu}, \qquad \text{if} \quad \nu > 1 \tag{2.164}$$

$$\mathrm{cov}[\mathbf{x}] = \frac{\nu}{(\nu-2)} \boldsymbol{\Lambda}^{-1}, \qquad \text{if} \quad \nu > 2 \tag{2.165}$$

$$\mathrm{mode}[\mathbf{x}] = \boldsymbol{\mu} \tag{2.166}$$

with corresponding results for the univariate case.

2.3.8 Periodic variables

Although Gaussian distributions are of great practical significance, both in their own right and as building blocks for more complex probabilistic models, there are situations in which they are inappropriate as density models for continuous variables. One important case, which arises in practical applications, is that of periodic variables.

An example of a periodic variable would be the wind direction at a particular geographical location. We might, for instance, measure values of wind direction on a number of days and wish to summarize this using a parametric distribution. Another example is calendar time, where we may be interested in modelling quantities that are believed to be periodic over 24 hours or over an annual cycle. Such quantities can conveniently be represented using an angular (polar) coordinate $0 \leqslant \theta < 2\pi$.

We might be tempted to treat periodic variables by choosing some direction as the origin and then applying a conventional distribution such as the Gaussian. Such an approach, however, would give results that were strongly dependent on the arbitrary choice of origin. Suppose, for instance, that we have two observations at $\theta_1 = 1°$ and $\theta_2 = 359°$, and we model them using a standard univariate Gaussian distribution. If we choose the origin at $0°$, then the sample mean of this data set will be $180°$ with standard deviation $179°$, whereas if we choose the origin at $180°$, then the mean will be $0°$ and the standard deviation will be $1°$. We clearly need to develop a special approach for the treatment of periodic variables.

Let us consider the problem of evaluating the mean of a set of observations $\mathcal{D} = \{\theta_1, \ldots, \theta_N\}$ of a periodic variable. From now on, we shall assume that θ is measured in radians. We have already seen that the simple average $(\theta_1 + \cdots + \theta_N)/N$ will be strongly coordinate dependent. To find an invariant measure of the mean, we note that the observations can be viewed as points on the unit circle and can therefore be described instead by two-dimensional unit vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ where $\|\mathbf{x}_n\| = 1$ for $n = 1, \ldots, N$, as illustrated in Figure 2.17. We can average the vectors $\{\mathbf{x}_n\}$

Figure 2.17 Illustration of the representation of values θ_n of a periodic variable as two-dimensional vectors $\mathbf{x}_n$ living on the unit circle. Also shown is the average $\overline{\mathbf{x}}$ of those vectors.

instead to give

$$\overline{\mathbf{x}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n \tag{2.167}$$

and then find the corresponding angle $\overline{\theta}$ of this average. Clearly, this definition will ensure that the location of the mean is independent of the origin of the angular coordinate. Note that $\overline{\mathbf{x}}$ will typically lie inside the unit circle. The Cartesian coordinates of the observations are given by $\mathbf{x}_n = (\cos\theta_n, \sin\theta_n)$, and we can write the Cartesian coordinates of the sample mean in the form $\overline{\mathbf{x}} = (\overline{r}\cos\overline{\theta}, \overline{r}\sin\overline{\theta})$. Substituting into (2.167) and equating the x_1 and x_2 components then gives

$$\overline{r}\cos\overline{\theta} = \frac{1}{N}\sum_{n=1}^{N}\cos\theta_n, \qquad \overline{r}\sin\overline{\theta} = \frac{1}{N}\sum_{n=1}^{N}\sin\theta_n. \tag{2.168}$$

Taking the ratio, and using the identity $\tan\theta = \sin\theta/\cos\theta$, we can solve for $\overline{\theta}$ to give

$$\overline{\theta} = \tan^{-1}\left\{\frac{\sum_n \sin\theta_n}{\sum_n \cos\theta_n}\right\}. \tag{2.169}$$

Shortly, we shall see how this result arises naturally as the maximum likelihood estimator for an appropriately defined distribution over a periodic variable.

We now consider a periodic generalization of the Gaussian called the *von Mises* distribution. Here we shall limit our attention to univariate distributions, although periodic distributions can also be found over hyperspheres of arbitrary dimension. For an extensive discussion of periodic distributions, see Mardia and Jupp (2000).

By convention, we will consider distributions $p(\theta)$ that have period 2π. Any probability density $p(\theta)$ defined over θ must not only be nonnegative and integrate

Figure 2.18 The von Mises distribution can be derived by considering a two-dimensional Gaussian of the form (2.173), whose density contours are shown in blue and conditioning on the unit circle shown in red.

to one, but it must also be periodic. Thus $p(\theta)$ must satisfy the three conditions

$$p(\theta) \geqslant 0 \tag{2.170}$$

$$\int_0^{2\pi} p(\theta)\,\mathrm{d}\theta = 1 \tag{2.171}$$

$$p(\theta + 2\pi) = p(\theta). \tag{2.172}$$

From (2.172), it follows that $p(\theta + M2\pi) = p(\theta)$ for any integer M.

We can easily obtain a Gaussian-like distribution that satisfies these three properties as follows. Consider a Gaussian distribution over two variables $\mathbf{x} = (x_1, x_2)$ having mean $\boldsymbol{\mu} = (\mu_1, \mu_2)$ and a covariance matrix $\boldsymbol{\Sigma} = \sigma^2\mathbf{I}$ where $\mathbf{I}$ is the 2×2 identity matrix, so that

$$p(x_1, x_2) = \frac{1}{2\pi\sigma^2} \exp\left\{-\frac{(x_1 - \mu_1)^2 + (x_2 - \mu_2)^2}{2\sigma^2}\right\}. \tag{2.173}$$

The contours of constant $p(\mathbf{x})$ are circles, as illustrated in Figure 2.18. Now suppose we consider the value of this distribution along a circle of fixed radius. Then by construction this distribution will be periodic, although it will not be normalized. We can determine the form of this distribution by transforming from Cartesian coordinates (x_1, x_2) to polar coordinates (r, θ) so that

$$x_1 = r\cos\theta, \qquad x_2 = r\sin\theta. \tag{2.174}$$

We also map the mean $\boldsymbol{\mu}$ into polar coordinates by writing

$$\mu_1 = r_0\cos\theta_0, \qquad \mu_2 = r_0\sin\theta_0. \tag{2.175}$$

Next we substitute these transformations into the two-dimensional Gaussian distribution (2.173), and then condition on the unit circle $r = 1$, noting that we are interested only in the dependence on θ. Focussing on the exponent in the Gaussian distribution we have

$$\begin{aligned} &-\frac{1}{2\sigma^2}\left\{(r\cos\theta - r_0\cos\theta_0)^2 + (r\sin\theta - r_0\sin\theta_0)^2\right\} \\ &= -\frac{1}{2\sigma^2}\left\{1 + r_0^2 - 2r_0\cos\theta\cos\theta_0 - 2r_0\sin\theta\sin\theta_0\right\} \\ &= \frac{r_0}{\sigma^2}\cos(\theta - \theta_0) + \text{const} \end{aligned} \tag{2.176}$$

Figure 2.19 The von Mises distribution plotted for two different parameter values, shown as a Cartesian plot on the left and as the corresponding polar plot on the right.

where 'const' denotes terms independent of θ, and we have made use of the following trigonometrical identities

Exercise 2.51

$$\cos^2 A + \sin^2 A = 1 \tag{2.177}$$

$$\cos A \cos B + \sin A \sin B = \cos(A - B). \tag{2.178}$$

If we now define $m = r_0/\sigma^2$, we obtain our final expression for the distribution of $p(\theta)$ along the unit circle $r = 1$ in the form

$$p(\theta|\theta_0, m) = \frac{1}{2\pi I_0(m)} \exp\{m \cos(\theta - \theta_0)\} \tag{2.179}$$

which is called the *von Mises* distribution, or the *circular normal*. Here the parameter θ_0 corresponds to the mean of the distribution, while m, which is known as the *concentration* parameter, is analogous to the inverse variance (precision) for the Gaussian. The normalization coefficient in (2.179) is expressed in terms of $I_0(m)$, which is the zeroth-order Bessel function of the first kind (Abramowitz and Stegun, 1965) and is defined by

$$I_0(m) = \frac{1}{2\pi} \int_0^{2\pi} \exp\{m \cos\theta\}\, \mathrm{d}\theta. \tag{2.180}$$

Exercise 2.52

For large m, the distribution becomes approximately Gaussian. The von Mises distribution is plotted in Figure 2.19, and the function $I_0(m)$ is plotted in Figure 2.20.

Now consider the maximum likelihood estimators for the parameters θ_0 and m for the von Mises distribution. The log likelihood function is given by

$$\ln p(\mathcal{D}|\theta_0, m) = -N \ln(2\pi) - N \ln I_0(m) + m \sum_{n=1}^{N} \cos(\theta_n - \theta_0). \tag{2.181}$$

Figure 2.20 Plot of the Bessel function $I_0(m)$ defined by (2.180), together with the function $A(m)$ defined by (2.186).

Setting the derivative with respect to θ_0 equal to zero gives

$$\sum_{n=1}^{N} \sin(\theta_n - \theta_0) = 0. \tag{2.182}$$

To solve for θ_0, we make use of the trigonometric identity

$$\sin(A - B) = \cos B \sin A - \cos A \sin B \tag{2.183}$$

Exercise 2.53 from which we obtain

$$\theta_0^{\mathrm{ML}} = \tan^{-1}\left\{\frac{\sum_n \sin\theta_n}{\sum_n \cos\theta_n}\right\} \tag{2.184}$$

which we recognize as the result (2.169) obtained earlier for the mean of the observations viewed in a two-dimensional Cartesian space.

Similarly, maximizing (2.181) with respect to m, and making use of $I_0'(m) = I_1(m)$ (Abramowitz and Stegun, 1965), we have

$$A(m) = \frac{1}{N}\sum_{n=1}^{N} \cos(\theta_n - \theta_0^{\mathrm{ML}}) \tag{2.185}$$

where we have substituted for the maximum likelihood solution for θ_0^{ML} (recalling that we are performing a joint optimization over θ and m), and we have defined

$$A(m) = \frac{I_1(m)}{I_0(m)}. \tag{2.186}$$

The function $A(m)$ is plotted in Figure 2.20. Making use of the trigonometric identity (2.178), we can write (2.185) in the form

$$A(m_{\mathrm{ML}}) = \left(\frac{1}{N}\sum_{n=1}^{N} \cos\theta_n\right)\cos\theta_0^{\mathrm{ML}} - \left(\frac{1}{N}\sum_{n=1}^{N} \sin\theta_n\right)\sin\theta_0^{\mathrm{ML}}. \tag{2.187}$$

Figure 2.21 Plots of the 'old faithful' data in which the blue curves show contours of constant probability density. On the left is a single Gaussian distribution which has been fitted to the data using maximum likelihood. Note that this distribution fails to capture the two clumps in the data and indeed places much of its probability mass in the central region between the clumps where the data are relatively sparse. On the right the distribution is given by a linear combination of two Gaussians which has been fitted to the data by maximum likelihood using techniques discussed Chapter 9, and which gives a better representation of the data.

The right-hand side of (2.187) is easily evaluated, and the function $A(m)$ can be inverted numerically.

For completeness, we mention briefly some alternative techniques for the construction of periodic distributions. The simplest approach is to use a histogram of observations in which the angular coordinate is divided into fixed bins. This has the virtue of simplicity and flexibility but also suffers from significant limitations, as we shall see when we discuss histogram methods in more detail in Section 2.5. Another approach starts, like the von Mises distribution, from a Gaussian distribution over a Euclidean space but now marginalizes onto the unit circle rather than conditioning (Mardia and Jupp, 2000). However, this leads to more complex forms of distribution and will not be discussed further. Finally, any valid distribution over the real axis (such as a Gaussian) can be turned into a periodic distribution by mapping successive intervals of width 2π onto the periodic variable $(0, 2\pi)$, which corresponds to 'wrapping' the real axis around unit circle. Again, the resulting distribution is more complex to handle than the von Mises distribution.

One limitation of the von Mises distribution is that it is unimodal. By forming *mixtures* of von Mises distributions, we obtain a flexible framework for modelling periodic variables that can handle multimodality. For an example of a machine learning application that makes use of von Mises distributions, see Lawrence *et al.* (2002), and for extensions to modelling conditional densities for regression problems, see Bishop and Nabney (1996).

2.3.9 Mixtures of Gaussians

While the Gaussian distribution has some important analytical properties, it suffers from significant limitations when it comes to modelling real data sets. Consider the example shown in Figure 2.21. This is known as the 'Old Faithful' data set, and comprises 272 measurements of the eruption of the Old Faithful geyser at Yellowstone National Park in the USA. Each measurement comprises the duration of

Appendix A

Figure 2.22 Example of a Gaussian mixture distribution in one dimension showing three Gaussians (each scaled by a coefficient) in blue and their sum in red.

the eruption in minutes (horizontal axis) and the time in minutes to the next eruption (vertical axis). We see that the data set forms two dominant clumps, and that a simple Gaussian distribution is unable to capture this structure, whereas a linear superposition of two Gaussians gives a better characterization of the data set.

Such superpositions, formed by taking linear combinations of more basic distributions such as Gaussians, can be formulated as probabilistic models known as *mixture distributions* (McLachlan and Basford, 1988; McLachlan and Peel, 2000). In Figure 2.22 we see that a linear combination of Gaussians can give rise to very complex densities. By using a sufficient number of Gaussians, and by adjusting their means and covariances as well as the coefficients in the linear combination, almost any continuous density can be approximated to arbitrary accuracy.

We therefore consider a superposition of K Gaussian densities of the form

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{2.188}$$

which is called a *mixture of Gaussians*. Each Gaussian density $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ is called a *component* of the mixture and has its own mean $\boldsymbol{\mu}_k$ and covariance $\boldsymbol{\Sigma}_k$. Contour and surface plots for a Gaussian mixture having 3 components are shown in Figure 2.23.

In this section we shall consider Gaussian components to illustrate the framework of mixture models. More generally, mixture models can comprise linear combinations of other distributions. For instance, in Section 9.3.3 we shall consider mixtures of Bernoulli distributions as an example of a mixture model for discrete variables.

Section 9.3.3

The parameters π_k in (2.188) are called *mixing coefficients*. If we integrate both sides of (2.188) with respect to $\mathbf{x}$, and note that both $p(\mathbf{x})$ and the individual Gaussian components are normalized, we obtain

$$\sum_{k=1}^{K} \pi_k = 1. \tag{2.189}$$

Also, the requirement that $p(\mathbf{x}) \geqslant 0$, together with $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \geqslant 0$, implies $\pi_k \geqslant 0$ for all k. Combining this with the condition (2.189) we obtain

$$0 \leqslant \pi_k \leqslant 1. \tag{2.190}$$

Figure 2.23 Illustration of a mixture of 3 Gaussians in a two-dimensional space. (a) Contours of constant density for each of the mixture components, in which the 3 components are denoted red, blue and green, and the values of the mixing coefficients are shown below each component. (b) Contours of the marginal probability density $p(\mathbf{x})$ of the mixture distribution. (c) A surface plot of the distribution $p(\mathbf{x})$.

We therefore see that the mixing coefficients satisfy the requirements to be probabilities.

From the sum and product rules, the marginal density is given by

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(k)p(\mathbf{x}|k) \tag{2.191}$$

which is equivalent to (2.188) in which we can view $\pi_k = p(k)$ as the prior probability of picking the kth component, and the density $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = p(\mathbf{x}|k)$ as the probability of $\mathbf{x}$ conditioned on k. As we shall see in later chapters, an important role is played by the posterior probabilities $p(k|\mathbf{x})$, which are also known as *responsibilities*. From Bayes' theorem these are given by

$$\begin{aligned} \gamma_k(\mathbf{x}) &\equiv p(k|\mathbf{x}) \\ &= \frac{p(k)p(\mathbf{x}|k)}{\sum_l p(l)p(\mathbf{x}|l)} \\ &= \frac{\pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_l \pi_l \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_l, \boldsymbol{\Sigma}_l)}. \end{aligned} \tag{2.192}$$

We shall discuss the probabilistic interpretation of the mixture distribution in greater detail in Chapter 9.

The form of the Gaussian mixture distribution is governed by the parameters $\boldsymbol{\pi}$, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$, where we have used the notation $\boldsymbol{\pi} \equiv \{\pi_1, \ldots, \pi_K\}$, $\boldsymbol{\mu} \equiv \{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K\}$ and $\boldsymbol{\Sigma} \equiv \{\boldsymbol{\Sigma}_1, \ldots \boldsymbol{\Sigma}_K\}$. One way to set the values of these parameters is to use maximum likelihood. From (2.188) the log of the likelihood function is given by

$$\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\} \tag{2.193}$$

where $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$. We immediately see that the situation is now much more complex than with a single Gaussian, due to the presence of the summation over k inside the logarithm. As a result, the maximum likelihood solution for the parameters no longer has a closed-form analytical solution. One approach to maximizing the likelihood function is to use iterative numerical optimization techniques (Fletcher, 1987; Nocedal and Wright, 1999; Bishop and Nabney, 2008). Alternatively we can employ a powerful framework called *expectation maximization*, which will be discussed at length in Chapter 9.

2.4. The Exponential Family

The probability distributions that we have studied so far in this chapter (with the exception of the Gaussian mixture) are specific examples of a broad class of distributions called the *exponential family* (Duda and Hart, 1973; Bernardo and Smith, 1994). Members of the exponential family have many important properties in common, and it is illuminating to discuss these properties in some generality.

The exponential family of distributions over $\mathbf{x}$, given parameters $\boldsymbol{\eta}$, is defined to be the set of distributions of the form

$$p(\mathbf{x}|\boldsymbol{\eta}) = h(\mathbf{x})g(\boldsymbol{\eta}) \exp\left\{\boldsymbol{\eta}^{\mathrm{T}}\mathbf{u}(\mathbf{x})\right\} \tag{2.194}$$

where $\mathbf{x}$ may be scalar or vector, and may be discrete or continuous. Here $\boldsymbol{\eta}$ are called the *natural parameters* of the distribution, and $\mathbf{u}(\mathbf{x})$ is some function of $\mathbf{x}$. The function $g(\boldsymbol{\eta})$ can be interpreted as the coefficient that ensures that the distribution is normalized and therefore satisfies

$$g(\boldsymbol{\eta}) \int h(\mathbf{x}) \exp\left\{\boldsymbol{\eta}^{\mathrm{T}}\mathbf{u}(\mathbf{x})\right\} \mathrm{d}\mathbf{x} = 1 \tag{2.195}$$

where the integration is replaced by summation if $\mathbf{x}$ is a discrete variable.

We begin by taking some examples of the distributions introduced earlier in the chapter and showing that they are indeed members of the exponential family. Consider first the Bernoulli distribution

$$p(x|\mu) = \mathrm{Bern}(x|\mu) = \mu^x(1-\mu)^{1-x}. \tag{2.196}$$

Expressing the right-hand side as the exponential of the logarithm, we have

$$\begin{aligned} p(x|\mu) &= \exp\{x \ln \mu + (1-x)\ln(1-\mu)\} \\ &= (1-\mu)\exp\left\{\ln\left(\frac{\mu}{1-\mu}\right)x\right\}. \end{aligned} \tag{2.197}$$

Comparison with (2.194) allows us to identify

$$\eta = \ln\left(\frac{\mu}{1-\mu}\right) \tag{2.198}$$

which we can solve for μ to give $\mu = \sigma(\eta)$, where

$$\sigma(\eta) = \frac{1}{1 + \exp(-\eta)} \tag{2.199}$$

is called the *logistic sigmoid* function. Thus we can write the Bernoulli distribution using the standard representation (2.194) in the form

$$p(x|\eta) = \sigma(-\eta)\exp(\eta x) \tag{2.200}$$

where we have used $1 - \sigma(\eta) = \sigma(-\eta)$, which is easily proved from (2.199). Comparison with (2.194) shows that

$$u(x) = x \tag{2.201}$$
$$h(x) = 1 \tag{2.202}$$
$$g(\eta) = \sigma(-\eta). \tag{2.203}$$

Next consider the multinomial distribution that, for a single observation $\mathbf{x}$, takes the form

$$p(\mathbf{x}|\boldsymbol{\mu}) = \prod_{k=1}^{M} \mu_k^{x_k} = \exp\left\{\sum_{k=1}^{M} x_k \ln \mu_k\right\} \tag{2.204}$$

where $\mathbf{x} = (x_1, \ldots, x_N)^{\mathrm{T}}$. Again, we can write this in the standard representation (2.194) so that

$$p(\mathbf{x}|\boldsymbol{\eta}) = \exp(\boldsymbol{\eta}^{\mathrm{T}}\mathbf{x}) \tag{2.205}$$

where $\eta_k = \ln \mu_k$, and we have defined $\boldsymbol{\eta} = (\eta_1, \ldots, \eta_M)^{\mathrm{T}}$. Again, comparing with (2.194) we have

$$\mathbf{u}(\mathbf{x}) = \mathbf{x} \tag{2.206}$$
$$h(\mathbf{x}) = 1 \tag{2.207}$$
$$g(\boldsymbol{\eta}) = 1. \tag{2.208}$$

Note that the parameters η_k are not independent because the parameters μ_k are subject to the constraint

$$\sum_{k=1}^{M} \mu_k = 1 \tag{2.209}$$

so that, given any $M - 1$ of the parameters μ_k, the value of the remaining parameter is fixed. In some circumstances, it will be convenient to remove this constraint by expressing the distribution in terms of only $M - 1$ parameters. This can be achieved by using the relationship (2.209) to eliminate μ_M by expressing it in terms of the remaining $\{\mu_k\}$ where $k = 1, \ldots, M - 1$, thereby leaving $M - 1$ parameters. Note that these remaining parameters are still subject to the constraints

$$0 \leqslant \mu_k \leqslant 1, \qquad \sum_{k=1}^{M-1} \mu_k \leqslant 1. \tag{2.210}$$

Making use of the constraint (2.209), the multinomial distribution in this representation then becomes

$$\begin{aligned}
&\exp\left\{\sum_{k=1}^{M} x_k \ln \mu_k\right\} \\
&= \exp\left\{\sum_{k=1}^{M-1} x_k \ln \mu_k + \left(1 - \sum_{k=1}^{M-1} x_k\right) \ln\left(1 - \sum_{k=1}^{M-1} \mu_k\right)\right\} \\
&= \exp\left\{\sum_{k=1}^{M-1} x_k \ln\left(\frac{\mu_k}{1 - \sum_{j=1}^{M-1} \mu_j}\right) + \ln\left(1 - \sum_{k=1}^{M-1} \mu_k\right)\right\}. \qquad (2.211)
\end{aligned}$$

We now identify

$$\ln\left(\frac{\mu_k}{1 - \sum_j \mu_j}\right) = \eta_k \qquad (2.212)$$

which we can solve for μ_k by first summing both sides over k and then rearranging and back-substituting to give

$$\mu_k = \frac{\exp(\eta_k)}{1 + \sum_j \exp(\eta_j)}. \qquad (2.213)$$

This is called the *softmax* function, or the *normalized exponential*. In this representation, the multinomial distribution therefore takes the form

$$p(\mathbf{x}|\boldsymbol{\eta}) = \left(1 + \sum_{k=1}^{M-1} \exp(\eta_k)\right)^{-1} \exp(\boldsymbol{\eta}^{\mathrm{T}}\mathbf{x}). \qquad (2.214)$$

This is the standard form of the exponential family, with parameter vector $\boldsymbol{\eta} = (\eta_1, \ldots, \eta_{M-1})^{\mathrm{T}}$ in which

$$\begin{aligned}
\mathbf{u}(\mathbf{x}) &= \mathbf{x} && (2.215)\\
h(\mathbf{x}) &= 1 && (2.216)\\
g(\boldsymbol{\eta}) &= \left(1 + \sum_{k=1}^{M-1} \exp(\eta_k)\right)^{-1}. && (2.217)
\end{aligned}$$

Finally, let us consider the Gaussian distribution. For the univariate Gaussian, we have

$$\begin{aligned}
p(x|\mu, \sigma^2) &= \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\} && (2.218)\\
&= \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{-\frac{1}{2\sigma^2}x^2 + \frac{\mu}{\sigma^2}x - \frac{1}{2\sigma^2}\mu^2\right\} && (2.219)
\end{aligned}$$

Exercise 2.57

which, after some simple rearrangement, can be cast in the standard exponential family form (2.194) with

$$\boldsymbol{\eta} = \begin{pmatrix} \mu/\sigma^2 \\ -1/2\sigma^2 \end{pmatrix} \tag{2.220}$$

$$\mathbf{u}(x) = \begin{pmatrix} x \\ x^2 \end{pmatrix} \tag{2.221}$$

$$h(\mathbf{x}) = (2\pi)^{-1/2} \tag{2.222}$$

$$g(\boldsymbol{\eta}) = (-2\eta_2)^{1/2} \exp\left(\frac{\eta_1^2}{4\eta_2}\right). \tag{2.223}$$

2.4.1 Maximum likelihood and sufficient statistics

Let us now consider the problem of estimating the parameter vector $\boldsymbol{\eta}$ in the general exponential family distribution (2.194) using the technique of maximum likelihood. Taking the gradient of both sides of (2.195) with respect to $\boldsymbol{\eta}$, we have

$$\begin{aligned} &\nabla g(\boldsymbol{\eta}) \int h(\mathbf{x}) \exp\left\{\boldsymbol{\eta}^{\mathrm{T}}\mathbf{u}(\mathbf{x})\right\} \mathrm{d}\mathbf{x} \\ &+ \quad g(\boldsymbol{\eta}) \int h(\mathbf{x}) \exp\left\{\boldsymbol{\eta}^{\mathrm{T}}\mathbf{u}(\mathbf{x})\right\} \mathbf{u}(\mathbf{x}) \, \mathrm{d}\mathbf{x} = 0. \end{aligned} \tag{2.224}$$

Rearranging, and making use again of (2.195) then gives

$$-\frac{1}{g(\boldsymbol{\eta})} \nabla g(\boldsymbol{\eta}) = g(\boldsymbol{\eta}) \int h(\mathbf{x}) \exp\left\{\boldsymbol{\eta}^{\mathrm{T}}\mathbf{u}(\mathbf{x})\right\} \mathbf{u}(\mathbf{x}) \, \mathrm{d}\mathbf{x} = \mathbb{E}[\mathbf{u}(\mathbf{x})] \tag{2.225}$$

where we have used (2.194). We therefore obtain the result

$$-\nabla \ln g(\boldsymbol{\eta}) = \mathbb{E}[\mathbf{u}(\mathbf{x})]. \tag{2.226}$$

Exercise 2.58

Note that the covariance of $\mathbf{u}(\mathbf{x})$ can be expressed in terms of the second derivatives of $g(\boldsymbol{\eta})$, and similarly for higher order moments. Thus, provided we can normalize a distribution from the exponential family, we can always find its moments by simple differentiation.

Now consider a set of independent identically distributed data denoted by $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, for which the likelihood function is given by

$$p(\mathbf{X}|\boldsymbol{\eta}) = \left(\prod_{n=1}^{N} h(\mathbf{x}_n)\right) g(\boldsymbol{\eta})^N \exp\left\{\boldsymbol{\eta}^{\mathrm{T}} \sum_{n=1}^{N} \mathbf{u}(\mathbf{x}_n)\right\}. \tag{2.227}$$

Setting the gradient of $\ln p(\mathbf{X}|\boldsymbol{\eta})$ with respect to $\boldsymbol{\eta}$ to zero, we get the following condition to be satisfied by the maximum likelihood estimator $\boldsymbol{\eta}_{\mathrm{ML}}$

$$-\nabla \ln g(\boldsymbol{\eta}_{\mathrm{ML}}) = \frac{1}{N} \sum_{n=1}^{N} \mathbf{u}(\mathbf{x}_n) \tag{2.228}$$

which can in principle be solved to obtain $\boldsymbol{\eta}_{\mathrm{ML}}$. We see that the solution for the maximum likelihood estimator depends on the data only through $\sum_n \mathbf{u}(\mathbf{x}_n)$, which is therefore called the *sufficient statistic* of the distribution (2.194). We do not need to store the entire data set itself but only the value of the sufficient statistic. For the Bernoulli distribution, for example, the function $\mathbf{u}(x)$ is given just by x and so we need only keep the sum of the data points $\{x_n\}$, whereas for the Gaussian $\mathbf{u}(x) = (x, x^2)^{\mathrm{T}}$, and so we should keep both the sum of $\{x_n\}$ and the sum of $\{x_n^2\}$.

If we consider the limit $N \to \infty$, then the right-hand side of (2.228) becomes $\mathbb{E}[\mathbf{u}(\mathbf{x})]$, and so by comparing with (2.226) we see that in this limit $\boldsymbol{\eta}_{\mathrm{ML}}$ will equal the true value $\boldsymbol{\eta}$.

In fact, this sufficiency property holds also for Bayesian inference, although we shall defer discussion of this until Chapter 8 when we have equipped ourselves with the tools of graphical models and can thereby gain a deeper insight into these important concepts.

2.4.2 Conjugate priors

We have already encountered the concept of a conjugate prior several times, for example in the context of the Bernoulli distribution (for which the conjugate prior is the beta distribution) or the Gaussian (where the conjugate prior for the mean is a Gaussian, and the conjugate prior for the precision is the Wishart distribution). In general, for a given probability distribution $p(\mathbf{x}|\boldsymbol{\eta})$, we can seek a prior $p(\boldsymbol{\eta})$ that is conjugate to the likelihood function, so that the posterior distribution has the same functional form as the prior. For any member of the exponential family (2.194), there exists a conjugate prior that can be written in the form

$$p(\boldsymbol{\eta}|\boldsymbol{\chi}, \nu) = f(\boldsymbol{\chi}, \nu) g(\boldsymbol{\eta})^{\nu} \exp\left\{\nu \boldsymbol{\eta}^{\mathrm{T}} \boldsymbol{\chi}\right\} \tag{2.229}$$

where $f(\boldsymbol{\chi}, \nu)$ is a normalization coefficient, and $g(\boldsymbol{\eta})$ is the same function as appears in (2.194). To see that this is indeed conjugate, let us multiply the prior (2.229) by the likelihood function (2.227) to obtain the posterior distribution, up to a normalization coefficient, in the form

$$p(\boldsymbol{\eta}|\mathbf{X}, \boldsymbol{\chi}, \nu) \propto g(\boldsymbol{\eta})^{\nu+N} \exp\left\{\boldsymbol{\eta}^{\mathrm{T}} \left(\sum_{n=1}^{N} \mathbf{u}(\mathbf{x}_n) + \nu \boldsymbol{\chi}\right)\right\}. \tag{2.230}$$

This again takes the same functional form as the prior (2.229), confirming conjugacy. Furthermore, we see that the parameter ν can be interpreted as a effective number of pseudo-observations in the prior, each of which has a value for the sufficient statistic $\mathbf{u}(\mathbf{x})$ given by $\boldsymbol{\chi}$.

2.4.3 Noninformative priors

In some applications of probabilistic inference, we may have prior knowledge that can be conveniently expressed through the prior distribution. For example, if the prior assigns zero probability to some value of variable, then the posterior distribution will necessarily also assign zero probability to that value, irrespective of

any subsequent observations of data. In many cases, however, we may have little idea of what form the distribution should take. We may then seek a form of prior distribution, called a *noninformative prior*, which is intended to have as little influence on the posterior distribution as possible (Jeffries, 1946; Box and Tao, 1973; Bernardo and Smith, 1994). This is sometimes referred to as 'letting the data speak for themselves'.

If we have a distribution $p(x|\lambda)$ governed by a parameter λ, we might be tempted to propose a prior distribution $p(\lambda) = \text{const}$ as a suitable prior. If λ is a discrete variable with K states, this simply amounts to setting the prior probability of each state to $1/K$. In the case of continuous parameters, however, there are two potential difficulties with this approach. The first is that, if the domain of λ is unbounded, this prior distribution cannot be correctly normalized because the integral over λ diverges. Such priors are called *improper*. In practice, improper priors can often be used provided the corresponding posterior distribution is *proper*, i.e., that it can be correctly normalized. For instance, if we put a uniform prior distribution over the mean of a Gaussian, then the posterior distribution for the mean, once we have observed at least one data point, will be proper.

A second difficulty arises from the transformation behaviour of a probability density under a nonlinear change of variables, given by (1.27). If a function $h(\lambda)$ is constant, and we change variables to $\lambda = \eta^2$, then $\widehat{h}(\eta) = h(\eta^2)$ will also be constant. However, if we choose the density $p_\lambda(\lambda)$ to be constant, then the density of η will be given, from (1.27), by

$$p_\eta(\eta) = p_\lambda(\lambda) \left| \frac{\mathrm{d}\lambda}{\mathrm{d}\eta} \right| = p_\lambda(\eta^2) 2\eta \propto \eta \tag{2.231}$$

and so the density over η will not be constant. This issue does not arise when we use maximum likelihood, because the likelihood function $p(x|\lambda)$ is a simple function of λ and so we are free to use any convenient parameterization. If, however, we are to choose a prior distribution that is constant, we must take care to use an appropriate representation for the parameters.

Here we consider two simple examples of noninformative priors (Berger, 1985). First of all, if a density takes the form

$$p(x|\mu) = f(x - \mu) \tag{2.232}$$

then the parameter μ is known as a *location parameter*. This family of densities exhibits *translation invariance* because if we shift x by a constant to give $\widehat{x} = x + c$, then

$$p(\widehat{x}|\widehat{\mu}) = f(\widehat{x} - \widehat{\mu}) \tag{2.233}$$

where we have defined $\widehat{\mu} = \mu + c$. Thus the density takes the same form in the new variable as in the original one, and so the density is independent of the choice of origin. We would like to choose a prior distribution that reflects this translation invariance property, and so we choose a prior that assigns equal probability mass to

an interval $A \leqslant \mu \leqslant B$ as to the shifted interval $A - c \leqslant \mu \leqslant B - c$. This implies

$$\int_A^B p(\mu)\,\mathrm{d}\mu = \int_{A-c}^{B-c} p(\mu)\,\mathrm{d}\mu = \int_A^B p(\mu - c)\,\mathrm{d}\mu \tag{2.234}$$

and because this must hold for all choices of A and B, we have

$$p(\mu - c) = p(\mu) \tag{2.235}$$

which implies that $p(\mu)$ is constant. An example of a location parameter would be the mean μ of a Gaussian distribution. As we have seen, the conjugate prior distribution for μ in this case is a Gaussian $p(\mu|\mu_0, \sigma_0^2) = \mathcal{N}(\mu|\mu_0, \sigma_0^2)$, and we obtain a noninformative prior by taking the limit $\sigma_0^2 \to \infty$. Indeed, from (2.141) and (2.142) we see that this gives a posterior distribution over μ in which the contributions from the prior vanish.

As a second example, consider a density of the form

$$p(x|\sigma) = \frac{1}{\sigma} f\left(\frac{x}{\sigma}\right) \tag{2.236}$$

Exercise 2.59

where $\sigma > 0$. Note that this will be a normalized density provided $f(x)$ is correctly normalized. The parameter σ is known as a *scale parameter*, and the density exhibits *scale invariance* because if we scale x by a constant to give $\widehat{x} = cx$, then

$$p(\widehat{x}|\widehat{\sigma}) = \frac{1}{\widehat{\sigma}} f\left(\frac{\widehat{x}}{\widehat{\sigma}}\right) \tag{2.237}$$

where we have defined $\widehat{\sigma} = c\sigma$. This transformation corresponds to a change of scale, for example from meters to kilometers if x is a length, and we would like to choose a prior distribution that reflects this scale invariance. If we consider an interval $A \leqslant \sigma \leqslant B$, and a scaled interval $A/c \leqslant \sigma \leqslant B/c$, then the prior should assign equal probability mass to these two intervals. Thus we have

$$\int_A^B p(\sigma)\,\mathrm{d}\sigma = \int_{A/c}^{B/c} p(\sigma)\,\mathrm{d}\sigma = \int_A^B p\left(\frac{1}{c}\sigma\right)\frac{1}{c}\,\mathrm{d}\sigma \tag{2.238}$$

and because this must hold for choices of A and B, we have

$$p(\sigma) = p\left(\frac{1}{c}\sigma\right)\frac{1}{c} \tag{2.239}$$

and hence $p(\sigma) \propto 1/\sigma$. Note that again this is an improper prior because the integral of the distribution over $0 \leqslant \sigma \leqslant \infty$ is divergent. It is sometimes also convenient to think of the prior distribution for a scale parameter in terms of the density of the log of the parameter. Using the transformation rule (1.27) for densities we see that $p(\ln \sigma) = \text{const}$. Thus, for this prior there is the same probability mass in the range $1 \leqslant \sigma \leqslant 10$ as in the range $10 \leqslant \sigma \leqslant 100$ and in $100 \leqslant \sigma \leqslant 1000$.

An example of a scale parameter would be the standard deviation σ of a Gaussian distribution, after we have taken account of the location parameter μ, because

$$\mathcal{N}(x|\mu, \sigma^2) \propto \sigma^{-1} \exp\left\{-(\widetilde{x}/\sigma)^2\right\} \tag{2.240}$$

where $\widetilde{x} = x - \mu$. As discussed earlier, it is often more convenient to work in terms of the precision $\lambda = 1/\sigma^2$ rather than σ itself. Using the transformation rule for densities, we see that a distribution $p(\sigma) \propto 1/\sigma$ corresponds to a distribution over λ of the form $p(\lambda) \propto 1/\lambda$. We have seen that the conjugate prior for λ was the gamma distribution $\mathrm{Gam}(\lambda|a_0, b_0)$ given by (2.146). The noninformative prior is obtained as the special case $a_0 = b_0 = 0$. Again, if we examine the results (2.150) and (2.151) for the posterior distribution of λ, we see that for $a_0 = b_0 = 0$, the posterior depends only on terms arising from the data and not from the prior.

Section 2.3

2.5. Nonparametric Methods

Throughout this chapter, we have focussed on the use of probability distributions having specific functional forms governed by a small number of parameters whose values are to be determined from a data set. This is called the *parametric* approach to density modelling. An important limitation of this approach is that the chosen density might be a poor model of the distribution that generates the data, which can result in poor predictive performance. For instance, if the process that generates the data is multimodal, then this aspect of the distribution can never be captured by a Gaussian, which is necessarily unimodal.

In this final section, we consider some *nonparametric* approaches to density estimation that make few assumptions about the form of the distribution. Here we shall focus mainly on simple frequentist methods. The reader should be aware, however, that nonparametric Bayesian methods are attracting increasing interest (Walker *et al.*, 1999; Neal, 2000; Müller and Quintana, 2004; Teh *et al.*, 2006).

Let us start with a discussion of histogram methods for density estimation, which we have already encountered in the context of marginal and conditional distributions in Figure 1.11 and in the context of the central limit theorem in Figure 2.6. Here we explore the properties of histogram density models in more detail, focussing on the case of a single continuous variable x. Standard histograms simply partition x into distinct bins of width Δ_i and then count the number n_i of observations of x falling in bin i. In order to turn this count into a normalized probability density, we simply divide by the total number N of observations and by the width Δ_i of the bins to obtain probability values for each bin given by

$$p_i = \frac{n_i}{N\Delta_i} \tag{2.241}$$

for which it is easily seen that $\int p(x)\,\mathrm{d}x = 1$. This gives a model for the density $p(x)$ that is constant over the width of each bin, and often the bins are chosen to have the same width $\Delta_i = \Delta$.

Figure 2.24 An illustration of the histogram approach to density estimation, in which a data set of 50 data points is generated from the distribution shown by the green curve. Histogram density estimates, based on (2.241), with a common bin width Δ are shown for various values of Δ.

In Figure 2.24, we show an example of histogram density estimation. Here the data is drawn from the distribution, corresponding to the green curve, which is formed from a mixture of two Gaussians. Also shown are three examples of histogram density estimates corresponding to three different choices for the bin width Δ. We see that when Δ is very small (top figure), the resulting density model is very spiky, with a lot of structure that is not present in the underlying distribution that generated the data set. Conversely, if Δ is too large (bottom figure) then the result is a model that is too smooth and that consequently fails to capture the bimodal property of the green curve. The best results are obtained for some intermediate value of Δ (middle figure). In principle, a histogram density model is also dependent on the choice of edge location for the bins, though this is typically much less significant than the value of Δ.

Note that the histogram method has the property (unlike the methods to be discussed shortly) that, once the histogram has been computed, the data set itself can be discarded, which can be advantageous if the data set is large. Also, the histogram approach is easily applied if the data points are arriving sequentially.

In practice, the histogram technique can be useful for obtaining a quick visualization of data in one or two dimensions but is unsuited to most density estimation applications. One obvious problem is that the estimated density has discontinuities that are due to the bin edges rather than any property of the underlying distribution that generated the data. Another major limitation of the histogram approach is its scaling with dimensionality. If we divide each variable in a D-dimensional space into M bins, then the total number of bins will be M^D. This exponential scaling with D is an example of the curse of dimensionality. In a space of high dimensionality, the quantity of data needed to provide meaningful estimates of local probability density would be prohibitive.

Section 1.4

The histogram approach to density estimation does, however, teach us two important lessons. First, to estimate the probability density at a particular location, we should consider the data points that lie within some local neighbourhood of that point. Note that the concept of locality requires that we assume some form of distance measure, and here we have been assuming Euclidean distance. For histograms,

this neighbourhood property was defined by the bins, and there is a natural 'smoothing' parameter describing the spatial extent of the local region, in this case the bin width. Second, the value of the smoothing parameter should be neither too large nor too small in order to obtain good results. This is reminiscent of the choice of model complexity in polynomial curve fitting discussed in Chapter 1 where the degree M of the polynomial, or alternatively the value α of the regularization parameter, was optimal for some intermediate value, neither too large nor too small. Armed with these insights, we turn now to a discussion of two widely used nonparametric techniques for density estimation, kernel estimators and nearest neighbours, which have better scaling with dimensionality than the simple histogram model.

2.5.1 Kernel density estimators

Let us suppose that observations are being drawn from some unknown probability density $p(\mathbf{x})$ in some D-dimensional space, which we shall take to be Euclidean, and we wish to estimate the value of $p(\mathbf{x})$. From our earlier discussion of locality, let us consider some small region $\mathcal{R}$ containing $\mathbf{x}$. The probability mass associated with this region is given by

$$P = \int_{\mathcal{R}} p(\mathbf{x})\,\mathrm{d}\mathbf{x}. \tag{2.242}$$

Now suppose that we have collected a data set comprising N observations drawn from $p(\mathbf{x})$. Because each data point has a probability P of falling within $\mathcal{R}$, the total number K of points that lie inside $\mathcal{R}$ will be distributed according to the binomial distribution

Section 2.1

$$\mathrm{Bin}(K|N,P) = \frac{N!}{K!(N-K)!}P^K(1-P)^{1-K}. \tag{2.243}$$

Using (2.11), we see that the mean fraction of points falling inside the region is $\mathbb{E}[K/N] = P$, and similarly using (2.12) we see that the variance around this mean is $\mathrm{var}[K/N] = P(1-P)/N$. For large N, this distribution will be sharply peaked around the mean and so

$$K \simeq NP. \tag{2.244}$$

If, however, we also assume that the region $\mathcal{R}$ is sufficiently small that the probability density $p(\mathbf{x})$ is roughly constant over the region, then we have

$$P \simeq p(\mathbf{x})V \tag{2.245}$$

where V is the volume of $\mathcal{R}$. Combining (2.244) and (2.245), we obtain our density estimate in the form

$$p(\mathbf{x}) = \frac{K}{NV}. \tag{2.246}$$

Note that the validity of (2.246) depends on two contradictory assumptions, namely that the region $\mathcal{R}$ be sufficiently small that the density is approximately constant over the region and yet sufficiently large (in relation to the value of that density) that the number K of points falling inside the region is sufficient for the binomial distribution to be sharply peaked.

We can exploit the result (2.246) in two different ways. Either we can fix K and determine the value of V from the data, which gives rise to the K-nearest-neighbour technique discussed shortly, or we can fix V and determine K from the data, giving rise to the kernel approach. It can be shown that both the K-nearest-neighbour density estimator and the kernel density estimator converge to the true probability density in the limit $N \to \infty$ provided V shrinks suitably with N, and K grows with N (Duda and Hart, 1973).

We begin by discussing the kernel method in detail, and to start with we take the region $\mathcal{R}$ to be a small hypercube centred on the point $\mathbf{x}$ at which we wish to determine the probability density. In order to count the number K of points falling within this region, it is convenient to define the following function

$$k(\mathbf{u}) = \begin{cases} 1, & |u_i| \leqslant 1/2, \qquad i = 1, \ldots, D, \\ 0, & \text{otherwise} \end{cases} \tag{2.247}$$

which represents a unit cube centred on the origin. The function $k(\mathbf{u})$ is an example of a *kernel function*, and in this context is also called a *Parzen window*. From (2.247), the quantity $k((\mathbf{x} - \mathbf{x}_n)/h)$ will be one if the data point $\mathbf{x}_n$ lies inside a cube of side h centred on $\mathbf{x}$, and zero otherwise. The total number of data points lying inside this cube will therefore be

$$K = \sum_{n=1}^{N} k\left(\frac{\mathbf{x} - \mathbf{x}_n}{h}\right). \tag{2.248}$$

Substituting this expression into (2.246) then gives the following result for the estimated density at $\mathbf{x}$

$$p(\mathbf{x}) = \frac{1}{N} \sum_{n=1}^{N} \frac{1}{h^D} k\left(\frac{\mathbf{x} - \mathbf{x}_n}{h}\right) \tag{2.249}$$

where we have used $V = h^D$ for the volume of a hypercube of side h in D dimensions. Using the symmetry of the function $k(\mathbf{u})$, we can now re-interpret this equation, not as a single cube centred on $\mathbf{x}$ but as the sum over N cubes centred on the N data points $\mathbf{x}_n$.

As it stands, the kernel density estimator (2.249) will suffer from one of the same problems that the histogram method suffered from, namely the presence of artificial discontinuities, in this case at the boundaries of the cubes. We can obtain a smoother density model if we choose a smoother kernel function, and a common choice is the Gaussian, which gives rise to the following kernel density model

$$p(\mathbf{x}) = \frac{1}{N} \sum_{n=1}^{N} \frac{1}{(2\pi h^2)^{1/2}} \exp\left\{-\frac{\|\mathbf{x} - \mathbf{x}_n\|^2}{2h^2}\right\} \tag{2.250}$$

where h represents the standard deviation of the Gaussian components. Thus our density model is obtained by placing a Gaussian over each data point and then adding up the contributions over the whole data set, and then dividing by N so that the density is correctly normalized. In Figure 2.25, we apply the model (2.250) to the data

Figure 2.25 Illustration of the kernel density model (2.250) applied to the same data set used to demonstrate the histogram approach in Figure 2.24. We see that h acts as a smoothing parameter and that if it is set too small (top panel), the result is a very noisy density model, whereas if it is set too large (bottom panel), then the bimodal nature of the underlying distribution from which the data is generated (shown by the green curve) is washed out. The best density model is obtained for some intermediate value of h (middle panel).

set used earlier to demonstrate the histogram technique. We see that, as expected, the parameter h plays the role of a smoothing parameter, and there is a trade-off between sensitivity to noise at small h and over-smoothing at large h. Again, the optimization of h is a problem in model complexity, analogous to the choice of bin width in histogram density estimation, or the degree of the polynomial used in curve fitting.

We can choose any other kernel function $k(\mathbf{u})$ in (2.249) subject to the conditions

$$k(\mathbf{u}) \geqslant 0, \tag{2.251}$$

$$\int k(\mathbf{u})\,\mathrm{d}\mathbf{u} = 1 \tag{2.252}$$

which ensure that the resulting probability distribution is nonnegative everywhere and integrates to one. The class of density model given by (2.249) is called a kernel density estimator, or *Parzen* estimator. It has a great merit that there is no computation involved in the 'training' phase because this simply requires storage of the training set. However, this is also one of its great weaknesses because the computational cost of evaluating the density grows linearly with the size of the data set.

2.5.2 Nearest-neighbour methods

One of the difficulties with the kernel approach to density estimation is that the parameter h governing the kernel width is fixed for all kernels. In regions of high data density, a large value of h may lead to over-smoothing and a washing out of structure that might otherwise be extracted from the data. However, reducing h may lead to noisy estimates elsewhere in data space where the density is smaller. Thus the optimal choice for h may be dependent on location within the data space. This issue is addressed by nearest-neighbour methods for density estimation.

We therefore return to our general result (2.246) for local density estimation, and instead of fixing V and determining the value of K from the data, we consider a fixed value of K and use the data to find an appropriate value for V. To do this, we consider a small sphere centred on the point $\mathbf{x}$ at which we wish to estimate the

Figure 2.26 Illustration of K-nearest-neighbour density estimation using the same data set as in Figures 2.25 and 2.24. We see that the parameter K governs the degree of smoothing, so that a small value of K leads to a very noisy density model (top panel), whereas a large value (bottom panel) smoothes out the bimodal nature of the true distribution (shown by the green curve) from which the data set was generated.

density $p(\mathbf{x})$, and we allow the radius of the sphere to grow until it contains precisely K data points. The estimate of the density $p(\mathbf{x})$ is then given by (2.246) with V set to the volume of the resulting sphere. This technique is known as K *nearest neighbours* and is illustrated in Figure 2.26, for various choices of the parameter K, using the same data set as used in Figure 2.24 and Figure 2.25. We see that the value of K now governs the degree of smoothing and that again there is an optimum choice for K that is neither too large nor too small. Note that the model produced by K nearest neighbours is not a true density model because the integral over all space diverges.

Exercise 2.61

We close this chapter by showing how the K-nearest-neighbour technique for density estimation can be extended to the problem of classification. To do this, we apply the K-nearest-neighbour density estimation technique to each class separately and then make use of Bayes' theorem. Let us suppose that we have a data set comprising N_k points in class $\mathcal{C}_k$ with N points in total, so that $\sum_k N_k = N$. If we wish to classify a new point $\mathbf{x}$, we draw a sphere centred on $\mathbf{x}$ containing precisely K points irrespective of their class. Suppose this sphere has volume V and contains K_k points from class $\mathcal{C}_k$. Then (2.246) provides an estimate of the density associated with each class

$$p(\mathbf{x}|\mathcal{C}_k) = \frac{K_k}{N_k V}. \tag{2.253}$$

Similarly, the unconditional density is given by

$$p(\mathbf{x}) = \frac{K}{NV} \tag{2.254}$$

while the class priors are given by

$$p(\mathcal{C}_k) = \frac{N_k}{N}. \tag{2.255}$$

We can now combine (2.253), (2.254), and (2.255) using Bayes' theorem to obtain the posterior probability of class membership

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})} = \frac{K_k}{K}. \tag{2.256}$$

Figure 2.27 (a) In the K-nearest-neighbour classifier, a new point, shown by the black diamond, is classified according to the majority class membership of the K closest training data points, in this case $K = 3$. (b) In the nearest-neighbour ($K = 1$) approach to classification, the resulting decision boundary is composed of hyperplanes that form perpendicular bisectors of pairs of points from different classes.

If we wish to minimize the probability of misclassification, this is done by assigning the test point $\mathbf{x}$ to the class having the largest posterior probability, corresponding to the largest value of K_k/K. Thus to classify a new point, we identify the K nearest points from the training data set and then assign the new point to the class having the largest number of representatives amongst this set. Ties can be broken at random. The particular case of $K = 1$ is called the *nearest-neighbour* rule, because a test point is simply assigned to the same class as the nearest point from the training set. These concepts are illustrated in Figure 2.27.

In Figure 2.28, we show the results of applying the K-nearest-neighbour algorithm to the oil flow data, introduced in Chapter 1, for various values of K. As expected, we see that K controls the degree of smoothing, so that small K produces many small regions of each class, whereas large K leads to fewer larger regions.

Figure 2.28 Plot of 200 data points from the oil data set showing values of x_6 plotted against x_7, where the red, green, and blue points correspond to the 'laminar', 'annular', and 'homogeneous' classes, respectively. Also shown are the classifications of the input space given by the K-nearest-neighbour algorithm for various values of K.

An interesting property of the nearest-neighbour ($K = 1$) classifier is that, in the limit $N \to \infty$, the error rate is never more than twice the minimum achievable error rate of an optimal classifier, i.e., one that uses the true class distributions (Cover and Hart, 1967) .

As discussed so far, both the K-nearest-neighbour method, and the kernel density estimator, require the entire training data set to be stored, leading to expensive computation if the data set is large. This effect can be offset, at the expense of some additional one-off computation, by constructing tree-based search structures to allow (approximate) near neighbours to be found efficiently without doing an exhaustive search of the data set. Nevertheless, these nonparametric methods are still severely limited. On the other hand, we have seen that simple parametric models are very restricted in terms of the forms of distribution that they can represent. We therefore need to find density models that are very flexible and yet for which the complexity of the models can be controlled independently of the size of the training set, and we shall see in subsequent chapters how to achieve this.

Exercises

2.1 ($\star$) **www** Verify that the Bernoulli distribution (2.2) satisfies the following properties

$$\sum_{x=0}^{1} p(x|\mu) = 1 \tag{2.257}$$

$$\mathbb{E}[x] = \mu \tag{2.258}$$

$$\mathrm{var}[x] = \mu(1-\mu). \tag{2.259}$$

Show that the entropy $\mathrm{H}[x]$ of a Bernoulli distributed random binary variable x is given by

$$\mathrm{H}[x] = -\mu \ln \mu - (1-\mu)\ln(1-\mu). \tag{2.260}$$

2.2 ($\star\star$) The form of the Bernoulli distribution given by (2.2) is not symmetric between the two values of x. In some situations, it will be more convenient to use an equivalent formulation for which $x \in \{-1, 1\}$, in which case the distribution can be written

$$p(x|\mu) = \left(\frac{1-\mu}{2}\right)^{(1-x)/2} \left(\frac{1+\mu}{2}\right)^{(1+x)/2} \tag{2.261}$$

where $\mu \in [-1, 1]$. Show that the distribution (2.261) is normalized, and evaluate its mean, variance, and entropy.

2.3 ($\star\star$) **www** In this exercise, we prove that the binomial distribution (2.9) is normalized. First use the definition (2.10) of the number of combinations of m identical objects chosen from a total of N to show that

$$\binom{N}{m} + \binom{N}{m-1} = \binom{N+1}{m}. \tag{2.262}$$

Use this result to prove by induction the following result

$$(1+x)^N = \sum_{m=0}^{N} \binom{N}{m} x^m \tag{2.263}$$

which is known as the *binomial theorem*, and which is valid for all real values of x. Finally, show that the binomial distribution is normalized, so that

$$\sum_{m=0}^{N} \binom{N}{m} \mu^m (1-\mu)^{N-m} = 1 \tag{2.264}$$

which can be done by first pulling out a factor $(1-\mu)^N$ out of the summation and then making use of the binomial theorem.

2.4 ($\star\star$) Show that the mean of the binomial distribution is given by (2.11). To do this, differentiate both sides of the normalization condition (2.264) with respect to μ and then rearrange to obtain an expression for the mean of n. Similarly, by differentiating (2.264) twice with respect to μ and making use of the result (2.11) for the mean of the binomial distribution prove the result (2.12) for the variance of the binomial.

2.5 ($\star\star$) **WWW** In this exercise, we prove that the beta distribution, given by (2.13), is correctly normalized, so that (2.14) holds. This is equivalent to showing that

$$\int_0^1 \mu^{a-1}(1-\mu)^{b-1}\,\mathrm{d}\mu = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}. \tag{2.265}$$

From the definition (1.141) of the gamma function, we have

$$\Gamma(a)\Gamma(b) = \int_0^\infty \exp(-x)x^{a-1}\,\mathrm{d}x \int_0^\infty \exp(-y)y^{b-1}\,\mathrm{d}y. \tag{2.266}$$

Use this expression to prove (2.265) as follows. First bring the integral over y inside the integrand of the integral over x, next make the change of variable $t = y + x$ where x is fixed, then interchange the order of the x and t integrations, and finally make the change of variable $x = t\mu$ where t is fixed.

2.6 ($\star$) Make use of the result (2.265) to show that the mean, variance, and mode of the beta distribution (2.13) are given respectively by

$$\mathbb{E}[\mu] = \frac{a}{a+b} \tag{2.267}$$

$$\mathrm{var}[\mu] = \frac{ab}{(a+b)^2(a+b+1)} \tag{2.268}$$

$$\mathrm{mode}[\mu] = \frac{a-1}{a+b-2}. \tag{2.269}$$

2.7 ($\star\star$) Consider a binomial random variable x given by (2.9), with prior distribution for μ given by the beta distribution (2.13), and suppose we have observed m occurrences of $x = 1$ and l occurrences of $x = 0$. Show that the posterior mean value of x lies between the prior mean and the maximum likelihood estimate for μ. To do this, show that the posterior mean can be written as λ times the prior mean plus $(1 - \lambda)$ times the maximum likelihood estimate, where $0 \leqslant \lambda \leqslant 1$. This illustrates the concept of the posterior distribution being a compromise between the prior distribution and the maximum likelihood solution.

2.8 ($\star$) Consider two variables x and y with joint distribution $p(x, y)$. Prove the following two results

$$\mathbb{E}[x] = \mathbb{E}_y\left[\mathbb{E}_x[x|y]\right] \tag{2.270}$$

$$\mathrm{var}[x] = \mathbb{E}_y\left[\mathrm{var}_x[x|y]\right] + \mathrm{var}_y\left[\mathbb{E}_x[x|y]\right]. \tag{2.271}$$

Here $\mathbb{E}_x[x|y]$ denotes the expectation of x under the conditional distribution $p(x|y)$, with a similar notation for the conditional variance.

2.9 ($\star\star\star$) **www** . In this exercise, we prove the normalization of the Dirichlet distribution (2.38) using induction. We have already shown in Exercise 2.5 that the beta distribution, which is a special case of the Dirichlet for $M = 2$, is normalized. We now assume that the Dirichlet distribution is normalized for $M - 1$ variables and prove that it is normalized for M variables. To do this, consider the Dirichlet distribution over M variables, and take account of the constraint $\sum_{k=1}^{M} \mu_k = 1$ by eliminating μ_M, so that the Dirichlet is written

$$p_M(\mu_1, \ldots, \mu_{M-1}) = C_M \prod_{k=1}^{M-1} \mu_k^{\alpha_k - 1} \left(1 - \sum_{j=1}^{M-1} \mu_j\right)^{\alpha_M - 1} \tag{2.272}$$

and our goal is to find an expression for C_M. To do this, integrate over μ_{M-1}, taking care over the limits of integration, and then make a change of variable so that this integral has limits 0 and 1. By assuming the correct result for C_{M-1} and making use of (2.265), derive the expression for C_M.

2.10 ($\star\star$) Using the property $\Gamma(x + 1) = x\Gamma(x)$ of the gamma function, derive the following results for the mean, variance, and covariance of the Dirichlet distribution given by (2.38)

$$\mathbb{E}[\mu_j] = \frac{\alpha_j}{\alpha_0} \tag{2.273}$$

$$\mathrm{var}[\mu_j] = \frac{\alpha_j(\alpha_0 - \alpha_j)}{\alpha_0^2(\alpha_0 + 1)} \tag{2.274}$$

$$\mathrm{cov}[\mu_j \mu_l] = -\frac{\alpha_j \alpha_l}{\alpha_0^2(\alpha_0 + 1)}, \qquad j \neq l \tag{2.275}$$

where α_0 is defined by (2.39).

2.11 ($\star$) **www** By expressing the expectation of $\ln \mu_j$ under the Dirichlet distribution (2.38) as a derivative with respect to α_j, show that

$$\mathbb{E}[\ln \mu_j] = \psi(\alpha_j) - \psi(\alpha_0) \tag{2.276}$$

where α_0 is given by (2.39) and

$$\psi(a) \equiv \frac{d}{da} \ln \Gamma(a) \tag{2.277}$$

is the *digamma* function.

2.12 ($\star$) The uniform distribution for a continuous variable x is defined by

$$\mathrm{U}(x|a, b) = \frac{1}{b-a}, \qquad a \leqslant x \leqslant b. \tag{2.278}$$

Verify that this distribution is normalized, and find expressions for its mean and variance.

2.13 ($\star\star$) Evaluate the Kullback-Leibler divergence (1.113) between two Gaussians $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ and $q(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\mathbf{m}, \mathbf{L})$.

2.14 ($\star\star$) **www** This exercise demonstrates that the multivariate distribution with maximum entropy, for a given covariance, is a Gaussian. The entropy of a distribution $p(\mathbf{x})$ is given by

$$\mathrm{H}[\mathbf{x}] = -\int p(\mathbf{x}) \ln p(\mathbf{x}) \, \mathrm{d}\mathbf{x}. \tag{2.279}$$

We wish to maximize $\mathrm{H}[\mathbf{x}]$ over all distributions $p(\mathbf{x})$ subject to the constraints that $p(\mathbf{x})$ be normalized and that it have a specific mean and covariance, so that

$$\int p(\mathbf{x}) \, \mathrm{d}\mathbf{x} = 1 \tag{2.280}$$

$$\int p(\mathbf{x}) \mathbf{x} \, \mathrm{d}\mathbf{x} = \boldsymbol{\mu} \tag{2.281}$$

$$\int p(\mathbf{x}) (\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}} \, \mathrm{d}\mathbf{x} = \boldsymbol{\Sigma}. \tag{2.282}$$

By performing a variational maximization of (2.279) and using Lagrange multipliers to enforce the constraints (2.280), (2.281), and (2.282), show that the maximum likelihood distribution is given by the Gaussian (2.43).

2.15 ($\star\star$) Show that the entropy of the multivariate Gaussian $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is given by

$$\mathrm{H}[\mathbf{x}] = \frac{1}{2} \ln |\boldsymbol{\Sigma}| + \frac{D}{2} (1 + \ln(2\pi)) \tag{2.283}$$

where D is the dimensionality of $\mathbf{x}$.

2.16 ($\star\star\star$) **www** Consider two random variables x_1 and x_2 having Gaussian distributions with means μ_1, μ_2 and precisions τ_1, τ_2 respectively. Derive an expression for the differential entropy of the variable $x = x_1 + x_2$. To do this, first find the distribution of x by using the relation

$$p(x) = \int_{-\infty}^{\infty} p(x|x_2)p(x_2)\,\mathrm{d}x_2 \tag{2.284}$$

and completing the square in the exponent. Then observe that this represents the convolution of two Gaussian distributions, which itself will be Gaussian, and finally make use of the result (1.110) for the entropy of the univariate Gaussian.

2.17 ($\star$) **www** Consider the multivariate Gaussian distribution given by (2.43). By writing the precision matrix (inverse covariance matrix) $\boldsymbol{\Sigma}^{-1}$ as the sum of a symmetric and an anti-symmetric matrix, show that the anti-symmetric term does not appear in the exponent of the Gaussian, and hence that the precision matrix may be taken to be symmetric without loss of generality. Because the inverse of a symmetric matrix is also symmetric (see Exercise 2.22), it follows that the covariance matrix may also be chosen to be symmetric without loss of generality.

2.18 ($\star\star\star$) Consider a real, symmetric matrix $\boldsymbol{\Sigma}$ whose eigenvalue equation is given by (2.45). By taking the complex conjugate of this equation and subtracting the original equation, and then forming the inner product with eigenvector $\mathbf{u}_i$, show that the eigenvalues λ_i are real. Similarly, use the symmetry property of $\boldsymbol{\Sigma}$ to show that two eigenvectors $\mathbf{u}_i$ and $\mathbf{u}_j$ will be orthogonal provided $\lambda_j \neq \lambda_i$. Finally, show that without loss of generality, the set of eigenvectors can be chosen to be orthonormal, so that they satisfy (2.46), even if some of the eigenvalues are zero.

2.19 ($\star\star$) Show that a real, symmetric matrix $\boldsymbol{\Sigma}$ having the eigenvector equation (2.45) can be expressed as an expansion in the eigenvectors, with coefficients given by the eigenvalues, of the form (2.48). Similarly, show that the inverse matrix $\boldsymbol{\Sigma}^{-1}$ has a representation of the form (2.49).

2.20 ($\star\star$) **www** A positive definite matrix $\boldsymbol{\Sigma}$ can be defined as one for which the quadratic form

$$\mathbf{a}^{\mathrm{T}}\boldsymbol{\Sigma}\mathbf{a} \tag{2.285}$$

is positive for any real value of the vector $\mathbf{a}$. Show that a necessary and sufficient condition for $\boldsymbol{\Sigma}$ to be positive definite is that all of the eigenvalues λ_i of $\boldsymbol{\Sigma}$, defined by (2.45), are positive.

2.21 ($\star$) Show that a real, symmetric matrix of size $D \times D$ has $D(D+1)/2$ independent parameters.

2.22 ($\star$) **www** Show that the inverse of a symmetric matrix is itself symmetric.

2.23 ($\star\star$) By diagonalizing the coordinate system using the eigenvector expansion (2.45), show that the volume contained within the hyperellipsoid corresponding to a constant

Mahalanobis distance Δ is given by

$$V_D|\boldsymbol{\Sigma}|^{1/2}\Delta^D \tag{2.286}$$

where V_D is the volume of the unit sphere in D dimensions, and the Mahalanobis distance is defined by (2.44).

2.24 ($\star\star$) **WWW** Prove the identity (2.76) by multiplying both sides by the matrix

$$\begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{pmatrix} \tag{2.287}$$

and making use of the definition (2.77).

2.25 ($\star\star$) In Sections 2.3.1 and 2.3.2, we considered the conditional and marginal distributions for a multivariate Gaussian. More generally, we can consider a partitioning of the components of $\mathbf{x}$ into three groups $\mathbf{x}_a$, $\mathbf{x}_b$, and $\mathbf{x}_c$, with a corresponding partitioning of the mean vector $\boldsymbol{\mu}$ and of the covariance matrix $\boldsymbol{\Sigma}$ in the form

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \\ \boldsymbol{\mu}_c \end{pmatrix}, \qquad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} & \boldsymbol{\Sigma}_{ac} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} & \boldsymbol{\Sigma}_{bc} \\ \boldsymbol{\Sigma}_{ca} & \boldsymbol{\Sigma}_{cb} & \boldsymbol{\Sigma}_{cc} \end{pmatrix}. \tag{2.288}$$

By making use of the results of Section 2.3, find an expression for the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ in which $\mathbf{x}_c$ has been marginalized out.

2.26 ($\star\star$) A very useful result from linear algebra is the *Woodbury* matrix inversion formula given by

$$(\mathbf{A} + \mathbf{BCD})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{B}(\mathbf{C}^{-1} + \mathbf{DA}^{-1}\mathbf{B})^{-1}\mathbf{DA}^{-1}. \tag{2.289}$$

By multiplying both sides by $(\mathbf{A} + \mathbf{BCD})$ prove the correctness of this result.

2.27 ($\star$) Let $\mathbf{x}$ and $\mathbf{z}$ be two independent random vectors, so that $p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x})p(\mathbf{z})$. Show that the mean of their sum $\mathbf{y} = \mathbf{x} + \mathbf{z}$ is given by the sum of the means of each of the variable separately. Similarly, show that the covariance matrix of $\mathbf{y}$ is given by the sum of the covariance matrices of $\mathbf{x}$ and $\mathbf{z}$. Confirm that this result agrees with that of Exercise 1.10.

2.28 ($\star\star\star$) **WWW** Consider a joint distribution over the variable

$$\mathbf{z} = \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \tag{2.290}$$

whose mean and covariance are given by (2.108) and (2.105) respectively. By making use of the results (2.92) and (2.93) show that the marginal distribution $p(\mathbf{x})$ is given (2.99). Similarly, by making use of the results (2.81) and (2.82) show that the conditional distribution $p(\mathbf{y}|\mathbf{x})$ is given by (2.100).

2.29 ($\star\star$) Using the partitioned matrix inversion formula (2.76), show that the inverse of the precision matrix (2.104) is given by the covariance matrix (2.105).

2.30 ($\star$) By starting from (2.107) and making use of the result (2.105), verify the result (2.108).

2.31 ($\star\star$) Consider two multidimensional random vectors $\mathbf{x}$ and $\mathbf{z}$ having Gaussian distributions $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_{\mathbf{x}}, \boldsymbol{\Sigma}_{\mathbf{x}})$ and $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}_{\mathbf{z}}, \boldsymbol{\Sigma}_{\mathbf{z}})$ respectively, together with their sum $\mathbf{y} = \mathbf{x}+\mathbf{z}$. Use the results (2.109) and (2.110) to find an expression for the marginal distribution $p(\mathbf{y})$ by considering the linear-Gaussian model comprising the product of the marginal distribution $p(\mathbf{x})$ and the conditional distribution $p(\mathbf{y}|\mathbf{x})$.

2.32 ($\star\star\star$) **www** This exercise and the next provide practice at manipulating the quadratic forms that arise in linear-Gaussian models, as well as giving an independent check of results derived in the main text. Consider a joint distribution $p(\mathbf{x}, \mathbf{y})$ defined by the marginal and conditional distributions given by (2.99) and (2.100). By examining the quadratic form in the exponent of the joint distribution, and using the technique of 'completing the square' discussed in Section 2.3, find expressions for the mean and covariance of the marginal distribution $p(\mathbf{y})$ in which the variable $\mathbf{x}$ has been integrated out. To do this, make use of the Woodbury matrix inversion formula (2.289). Verify that these results agree with (2.109) and (2.110) obtained using the results of Chapter 2.

2.33 ($\star\star\star$) Consider the same joint distribution as in Exercise 2.32, but now use the technique of completing the square to find expressions for the mean and covariance of the conditional distribution $p(\mathbf{x}|\mathbf{y})$. Again, verify that these agree with the corresponding expressions (2.111) and (2.112).

2.34 ($\star\star$) **www** To find the maximum likelihood solution for the covariance matrix of a multivariate Gaussian, we need to maximize the log likelihood function (2.118) with respect to $\boldsymbol{\Sigma}$, noting that the covariance matrix must be symmetric and positive definite. Here we proceed by ignoring these constraints and doing a straightforward maximization. Using the results (C.21), (C.26), and (C.28) from Appendix C, show that the covariance matrix $\boldsymbol{\Sigma}$ that maximizes the log likelihood function (2.118) is given by the sample covariance (2.122). We note that the final result is necessarily symmetric and positive definite (provided the sample covariance is nonsingular).

2.35 ($\star\star$) Use the result (2.59) to prove (2.62). Now, using the results (2.59), and (2.62), show that

$$\mathbb{E}[\mathbf{x}_n \mathbf{x}_m] = \boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}} + I_{nm}\boldsymbol{\Sigma} \tag{2.291}$$

where $\mathbf{x}_n$ denotes a data point sampled from a Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$, and I_{nm} denotes the (n, m) element of the identity matrix. Hence prove the result (2.124).

2.36 ($\star\star$) **www** Using an analogous procedure to that used to obtain (2.126), derive an expression for the sequential estimation of the variance of a univariate Gaussian

distribution, by starting with the maximum likelihood expression

$$\sigma^2_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}(x_n - \mu)^2. \tag{2.292}$$

Verify that substituting the expression for a Gaussian distribution into the Robbins-Monro sequential estimation formula (2.135) gives a result of the same form, and hence obtain an expression for the corresponding coefficients a_N.

2.37 ($\star\star$) Using an analogous procedure to that used to obtain (2.126), derive an expression for the sequential estimation of the covariance of a multivariate Gaussian distribution, by starting with the maximum likelihood expression (2.122). Verify that substituting the expression for a Gaussian distribution into the Robbins-Monro sequential estimation formula (2.135) gives a result of the same form, and hence obtain an expression for the corresponding coefficients a_N.

2.38 ($\star$) Use the technique of completing the square for the quadratic form in the exponent to derive the results (2.141) and (2.142).

2.39 ($\star\star$) Starting from the results (2.141) and (2.142) for the posterior distribution of the mean of a Gaussian random variable, dissect out the contributions from the first $N-1$ data points and hence obtain expressions for the sequential update of μ_N and σ^2_N. Now derive the same results starting from the posterior distribution $p(\mu|x_1,\ldots,x_{N-1}) = \mathcal{N}(\mu|\mu_{N-1},\sigma^2_{N-1})$ and multiplying by the likelihood function $p(x_N|\mu) = \mathcal{N}(x_N|\mu,\sigma^2)$ and then completing the square and normalizing to obtain the posterior distribution after N observations.

2.40 ($\star\star$) **www** Consider a D-dimensional Gaussian random variable $\mathbf{x}$ with distribution $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\Sigma})$ in which the covariance $\boldsymbol{\Sigma}$ is known and for which we wish to infer the mean $\boldsymbol{\mu}$ from a set of observations $\mathbf{X} = \{\mathbf{x}_1,\ldots,\mathbf{x}_N\}$. Given a prior distribution $p(\boldsymbol{\mu}) = \mathcal{N}(\boldsymbol{\mu}|\boldsymbol{\mu}_0,\boldsymbol{\Sigma}_0)$, find the corresponding posterior distribution $p(\boldsymbol{\mu}|\mathbf{X})$.

2.41 ($\star$) Use the definition of the gamma function (1.141) to show that the gamma distribution (2.146) is normalized.

2.42 ($\star\star$) Evaluate the mean, variance, and mode of the gamma distribution (2.146).

2.43 ($\star$) The following distribution

$$p(x|\sigma^2,q) = \frac{q}{2(2\sigma^2)^{1/q}\Gamma(1/q)}\exp\left(-\frac{|x|^q}{2\sigma^2}\right) \tag{2.293}$$

is a generalization of the univariate Gaussian distribution. Show that this distribution is normalized so that

$$\int_{-\infty}^{\infty} p(x|\sigma^2,q)\,\mathrm{d}x = 1 \tag{2.294}$$

and that it reduces to the Gaussian when $q=2$. Consider a regression model in which the target variable is given by $t = y(\mathbf{x},\mathbf{w}) + \epsilon$ and ϵ is a random noise

variable drawn from the distribution (2.293). Show that the log likelihood function over $\mathbf{w}$ and σ^2, for an observed data set of input vectors $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ and corresponding target variables $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$, is given by

$$\ln p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \sigma^2) = -\frac{1}{2\sigma^2} \sum_{n=1}^{N} |y(\mathbf{x}_n, \mathbf{w}) - t_n|^q - \frac{N}{q} \ln(2\sigma^2) + \mathrm{const} \tag{2.295}$$

where 'const' denotes terms independent of both $\mathbf{w}$ and σ^2. Note that, as a function of $\mathbf{w}$, this is the L_q error function considered in Section 1.5.5.

2.44 ($\star\star$) Consider a univariate Gaussian distribution $\mathcal{N}(x|\mu, \tau^{-1})$ having conjugate Gaussian-gamma prior given by (2.154), and a data set $\mathbf{x} = \{x_1, \ldots, x_N\}$ of i.i.d. observations. Show that the posterior distribution is also a Gaussian-gamma distribution of the same functional form as the prior, and write down expressions for the parameters of this posterior distribution.

2.45 ($\star$) Verify that the Wishart distribution defined by (2.155) is indeed a conjugate prior for the precision matrix of a multivariate Gaussian.

2.46 ($\star$) **www** Verify that evaluating the integral in (2.158) leads to the result (2.159).

2.47 ($\star$) **www** Show that in the limit $\nu \to \infty$, the t-distribution (2.159) becomes a Gaussian. Hint: ignore the normalization coefficient, and simply look at the dependence on x.

2.48 ($\star$) By following analogous steps to those used to derive the univariate Student's t-distribution (2.159), verify the result (2.162) for the multivariate form of the Student's t-distribution, by marginalizing over the variable η in (2.161). Using the definition (2.161), show by exchanging integration variables that the multivariate t-distribution is correctly normalized.

2.49 ($\star\star$) By using the definition (2.161) of the multivariate Student's t-distribution as a convolution of a Gaussian with a gamma distribution, verify the properties (2.164), (2.165), and (2.166) for the multivariate t-distribution defined by (2.162).

2.50 ($\star$) Show that in the limit $\nu \to \infty$, the multivariate Student's t-distribution (2.162) reduces to a Gaussian with mean $\boldsymbol{\mu}$ and precision $\boldsymbol{\Lambda}$.

2.51 ($\star$) **www** The various trigonometric identities used in the discussion of periodic variables in this chapter can be proven easily from the relation

$$\exp(iA) = \cos A + i \sin A \tag{2.296}$$

in which i is the square root of minus one. By considering the identity

$$\exp(iA) \exp(-iA) = 1 \tag{2.297}$$

prove the result (2.177). Similarly, using the identity

$$\cos(A - B) = \Re \exp\{i(A - B)\} \tag{2.298}$$

where $\Re$ denotes the real part, prove (2.178). Finally, by using $\sin(A - B) = \Im \exp\{i(A - B)\}$, where $\Im$ denotes the imaginary part, prove the result (2.183).

2.52 ($\star\star$) For large m, the von Mises distribution (2.179) becomes sharply peaked around the mode θ_0. By defining $\xi = m^{1/2}(\theta - \theta_0)$ and making the Taylor expansion of the cosine function given by

$$\cos\alpha = 1 - \frac{\alpha^2}{2} + O(\alpha^4) \tag{2.299}$$

show that as $m \to \infty$, the von Mises distribution tends to a Gaussian.

2.53 ($\star$) Using the trigonometric identity (2.183), show that solution of (2.182) for θ_0 is given by (2.184).

2.54 ($\star$) By computing first and second derivatives of the von Mises distribution (2.179), and using $I_0(m) > 0$ for $m > 0$, show that the maximum of the distribution occurs when $\theta = \theta_0$ and that the minimum occurs when $\theta = \theta_0 + \pi \pmod{2\pi}$.

2.55 ($\star$) By making use of the result (2.168), together with (2.184) and the trigonometric identity (2.178), show that the maximum likelihood solution m_{ML} for the concentration of the von Mises distribution satisfies $A(m_{\mathrm{ML}}) = \overline{r}$ where $\overline{r}$ is the radius of the mean of the observations viewed as unit vectors in the two-dimensional Euclidean plane, as illustrated in Figure 2.17.

2.56 ($\star\star$) **www** Express the beta distribution (2.13), the gamma distribution (2.146), and the von Mises distribution (2.179) as members of the exponential family (2.194) and thereby identify their natural parameters.

2.57 ($\star$) Verify that the multivariate Gaussian distribution can be cast in exponential family form (2.194) and derive expressions for $\boldsymbol{\eta}$, $\mathbf{u}(\mathbf{x})$, $h(\mathbf{x})$ and $g(\boldsymbol{\eta})$ analogous to (2.220)–(2.223).

2.58 ($\star$) The result (2.226) showed that the negative gradient of $\ln g(\boldsymbol{\eta})$ for the exponential family is given by the expectation of $\mathbf{u}(\mathbf{x})$. By taking the second derivatives of (2.195), show that

$$-\nabla\nabla \ln g(\boldsymbol{\eta}) = \mathbb{E}[\mathbf{u}(\mathbf{x})\mathbf{u}(\mathbf{x})^{\mathrm{T}}] - \mathbb{E}[\mathbf{u}(\mathbf{x})]\mathbb{E}[\mathbf{u}(\mathbf{x})^{\mathrm{T}}] = \mathrm{cov}[\mathbf{u}(\mathbf{x})]. \tag{2.300}$$

2.59 ($\star$) By changing variables using $y = x/\sigma$, show that the density (2.236) will be correctly normalized, provided $f(x)$ is correctly normalized.

2.60 ($\star\star$) **www** Consider a histogram-like density model in which the space $\mathbf{x}$ is divided into fixed regions for which the density $p(\mathbf{x})$ takes the constant value h_i over the ith region, and that the volume of region i is denoted Δ_i. Suppose we have a set of N observations of $\mathbf{x}$ such that n_i of these observations fall in region i. Using a Lagrange multiplier to enforce the normalization constraint on the density, derive an expression for the maximum likelihood estimator for the $\{h_i\}$.

2.61 ($\star$) Show that the K-nearest-neighbour density model defines an improper distribution whose integral over all space is divergent.

3 Linear Models for Regression

The focus so far in this book has been on unsupervised learning, including topics such as density estimation and data clustering. We turn now to a discussion of supervised learning, starting with regression. The goal of regression is to predict the value of one or more continuous *target* variables t given the value of a D-dimensional vector $\mathbf{x}$ of *input* variables. We have already encountered an example of a regression problem when we considered polynomial curve fitting in Chapter 1. The polynomial is a specific example of a broad class of functions called linear regression models, which share the property of being linear functions of the adjustable parameters, and which will form the focus of this chapter. The simplest form of linear regression models are also linear functions of the input variables. However, we can obtain a much more useful class of functions by taking linear combinations of a fixed set of nonlinear functions of the input variables, known as *basis functions*. Such models are linear functions of the parameters, which gives them simple analytical properties, and yet can be nonlinear with respect to the input variables.

Given a training data set comprising N observations $\{\mathbf{x}_n\}$, where $n = 1, \ldots, N$, together with corresponding target values $\{t_n\}$, the goal is to predict the value of t for a new value of $\mathbf{x}$. In the simplest approach, this can be done by directly constructing an appropriate function $y(\mathbf{x})$ whose values for new inputs $\mathbf{x}$ constitute the predictions for the corresponding values of t. More generally, from a probabilistic perspective, we aim to model the predictive distribution $p(t|\mathbf{x})$ because this expresses our uncertainty about the value of t for each value of $\mathbf{x}$. From this conditional distribution we can make predictions of t, for any new value of $\mathbf{x}$, in such a way as to minimize the expected value of a suitably chosen loss function. As discussed in Section 1.5.5, a common choice of loss function for real-valued variables is the squared loss, for which the optimal solution is given by the conditional expectation of t.

Although linear models have significant limitations as practical techniques for pattern recognition, particularly for problems involving input spaces of high dimensionality, they have nice analytical properties and form the foundation for more sophisticated models to be discussed in later chapters.

3.1. Linear Basis Function Models

The simplest linear model for regression is one that involves a linear combination of the input variables

$$y(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + \ldots + w_D x_D \tag{3.1}$$

where $\mathbf{x} = (x_1, \ldots, x_D)^{\mathrm{T}}$. This is often simply known as *linear regression*. The key property of this model is that it is a linear function of the parameters $w_0, \ldots, w_D$. It is also, however, a linear function of the input variables x_i, and this imposes significant limitations on the model. We therefore extend the class of models by considering linear combinations of fixed nonlinear functions of the input variables, of the form

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j=1}^{M-1} w_j \phi_j(\mathbf{x}) \tag{3.2}$$

where $\phi_j(\mathbf{x})$ are known as *basis functions*. By denoting the maximum value of the index j by $M - 1$, the total number of parameters in this model will be M.

The parameter w_0 allows for any fixed offset in the data and is sometimes called a *bias* parameter (not to be confused with 'bias' in a statistical sense). It is often convenient to define an additional dummy 'basis function' $\phi_0(\mathbf{x}) = 1$ so that

$$y(\mathbf{x}, \mathbf{w}) = \sum_{j=0}^{M-1} w_j \phi_j(\mathbf{x}) = \mathbf{w}^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}) \tag{3.3}$$

where $\mathbf{w} = (w_0, \ldots, w_{M-1})^{\mathrm{T}}$ and $\boldsymbol{\phi} = (\phi_0, \ldots, \phi_{M-1})^{\mathrm{T}}$. In many practical applications of pattern recognition, we will apply some form of fixed pre-processing,

or feature extraction, to the original data variables. If the original variables comprise the vector $\mathbf{x}$, then the features can be expressed in terms of the basis functions $\{\phi_j(\mathbf{x})\}$.

By using nonlinear basis functions, we allow the function $y(\mathbf{x}, \mathbf{w})$ to be a nonlinear function of the input vector $\mathbf{x}$. Functions of the form (3.2) are called linear models, however, because this function is linear in $\mathbf{w}$. It is this linearity in the parameters that will greatly simplify the analysis of this class of models. However, it also leads to some significant limitations, as we discuss in Section 3.6.

The example of polynomial regression considered in Chapter 1 is a particular example of this model in which there is a single input variable x, and the basis functions take the form of powers of x so that $\phi_j(x) = x^j$. One limitation of polynomial basis functions is that they are global functions of the input variable, so that changes in one region of input space affect all other regions. This can be resolved by dividing the input space up into regions and fit a different polynomial in each region, leading to *spline functions* (Hastie *et al.*, 2001).

There are many other possible choices for the basis functions, for example

$$\phi_j(x) = \exp\left\{-\frac{(x-\mu_j)^2}{2s^2}\right\} \tag{3.4}$$

where the μ_j govern the locations of the basis functions in input space, and the parameter s governs their spatial scale. These are usually referred to as 'Gaussian' basis functions, although it should be noted that they are not required to have a probabilistic interpretation, and in particular the normalization coefficient is unimportant because these basis functions will be multiplied by adaptive parameters w_j.

Another possibility is the sigmoidal basis function of the form

$$\phi_j(x) = \sigma\left(\frac{x-\mu_j}{s}\right) \tag{3.5}$$

where $\sigma(a)$ is the logistic sigmoid function defined by

$$\sigma(a) = \frac{1}{1+\exp(-a)}. \tag{3.6}$$

Equivalently, we can use the '$\tanh$' function because this is related to the logistic sigmoid by $\tanh(a) = 2\sigma(a) - 1$, and so a general linear combination of logistic sigmoid functions is equivalent to a general linear combination of '$\tanh$' functions. These various choices of basis function are illustrated in Figure 3.1.

Yet another possible choice of basis function is the Fourier basis, which leads to an expansion in sinusoidal functions. Each basis function represents a specific frequency and has infinite spatial extent. By contrast, basis functions that are localized to finite regions of input space necessarily comprise a spectrum of different spatial frequencies. In many signal processing applications, it is of interest to consider basis functions that are localized in both space and frequency, leading to a class of functions known as *wavelets*. These are also defined to be mutually orthogonal, to simplify their application. Wavelets are most applicable when the input values live

Figure 3.1 Examples of basis functions, showing polynomials on the left, Gaussians of the form (3.4) in the centre, and sigmoidal of the form (3.5) on the right.

on a regular lattice, such as the successive time points in a temporal sequence, or the pixels in an image. Useful texts on wavelets include Ogden (1997), Mallat (1999), and Vidakovic (1999).

Most of the discussion in this chapter, however, is independent of the particular choice of basis function set, and so for most of our discussion we shall not specify the particular form of the basis functions, except for the purposes of numerical illustration. Indeed, much of our discussion will be equally applicable to the situation in which the vector $\boldsymbol{\phi}(\mathbf{x})$ of basis functions is simply the identity $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$. Furthermore, in order to keep the notation simple, we shall focus on the case of a single target variable t. However, in Section 3.1.5, we consider briefly the modifications needed to deal with multiple target variables.

3.1.1 Maximum likelihood and least squares

In Chapter 1, we fitted polynomial functions to data sets by minimizing a sum-of-squares error function. We also showed that this error function could be motivated as the maximum likelihood solution under an assumed Gaussian noise model. Let us return to this discussion and consider the least squares approach, and its relation to maximum likelihood, in more detail.

As before, we assume that the target variable t is given by a deterministic function $y(\mathbf{x}, \mathbf{w})$ with additive Gaussian noise so that

$$t = y(\mathbf{x}, \mathbf{w}) + \epsilon \tag{3.7}$$

where ϵ is a zero mean Gaussian random variable with precision (inverse variance) β. Thus we can write

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}). \tag{3.8}$$

Section 1.5.5

Recall that, if we assume a squared loss function, then the optimal prediction, for a new value of $\mathbf{x}$, will be given by the conditional mean of the target variable. In the case of a Gaussian conditional distribution of the form (3.8), the conditional mean

will be simply

$$\mathbb{E}[t|\mathbf{x}] = \int tp(t|\mathbf{x})\,\mathrm{d}t = y(\mathbf{x}, \mathbf{w}). \tag{3.9}$$

Note that the Gaussian noise assumption implies that the conditional distribution of t given $\mathbf{x}$ is unimodal, which may be inappropriate for some applications. An extension to mixtures of conditional Gaussian distributions, which permit multimodal conditional distributions, will be discussed in Section 14.5.1.

Now consider a data set of inputs $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ with corresponding target values $t_1, \ldots, t_N$. We group the target variables $\{t_n\}$ into a column vector that we denote by $\mathbf{t}$ where the typeface is chosen to distinguish it from a single observation of a multivariate target, which would be denoted $\mathbf{t}$. Making the assumption that these data points are drawn independently from the distribution (3.8), we obtain the following expression for the likelihood function, which is a function of the adjustable parameters $\mathbf{w}$ and β, in the form

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}(t_n|\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n), \beta^{-1}) \tag{3.10}$$

where we have used (3.3). Note that in supervised learning problems such as regression (and classification), we are not seeking to model the distribution of the input variables. Thus $\mathbf{x}$ will always appear in the set of conditioning variables, and so from now on we will drop the explicit $\mathbf{x}$ from expressions such as $p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \beta)$ in order to keep the notation uncluttered. Taking the logarithm of the likelihood function, and making use of the standard form (1.46) for the univariate Gaussian, we have

$$\begin{aligned} \ln p(\mathbf{t}|\mathbf{w}, \beta) &= \sum_{n=1}^{N} \ln \mathcal{N}(t_n|\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n), \beta^{-1}) \\ &= \frac{N}{2}\ln\beta - \frac{N}{2}\ln(2\pi) - \beta E_D(\mathbf{w}) \end{aligned} \tag{3.11}$$

where the sum-of-squares error function is defined by

$$E_D(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2. \tag{3.12}$$

Having written down the likelihood function, we can use maximum likelihood to determine $\mathbf{w}$ and β. Consider first the maximization with respect to $\mathbf{w}$. As observed already in Section 1.2.5, we see that maximization of the likelihood function under a conditional Gaussian noise distribution for a linear model is equivalent to minimizing a sum-of-squares error function given by $E_D(\mathbf{w})$. The gradient of the log likelihood function (3.11) takes the form

$$\nabla \ln p(\mathbf{t}|\mathbf{w}, \beta) = \sum_{n=1}^{N} \left\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\right\}\boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}}. \tag{3.13}$$

Setting this gradient to zero gives

$$0 = \sum_{n=1}^{N} t_n \boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}} - \mathbf{w}^{\mathrm{T}} \left(\sum_{n=1}^{N} \boldsymbol{\phi}(\mathbf{x}_n) \boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}} \right). \tag{3.14}$$

Solving for $\mathbf{w}$ we obtain

$$\mathbf{w}_{\mathrm{ML}} = \left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1} \boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t} \tag{3.15}$$

which are known as the *normal equations* for the least squares problem. Here $\boldsymbol{\Phi}$ is an $N \times M$ matrix, called the *design matrix*, whose elements are given by $\Phi_{nj} = \phi_j(\mathbf{x}_n)$, so that

$$\boldsymbol{\Phi} = \begin{pmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \cdots & \phi_{M-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \cdots & \phi_{M-1}(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_N) & \phi_1(\mathbf{x}_N) & \cdots & \phi_{M-1}(\mathbf{x}_N) \end{pmatrix}. \tag{3.16}$$

The quantity

$$\boldsymbol{\Phi}^{\dagger} \equiv \left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1} \boldsymbol{\Phi}^{\mathrm{T}} \tag{3.17}$$

is known as the *Moore-Penrose pseudo-inverse* of the matrix $\boldsymbol{\Phi}$ (Rao and Mitra, 1971; Golub and Van Loan, 1996). It can be regarded as a generalization of the notion of matrix inverse to nonsquare matrices. Indeed, if $\boldsymbol{\Phi}$ is square and invertible, then using the property $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$ we see that $\boldsymbol{\Phi}^{\dagger} \equiv \boldsymbol{\Phi}^{-1}$.

At this point, we can gain some insight into the role of the bias parameter w_0. If we make the bias parameter explicit, then the error function (3.12) becomes

$$E_D(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \{t_n - w_0 - \sum_{j=1}^{M-1} w_j \phi_j(\mathbf{x}_n)\}^2. \tag{3.18}$$

Setting the derivative with respect to w_0 equal to zero, and solving for w_0, we obtain

$$w_0 = \bar{t} - \sum_{j=1}^{M-1} w_j \overline{\phi_j} \tag{3.19}$$

where we have defined

$$\bar{t} = \frac{1}{N} \sum_{n=1}^{N} t_n, \qquad \overline{\phi_j} = \frac{1}{N} \sum_{n=1}^{N} \phi_j(\mathbf{x}_n). \tag{3.20}$$

Thus the bias w_0 compensates for the difference between the averages (over the training set) of the target values and the weighted sum of the averages of the basis function values.

We can also maximize the log likelihood function (3.11) with respect to the noise precision parameter β, giving

$$\frac{1}{\beta_{\mathrm{ML}}} = \frac{1}{N} \sum_{n=1}^{N} \{t_n - \mathbf{w}_{\mathrm{ML}}^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}_n)\}^2 \tag{3.21}$$

Figure 3.2 Geometrical interpretation of the least-squares solution, in an N-dimensional space whose axes are the values of $t_1, \ldots, t_N$. The least-squares regression function is obtained by finding the orthogonal projection of the data vector $\mathbf{t}$ onto the subspace spanned by the basis functions $\phi_j(\mathbf{x})$ in which each basis function is viewed as a vector $\boldsymbol{\varphi}_j$ of length N with elements $\phi_j(\mathbf{x}_n)$.

and so we see that the inverse of the noise precision is given by the residual variance of the target values around the regression function.

3.1.2 Geometry of least squares

At this point, it is instructive to consider the geometrical interpretation of the least-squares solution. To do this we consider an N-dimensional space whose axes are given by the t_n, so that $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$ is a vector in this space. Each basis function $\phi_j(\mathbf{x}_n)$, evaluated at the N data points, can also be represented as a vector in the same space, denoted by $\boldsymbol{\varphi}_j$, as illustrated in Figure 3.2. Note that $\boldsymbol{\varphi}_j$ corresponds to the jth column of $\boldsymbol{\Phi}$, whereas $\boldsymbol{\phi}(\mathbf{x}_n)$ corresponds to the nth row of $\boldsymbol{\Phi}$. If the number M of basis functions is smaller than the number N of data points, then the M vectors $\phi_j(\mathbf{x}_n)$ will span a linear subspace $\mathcal{S}$ of dimensionality M. We define $\mathbf{y}$ to be an N-dimensional vector whose n^{th} element is given by $y(\mathbf{x}_n, \mathbf{w})$, where $n = 1, \ldots, N$. Because $\mathbf{y}$ is an arbitrary linear combination of the vectors $\boldsymbol{\varphi}_j$, it can live anywhere in the M-dimensional subspace. The sum-of-squares error (3.12) is then equal (up to a factor of $1/2$) to the squared Euclidean distance between $\mathbf{y}$ and $\mathbf{t}$. Thus the least-squares solution for $\mathbf{w}$ corresponds to that choice of $\mathbf{y}$ that lies in subspace $\mathcal{S}$ and that is closest to $\mathbf{t}$. Intuitively, from Figure 3.2, we anticipate that this solution corresponds to the orthogonal projection of $\mathbf{t}$ onto the subspace $\mathcal{S}$. This is indeed the case, as can easily be verified by noting that the solution for $\mathbf{y}$ is given by $\boldsymbol{\Phi}\mathbf{w}_{\mathrm{ML}}$, and then confirming that this takes the form of an orthogonal projection.

Exercise 3.2

In practice, a direct solution of the normal equations can lead to numerical difficulties when $\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}$ is close to singular. In particular, when two or more of the basis vectors $\boldsymbol{\varphi}_j$ are co-linear, or nearly so, the resulting parameter values can have large magnitudes. Such near degeneracies will not be uncommon when dealing with real data sets. The resulting numerical difficulties can be addressed using the technique of *singular value decomposition*, or *SVD* (Press *et al.*, 1992; Bishop and Nabney, 2008). Note that the addition of a regularization term ensures that the matrix is non-singular, even in the presence of degeneracies.

3.1.3 Sequential learning

Batch techniques, such as the maximum likelihood solution (3.15), which involve processing the entire training set in one go, can be computationally costly for large data sets. As we have discussed in Chapter 1, if the data set is sufficiently large, it may be worthwhile to use *sequential* algorithms, also known as *on-line* algorithms,

in which the data points are considered one at a time, and the model parameters updated after each such presentation. Sequential learning is also appropriate for real-time applications in which the data observations are arriving in a continuous stream, and predictions must be made before all of the data points are seen.

We can obtain a sequential learning algorithm by applying the technique of *stochastic gradient descent*, also known as *sequential gradient descent*, as follows. If the error function comprises a sum over data points $E = \sum_n E_n$, then after presentation of pattern n, the stochastic gradient descent algorithm updates the parameter vector $\mathbf{w}$ using

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n \tag{3.22}$$

where τ denotes the iteration number, and η is a learning rate parameter. We shall discuss the choice of value for η shortly. The value of $\mathbf{w}$ is initialized to some starting vector $\mathbf{w}^{(0)}$. For the case of the sum-of-squares error function (3.12), this gives

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \eta(t_n - \mathbf{w}^{(\tau)\mathrm{T}}\boldsymbol{\phi}_n)\boldsymbol{\phi}_n \tag{3.23}$$

where $\boldsymbol{\phi}_n = \boldsymbol{\phi}(\mathbf{x}_n)$. This is known as *least-mean-squares* or the *LMS algorithm*. The value of η needs to be chosen with care to ensure that the algorithm converges (Bishop and Nabney, 2008).

3.1.4 Regularized least squares

In Section 1.1, we introduced the idea of adding a regularization term to an error function in order to control over-fitting, so that the total error function to be minimized takes the form

$$E_D(\mathbf{w}) + \lambda E_W(\mathbf{w}) \tag{3.24}$$

where λ is the regularization coefficient that controls the relative importance of the data-dependent error $E_D(\mathbf{w})$ and the regularization term $E_W(\mathbf{w})$. One of the simplest forms of regularizer is given by the sum-of-squares of the weight vector elements

$$E_W(\mathbf{w}) = \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}. \tag{3.25}$$

If we also consider the sum-of-squares error function given by

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2 \tag{3.26}$$

then the total error function becomes

$$\frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2 + \frac{\lambda}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}. \tag{3.27}$$

This particular choice of regularizer is known in the machine learning literature as *weight decay* because in sequential learning algorithms, it encourages weight values to decay towards zero, unless supported by the data. In statistics, it provides an example of a *parameter shrinkage* method because it shrinks parameter values towards

Figure 3.3 Contours of the regularization term in (3.29) for various values of the parameter q.

zero. It has the advantage that the error function remains a quadratic function of $\mathbf{w}$, and so its exact minimizer can be found in closed form. Specifically, setting the gradient of (3.27) with respect to $\mathbf{w}$ to zero, and solving for $\mathbf{w}$ as before, we obtain

$$\mathbf{w} = \left(\lambda \mathbf{I} + \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1} \boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t}. \tag{3.28}$$

This represents a simple extension of the least-squares solution (3.15).

A more general regularizer is sometimes used, for which the regularized error takes the form

$$\frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2 + \frac{\lambda}{2}\sum_{j=1}^{M}|w_j|^q \tag{3.29}$$

where $q = 2$ corresponds to the quadratic regularizer (3.27). Figure 3.3 shows contours of the regularization function for different values of q.

The case of $q = 1$ is know as the *lasso* in the statistics literature (Tibshirani, 1996). It has the property that if λ is sufficiently large, some of the coefficients w_j are driven to zero, leading to a *sparse* model in which the corresponding basis functions play no role. To see this, we first note that minimizing (3.29) is equivalent to minimizing the unregularized sum-of-squares error (3.12) subject to the constraint

Exercise 3.5

$$\sum_{j=1}^{M}|w_j|^q \leqslant \eta \tag{3.30}$$

for an appropriate value of the parameter η, where the two approaches can be related using Lagrange multipliers. The origin of the sparsity can be seen from Figure 3.4, which shows that the minimum of the error function, subject to the constraint (3.30). As λ is increased, so an increasing number of parameters are driven to zero.

Appendix E

Regularization allows complex models to be trained on data sets of limited size without severe over-fitting, essentially by limiting the effective model complexity. However, the problem of determining the optimal model complexity is then shifted from one of finding the appropriate number of basis functions to one of determining a suitable value of the regularization coefficient λ. We shall return to the issue of model complexity later in this chapter.

Figure 3.4 Plot of the contours of the unregularized error function (blue) along with the constraint region (3.30) for the quadratic regularizer $q = 2$ on the left and the lasso regularizer $q = 1$ on the right, in which the optimum value for the parameter vector $\mathbf{w}$ is denoted by $\mathbf{w}^\star$. The lasso gives a sparse solution in which $w_1^\star = 0$.

For the remainder of this chapter we shall focus on the quadratic regularizer (3.27) both for its practical importance and its analytical tractability.

3.1.5 Multiple outputs

So far, we have considered the case of a single target variable t. In some applications, we may wish to predict $K > 1$ target variables, which we denote collectively by the target vector $\mathbf{t}$. This could be done by introducing a different set of basis functions for each component of $\mathbf{t}$, leading to multiple, independent regression problems. However, a more interesting, and more common, approach is to use the same set of basis functions to model all of the components of the target vector so that

$$\mathbf{y}(\mathbf{x}, \mathbf{w}) = \mathbf{W}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}) \tag{3.31}$$

where $\mathbf{y}$ is a K-dimensional column vector, $\mathbf{W}$ is an $M \times K$ matrix of parameters, and $\boldsymbol{\phi}(\mathbf{x})$ is an M-dimensional column vector with elements $\phi_j(\mathbf{x})$, with $\phi_0(\mathbf{x}) = 1$ as before. Suppose we take the conditional distribution of the target vector to be an isotropic Gaussian of the form

$$p(\mathbf{t}|\mathbf{x}, \mathbf{W}, \beta) = \mathcal{N}(\mathbf{t}|\mathbf{W}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}), \beta^{-1}\mathbf{I}). \tag{3.32}$$

If we have a set of observations $\mathbf{t}_1, \ldots, \mathbf{t}_N$, we can combine these into a matrix $\mathbf{T}$ of size $N \times K$ such that the nth row is given by $\mathbf{t}_n^{\mathrm{T}}$. Similarly, we can combine the input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ into a matrix $\mathbf{X}$. The log likelihood function is then given by

$$\begin{aligned} \ln p(\mathbf{T}|\mathbf{X}, \mathbf{W}, \beta) &= \sum_{n=1}^{N} \ln \mathcal{N}(\mathbf{t}_n|\mathbf{W}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n), \beta^{-1}\mathbf{I}) \\ &= \frac{NK}{2}\ln\left(\frac{\beta}{2\pi}\right) - \frac{\beta}{2}\sum_{n=1}^{N}\left\|\mathbf{t}_n - \mathbf{W}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\right\|^2. \end{aligned} \tag{3.33}$$

As before, we can maximize this function with respect to $\mathbf{W}$, giving

$$\mathbf{W}_{\mathrm{ML}} = \left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{T}. \tag{3.34}$$

If we examine this result for each target variable t_k, we have

$$\mathbf{w}_k = \left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t}_k = \boldsymbol{\Phi}^{\dagger}\mathbf{t}_k \tag{3.35}$$

where $\mathbf{t}_k$ is an N-dimensional column vector with components t_{nk} for $n = 1, \ldots N$. Thus the solution to the regression problem decouples between the different target variables, and we need only compute a single pseudo-inverse matrix $\boldsymbol{\Phi}^{\dagger}$, which is shared by all of the vectors $\mathbf{w}_k$.

Exercise 3.6

The extension to general Gaussian noise distributions having arbitrary covariance matrices is straightforward. Again, this leads to a decoupling into K independent regression problems. This result is unsurprising because the parameters $\mathbf{W}$ define only the mean of the Gaussian noise distribution, and we know from Section 2.3.4 that the maximum likelihood solution for the mean of a multivariate Gaussian is independent of the covariance. From now on, we shall therefore consider a single target variable t for simplicity.

3.2. The Bias-Variance Decomposition

So far in our discussion of linear models for regression, we have assumed that the form and number of basis functions are both fixed. As we have seen in Chapter 1, the use of maximum likelihood, or equivalently least squares, can lead to severe over-fitting if complex models are trained using data sets of limited size. However, limiting the number of basis functions in order to avoid over-fitting has the side effect of limiting the flexibility of the model to capture interesting and important trends in the data. Although the introduction of regularization terms can control over-fitting for models with many parameters, this raises the question of how to determine a suitable value for the regularization coefficient λ. Seeking the solution that minimizes the regularized error function with respect to both the weight vector $\mathbf{w}$ and the regularization coefficient λ is clearly not the right approach since this leads to the unregularized solution with $\lambda = 0$.

As we have seen in earlier chapters, the phenomenon of over-fitting is really an unfortunate property of maximum likelihood and does not arise when we marginalize over parameters in a Bayesian setting. In this chapter, we shall consider the Bayesian view of model complexity in some depth. Before doing so, however, it is instructive to consider a frequentist viewpoint of the model complexity issue, known as the *bias-variance* trade-off. Although we shall introduce this concept in the context of linear basis function models, where it is easy to illustrate the ideas using simple examples, the discussion has more general applicability.

In Section 1.5.5, when we discussed decision theory for regression problems, we considered various loss functions each of which leads to a corresponding optimal prediction once we are given the conditional distribution $p(t|\mathbf{x})$. A popular choice is

the squared loss function, for which the optimal prediction is given by the conditional expectation, which we denote by $h(\mathbf{x})$ and which is given by

$$h(\mathbf{x}) = \mathbb{E}[t|\mathbf{x}] = \int tp(t|\mathbf{x})\,\mathrm{d}t. \tag{3.36}$$

At this point, it is worth distinguishing between the squared loss function arising from decision theory and the sum-of-squares error function that arose in the maximum likelihood estimation of model parameters. We might use more sophisticated techniques than least squares, for example regularization or a fully Bayesian approach, to determine the conditional distribution $p(t|\mathbf{x})$. These can all be combined with the squared loss function for the purpose of making predictions.

We showed in Section 1.5.5 that the expected squared loss can be written in the form

$$\mathbb{E}[L] = \int \{y(\mathbf{x}) - h(\mathbf{x})\}^2 p(\mathbf{x})\,\mathrm{d}\mathbf{x} + \int \{h(\mathbf{x}) - t\}^2 p(\mathbf{x}, t)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t. \tag{3.37}$$

Recall that the second term, which is independent of $y(\mathbf{x})$, arises from the intrinsic noise on the data and represents the minimum achievable value of the expected loss. The first term depends on our choice for the function $y(\mathbf{x})$, and we will seek a solution for $y(\mathbf{x})$ which makes this term a minimum. Because it is nonnegative, the smallest that we can hope to make this term is zero. If we had an unlimited supply of data (and unlimited computational resources), we could in principle find the regression function $h(\mathbf{x})$ to any desired degree of accuracy, and this would represent the optimal choice for $y(\mathbf{x})$. However, in practice we have a data set $\mathcal{D}$ containing only a finite number N of data points, and consequently we do not know the regression function $h(\mathbf{x})$ exactly.

If we model the $h(\mathbf{x})$ using a parametric function $y(\mathbf{x}, \mathbf{w})$ governed by a parameter vector $\mathbf{w}$, then from a Bayesian perspective the uncertainty in our model is expressed through a posterior distribution over $\mathbf{w}$. A frequentist treatment, however, involves making a point estimate of $\mathbf{w}$ based on the data set $\mathcal{D}$, and tries instead to interpret the uncertainty of this estimate through the following thought experiment. Suppose we had a large number of data sets each of size N and each drawn independently from the distribution $p(t, \mathbf{x})$. For any given data set $\mathcal{D}$, we can run our learning algorithm and obtain a prediction function $y(\mathbf{x}; \mathcal{D})$. Different data sets from the ensemble will give different functions and consequently different values of the squared loss. The performance of a particular learning algorithm is then assessed by taking the average over this ensemble of data sets.

Consider the integrand of the first term in (3.37), which for a particular data set $\mathcal{D}$ takes the form

$$\{y(\mathbf{x}; \mathcal{D}) - h(\mathbf{x})\}^2. \tag{3.38}$$

Because this quantity will be dependent on the particular data set $\mathcal{D}$, we take its average over the ensemble of data sets. If we add and subtract the quantity $\mathbb{E}_{\mathcal{D}}[y(\mathbf{x}; \mathcal{D})]$

inside the braces, and then expand, we obtain

$$\begin{aligned}
&\{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] + \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2 \\
&\quad = \{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}^2 + \{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2 \\
&\qquad +2\{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}. \qquad (3.39)
\end{aligned}$$

We now take the expectation of this expression with respect to $\mathcal{D}$ and note that the final term will vanish, giving

$$\begin{aligned}
&\mathbb{E}_{\mathcal{D}}\left[\{y(\mathbf{x};\mathcal{D}) - h(\mathbf{x})\}^2\right] \\
&\quad = \underbrace{\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2}_{(\text{bias})^2} + \underbrace{\mathbb{E}_{\mathcal{D}}\left[\{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}^2\right]}_{\text{variance}}. \qquad (3.40)
\end{aligned}$$

We see that the expected squared difference between $y(\mathbf{x};\mathcal{D})$ and the regression function $h(\mathbf{x})$ can be expressed as the sum of two terms. The first term, called the squared *bias*, represents the extent to which the average prediction over all data sets differs from the desired regression function. The second term, called the *variance*, measures the extent to which the solutions for individual data sets vary around their average, and hence this measures the extent to which the function $y(\mathbf{x};\mathcal{D})$ is sensitive to the particular choice of data set. We shall provide some intuition to support these definitions shortly when we consider a simple example.

So far, we have considered a single input value $\mathbf{x}$. If we substitute this expansion back into (3.37), we obtain the following decomposition of the expected squared loss

$$\text{expected loss} = (\text{bias})^2 + \text{variance} + \text{noise} \qquad (3.41)$$

where

$$(\text{bias})^2 = \int \{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2 p(\mathbf{x})\,\mathrm{d}\mathbf{x} \qquad (3.42)$$

$$\text{variance} = \int \mathbb{E}_{\mathcal{D}}\left[\{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}^2\right] p(\mathbf{x})\,\mathrm{d}\mathbf{x} \qquad (3.43)$$

$$\text{noise} = \int \{h(\mathbf{x}) - t\}^2 p(\mathbf{x}, t)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t \qquad (3.44)$$

and the bias and variance terms now refer to integrated quantities.

Our goal is to minimize the expected loss, which we have decomposed into the sum of a (squared) bias, a variance, and a constant noise term. As we shall see, there is a trade-off between bias and variance, with very flexible models having low bias and high variance, and relatively rigid models having high bias and low variance. The model with the optimal predictive capability is the one that leads to the best balance between bias and variance. This is illustrated by considering the sinusoidal data set from Chapter 1. Here we generate 100 data sets, each containing $N = 25$ data points, independently from the sinusoidal curve $h(x) = \sin(2\pi x)$. The data sets are indexed by $l = 1, \ldots, L$, where $L = 100$, and for each data set $\mathcal{D}^{(l)}$ we

Appendix A

Figure 3.5 Illustration of the dependence of bias and variance on model complexity, governed by a regularization parameter λ, using the sinusoidal data set from Chapter 1. There are $L = 100$ data sets, each having $N = 25$ data points, and there are 24 Gaussian basis functions in the model so that the total number of parameters is $M = 25$ including the bias parameter. The left column shows the result of fitting the model to the data sets for various values of $\ln \lambda$ (for clarity, only 20 of the 100 fits are shown). The right column shows the corresponding average of the 100 fits (red) along with the sinusoidal function from which the data sets were generated (green).

Figure 3.6 Plot of squared bias and variance, together with their sum, corresponding to the results shown in Figure 3.5. Also shown is the average test set error for a test data set size of 1000 points. The minimum value of $(\text{bias})^2 + \text{variance}$ occurs around $\ln \lambda = -0.31$, which is close to the value that gives the minimum error on the test data.

fit a model with 24 Gaussian basis functions by minimizing the regularized error function (3.27) to give a prediction function $y^{(l)}(x)$ as shown in Figure 3.5. The top row corresponds to a large value of the regularization coefficient λ that gives low variance (because the red curves in the left plot look similar) but high bias (because the two curves in the right plot are very different). Conversely on the bottom row, for which λ is small, there is large variance (shown by the high variability between the red curves in the left plot) but low bias (shown by the good fit between the average model fit and the original sinusoidal function). Note that the result of averaging many solutions for the complex model with $M = 25$ is a very good fit to the regression function, which suggests that averaging may be a beneficial procedure. Indeed, a weighted averaging of multiple solutions lies at the heart of a Bayesian approach, although the averaging is with respect to the posterior distribution of parameters, not with respect to multiple data sets.

We can also examine the bias-variance trade-off quantitatively for this example. The average prediction is estimated from

$$\overline{y}(x) = \frac{1}{L}\sum_{l=1}^{L} y^{(l)}(x) \tag{3.45}$$

and the integrated squared bias and integrated variance are then given by

$$(\text{bias})^2 = \frac{1}{N}\sum_{n=1}^{N} \{\overline{y}(x_n) - h(x_n)\}^2 \tag{3.46}$$

$$\text{variance} = \frac{1}{N}\sum_{n=1}^{N} \frac{1}{L}\sum_{l=1}^{L} \left\{y^{(l)}(x_n) - \overline{y}(x_n)\right\}^2 \tag{3.47}$$

where the integral over x weighted by the distribution $p(x)$ is approximated by a finite sum over data points drawn from that distribution. These quantities, along with their sum, are plotted as a function of $\ln \lambda$ in Figure 3.6. We see that small values of λ allow the model to become finely tuned to the noise on each individual

data set leading to large variance. Conversely, a large value of λ pulls the weight parameters towards zero leading to large bias.

Although the bias-variance decomposition may provide some interesting insights into the model complexity issue from a frequentist perspective, it is of limited practical value, because the bias-variance decomposition is based on averages with respect to ensembles of data sets, whereas in practice we have only the single observed data set. If we had a large number of independent training sets of a given size, we would be better off combining them into a single large training set, which of course would reduce the level of over-fitting for a given model complexity.

Given these limitations, we turn in the next section to a Bayesian treatment of linear basis function models, which not only provides powerful insights into the issues of over-fitting but which also leads to practical techniques for addressing the question model complexity.

3.3. Bayesian Linear Regression

In our discussion of maximum likelihood for setting the parameters of a linear regression model, we have seen that the effective model complexity, governed by the number of basis functions, needs to be controlled according to the size of the data set. Adding a regularization term to the log likelihood function means the effective model complexity can then be controlled by the value of the regularization coefficient, although the choice of the number and form of the basis functions is of course still important in determining the overall behaviour of the model.

This leaves the issue of deciding the appropriate model complexity for the particular problem, which cannot be decided simply by maximizing the likelihood function, because this always leads to excessively complex models and over-fitting. Independent hold-out data can be used to determine model complexity, as discussed in Section 1.3, but this can be both computationally expensive and wasteful of valuable data. We therefore turn to a Bayesian treatment of linear regression, which will avoid the over-fitting problem of maximum likelihood, and which will also lead to automatic methods of determining model complexity using the training data alone. Again, for simplicity we will focus on the case of a single target variable t. Extension to multiple target variables is straightforward and follows the discussion of Section 3.1.5.

3.3.1 Parameter distribution

We begin our discussion of the Bayesian treatment of linear regression by introducing a prior probability distribution over the model parameters $\mathbf{w}$. For the moment, we shall treat the noise precision parameter β as a known constant. First note that the likelihood function $p(\mathbf{t}|\mathbf{w})$ defined by (3.10) is the exponential of a quadratic function of $\mathbf{w}$. The corresponding conjugate prior is therefore given by a Gaussian distribution of the form

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{m}_0, \mathbf{S}_0) \tag{3.48}$$

having mean $\mathbf{m}_0$ and covariance $\mathbf{S}_0$.

Next we compute the posterior distribution, which is proportional to the product of the likelihood function and the prior. Due to the choice of a conjugate Gaussian prior distribution, the posterior will also be Gaussian. We can evaluate this distribution by the usual procedure of completing the square in the exponential, and then finding the normalization coefficient using the standard result for a normalized Gaussian. However, we have already done the necessary work in deriving the general result (2.116), which allows us to write down the posterior distribution directly in the form

Exercise 3.7

$$p(\mathbf{w}|\mathbf{t}) = \mathcal{N}(\mathbf{w}|\mathbf{m}_N, \mathbf{S}_N) \tag{3.49}$$

where

$$\mathbf{m}_N = \mathbf{S}_N \left(\mathbf{S}_0^{-1}\mathbf{m}_0 + \beta\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t}\right) \tag{3.50}$$

$$\mathbf{S}_N^{-1} = \mathbf{S}_0^{-1} + \beta\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}. \tag{3.51}$$

Note that because the posterior distribution is Gaussian, its mode coincides with its mean. Thus the maximum posterior weight vector is simply given by $\mathbf{w}_{\mathrm{MAP}} = \mathbf{m}_N$. If we consider an infinitely broad prior $\mathbf{S}_0 = \alpha^{-1}\mathbf{I}$ with $\alpha \to 0$, the mean $\mathbf{m}_N$ of the posterior distribution reduces to the maximum likelihood value $\mathbf{w}_{\mathrm{ML}}$ given by (3.15). Similarly, if $N = 0$, then the posterior distribution reverts to the prior. Furthermore, if data points arrive sequentially, then the posterior distribution at any stage acts as the prior distribution for the subsequent data point, such that the new posterior distribution is again given by (3.49).

Exercise 3.8

For the remainder of this chapter, we shall consider a particular form of Gaussian prior in order to simplify the treatment. Specifically, we consider a zero-mean isotropic Gaussian governed by a single precision parameter α so that

$$p(\mathbf{w}|\alpha) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I}) \tag{3.52}$$

and the corresponding posterior distribution over $\mathbf{w}$ is then given by (3.49) with

$$\mathbf{m}_N = \beta\mathbf{S}_N\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t} \tag{3.53}$$

$$\mathbf{S}_N^{-1} = \alpha\mathbf{I} + \beta\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}. \tag{3.54}$$

The log of the posterior distribution is given by the sum of the log likelihood and the log of the prior and, as a function of $\mathbf{w}$, takes the form

$$\ln p(\mathbf{w}|\mathbf{t}) = -\frac{\beta}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\}^2 - \frac{\alpha}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} + \mathrm{const}. \tag{3.55}$$

Maximization of this posterior distribution with respect to $\mathbf{w}$ is therefore equivalent to the minimization of the sum-of-squares error function with the addition of a quadratic regularization term, corresponding to (3.27) with $\lambda = \alpha/\beta$.

We can illustrate Bayesian learning in a linear basis function model, as well as the sequential update of a posterior distribution, using a simple example involving straight-line fitting. Consider a single input variable x, a single target variable t and

a linear model of the form $y(x, \mathbf{w}) = w_0 + w_1 x$. Because this has just two adaptive parameters, we can plot the prior and posterior distributions directly in parameter space. We generate synthetic data from the function $f(x, \mathbf{a}) = a_0 + a_1 x$ with parameter values $a_0 = -0.3$ and $a_1 = 0.5$ by first choosing values of x_n from the uniform distribution $\mathrm{U}(x|-1, 1)$, then evaluating $f(x_n, \mathbf{a})$, and finally adding Gaussian noise with standard deviation of 0.2 to obtain the target values t_n. Our goal is to recover the values of a_0 and a_1 from such data, and we will explore the dependence on the size of the data set. We assume here that the noise variance is known and hence we set the precision parameter to its true value $\beta = (1/0.2)^2 = 25$. Similarly, we fix the parameter α to 2.0. We shall shortly discuss strategies for determining α and β from the training data. Figure 3.7 shows the results of Bayesian learning in this model as the size of the data set is increased and demonstrates the sequential nature of Bayesian learning in which the current posterior distribution forms the prior when a new data point is observed. It is worth taking time to study this figure in detail as it illustrates several important aspects of Bayesian inference. The first row of this figure corresponds to the situation before any data points are observed and shows a plot of the prior distribution in $\mathbf{w}$ space together with six samples of the function $y(x, \mathbf{w})$ in which the values of $\mathbf{w}$ are drawn from the prior. In the second row, we see the situation after observing a single data point. The location (x, t) of the data point is shown by a blue circle in the right-hand column. In the left-hand column is a plot of the likelihood function $p(t|x, \mathbf{w})$ for this data point as a function of $\mathbf{w}$. Note that the likelihood function provides a soft constraint that the line must pass close to the data point, where close is determined by the noise precision β. For comparison, the true parameter values $a_0 = -0.3$ and $a_1 = 0.5$ used to generate the data set are shown by a white cross in the plots in the left column of Figure 3.7. When we multiply this likelihood function by the prior from the top row, and normalize, we obtain the posterior distribution shown in the middle plot on the second row. Samples of the regression function $y(x, \mathbf{w})$ obtained by drawing samples of $\mathbf{w}$ from this posterior distribution are shown in the right-hand plot. Note that these sample lines all pass close to the data point. The third row of this figure shows the effect of observing a second data point, again shown by a blue circle in the plot in the right-hand column. The corresponding likelihood function for this second data point alone is shown in the left plot. When we multiply this likelihood function by the posterior distribution from the second row, we obtain the posterior distribution shown in the middle plot of the third row. Note that this is exactly the same posterior distribution as would be obtained by combining the original prior with the likelihood function for the two data points. This posterior has now been influenced by two data points, and because two points are sufficient to define a line this already gives a relatively compact posterior distribution. Samples from this posterior distribution give rise to the functions shown in red in the third column, and we see that these functions pass close to both of the data points. The fourth row shows the effect of observing a total of 20 data points. The left-hand plot shows the likelihood function for the 20th data point alone, and the middle plot shows the resulting posterior distribution that has now absorbed information from all 20 observations. Note how the posterior is much sharper than in the third row. In the limit of an infinite number of data points, the

Figure 3.7 Illustration of sequential Bayesian learning for a simple linear model of the form $y(x, \mathbf{w}) = w_0 + w_1 x$. A detailed description of this figure is given in the text.

posterior distribution would become a delta function centred on the true parameter values, shown by the white cross.

Other forms of prior over the parameters can be considered. For instance, we can generalize the Gaussian prior to give

$$p(\mathbf{w}|\alpha) = \left[\frac{q}{2}\left(\frac{\alpha}{2}\right)^{1/q}\frac{1}{\Gamma(1/q)}\right]^M \exp\left(-\frac{\alpha}{2}\sum_{j=1}^{M}|w_j|^q\right) \tag{3.56}$$

in which $q = 2$ corresponds to the Gaussian distribution, and only in this case is the prior conjugate to the likelihood function (3.10). Finding the maximum of the posterior distribution over $\mathbf{w}$ corresponds to minimization of the regularized error function (3.29). In the case of the Gaussian prior, the mode of the posterior distribution was equal to the mean, although this will no longer hold if $q \neq 2$.

3.3.2 Predictive distribution

In practice, we are not usually interested in the value of $\mathbf{w}$ itself but rather in making predictions of t for new values of $\mathbf{x}$. This requires that we evaluate the *predictive distribution* defined by

$$p(t|\mathbf{t}, \alpha, \beta) = \int p(t|\mathbf{w}, \beta)p(\mathbf{w}|\mathbf{t}, \alpha, \beta)\,\mathrm{d}\mathbf{w} \tag{3.57}$$

in which $\mathbf{t}$ is the vector of target values from the training set, and we have omitted the corresponding input vectors from the right-hand side of the conditioning statements to simplify the notation. The conditional distribution $p(t|\mathbf{x}, \mathbf{w}, \beta)$ of the target variable is given by (3.8), and the posterior weight distribution is given by (3.49). We see that (3.57) involves the convolution of two Gaussian distributions, and so making use of the result (2.115) from Section 8.1.4, we see that the predictive distribution takes the form

Exercise 3.10

$$p(t|\mathbf{x}, \mathbf{t}, \alpha, \beta) = \mathcal{N}(t|\mathbf{m}_N^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}), \sigma_N^2(\mathbf{x})) \tag{3.58}$$

where the variance $\sigma_N^2(\mathbf{x})$ of the predictive distribution is given by

$$\sigma_N^2(\mathbf{x}) = \frac{1}{\beta} + \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}}\mathbf{S}_N\boldsymbol{\phi}(\mathbf{x}). \tag{3.59}$$

The first term in (3.59) represents the noise on the data whereas the second term reflects the uncertainty associated with the parameters $\mathbf{w}$. Because the noise process and the distribution of $\mathbf{w}$ are independent Gaussians, their variances are additive. Note that, as additional data points are observed, the posterior distribution becomes narrower. As a consequence it can be shown (Qazaz *et al.*, 1997) that $\sigma_{N+1}^2(\mathbf{x}) \leqslant \sigma_N^2(\mathbf{x})$. In the limit $N \to \infty$, the second term in (3.59) goes to zero, and the variance of the predictive distribution arises solely from the additive noise governed by the parameter β.

Exercise 3.11

As an illustration of the predictive distribution for Bayesian linear regression models, let us return to the synthetic sinusoidal data set of Section 1.1. In Figure 3.8,

Figure 3.8 Examples of the predictive distribution (3.58) for a model consisting of 9 Gaussian basis functions of the form (3.4) using the synthetic sinusoidal data set of Section 1.1. See the text for a detailed discussion.

we fit a model comprising a linear combination of Gaussian basis functions to data sets of various sizes and then look at the corresponding posterior distributions. Here the green curves correspond to the function $\sin(2\pi x)$ from which the data points were generated (with the addition of Gaussian noise). Data sets of size $N = 1$, $N = 2$, $N = 4$, and $N = 25$ are shown in the four plots by the blue circles. For each plot, the red curve shows the mean of the corresponding Gaussian predictive distribution, and the red shaded region spans one standard deviation either side of the mean. Note that the predictive uncertainty depends on x and is smallest in the neighbourhood of the data points. Also note that the level of uncertainty decreases as more data points are observed.

The plots in Figure 3.8 only show the point-wise predictive variance as a function of x. In order to gain insight into the covariance between the predictions at different values of x, we can draw samples from the posterior distribution over $\mathbf{w}$, and then plot the corresponding functions $y(x, \mathbf{w})$, as shown in Figure 3.9.

Figure 3.9 Plots of the function $y(x, \mathbf{w})$ using samples from the posterior distributions over $\mathbf{w}$ corresponding to the plots in Figure 3.8.

If we used localized basis functions such as Gaussians, then in regions away from the basis function centres, the contribution from the second term in the predictive variance (3.59) will go to zero, leaving only the noise contribution β^{-1}. Thus, the model becomes very confident in its predictions when extrapolating outside the region occupied by the basis functions, which is generally an undesirable behaviour. This problem can be avoided by adopting an alternative Bayesian approach to regression known as a Gaussian process.

Section 6.4

Note that, if both $\mathbf{w}$ and β are treated as unknown, then we can introduce a conjugate prior distribution $p(\mathbf{w}, \beta)$ that, from the discussion in Section 2.3.6, will be given by a Gaussian-gamma distribution (Denison *et al.*, 2002). In this case, the predictive distribution is a Student's t-distribution.

Exercise 3.12
Exercise 3.13

Figure 3.10 The equivalent kernel $k(x, x')$ for the Gaussian basis functions in Figure 3.1, shown as a plot of x versus x', together with three slices through this matrix corresponding to three different values of x. The data set used to generate this kernel comprised 200 values of x equally spaced over the interval $(-1, 1)$.

3.3.3 Equivalent kernel

The posterior mean solution (3.53) for the linear basis function model has an interesting interpretation that will set the stage for kernel methods, including Gaussian processes. If we substitute (3.53) into the expression (3.3), we see that the predictive mean can be written in the form

Chapter 6

$$y(\mathbf{x}, \mathbf{m}_N) = \mathbf{m}_N^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}) = \beta \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}} \mathbf{S}_N \boldsymbol{\Phi}^{\mathrm{T}} \mathbf{t} = \sum_{n=1}^{N} \beta \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}} \mathbf{S}_N \boldsymbol{\phi}(\mathbf{x}_n) t_n \tag{3.60}$$

where $\mathbf{S}_N$ is defined by (3.51). Thus the mean of the predictive distribution at a point $\mathbf{x}$ is given by a linear combination of the training set target variables t_n, so that we can write

$$y(\mathbf{x}, \mathbf{m}_N) = \sum_{n=1}^{N} k(\mathbf{x}, \mathbf{x}_n) t_n \tag{3.61}$$

where the function

$$k(\mathbf{x}, \mathbf{x}') = \beta \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}} \mathbf{S}_N \boldsymbol{\phi}(\mathbf{x}') \tag{3.62}$$

is known as the *smoother matrix* or the *equivalent kernel*. Regression functions, such as this, which make predictions by taking linear combinations of the training set target values are known as *linear smoothers*. Note that the equivalent kernel depends on the input values $\mathbf{x}_n$ from the data set because these appear in the definition of $\mathbf{S}_N$. The equivalent kernel is illustrated for the case of Gaussian basis functions in Figure 3.10 in which the kernel functions $k(x, x')$ have been plotted as a function of x' for three different values of x. We see that they are localized around x, and so the mean of the predictive distribution at x, given by $y(x, \mathbf{m}_N)$, is obtained by forming a weighted combination of the target values in which data points close to x are given higher weight than points further removed from x. Intuitively, it seems reasonable that we should weight local evidence more strongly than distant evidence. Note that this localization property holds not only for the localized Gaussian basis functions but also for the nonlocal polynomial and sigmoidal basis functions, as illustrated in Figure 3.11.

Figure 3.11 Examples of equivalent kernels $k(x, x')$ for $x = 0$ plotted as a function of x', corresponding (left) to the polynomial basis functions and (right) to the sigmoidal basis functions shown in Figure 3.1. Note that these are localized functions of x' even though the corresponding basis functions are nonlocal.

Further insight into the role of the equivalent kernel can be obtained by considering the covariance between $y(\mathbf{x})$ and $y(\mathbf{x}')$, which is given by

$$\begin{aligned} \operatorname{cov}[y(\mathbf{x}), y(\mathbf{x}')] &= \operatorname{cov}[\boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}}\mathbf{w}, \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}')] \\ &= \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}}\mathbf{S}_N\boldsymbol{\phi}(\mathbf{x}') = \beta^{-1}k(\mathbf{x}, \mathbf{x}') \end{aligned} \tag{3.63}$$

where we have made use of (3.49) and (3.62). From the form of the equivalent kernel, we see that the predictive mean at nearby points will be highly correlated, whereas for more distant pairs of points the correlation will be smaller.

The predictive distribution shown in Figure 3.8 allows us to visualize the pointwise uncertainty in the predictions, governed by (3.59). However, by drawing samples from the posterior distribution over $\mathbf{w}$, and plotting the corresponding model functions $y(\mathbf{x}, \mathbf{w})$ as in Figure 3.9, we are visualizing the joint uncertainty in the posterior distribution between the y values at two (or more) x values, as governed by the equivalent kernel.

The formulation of linear regression in terms of a kernel function suggests an alternative approach to regression as follows. Instead of introducing a set of basis functions, which implicitly determines an equivalent kernel, we can instead define a localized kernel directly and use this to make predictions for new input vectors $\mathbf{x}$, given the observed training set. This leads to a practical framework for regression (and classification) called *Gaussian processes*, which will be discussed in detail in Section 6.4.

We have seen that the effective kernel defines the weights by which the training set target values are combined in order to make a prediction at a new value of $\mathbf{x}$, and it can be shown that these weights sum to one, in other words

$$\sum_{n=1}^{N} k(\mathbf{x}, \mathbf{x}_n) = 1 \tag{3.64}$$

Exercise 3.14 for all values of $\mathbf{x}$. This intuitively pleasing result can easily be proven informally by noting that the summation is equivalent to considering the predictive mean $\widehat{y}(\mathbf{x})$ for a set of target data in which $t_n = 1$ for all n. Provided the basis functions are linearly independent, that there are more data points than basis functions, and that one of the basis functions is constant (corresponding to the bias parameter), then it is clear that we can fit the training data exactly and hence that the predictive mean will

be simply $\widehat{y}(\mathbf{x}) = 1$, from which we obtain (3.64). Note that the kernel function can be negative as well as positive, so although it satisfies a summation constraint, the corresponding predictions are not necessarily convex combinations of the training set target variables.

Finally, we note that the equivalent kernel (3.62) satisfies an important property shared by kernel functions in general, namely that it can be expressed in the form an inner product with respect to a vector $\boldsymbol{\psi}(\mathbf{x})$ of nonlinear functions, so that

Chapter 6

$$k(\mathbf{x}, \mathbf{z}) = \boldsymbol{\psi}(\mathbf{x})^{\mathrm{T}} \boldsymbol{\psi}(\mathbf{z}) \tag{3.65}$$

where $\boldsymbol{\psi}(\mathbf{x}) = \beta^{1/2} \mathbf{S}_N^{1/2} \boldsymbol{\phi}(\mathbf{x})$.

3.4. Bayesian Model Comparison

In Chapter 1, we highlighted the problem of over-fitting as well as the use of cross-validation as a technique for setting the values of regularization parameters or for choosing between alternative models. Here we consider the problem of model selection from a Bayesian perspective. In this section, our discussion will be very general, and then in Section 3.5 we shall see how these ideas can be applied to the determination of regularization parameters in linear regression.

As we shall see, the over-fitting associated with maximum likelihood can be avoided by marginalizing (summing or integrating) over the model parameters instead of making point estimates of their values. Models can then be compared directly on the training data, without the need for a validation set. This allows all available data to be used for training and avoids the multiple training runs for each model associated with cross-validation. It also allows multiple complexity parameters to be determined simultaneously as part of the training process. For example, in Chapter 7 we shall introduce the *relevance vector machine*, which is a Bayesian model having one complexity parameter for every training data point.

The Bayesian view of model comparison simply involves the use of probabilities to represent uncertainty in the choice of model, along with a consistent application of the sum and product rules of probability. Suppose we wish to compare a set of L models $\{\mathcal{M}_i\}$ where $i = 1, \ldots, L$. Here a model refers to a probability distribution over the observed data $\mathcal{D}$. In the case of the polynomial curve-fitting problem, the distribution is defined over the set of target values $\mathbf{t}$, while the set of input values $\mathbf{X}$ is assumed to be known. Other types of model define a joint distributions over $\mathbf{X}$ and $\mathbf{t}$. We shall suppose that the data is generated from one of these models but we are uncertain which one. Our uncertainty is expressed through a prior probability distribution $p(\mathcal{M}_i)$. Given a training set $\mathcal{D}$, we then wish to evaluate the posterior distribution

Section 1.5.4

$$p(\mathcal{M}_i|\mathcal{D}) \propto p(\mathcal{M}_i)p(\mathcal{D}|\mathcal{M}_i). \tag{3.66}$$

The prior allows us to express a preference for different models. Let us simply assume that all models are given equal prior probability. The interesting term is the *model evidence* $p(\mathcal{D}|\mathcal{M}_i)$ which expresses the preference shown by the data for

different models, and we shall examine this term in more detail shortly. The model evidence is sometimes also called the *marginal likelihood* because it can be viewed as a likelihood function over the space of models, in which the parameters have been marginalized out. The ratio of model evidences $p(\mathcal{D}|\mathcal{M}_i)/p(\mathcal{D}|\mathcal{M}_j)$ for two models is known as a *Bayes factor* (Kass and Raftery, 1995).

Once we know the posterior distribution over models, the predictive distribution is given, from the sum and product rules, by

$$p(t|\mathbf{x}, \mathcal{D}) = \sum_{i=1}^{L} p(t|\mathbf{x}, \mathcal{M}_i, \mathcal{D})p(\mathcal{M}_i|\mathcal{D}). \tag{3.67}$$

This is an example of a *mixture distribution* in which the overall predictive distribution is obtained by averaging the predictive distributions $p(t|\mathbf{x}, \mathcal{M}_i, \mathcal{D})$ of individual models, weighted by the posterior probabilities $p(\mathcal{M}_i|\mathcal{D})$ of those models. For instance, if we have two models that are a-posteriori equally likely and one predicts a narrow distribution around $t = a$ while the other predicts a narrow distribution around $t = b$, the overall predictive distribution will be a bimodal distribution with modes at $t = a$ and $t = b$, not a single model at $t = (a + b)/2$.

A simple approximation to model averaging is to use the single most probable model alone to make predictions. This is known as *model selection*.

For a model governed by a set of parameters $\mathbf{w}$, the model evidence is given, from the sum and product rules of probability, by

$$p(\mathcal{D}|\mathcal{M}_i) = \int p(\mathcal{D}|\mathbf{w}, \mathcal{M}_i)p(\mathbf{w}|\mathcal{M}_i)\,\mathrm{d}\mathbf{w}. \tag{3.68}$$

Chapter 11

From a sampling perspective, the marginal likelihood can be viewed as the probability of generating the data set $\mathcal{D}$ from a model whose parameters are sampled at random from the prior. It is also interesting to note that the evidence is precisely the normalizing term that appears in the denominator in Bayes' theorem when evaluating the posterior distribution over parameters because

$$p(\mathbf{w}|\mathcal{D}, \mathcal{M}_i) = \frac{p(\mathcal{D}|\mathbf{w}, \mathcal{M}_i)p(\mathbf{w}|\mathcal{M}_i)}{p(\mathcal{D}|\mathcal{M}_i)}. \tag{3.69}$$

We can obtain some insight into the model evidence by making a simple approximation to the integral over parameters. Consider first the case of a model having a single parameter w. The posterior distribution over parameters is proportional to $p(\mathcal{D}|w)p(w)$, where we omit the dependence on the model $\mathcal{M}_i$ to keep the notation uncluttered. If we assume that the posterior distribution is sharply peaked around the most probable value w_{MAP}, with width $\Delta w_{\mathrm{posterior}}$, then we can approximate the integral by the value of the integrand at its maximum times the width of the peak. If we further assume that the prior is flat with width $\Delta w_{\mathrm{prior}}$ so that $p(w) = 1/\Delta w_{\mathrm{prior}}$, then we have

$$p(\mathcal{D}) = \int p(\mathcal{D}|w)p(w)\,\mathrm{d}w \simeq p(\mathcal{D}|w_{\mathrm{MAP}})\frac{\Delta w_{\mathrm{posterior}}}{\Delta w_{\mathrm{prior}}} \tag{3.70}$$

Figure 3.12 We can obtain a rough approximation to the model evidence if we assume that the posterior distribution over parameters is sharply peaked around its mode w_{MAP}.

and so taking logs we obtain

$$\ln p(\mathcal{D}) \simeq \ln p(\mathcal{D}|w_{\text{MAP}}) + \ln\left(\frac{\Delta w_{\text{posterior}}}{\Delta w_{\text{prior}}}\right). \tag{3.71}$$

This approximation is illustrated in Figure 3.12. The first term represents the fit to the data given by the most probable parameter values, and for a flat prior this would correspond to the log likelihood. The second term penalizes the model according to its complexity. Because $\Delta w_{\text{posterior}} < \Delta w_{\text{prior}}$ this term is negative, and it increases in magnitude as the ratio $\Delta w_{\text{posterior}}/\Delta w_{\text{prior}}$ gets smaller. Thus, if parameters are finely tuned to the data in the posterior distribution, then the penalty term is large.

For a model having a set of M parameters, we can make a similar approximation for each parameter in turn. Assuming that all parameters have the same ratio of $\Delta w_{\text{posterior}}/\Delta w_{\text{prior}}$, we obtain

$$\ln p(\mathcal{D}) \simeq \ln p(\mathcal{D}|\mathbf{w}_{\text{MAP}}) + M \ln\left(\frac{\Delta w_{\text{posterior}}}{\Delta w_{\text{prior}}}\right). \tag{3.72}$$

Thus, in this very simple approximation, the size of the complexity penalty increases linearly with the number M of adaptive parameters in the model. As we increase the complexity of the model, the first term will typically decrease, because a more complex model is better able to fit the data, whereas the second term will increase due to the dependence on M. The optimal model complexity, as determined by the maximum evidence, will be given by a trade-off between these two competing terms. We shall later develop a more refined version of this approximation, based on a Gaussian approximation to the posterior distribution.

Section 4.4.1

We can gain further insight into Bayesian model comparison and understand how the marginal likelihood can favour models of intermediate complexity by considering Figure 3.13. Here the horizontal axis is a one-dimensional representation of the space of possible data sets, so that each point on this axis corresponds to a specific data set. We now consider three models $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$ of successively increasing complexity. Imagine running these models generatively to produce example data sets, and then looking at the distribution of data sets that result. Any given

Figure 3.13 Schematic illustration of the distribution of data sets for three models of different complexity, in which $\mathcal{M}_1$ is the simplest and $\mathcal{M}_3$ is the most complex. Note that the distributions are normalized. In this example, for the particular observed data set $\mathcal{D}_0$, the model $\mathcal{M}_2$ with intermediate complexity has the largest evidence.

model can generate a variety of different data sets since the parameters are governed by a prior probability distribution, and for any choice of the parameters there may be random noise on the target variables. To generate a particular data set from a specific model, we first choose the values of the parameters from their prior distribution $p(\mathbf{w})$, and then for these parameter values we sample the data from $p(\mathcal{D}|\mathbf{w})$. A simple model (for example, based on a first order polynomial) has little variability and so will generate data sets that are fairly similar to each other. Its distribution $p(\mathcal{D})$ is therefore confined to a relatively small region of the horizontal axis. By contrast, a complex model (such as a ninth order polynomial) can generate a great variety of different data sets, and so its distribution $p(\mathcal{D})$ is spread over a large region of the space of data sets. Because the distributions $p(\mathcal{D}|\mathcal{M}_i)$ are normalized, we see that the particular data set $\mathcal{D}_0$ can have the highest value of the evidence for the model of intermediate complexity. Essentially, the simpler model cannot fit the data well, whereas the more complex model spreads its predictive probability over too broad a range of data sets and so assigns relatively small probability to any one of them.

Implicit in the Bayesian model comparison framework is the assumption that the true distribution from which the data are generated is contained within the set of models under consideration. Provided this is so, we can show that Bayesian model comparison will on average favour the correct model. To see this, consider two models $\mathcal{M}_1$ and $\mathcal{M}_2$ in which the truth corresponds to $\mathcal{M}_1$. For a given finite data set, it is possible for the Bayes factor to be larger for the incorrect model. However, if we average the Bayes factor over the distribution of data sets, we obtain the expected Bayes factor in the form

$$\int p(\mathcal{D}|\mathcal{M}_1) \ln \frac{p(\mathcal{D}|\mathcal{M}_1)}{p(\mathcal{D}|\mathcal{M}_2)} \, \mathrm{d}\mathcal{D} \tag{3.73}$$

where the average has been taken with respect to the true distribution of the data. This quantity is an example of the *Kullback-Leibler* divergence and satisfies the property of always being positive unless the two distributions are equal in which case it is zero. Thus on average the Bayes factor will always favour the correct model.

Section 1.6.1

We have seen that the Bayesian framework avoids the problem of over-fitting and allows models to be compared on the basis of the training data alone. However,

a Bayesian approach, like any approach to pattern recognition, needs to make assumptions about the form of the model, and if these are invalid then the results can be misleading. In particular, we see from Figure 3.12 that the model evidence can be sensitive to many aspects of the prior, such as the behaviour in the tails. Indeed, the evidence is not defined if the prior is improper, as can be seen by noting that an improper prior has an arbitrary scaling factor (in other words, the normalization coefficient is not defined because the distribution cannot be normalized). If we consider a proper prior and then take a suitable limit in order to obtain an improper prior (for example, a Gaussian prior in which we take the limit of infinite variance) then the evidence will go to zero, as can be seen from (3.70) and Figure 3.12. It may, however, be possible to consider the evidence ratio between two models first and then take a limit to obtain a meaningful answer.

In a practical application, therefore, it will be wise to keep aside an independent test set of data on which to evaluate the overall performance of the final system.

3.5. The Evidence Approximation

In a fully Bayesian treatment of the linear basis function model, we would introduce prior distributions over the hyperparameters α and β and make predictions by marginalizing with respect to these hyperparameters as well as with respect to the parameters $\mathbf{w}$. However, although we can integrate analytically over either $\mathbf{w}$ or over the hyperparameters, the complete marginalization over all of these variables is analytically intractable. Here we discuss an approximation in which we set the hyperparameters to specific values determined by maximizing the *marginal likelihood function* obtained by first integrating over the parameters $\mathbf{w}$. This framework is known in the statistics literature as *empirical Bayes* (Bernardo and Smith, 1994; Gelman *et al.*, 2004), or *type 2 maximum likelihood* (Berger, 1985), or *generalized maximum likelihood* (Wahba, 1975), and in the machine learning literature is also called the *evidence approximation* (Gull, 1989; MacKay, 1992a).

If we introduce hyperpriors over α and β, the predictive distribution is obtained by marginalizing over $\mathbf{w}$, α and β so that

$$p(t|\mathbf{t}) = \iiint p(t|\mathbf{w},\beta)p(\mathbf{w}|\mathbf{t},\alpha,\beta)p(\alpha,\beta|\mathbf{t})\,\mathrm{d}\mathbf{w}\,\mathrm{d}\alpha\,\mathrm{d}\beta \tag{3.74}$$

where $p(t|\mathbf{w},\beta)$ is given by (3.8) and $p(\mathbf{w}|\mathbf{t},\alpha,\beta)$ is given by (3.49) with $\mathbf{m}_N$ and $\mathbf{S}_N$ defined by (3.53) and (3.54) respectively. Here we have omitted the dependence on the input variable $\mathbf{x}$ to keep the notation uncluttered. If the posterior distribution $p(\alpha,\beta|\mathbf{t})$ is sharply peaked around values $\widehat{\alpha}$ and $\widehat{\beta}$, then the predictive distribution is obtained simply by marginalizing over $\mathbf{w}$ in which α and β are fixed to the values $\widehat{\alpha}$ and $\widehat{\beta}$, so that

$$p(t|\mathbf{t}) \simeq p(t|\mathbf{t},\widehat{\alpha},\widehat{\beta}) = \int p(t|\mathbf{w},\widehat{\beta})p(\mathbf{w}|\mathbf{t},\widehat{\alpha},\widehat{\beta})\,\mathrm{d}\mathbf{w}. \tag{3.75}$$

From Bayes' theorem, the posterior distribution for α and β is given by

$$p(\alpha, \beta|\mathbf{t}) \propto p(\mathbf{t}|\alpha, \beta)p(\alpha, \beta). \tag{3.76}$$

If the prior is relatively flat, then in the evidence framework the values of $\widehat{\alpha}$ and $\widehat{\beta}$ are obtained by maximizing the marginal likelihood function $p(\mathbf{t}|\alpha, \beta)$. We shall proceed by evaluating the marginal likelihood for the linear basis function model and then finding its maxima. This will allow us to determine values for these hyperparameters from the training data alone, without recourse to cross-validation. Recall that the ratio α/β is analogous to a regularization parameter.

As an aside it is worth noting that, if we define conjugate (Gamma) prior distributions over α and β, then the marginalization over these hyperparameters in (3.74) can be performed analytically to give a Student's t-distribution over $\mathbf{w}$ (see Section 2.3.7). Although the resulting integral over $\mathbf{w}$ is no longer analytically tractable, it might be thought that approximating this integral, for example using the Laplace approximation discussed (Section 4.4) which is based on a local Gaussian approximation centred on the mode of the posterior distribution, might provide a practical alternative to the evidence framework (Buntine and Weigend, 1991). However, the integrand as a function of $\mathbf{w}$ typically has a strongly skewed mode so that the Laplace approximation fails to capture the bulk of the probability mass, leading to poorer results than those obtained by maximizing the evidence (MacKay, 1999).

Returning to the evidence framework, we note that there are two approaches that we can take to the maximization of the log evidence. We can evaluate the evidence function analytically and then set its derivative equal to zero to obtain re-estimation equations for α and β, which we shall do in Section 3.5.2. Alternatively we use a technique called the expectation maximization (EM) algorithm, which will be discussed in Section 9.3.4 where we shall also show that these two approaches converge to the same solution.

3.5.1 Evaluation of the evidence function

The marginal likelihood function $p(\mathbf{t}|\alpha, \beta)$ is obtained by integrating over the weight parameters $\mathbf{w}$, so that

$$p(\mathbf{t}|\alpha, \beta) = \int p(\mathbf{t}|\mathbf{w}, \beta)p(\mathbf{w}|\alpha)\,\mathrm{d}\mathbf{w}. \tag{3.77}$$

One way to evaluate this integral is to make use once again of the result (2.115) for the conditional distribution in a linear-Gaussian model. Here we shall evaluate the integral instead by completing the square in the exponent and making use of the standard form for the normalization coefficient of a Gaussian.

Exercise 3.16

From (3.11), (3.12), and (3.52), we can write the evidence function in the form

Exercise 3.17

$$p(\mathbf{t}|\alpha, \beta) = \left(\frac{\beta}{2\pi}\right)^{N/2} \left(\frac{\alpha}{2\pi}\right)^{M/2} \int \exp\left\{-E(\mathbf{w})\right\}\,\mathrm{d}\mathbf{w} \tag{3.78}$$

where M is the dimensionality of $\mathbf{w}$, and we have defined

$$\begin{aligned} E(\mathbf{w}) &= \beta E_D(\mathbf{w}) + \alpha E_W(\mathbf{w}) \\ &= \frac{\beta}{2}\|\mathbf{t} - \mathbf{\Phi}\mathbf{w}\|^2 + \frac{\alpha}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}. \end{aligned} \tag{3.79}$$

Exercise 3.18

We recognize (3.79) as being equal, up to a constant of proportionality, to the regularized sum-of-squares error function (3.27). We now complete the square over $\mathbf{w}$ giving

$$E(\mathbf{w}) = E(\mathbf{m}_N) + \frac{1}{2}(\mathbf{w} - \mathbf{m}_N)^{\mathrm{T}}\mathbf{A}(\mathbf{w} - \mathbf{m}_N) \tag{3.80}$$

where we have introduced

$$\mathbf{A} = \alpha\mathbf{I} + \beta\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi} \tag{3.81}$$

together with

$$E(\mathbf{m}_N) = \frac{\beta}{2}\|\mathbf{t} - \mathbf{\Phi}\mathbf{m}_N\|^2 + \frac{\alpha}{2}\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N. \tag{3.82}$$

Note that $\mathbf{A}$ corresponds to the matrix of second derivatives of the error function

$$\mathbf{A} = \nabla\nabla E(\mathbf{w}) \tag{3.83}$$

and is known as the *Hessian matrix*. Here we have also defined $\mathbf{m}_N$ given by

$$\mathbf{m}_N = \beta\mathbf{A}^{-1}\mathbf{\Phi}^{\mathrm{T}}\mathbf{t}. \tag{3.84}$$

Using (3.54), we see that $\mathbf{A} = \mathbf{S}_N^{-1}$, and hence (3.84) is equivalent to the previous definition (3.53), and therefore represents the mean of the posterior distribution.

Exercise 3.19

The integral over $\mathbf{w}$ can now be evaluated simply by appealing to the standard result for the normalization coefficient of a multivariate Gaussian, giving

$$\begin{aligned} &\int \exp\{-E(\mathbf{w})\}\,\mathrm{d}\mathbf{w} \\ &= \exp\{-E(\mathbf{m}_N)\}\int \exp\left\{-\frac{1}{2}(\mathbf{w} - \mathbf{m}_N)^{\mathrm{T}}\mathbf{A}(\mathbf{w} - \mathbf{m}_N)\right\}\mathrm{d}\mathbf{w} \\ &= \exp\{-E(\mathbf{m}_N)\}(2\pi)^{M/2}|\mathbf{A}|^{-1/2}. \end{aligned} \tag{3.85}$$

Using (3.78) we can then write the log of the marginal likelihood in the form

$$\ln p(\mathbf{t}|\alpha,\beta) = \frac{M}{2}\ln\alpha + \frac{N}{2}\ln\beta - E(\mathbf{m}_N) - \frac{1}{2}\ln|\mathbf{A}| - \frac{N}{2}\ln(2\pi) \tag{3.86}$$

which is the required expression for the evidence function.

Returning to the polynomial regression problem, we can plot the model evidence against the order of the polynomial, as shown in Figure 3.14. Here we have assumed a prior of the form (1.65) with the parameter α fixed at $\alpha = 5 \times 10^{-3}$. The form of this plot is very instructive. Referring back to Figure 1.4, we see that the $M = 0$ polynomial has very poor fit to the data and consequently gives a relatively low value

Figure 3.14 Plot of the model evidence versus the order M, for the polynomial regression model, showing that the evidence favours the model with $M = 3$.

for the evidence. Going to the $M = 1$ polynomial greatly improves the data fit, and hence the evidence is significantly higher. However, in going to $M = 2$, the data fit is improved only very marginally, due to the fact that the underlying sinusoidal function from which the data is generated is an odd function and so has no even terms in a polynomial expansion. Indeed, Figure 1.5 shows that the residual data error is reduced only slightly in going from $M = 1$ to $M = 2$. Because this richer model suffers a greater complexity penalty, the evidence actually falls in going from $M = 1$ to $M = 2$. When we go to $M = 3$ we obtain a significant further improvement in data fit, as seen in Figure 1.4, and so the evidence is increased again, giving the highest overall evidence for any of the polynomials. Further increases in the value of M produce only small improvements in the fit to the data but suffer increasing complexity penalty, leading overall to a decrease in the evidence values. Looking again at Figure 1.5, we see that the generalization error is roughly constant between $M = 3$ and $M = 8$, and it would be difficult to choose between these models on the basis of this plot alone. The evidence values, however, show a clear preference for $M = 3$, since this is the simplest model which gives a good explanation for the observed data.

3.5.2 Maximizing the evidence function

Let us first consider the maximization of $p(\mathbf{t}|\alpha, \beta)$ with respect to α. This can be done by first defining the following eigenvector equation

$$\left(\beta \mathbf{\Phi}^{\mathrm{T}} \mathbf{\Phi}\right) \mathbf{u}_i = \lambda_i \mathbf{u}_i. \tag{3.87}$$

From (3.81), it then follows that $\mathbf{A}$ has eigenvalues $\alpha + \lambda_i$. Now consider the derivative of the term involving $\ln |\mathbf{A}|$ in (3.86) with respect to α. We have

$$\frac{d}{d\alpha} \ln |\mathbf{A}| = \frac{d}{d\alpha} \ln \prod_i (\lambda_i + \alpha) = \frac{d}{d\alpha} \sum_i \ln(\lambda_i + \alpha) = \sum_i \frac{1}{\lambda_i + \alpha}. \tag{3.88}$$

Thus the stationary points of (3.86) with respect to α satisfy

$$0 = \frac{M}{2\alpha} - \frac{1}{2} \mathbf{m}_N^{\mathrm{T}} \mathbf{m}_N - \frac{1}{2} \sum_i \frac{1}{\lambda_i + \alpha}. \tag{3.89}$$

Multiplying through by 2α and rearranging, we obtain

$$\alpha \mathbf{m}_N^{\mathrm{T}} \mathbf{m}_N = M - \alpha \sum_i \frac{1}{\lambda_i + \alpha} = \gamma. \tag{3.90}$$

Since there are M terms in the sum over i, the quantity γ can be written

$$\gamma = \sum_i \frac{\lambda_i}{\alpha + \lambda_i}. \tag{3.91}$$

The interpretation of the quantity γ will be discussed shortly. From (3.90) we see that the value of α that maximizes the marginal likelihood satisfies

Exercise 3.20

$$\alpha = \frac{\gamma}{\mathbf{m}_N^{\mathrm{T}} \mathbf{m}_N}. \tag{3.92}$$

Note that this is an implicit solution for α not only because γ depends on α, but also because the mode $\mathbf{m}_N$ of the posterior distribution itself depends on the choice of α. We therefore adopt an iterative procedure in which we make an initial choice for α and use this to find $\mathbf{m}_N$, which is given by (3.53), and also to evaluate γ, which is given by (3.91). These values are then used to re-estimate α using (3.92), and the process repeated until convergence. Note that because the matrix $\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}$ is fixed, we can compute its eigenvalues once at the start and then simply multiply these by β to obtain the λ_i.

It should be emphasized that the value of α has been determined purely by looking at the training data. In contrast to maximum likelihood methods, no independent data set is required in order to optimize the model complexity.

We can similarly maximize the log marginal likelihood (3.86) with respect to β. To do this, we note that the eigenvalues λ_i defined by (3.87) are proportional to β, and hence $d\lambda_i/d\beta = \lambda_i/\beta$ giving

$$\frac{d}{d\beta} \ln |\mathbf{A}| = \frac{d}{d\beta} \sum_i \ln(\lambda_i + \alpha) = \frac{1}{\beta} \sum_i \frac{\lambda_i}{\lambda_i + \alpha} = \frac{\gamma}{\beta}. \tag{3.93}$$

The stationary point of the marginal likelihood therefore satisfies

$$0 = \frac{N}{2\beta} - \frac{1}{2} \sum_{n=1}^{N} \left\{ t_n - \mathbf{m}_N^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}_n) \right\}^2 - \frac{\gamma}{2\beta} \tag{3.94}$$

Exercise 3.22

and rearranging we obtain

$$\frac{1}{\beta} = \frac{1}{N - \gamma} \sum_{n=1}^{N} \left\{ t_n - \mathbf{m}_N^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}_n) \right\}^2. \tag{3.95}$$

Again, this is an implicit solution for β and can be solved by choosing an initial value for β and then using this to calculate $\mathbf{m}_N$ and γ and then re-estimate β using (3.95), repeating until convergence. If both α and β are to be determined from the data, then their values can be re-estimated together after each update of γ.

Figure 3.15 Contours of the likelihood function (red) and the prior (green) in which the axes in parameter space have been rotated to align with the eigenvectors $\mathbf{u}_i$ of the Hessian. For $\alpha = 0$, the mode of the posterior is given by the maximum likelihood solution $\mathbf{w}_{\mathrm{ML}}$, whereas for nonzero α the mode is at $\mathbf{w}_{\mathrm{MAP}} = \mathbf{m}_N$. In the direction w_1 the eigenvalue λ_1, defined by (3.87), is small compared with α and so the quantity $\lambda_1/(\lambda_1 + \alpha)$ is close to zero, and the corresponding MAP value of w_1 is also close to zero. By contrast, in the direction w_2 the eigenvalue λ_2 is large compared with α and so the quantity $\lambda_2/(\lambda_2+\alpha)$ is close to unity, and the MAP value of w_2 is close to its maximum likelihood value.

3.5.3 Effective number of parameters

The result (3.92) has an elegant interpretation (MacKay, 1992a), which provides insight into the Bayesian solution for α. To see this, consider the contours of the likelihood function and the prior as illustrated in Figure 3.15. Here we have implicitly transformed to a rotated set of axes in parameter space aligned with the eigenvectors $\mathbf{u}_i$ defined in (3.87). Contours of the likelihood function are then axis-aligned ellipses. The eigenvalues λ_i measure the curvature of the likelihood function, and so in Figure 3.15 the eigenvalue λ_1 is small compared with λ_2 (because a smaller curvature corresponds to a greater elongation of the contours of the likelihood function). Because $\beta\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi}$ is a positive definite matrix, it will have positive eigenvalues, and so the ratio $\lambda_i/(\lambda_i + \alpha)$ will lie between 0 and 1. Consequently, the quantity γ defined by (3.91) will lie in the range $0 \leqslant \gamma \leqslant M$. For directions in which $\lambda_i \gg \alpha$, the corresponding parameter w_i will be close to its maximum likelihood value, and the ratio $\lambda_i/(\lambda_i + \alpha)$ will be close to 1. Such parameters are called *well determined* because their values are tightly constrained by the data. Conversely, for directions in which $\lambda_i \ll \alpha$, the corresponding parameters w_i will be close to zero, as will the ratios $\lambda_i/(\lambda_i+\alpha)$. These are directions in which the likelihood function is relatively insensitive to the parameter value and so the parameter has been set to a small value by the prior. The quantity γ defined by (3.91) therefore measures the effective total number of well determined parameters.

We can obtain some insight into the result (3.95) for re-estimating β by comparing it with the corresponding maximum likelihood result given by (3.21). Both of these formulae express the variance (the inverse precision) as an average of the squared differences between the targets and the model predictions. However, they differ in that the number of data points N in the denominator of the maximum likelihood result is replaced by $N - \gamma$ in the Bayesian result. We recall from (1.56) that the maximum likelihood estimate of the variance for a Gaussian distribution over a

single variable x is given by

$$\sigma^2_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}(x_n - \mu_{\mathrm{ML}})^2 \tag{3.96}$$

and that this estimate is biased because the maximum likelihood solution μ_{ML} for the mean has fitted some of the noise on the data. In effect, this has used up one degree of freedom in the model. The corresponding unbiased estimate is given by (1.59) and takes the form

$$\sigma^2_{\mathrm{MAP}} = \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \mu_{\mathrm{ML}})^2. \tag{3.97}$$

We shall see in Section 10.1.3 that this result can be obtained from a Bayesian treatment in which we marginalize over the unknown mean. The factor of $N-1$ in the denominator of the Bayesian result takes account of the fact that one degree of freedom has been used in fitting the mean and removes the bias of maximum likelihood. Now consider the corresponding results for the linear regression model. The mean of the target distribution is now given by the function $\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})$, which contains M parameters. However, not all of these parameters are tuned to the data. The effective number of parameters that are determined by the data is γ, with the remaining $M-\gamma$ parameters set to small values by the prior. This is reflected in the Bayesian result for the variance that has a factor $N-\gamma$ in the denominator, thereby correcting for the bias of the maximum likelihood result.

We can illustrate the evidence framework for setting hyperparameters using the sinusoidal synthetic data set from Section 1.1, together with the Gaussian basis function model comprising 9 basis functions, so that the total number of parameters in the model is given by $M = 10$ including the bias. Here, for simplicity of illustration, we have set β to its true value of 11.1 and then used the evidence framework to determine α, as shown in Figure 3.16.

We can also see how the parameter α controls the magnitude of the parameters $\{w_i\}$, by plotting the individual parameters versus the effective number γ of parameters, as shown in Figure 3.17.

If we consider the limit $N \gg M$ in which the number of data points is large in relation to the number of parameters, then from (3.87) all of the parameters will be well determined by the data because $\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}$ involves an implicit sum over data points, and so the eigenvalues λ_i increase with the size of the data set. In this case, $\gamma = M$, and the re-estimation equations for α and β become

$$\alpha = \frac{M}{2E_W(\mathbf{m}_N)} \tag{3.98}$$

$$\beta = \frac{N}{2E_D(\mathbf{m}_N)} \tag{3.99}$$

where E_W and E_D are defined by (3.25) and (3.26), respectively. These results can be used as an easy-to-compute approximation to the full evidence re-estimation

Figure 3.16 The left plot shows γ (red curve) and $2\alpha E_W(\mathbf{m}_N)$ (blue curve) versus $\ln \alpha$ for the sinusoidal synthetic data set. It is the intersection of these two curves that defines the optimum value for α given by the evidence procedure. The right plot shows the corresponding graph of log evidence $\ln p(\mathbf{t}|\alpha, \beta)$ versus $\ln \alpha$ (red curve) showing that the peak coincides with the crossing point of the curves in the left plot. Also shown is the test set error (blue curve) showing that the evidence maximum occurs close to the point of best generalization.

formulae, because they do not require evaluation of the eigenvalue spectrum of the Hessian.

Figure 3.17 Plot of the 10 parameters w_i from the Gaussian basis function model versus the effective number of parameters γ, in which the hyperparameter α is varied in the range $0 \leqslant \alpha \leqslant \infty$ causing γ to vary in the range $0 \leqslant \gamma \leqslant M$.

3.6. Limitations of Fixed Basis Functions

Throughout this chapter, we have focussed on models comprising a linear combination of fixed, nonlinear basis functions. We have seen that the assumption of linearity in the parameters led to a range of useful properties including closed-form solutions to the least-squares problem, as well as a tractable Bayesian treatment. Furthermore, for a suitable choice of basis functions, we can model arbitrary nonlinearities in the

mapping from input variables to targets. In the next chapter, we shall study an analogous class of models for classification.

It might appear, therefore, that such linear models constitute a general purpose framework for solving problems in pattern recognition. Unfortunately, there are some significant shortcomings with linear models, which will cause us to turn in later chapters to more complex models such as support vector machines and neural networks.

The difficulty stems from the assumption that the basis functions $\phi_j(\mathbf{x})$ are fixed before the training data set is observed and is a manifestation of the curse of dimensionality discussed in Section 1.4. As a consequence, the number of basis functions needs to grow rapidly, often exponentially, with the dimensionality D of the input space.

Fortunately, there are two properties of real data sets that we can exploit to help alleviate this problem. First of all, the data vectors $\{\mathbf{x}_n\}$ typically lie close to a nonlinear manifold whose intrinsic dimensionality is smaller than that of the input space as a result of strong correlations between the input variables. We will see an example of this when we consider images of handwritten digits in Chapter 12. If we are using localized basis functions, we can arrange that they are scattered in input space only in regions containing data. This approach is used in radial basis function networks and also in support vector and relevance vector machines. Neural network models, which use adaptive basis functions having sigmoidal nonlinearities, can adapt the parameters so that the regions of input space over which the basis functions vary corresponds to the data manifold. The second property is that target variables may have significant dependence on only a small number of possible directions within the data manifold. Neural networks can exploit this property by choosing the directions in input space to which the basis functions respond.

Exercises

3.1 ($\star$) **www** Show that the 'tanh' function and the logistic sigmoid function (3.6) are related by

$$\tanh(a) = 2\sigma(2a) - 1. \tag{3.100}$$

Hence show that a general linear combination of logistic sigmoid functions of the form

$$y(x, \mathbf{w}) = w_0 + \sum_{j=1}^{M} w_j \sigma\left(\frac{x - \mu_j}{s}\right) \tag{3.101}$$

is equivalent to a linear combination of 'tanh' functions of the form

$$y(x, \mathbf{u}) = u_0 + \sum_{j=1}^{M} u_j \tanh\left(\frac{x - \mu_j}{s}\right) \tag{3.102}$$

and find expressions to relate the new parameters $\{u_1, \ldots, u_M\}$ to the original parameters $\{w_1, \ldots, w_M\}$.

3.2 ($\star\star$) Show that the matrix

$$\boldsymbol{\Phi}(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^{\mathrm{T}} \tag{3.103}$$

takes any vector $\mathbf{v}$ and projects it onto the space spanned by the columns of $\boldsymbol{\Phi}$. Use this result to show that the least-squares solution (3.15) corresponds to an orthogonal projection of the vector $\mathbf{t}$ onto the manifold $\mathcal{S}$ as shown in Figure 3.2.

3.3 ($\star$) Consider a data set in which each data point t_n is associated with a weighting factor $r_n > 0$, so that the sum-of-squares error function becomes

$$E_D(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} r_n \left\{t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)\right\}^2 . \tag{3.104}$$

Find an expression for the solution $\mathbf{w}^\star$ that minimizes this error function. Give two alternative interpretations of the weighted sum-of-squares error function in terms of (i) data dependent noise variance and (ii) replicated data points.

3.4 ($\star$) **WWW** Consider a linear model of the form

$$y(x, \mathbf{w}) = w_0 + \sum_{i=1}^{D} w_i x_i \tag{3.105}$$

together with a sum-of-squares error function of the form

$$E_D(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} \{y(x_n, \mathbf{w}) - t_n\}^2 . \tag{3.106}$$

Now suppose that Gaussian noise ϵ_i with zero mean and variance σ^2 is added independently to each of the input variables x_i. By making use of $\mathbb{E}[\epsilon_i] = 0$ and $\mathbb{E}[\epsilon_i\epsilon_j] = \delta_{ij}\sigma^2$, show that minimizing E_D averaged over the noise distribution is equivalent to minimizing the sum-of-squares error for noise-free input variables with the addition of a weight-decay regularization term, in which the bias parameter w_0 is omitted from the regularizer.

3.5 ($\star$) **WWW** Using the technique of Lagrange multipliers, discussed in Appendix E, show that minimization of the regularized error function (3.29) is equivalent to minimizing the unregularized sum-of-squares error (3.12) subject to the constraint (3.30). Discuss the relationship between the parameters η and λ.

3.6 ($\star$) **WWW** Consider a linear basis function regression model for a multivariate target variable $\mathbf{t}$ having a Gaussian distribution of the form

$$p(\mathbf{t}|\mathbf{W}, \boldsymbol{\Sigma}) = \mathcal{N}(\mathbf{t}|\mathbf{y}(\mathbf{x}, \mathbf{W}), \boldsymbol{\Sigma}) \tag{3.107}$$

where

$$\mathbf{y}(\mathbf{x}, \mathbf{W}) = \mathbf{W}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}) \tag{3.108}$$

together with a training data set comprising input basis vectors $\boldsymbol{\phi}(\mathbf{x}_n)$ and corresponding target vectors $\mathbf{t}_n$, with $n = 1, \ldots, N$. Show that the maximum likelihood solution $\mathbf{W}_{\mathrm{ML}}$ for the parameter matrix $\mathbf{W}$ has the property that each column is given by an expression of the form (3.15), which was the solution for an isotropic noise distribution. Note that this is independent of the covariance matrix $\boldsymbol{\Sigma}$. Show that the maximum likelihood solution for $\boldsymbol{\Sigma}$ is given by

$$\boldsymbol{\Sigma} = \frac{1}{N} \sum_{n=1}^{N} \left(\mathbf{t}_n - \mathbf{W}_{\mathrm{ML}}^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}_n)\right) \left(\mathbf{t}_n - \mathbf{W}_{\mathrm{ML}}^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}_n)\right)^{\mathrm{T}}. \tag{3.109}$$

3.7 ($\star$) By using the technique of completing the square, verify the result (3.49) for the posterior distribution of the parameters $\mathbf{w}$ in the linear basis function model in which $\mathbf{m}_N$ and $\mathbf{S}_N$ are defined by (3.50) and (3.51) respectively.

3.8 ($\star\star$) **www** Consider the linear basis function model in Section 3.1, and suppose that we have already observed N data points, so that the posterior distribution over $\mathbf{w}$ is given by (3.49). This posterior can be regarded as the prior for the next observation. By considering an additional data point $(\mathbf{x}_{N+1}, t_{N+1})$, and by completing the square in the exponential, show that the resulting posterior distribution is again given by (3.49) but with $\mathbf{S}_N$ replaced by $\mathbf{S}_{N+1}$ and $\mathbf{m}_N$ replaced by $\mathbf{m}_{N+1}$.

3.9 ($\star\star$) Repeat the previous exercise but instead of completing the square by hand, make use of the general result for linear-Gaussian models given by (2.116).

3.10 ($\star\star$) **www** By making use of the result (2.115) to evaluate the integral in (3.57), verify that the predictive distribution for the Bayesian linear regression model is given by (3.58) in which the input-dependent variance is given by (3.59).

3.11 ($\star\star$) We have seen that, as the size of a data set increases, the uncertainty associated with the posterior distribution over model parameters decreases. Make use of the matrix identity (Appendix C)

$$\left(\mathbf{M} + \mathbf{v}\mathbf{v}^{\mathrm{T}}\right)^{-1} = \mathbf{M}^{-1} - \frac{\left(\mathbf{M}^{-1}\mathbf{v}\right)\left(\mathbf{v}^{\mathrm{T}}\mathbf{M}^{-1}\right)}{1 + \mathbf{v}^{\mathrm{T}}\mathbf{M}^{-1}\mathbf{v}} \tag{3.110}$$

to show that the uncertainty $\sigma_N^2(\mathbf{x})$ associated with the linear regression function given by (3.59) satisfies

$$\sigma_{N+1}^2(\mathbf{x}) \leqslant \sigma_N^2(\mathbf{x}). \tag{3.111}$$

3.12 ($\star\star$) We saw in Section 2.3.6 that the conjugate prior for a Gaussian distribution with unknown mean and unknown precision (inverse variance) is a normal-gamma distribution. This property also holds for the case of the conditional Gaussian distribution $p(t|\mathbf{x}, \mathbf{w}, \beta)$ of the linear regression model. If we consider the likelihood function (3.10), then the conjugate prior for $\mathbf{w}$ and β is given by

$$p(\mathbf{w}, \beta) = \mathcal{N}(\mathbf{w}|\mathbf{m}_0, \beta^{-1}\mathbf{S}_0)\mathrm{Gam}(\beta|a_0, b_0). \tag{3.112}$$

Show that the corresponding posterior distribution takes the same functional form, so that

$$p(\mathbf{w}, \beta|\mathbf{t}) = \mathcal{N}(\mathbf{w}|\mathbf{m}_N, \beta^{-1}\mathbf{S}_N)\mathrm{Gam}(\beta|a_N, b_N) \tag{3.113}$$

and find expressions for the posterior parameters $\mathbf{m}_N$, $\mathbf{S}_N$, a_N, and b_N.

3.13 ($\star\star$) Show that the predictive distribution $p(t|\mathbf{x}, \mathbf{t})$ for the model discussed in Exercise 3.12 is given by a Student's t-distribution of the form

$$p(t|\mathbf{x}, \mathbf{t}) = \mathrm{St}(t|\mu, \lambda, \nu) \tag{3.114}$$

and obtain expressions for μ, λ and ν.

3.14 ($\star\star$) In this exercise, we explore in more detail the properties of the equivalent kernel defined by (3.62), where $\mathbf{S}_N$ is defined by (3.54). Suppose that the basis functions $\phi_j(\mathbf{x})$ are linearly independent and that the number N of data points is greater than the number M of basis functions. Furthermore, let one of the basis functions be constant, say $\phi_0(\mathbf{x}) = 1$. By taking suitable linear combinations of these basis functions, we can construct a new basis set $\psi_j(\mathbf{x})$ spanning the same space but that are orthonormal, so that

$$\sum_{n=1}^{N} \psi_j(\mathbf{x}_n)\psi_k(\mathbf{x}_n) = I_{jk} \tag{3.115}$$

where I_{jk} is defined to be 1 if $j = k$ and 0 otherwise, and we take $\psi_0(\mathbf{x}) = 1$. Show that for $\alpha = 0$, the equivalent kernel can be written as $k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\psi}(\mathbf{x})^{\mathrm{T}}\boldsymbol{\psi}(\mathbf{x}')$ where $\boldsymbol{\psi} = (\psi_1, \ldots, \psi_M)^{\mathrm{T}}$. Use this result to show that the kernel satisfies the summation constraint

$$\sum_{n=1}^{N} k(\mathbf{x}, \mathbf{x}_n) = 1. \tag{3.116}$$

3.15 ($\star$) **WWW** Consider a linear basis function model for regression in which the parameters α and β are set using the evidence framework. Show that the function $E(\mathbf{m}_N)$ defined by (3.82) satisfies the relation $2E(\mathbf{m}_N) = N$.

3.16 ($\star\star$) Derive the result (3.86) for the log evidence function $p(\mathbf{t}|\alpha, \beta)$ of the linear regression model by making use of (2.115) to evaluate the integral (3.77) directly.

3.17 ($\star$) Show that the evidence function for the Bayesian linear regression model can be written in the form (3.78) in which $E(\mathbf{w})$ is defined by (3.79).

3.18 ($\star\star$) **WWW** By completing the square over $\mathbf{w}$, show that the error function (3.79) in Bayesian linear regression can be written in the form (3.80).

3.19 ($\star\star$) Show that the integration over $\mathbf{w}$ in the Bayesian linear regression model gives the result (3.85). Hence show that the log marginal likelihood is given by (3.86).

3.20 ($\star\star$) **www** Starting from (3.86) verify all of the steps needed to show that maximization of the log marginal likelihood function (3.86) with respect to α leads to the re-estimation equation (3.92).

3.21 ($\star\star$) An alternative way to derive the result (3.92) for the optimal value of α in the evidence framework is to make use of the identity

$$\frac{d}{d\alpha}\ln|\mathbf{A}| = \mathrm{Tr}\left(\mathbf{A}^{-1}\frac{d}{d\alpha}\mathbf{A}\right). \tag{3.117}$$

Prove this identity by considering the eigenvalue expansion of a real, symmetric matrix $\mathbf{A}$, and making use of the standard results for the determinant and trace of $\mathbf{A}$ expressed in terms of its eigenvalues (Appendix C). Then make use of (3.117) to derive (3.92) starting from (3.86).

3.22 ($\star\star$) Starting from (3.86) verify all of the steps needed to show that maximization of the log marginal likelihood function (3.86) with respect to β leads to the re-estimation equation (3.95).

3.23 ($\star\star$) **www** Show that the marginal probability of the data, in other words the model evidence, for the model described in Exercise 3.12 is given by

$$p(\mathbf{t}) = \frac{1}{(2\pi)^{N/2}}\frac{b_0^{a_0}}{b_N^{a_N}}\frac{\Gamma(a_N)}{\Gamma(a_0)}\frac{|\mathbf{S}_N|^{1/2}}{|\mathbf{S}_0|^{1/2}} \tag{3.118}$$

by first marginalizing with respect to $\mathbf{w}$ and then with respect to β.

3.24 ($\star\star$) Repeat the previous exercise but now use Bayes' theorem in the form

$$p(\mathbf{t}) = \frac{p(\mathbf{t}|\mathbf{w},\beta)p(\mathbf{w},\beta)}{p(\mathbf{w},\beta|\mathbf{t})} \tag{3.119}$$

and then substitute for the prior and posterior distributions and the likelihood function in order to derive the result (3.118).

4 Linear Models for Classification

In the previous chapter, we explored a class of regression models having particularly simple analytical and computational properties. We now discuss an analogous class of models for solving classification problems. The goal in classification is to take an input vector $\mathbf{x}$ and to assign it to one of K discrete classes $\mathcal{C}_k$ where $k = 1, \ldots, K$. In the most common scenario, the classes are taken to be disjoint, so that each input is assigned to one and only one class. The input space is thereby divided into *decision regions* whose boundaries are called *decision boundaries* or *decision surfaces*. In this chapter, we consider linear models for classification, by which we mean that the decision surfaces are linear functions of the input vector $\mathbf{x}$ and hence are defined by $(D-1)$-dimensional hyperplanes within the D-dimensional input space. Data sets whose classes can be separated exactly by linear decision surfaces are said to be *linearly separable*.

For regression problems, the target variable $\mathbf{t}$ was simply the vector of real numbers whose values we wish to predict. In the case of classification, there are various

ways of using target values to represent class labels. For probabilistic models, the most convenient, in the case of two-class problems, is the binary representation in which there is a single target variable $t \in \{0, 1\}$ such that $t = 1$ represents class $\mathcal{C}_1$ and $t = 0$ represents class $\mathcal{C}_2$. We can interpret the value of t as the probability that the class is $\mathcal{C}_1$, with the values of probability taking only the extreme values of 0 and 1. For $K > 2$ classes, it is convenient to use a 1-of-K coding scheme in which $\mathbf{t}$ is a vector of length K such that if the class is $\mathcal{C}_j$, then all elements t_k of $\mathbf{t}$ are zero except element t_j, which takes the value 1. For instance, if we have $K = 5$ classes, then a pattern from class 2 would be given the target vector

$$\mathbf{t} = (0, 1, 0, 0, 0)^{\mathrm{T}}. \tag{4.1}$$

Again, we can interpret the value of t_k as the probability that the class is $\mathcal{C}_k$. For nonprobabilistic models, alternative choices of target variable representation will sometimes prove convenient.

In Chapter 1, we identified three distinct approaches to the classification problem. The simplest involves constructing a *discriminant function* that directly assigns each vector $\mathbf{x}$ to a specific class. A more powerful approach, however, models the conditional probability distribution $p(\mathcal{C}_k|\mathbf{x})$ in an inference stage, and then subsequently uses this distribution to make optimal decisions. By separating inference and decision, we gain numerous benefits, as discussed in Section 1.5.4. There are two different approaches to determining the conditional probabilities $p(\mathcal{C}_k|\mathbf{x})$. One technique is to model them directly, for example by representing them as parametric models and then optimizing the parameters using a training set. Alternatively, we can adopt a generative approach in which we model the class-conditional densities given by $p(\mathbf{x}|\mathcal{C}_k)$, together with the prior probabilities $p(\mathcal{C}_k)$ for the classes, and then we compute the required posterior probabilities using Bayes' theorem

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})}. \tag{4.2}$$

We shall discuss examples of all three approaches in this chapter.

In the linear regression models considered in Chapter 3, the model prediction $y(\mathbf{x}, \mathbf{w})$ was given by a linear function of the parameters $\mathbf{w}$. In the simplest case, the model is also linear in the input variables and therefore takes the form $y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0$, so that y is a real number. For classification problems, however, we wish to predict discrete class labels, or more generally posterior probabilities that lie in the range $(0, 1)$. To achieve this, we consider a generalization of this model in which we transform the linear function of $\mathbf{w}$ using a nonlinear function $f(\,\cdot\,)$ so that

$$y(\mathbf{x}) = f\left(\mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0\right). \tag{4.3}$$

In the machine learning literature $f(\,\cdot\,)$ is known as an *activation function*, whereas its inverse is called a *link function* in the statistics literature. The decision surfaces correspond to $y(\mathbf{x}) = \text{constant}$, so that $\mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0 = \text{constant}$ and hence the decision surfaces are linear functions of $\mathbf{x}$, even if the function $f(\cdot)$ is nonlinear. For this reason, the class of models described by (4.3) are called *generalized linear models*

(McCullagh and Nelder, 1989). Note, however, that in contrast to the models used for regression, they are no longer linear in the parameters due to the presence of the nonlinear function $f(\cdot)$. This will lead to more complex analytical and computational properties than for linear regression models. Nevertheless, these models are still relatively simple compared to the more general nonlinear models that will be studied in subsequent chapters.

The algorithms discussed in this chapter will be equally applicable if we first make a fixed nonlinear transformation of the input variables using a vector of basis functions $\boldsymbol{\phi}(\mathbf{x})$ as we did for regression models in Chapter 3. We begin by considering classification directly in the original input space $\mathbf{x}$, while in Section 4.3 we shall find it convenient to switch to a notation involving basis functions for consistency with later chapters.

4.1. Discriminant Functions

A discriminant is a function that takes an input vector $\mathbf{x}$ and assigns it to one of K classes, denoted $\mathcal{C}_k$. In this chapter, we shall restrict attention to *linear discriminants*, namely those for which the decision surfaces are hyperplanes. To simplify the discussion, we consider first the case of two classes and then investigate the extension to $K > 2$ classes.

4.1.1 Two classes

The simplest representation of a linear discriminant function is obtained by taking a linear function of the input vector so that

$$y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0 \tag{4.4}$$

where $\mathbf{w}$ is called a *weight vector*, and w_0 is a *bias* (not to be confused with bias in the statistical sense). The negative of the bias is sometimes called a *threshold*. An input vector $\mathbf{x}$ is assigned to class $\mathcal{C}_1$ if $y(\mathbf{x}) \geqslant 0$ and to class $\mathcal{C}_2$ otherwise. The corresponding decision boundary is therefore defined by the relation $y(\mathbf{x}) = 0$, which corresponds to a $(D-1)$-dimensional hyperplane within the D-dimensional input space. Consider two points $\mathbf{x}_{\mathrm{A}}$ and $\mathbf{x}_{\mathrm{B}}$ both of which lie on the decision surface. Because $y(\mathbf{x}_{\mathrm{A}}) = y(\mathbf{x}_{\mathrm{B}}) = 0$, we have $\mathbf{w}^{\mathrm{T}}(\mathbf{x}_{\mathrm{A}} - \mathbf{x}_{\mathrm{B}}) = 0$ and hence the vector $\mathbf{w}$ is orthogonal to every vector lying within the decision surface, and so $\mathbf{w}$ determines the orientation of the decision surface. Similarly, if $\mathbf{x}$ is a point on the decision surface, then $y(\mathbf{x}) = 0$, and so the normal distance from the origin to the decision surface is given by

$$\frac{\mathbf{w}^{\mathrm{T}}\mathbf{x}}{\|\mathbf{w}\|} = -\frac{w_0}{\|\mathbf{w}\|}. \tag{4.5}$$

We therefore see that the bias parameter w_0 determines the location of the decision surface. These properties are illustrated for the case of $D = 2$ in Figure 4.1.

Furthermore, we note that the value of $y(\mathbf{x})$ gives a signed measure of the perpendicular distance r of the point $\mathbf{x}$ from the decision surface. To see this, consider

Figure 4.1 Illustration of the geometry of a linear discriminant function in two dimensions. The decision surface, shown in red, is perpendicular to $\mathbf{w}$, and its displacement from the origin is controlled by the bias parameter w_0. Also, the signed orthogonal distance of a general point $\mathbf{x}$ from the decision surface is given by $y(\mathbf{x})/\|\mathbf{w}\|$.

an arbitrary point $\mathbf{x}$ and let $\mathbf{x}_\perp$ be its orthogonal projection onto the decision surface, so that

$$\mathbf{x} = \mathbf{x}_\perp + r\frac{\mathbf{w}}{\|\mathbf{w}\|}. \tag{4.6}$$

Multiplying both sides of this result by $\mathbf{w}^{\mathrm{T}}$ and adding w_0, and making use of $y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0$ and $y(\mathbf{x}_\perp) = \mathbf{w}^{\mathrm{T}}\mathbf{x}_\perp + w_0 = 0$, we have

$$r = \frac{y(\mathbf{x})}{\|\mathbf{w}\|}. \tag{4.7}$$

This result is illustrated in Figure 4.1.

As with the linear regression models in Chapter 3, it is sometimes convenient to use a more compact notation in which we introduce an additional dummy 'input' value $x_0 = 1$ and then define $\widetilde{\mathbf{w}} = (w_0, \mathbf{w})$ and $\widetilde{\mathbf{x}} = (x_0, \mathbf{x})$ so that

$$y(\mathbf{x}) = \widetilde{\mathbf{w}}^{\mathrm{T}}\widetilde{\mathbf{x}}. \tag{4.8}$$

In this case, the decision surfaces are D-dimensional hyperplanes passing through the origin of the $D+1$-dimensional expanded input space.

4.1.2 Multiple classes

Now consider the extension of linear discriminants to $K > 2$ classes. We might be tempted be to build a K-class discriminant by combining a number of two-class discriminant functions. However, this leads to some serious difficulties (Duda and Hart, 1973) as we now show.

Consider the use of $K-1$ classifiers each of which solves a two-class problem of separating points in a particular class $\mathcal{C}_k$ from points not in that class. This is known as a *one-versus-the-rest* classifier. The left-hand example in Figure 4.2 shows an

Figure 4.2 Attempting to construct a K class discriminant from a set of two class discriminants leads to ambiguous regions, shown in green. On the left is an example involving the use of two discriminants designed to distinguish points in class $\mathcal{C}_k$ from points not in class $\mathcal{C}_k$. On the right is an example involving three discriminant functions each of which is used to separate a pair of classes $\mathcal{C}_k$ and $\mathcal{C}_j$.

example involving three classes where this approach leads to regions of input space that are ambiguously classified.

An alternative is to introduce $K(K-1)/2$ binary discriminant functions, one for every possible pair of classes. This is known as a *one-versus-one* classifier. Each point is then classified according to a majority vote amongst the discriminant functions. However, this too runs into the problem of ambiguous regions, as illustrated in the right-hand diagram of Figure 4.2.

We can avoid these difficulties by considering a single K-class discriminant comprising K linear functions of the form

$$y_k(\mathbf{x}) = \mathbf{w}_k^{\mathrm{T}}\mathbf{x} + w_{k0} \tag{4.9}$$

and then assigning a point $\mathbf{x}$ to class $\mathcal{C}_k$ if $y_k(\mathbf{x}) > y_j(\mathbf{x})$ for all $j \neq k$. The decision boundary between class $\mathcal{C}_k$ and class $\mathcal{C}_j$ is therefore given by $y_k(\mathbf{x}) = y_j(\mathbf{x})$ and hence corresponds to a $(D-1)$-dimensional hyperplane defined by

$$(\mathbf{w}_k - \mathbf{w}_j)^{\mathrm{T}}\mathbf{x} + (w_{k0} - w_{j0}) = 0. \tag{4.10}$$

This has the same form as the decision boundary for the two-class case discussed in Section 4.1.1, and so analogous geometrical properties apply.

The decision regions of such a discriminant are always singly connected and convex. To see this, consider two points $\mathbf{x}_{\mathrm{A}}$ and $\mathbf{x}_{\mathrm{B}}$ both of which lie inside decision region $\mathcal{R}_k$, as illustrated in Figure 4.3. Any point $\widehat{\mathbf{x}}$ that lies on the line connecting $\mathbf{x}_{\mathrm{A}}$ and $\mathbf{x}_{\mathrm{B}}$ can be expressed in the form

$$\widehat{\mathbf{x}} = \lambda\mathbf{x}_{\mathrm{A}} + (1-\lambda)\mathbf{x}_{\mathrm{B}} \tag{4.11}$$

Figure 4.3 Illustration of the decision regions for a multiclass linear discriminant, with the decision boundaries shown in red. If two points $\mathbf{x}_\mathrm{A}$ and $\mathbf{x}_\mathrm{B}$ both lie inside the same decision region $\mathcal{R}_k$, then any point $\widehat{\mathbf{x}}$ that lies on the line connecting these two points must also lie in $\mathcal{R}_k$, and hence the decision region must be singly connected and convex.

where $0 \leqslant \lambda \leqslant 1$. From the linearity of the discriminant functions, it follows that

$$y_k(\widehat{\mathbf{x}}) = \lambda y_k(\mathbf{x}_\mathrm{A}) + (1-\lambda) y_k(\mathbf{x}_\mathrm{B}). \tag{4.12}$$

Because both $\mathbf{x}_\mathrm{A}$ and $\mathbf{x}_\mathrm{B}$ lie inside $\mathcal{R}_k$, it follows that $y_k(\mathbf{x}_\mathrm{A}) > y_j(\mathbf{x}_\mathrm{A})$, and $y_k(\mathbf{x}_\mathrm{B}) > y_j(\mathbf{x}_\mathrm{B})$, for all $j \neq k$, and hence $y_k(\widehat{\mathbf{x}}) > y_j(\widehat{\mathbf{x}})$, and so $\widehat{\mathbf{x}}$ also lies inside $\mathcal{R}_k$. Thus $\mathcal{R}_k$ is singly connected and convex.

Note that for two classes, we can either employ the formalism discussed here, based on two discriminant functions $y_1(\mathbf{x})$ and $y_2(\mathbf{x})$, or else use the simpler but equivalent formulation described in Section 4.1.1 based on a single discriminant function $y(\mathbf{x})$.

We now explore three approaches to learning the parameters of linear discriminant functions, based on least squares, Fisher's linear discriminant, and the perceptron algorithm.

4.1.3 Least squares for classification

In Chapter 3, we considered models that were linear functions of the parameters, and we saw that the minimization of a sum-of-squares error function led to a simple closed-form solution for the parameter values. It is therefore tempting to see if we can apply the same formalism to classification problems. Consider a general classification problem with K classes, with a 1-of-K binary coding scheme for the target vector $\mathbf{t}$. One justification for using least squares in such a context is that it approximates the conditional expectation $\mathbb{E}[\mathbf{t}|\mathbf{x}]$ of the target values given the input vector. For the binary coding scheme, this conditional expectation is given by the vector of posterior class probabilities. Unfortunately, however, these probabilities are typically approximated rather poorly, indeed the approximations can have values outside the range $(0, 1)$, due to the limited flexibility of a linear model as we shall see shortly.

Each class $\mathcal{C}_k$ is described by its own linear model so that

$$y_k(\mathbf{x}) = \mathbf{w}_k^\mathrm{T}\mathbf{x} + w_{k0} \tag{4.13}$$

where $k = 1, \ldots, K$. We can conveniently group these together using vector notation so that

$$\mathbf{y}(\mathbf{x}) = \widetilde{\mathbf{W}}^\mathrm{T}\widetilde{\mathbf{x}} \tag{4.14}$$

where $\widetilde{\mathbf{W}}$ is a matrix whose k^{th} column comprises the $D+1$-dimensional vector $\widetilde{\mathbf{w}}_k = (w_{k0}, \mathbf{w}_k^{\mathrm{T}})^{\mathrm{T}}$ and $\widetilde{\mathbf{x}}$ is the corresponding augmented input vector $(1, \mathbf{x}^{\mathrm{T}})^{\mathrm{T}}$ with a dummy input $x_0 = 1$. This representation was discussed in detail in Section 3.1. A new input $\mathbf{x}$ is then assigned to the class for which the output $y_k = \widetilde{\mathbf{w}}_k^{\mathrm{T}}\widetilde{\mathbf{x}}$ is largest.

We now determine the parameter matrix $\widetilde{\mathbf{W}}$ by minimizing a sum-of-squares error function, as we did for regression in Chapter 3. Consider a training data set $\{\mathbf{x}_n, \mathbf{t}_n\}$ where $n = 1, \ldots, N$, and define a matrix $\mathbf{T}$ whose n^{th} row is the vector $\mathbf{t}_n^{\mathrm{T}}$, together with a matrix $\widetilde{\mathbf{X}}$ whose n^{th} row is $\widetilde{\mathbf{x}}_n^{\mathrm{T}}$. The sum-of-squares error function can then be written as

$$E_D(\widetilde{\mathbf{W}}) = \frac{1}{2}\mathrm{Tr}\left\{(\widetilde{\mathbf{X}}\widetilde{\mathbf{W}} - \mathbf{T})^{\mathrm{T}}(\widetilde{\mathbf{X}}\widetilde{\mathbf{W}} - \mathbf{T})\right\}. \tag{4.15}$$

Setting the derivative with respect to $\widetilde{\mathbf{W}}$ to zero, and rearranging, we then obtain the solution for $\widetilde{\mathbf{W}}$ in the form

$$\widetilde{\mathbf{W}} = (\widetilde{\mathbf{X}}^{\mathrm{T}}\widetilde{\mathbf{X}})^{-1}\widetilde{\mathbf{X}}^{\mathrm{T}}\mathbf{T} = \widetilde{\mathbf{X}}^{\dagger}\mathbf{T} \tag{4.16}$$

where $\widetilde{\mathbf{X}}^{\dagger}$ is the pseudo-inverse of the matrix $\widetilde{\mathbf{X}}$, as discussed in Section 3.1.1. We then obtain the discriminant function in the form

$$\mathbf{y}(\mathbf{x}) = \widetilde{\mathbf{W}}^{\mathrm{T}}\widetilde{\mathbf{x}} = \mathbf{T}^{\mathrm{T}}\left(\widetilde{\mathbf{X}}^{\dagger}\right)^{\mathrm{T}}\widetilde{\mathbf{x}}. \tag{4.17}$$

An interesting property of least-squares solutions with multiple target variables is that if every target vector in the training set satisfies some linear constraint

$$\mathbf{a}^{\mathrm{T}}\mathbf{t}_n + b = 0 \tag{4.18}$$

for some constants $\mathbf{a}$ and b, then the model prediction for any value of $\mathbf{x}$ will satisfy the same constraint so that

Exercise 4.2

$$\mathbf{a}^{\mathrm{T}}\mathbf{y}(\mathbf{x}) + b = 0. \tag{4.19}$$

Thus if we use a 1-of-K coding scheme for K classes, then the predictions made by the model will have the property that the elements of $\mathbf{y}(\mathbf{x})$ will sum to 1 for any value of $\mathbf{x}$. However, this summation constraint alone is not sufficient to allow the model outputs to be interpreted as probabilities because they are not constrained to lie within the interval $(0, 1)$.

The least-squares approach gives an exact closed-form solution for the discriminant function parameters. However, even as a discriminant function (where we use it to make decisions directly and dispense with any probabilistic interpretation) it suffers from some severe problems. We have already seen that least-squares solutions lack robustness to outliers, and this applies equally to the classification application, as illustrated in Figure 4.4. Here we see that the additional data points in the right-hand figure produce a significant change in the location of the decision boundary, even though these point would be correctly classified by the original decision boundary in the left-hand figure. The sum-of-squares error function penalizes predictions that are 'too correct' in that they lie a long way on the correct side of the decision

Section 2.3.7

Figure 4.4 The left plot shows data from two classes, denoted by red crosses and blue circles, together with the decision boundary found by least squares (magenta curve) and also by the logistic regression model (green curve), which is discussed later in Section 4.3.2. The right-hand plot shows the corresponding results obtained when extra data points are added at the bottom left of the diagram, showing that least squares is highly sensitive to outliers, unlike logistic regression.

boundary. In Section 7.1.2, we shall consider several alternative error functions for classification and we shall see that they do not suffer from this difficulty.

However, problems with least squares can be more severe than simply lack of robustness, as illustrated in Figure 4.5. This shows a synthetic data set drawn from three classes in a two-dimensional input space (x_1, x_2), having the property that linear decision boundaries can give excellent separation between the classes. Indeed, the technique of logistic regression, described later in this chapter, gives a satisfactory solution as seen in the right-hand plot. However, the least-squares solution gives poor results, with only a small region of the input space assigned to the green class.

The failure of least squares should not surprise us when we recall that it corresponds to maximum likelihood under the assumption of a Gaussian conditional distribution, whereas binary target vectors clearly have a distribution that is far from Gaussian. By adopting more appropriate probabilistic models, we shall obtain classification techniques with much better properties than least squares. For the moment, however, we continue to explore alternative nonprobabilistic methods for setting the parameters in the linear classification models.

4.1.4 Fisher's linear discriminant

One way to view a linear classification model is in terms of dimensionality reduction. Consider first the case of two classes, and suppose we take the D-

Figure 4.5 Example of a synthetic data set comprising three classes, with training data points denoted in red ($\times$), green ($+$), and blue ($\circ$). Lines denote the decision boundaries, and the background colours denote the respective classes of the decision regions. On the left is the result of using a least-squares discriminant. We see that the region of input space assigned to the green class is too small and so most of the points from this class are misclassified. On the right is the result of using logistic regressions as described in Section 4.3.2 showing correct classification of the training data.

dimensional input vector $\mathbf{x}$ and project it down to one dimension using

$$y = \mathbf{w}^{\mathrm{T}}\mathbf{x}. \tag{4.20}$$

If we place a threshold on y and classify $y \geqslant -w_0$ as class $\mathcal{C}_1$, and otherwise class $\mathcal{C}_2$, then we obtain our standard linear classifier discussed in the previous section. In general, the projection onto one dimension leads to a considerable loss of information, and classes that are well separated in the original D-dimensional space may become strongly overlapping in one dimension. However, by adjusting the components of the weight vector $\mathbf{w}$, we can select a projection that maximizes the class separation. To begin with, consider a two-class problem in which there are N_1 points of class $\mathcal{C}_1$ and N_2 points of class $\mathcal{C}_2$, so that the mean vectors of the two classes are given by

$$\mathbf{m}_1 = \frac{1}{N_1}\sum_{n \in \mathcal{C}_1} \mathbf{x}_n, \qquad \mathbf{m}_2 = \frac{1}{N_2}\sum_{n \in \mathcal{C}_2} \mathbf{x}_n. \tag{4.21}$$

The simplest measure of the separation of the classes, when projected onto $\mathbf{w}$, is the separation of the projected class means. This suggests that we might choose $\mathbf{w}$ so as to maximize

$$m_2 - m_1 = \mathbf{w}^{\mathrm{T}}(\mathbf{m}_2 - \mathbf{m}_1) \tag{4.22}$$

where

$$m_k = \mathbf{w}^{\mathrm{T}}\mathbf{m}_k \tag{4.23}$$

Figure 4.6 The left plot shows samples from two classes (depicted in red and blue) along with the histograms resulting from projection onto the line joining the class means. Note that there is considerable class overlap in the projected space. The right plot shows the corresponding projection based on the Fisher linear discriminant, showing the greatly improved class separation.

is the mean of the projected data from class $\mathcal{C}_k$. However, this expression can be made arbitrarily large simply by increasing the magnitude of $\mathbf{w}$. To solve this problem, we could constrain $\mathbf{w}$ to have unit length, so that $\sum_i w_i^2 = 1$. Using a Lagrange multiplier to perform the constrained maximization, we then find that $\mathbf{w} \propto (\mathbf{m}_2 - \mathbf{m}_1)$. There is still a problem with this approach, however, as illustrated in Figure 4.6. This shows two classes that are well separated in the original two-dimensional space (x_1, x_2) but that have considerable overlap when projected onto the line joining their means. This difficulty arises from the strongly nondiagonal covariances of the class distributions. The idea proposed by Fisher is to maximize a function that will give a large separation between the projected class means while also giving a small variance within each class, thereby minimizing the class overlap.

Appendix E

Exercise 4.4

The projection formula (4.20) transforms the set of labelled data points in $\mathbf{x}$ into a labelled set in the one-dimensional space y. The within-class variance of the transformed data from class $\mathcal{C}_k$ is therefore given by

$$s_k^2 = \sum_{n \in \mathcal{C}_k} (y_n - m_k)^2 \tag{4.24}$$

where $y_n = \mathbf{w}^{\mathrm{T}}\mathbf{x}_n$. We can define the total within-class variance for the whole data set to be simply $s_1^2 + s_2^2$. The Fisher criterion is defined to be the ratio of the between-class variance to the within-class variance and is given by

$$J(\mathbf{w}) = \frac{(m_2 - m_1)^2}{s_1^2 + s_2^2}. \tag{4.25}$$

We can make the dependence on $\mathbf{w}$ explicit by using (4.20), (4.23), and (4.24) to rewrite the Fisher criterion in the form

Exercise 4.5

$$J(\mathbf{w}) = \frac{\mathbf{w}^{\mathrm{T}}\mathbf{S}_{\mathrm{B}}\mathbf{w}}{\mathbf{w}^{\mathrm{T}}\mathbf{S}_{\mathrm{W}}\mathbf{w}} \tag{4.26}$$

where $\mathbf{S}_{\mathrm{B}}$ is the *between-class* covariance matrix and is given by

$$\mathbf{S}_{\mathrm{B}} = (\mathbf{m}_2 - \mathbf{m}_1)(\mathbf{m}_2 - \mathbf{m}_1)^{\mathrm{T}} \tag{4.27}$$

and $\mathbf{S}_{\mathrm{W}}$ is the total *within-class* covariance matrix, given by

$$\mathbf{S}_{\mathrm{W}} = \sum_{n \in \mathcal{C}_1} (\mathbf{x}_n - \mathbf{m}_1)(\mathbf{x}_n - \mathbf{m}_1)^{\mathrm{T}} + \sum_{n \in \mathcal{C}_2} (\mathbf{x}_n - \mathbf{m}_2)(\mathbf{x}_n - \mathbf{m}_2)^{\mathrm{T}}. \tag{4.28}$$

Differentiating (4.26) with respect to $\mathbf{w}$, we find that $J(\mathbf{w})$ is maximized when

$$(\mathbf{w}^{\mathrm{T}}\mathbf{S}_{\mathrm{B}}\mathbf{w})\mathbf{S}_{\mathrm{W}}\mathbf{w} = (\mathbf{w}^{\mathrm{T}}\mathbf{S}_{\mathrm{W}}\mathbf{w})\mathbf{S}_{\mathrm{B}}\mathbf{w}. \tag{4.29}$$

From (4.27), we see that $\mathbf{S}_{\mathrm{B}}\mathbf{w}$ is always in the direction of $(\mathbf{m}_2 - \mathbf{m}_1)$. Furthermore, we do not care about the magnitude of $\mathbf{w}$, only its direction, and so we can drop the scalar factors $(\mathbf{w}^{\mathrm{T}}\mathbf{S}_{\mathrm{B}}\mathbf{w})$ and $(\mathbf{w}^{\mathrm{T}}\mathbf{S}_{\mathrm{W}}\mathbf{w})$. Multiplying both sides of (4.29) by $\mathbf{S}_{\mathrm{W}}^{-1}$ we then obtain

$$\mathbf{w} \propto \mathbf{S}_{\mathrm{W}}^{-1}(\mathbf{m}_2 - \mathbf{m}_1). \tag{4.30}$$

Note that if the within-class covariance is isotropic, so that $\mathbf{S}_{\mathrm{W}}$ is proportional to the unit matrix, we find that $\mathbf{w}$ is proportional to the difference of the class means, as discussed above.

The result (4.30) is known as *Fisher's linear discriminant*, although strictly it is not a discriminant but rather a specific choice of direction for projection of the data down to one dimension. However, the projected data can subsequently be used to construct a discriminant, by choosing a threshold y_0 so that we classify a new point as belonging to $\mathcal{C}_1$ if $y(\mathbf{x}) \geqslant y_0$ and classify it as belonging to $\mathcal{C}_2$ otherwise. For example, we can model the class-conditional densities $p(y|\mathcal{C}_k)$ using Gaussian distributions and then use the techniques of Section 1.2.4 to find the parameters of the Gaussian distributions by maximum likelihood. Having found Gaussian approximations to the projected classes, the formalism of Section 1.5.1 then gives an expression for the optimal threshold. Some justification for the Gaussian assumption comes from the central limit theorem by noting that $y = \mathbf{w}^{\mathrm{T}}\mathbf{x}$ is the sum of a set of random variables.

4.1.5 Relation to least squares

The least-squares approach to the determination of a linear discriminant was based on the goal of making the model predictions as close as possible to a set of target values. By contrast, the Fisher criterion was derived by requiring maximum class separation in the output space. It is interesting to see the relationship between these two approaches. In particular, we shall show that, for the two-class problem, the Fisher criterion can be obtained as a special case of least squares.

So far we have considered 1-of-K coding for the target values. If, however, we adopt a slightly different target coding scheme, then the least-squares solution for

the weights becomes equivalent to the Fisher solution (Duda and Hart, 1973). In particular, we shall take the targets for class $\mathcal{C}_1$ to be N/N_1, where N_1 is the number of patterns in class $\mathcal{C}_1$, and N is the total number of patterns. This target value approximates the reciprocal of the prior probability for class $\mathcal{C}_1$. For class $\mathcal{C}_2$, we shall take the targets to be $-N/N_2$, where N_2 is the number of patterns in class $\mathcal{C}_2$.

The sum-of-squares error function can be written

$$E = \frac{1}{2}\sum_{n=1}^{N}\left(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + w_0 - t_n\right)^2. \tag{4.31}$$

Setting the derivatives of E with respect to w_0 and $\mathbf{w}$ to zero, we obtain respectively

$$\sum_{n=1}^{N}\left(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + w_0 - t_n\right) = 0 \tag{4.32}$$

$$\sum_{n=1}^{N}\left(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + w_0 - t_n\right)\mathbf{x}_n = 0. \tag{4.33}$$

From (4.32), and making use of our choice of target coding scheme for the t_n, we obtain an expression for the bias in the form

$$w_0 = -\mathbf{w}^{\mathrm{T}}\mathbf{m} \tag{4.34}$$

where we have used

$$\sum_{n=1}^{N} t_n = N_1\frac{N}{N_1} - N_2\frac{N}{N_2} = 0 \tag{4.35}$$

and where $\mathbf{m}$ is the mean of the total data set and is given by

$$\mathbf{m} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n = \frac{1}{N}(N_1\mathbf{m}_1 + N_2\mathbf{m}_2). \tag{4.36}$$

After some straightforward algebra, and again making use of the choice of t_n, the second equation (4.33) becomes

Exercise 4.6

$$\left(\mathbf{S}_{\mathrm{W}} + \frac{N_1N_2}{N}\mathbf{S}_{\mathrm{B}}\right)\mathbf{w} = N(\mathbf{m}_1 - \mathbf{m}_2) \tag{4.37}$$

where $\mathbf{S}_{\mathrm{W}}$ is defined by (4.28), $\mathbf{S}_{\mathrm{B}}$ is defined by (4.27), and we have substituted for the bias using (4.34). Using (4.27), we note that $\mathbf{S}_{\mathrm{B}}\mathbf{w}$ is always in the direction of $(\mathbf{m}_2 - \mathbf{m}_1)$. Thus we can write

$$\mathbf{w} \propto \mathbf{S}_{\mathrm{W}}^{-1}(\mathbf{m}_2 - \mathbf{m}_1) \tag{4.38}$$

where we have ignored irrelevant scale factors. Thus the weight vector coincides with that found from the Fisher criterion. In addition, we have also found an expression for the bias value w_0 given by (4.34). This tells us that a new vector $\mathbf{x}$ should be classified as belonging to class $\mathcal{C}_1$ if $y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}(\mathbf{x} - \mathbf{m}) > 0$ and class $\mathcal{C}_2$ otherwise.

4.1.6 Fisher's discriminant for multiple classes

We now consider the generalization of the Fisher discriminant to $K > 2$ classes, and we shall assume that the dimensionality D of the input space is greater than the number K of classes. Next, we introduce $D' > 1$ linear 'features' $y_k = \mathbf{w}_k^{\mathrm{T}}\mathbf{x}$, where $k = 1, \ldots, D'$. These feature values can conveniently be grouped together to form a vector $\mathbf{y}$. Similarly, the weight vectors $\{\mathbf{w}_k\}$ can be considered to be the columns of a matrix $\mathbf{W}$, so that

$$\mathbf{y} = \mathbf{W}^{\mathrm{T}}\mathbf{x}. \tag{4.39}$$

Note that again we are not including any bias parameters in the definition of $\mathbf{y}$. The generalization of the within-class covariance matrix to the case of K classes follows from (4.28) to give

$$\mathbf{S}_{\mathrm{W}} = \sum_{k=1}^{K} \mathbf{S}_k \tag{4.40}$$

where

$$\mathbf{S}_k = \sum_{n \in \mathcal{C}_k} (\mathbf{x}_n - \mathbf{m}_k)(\mathbf{x}_n - \mathbf{m}_k)^{\mathrm{T}} \tag{4.41}$$

$$\mathbf{m}_k = \frac{1}{N_k} \sum_{n \in \mathcal{C}_k} \mathbf{x}_n \tag{4.42}$$

and N_k is the number of patterns in class $\mathcal{C}_k$. In order to find a generalization of the between-class covariance matrix, we follow Duda and Hart (1973) and consider first the total covariance matrix

$$\mathbf{S}_{\mathrm{T}} = \sum_{n=1}^{N} (\mathbf{x}_n - \mathbf{m})(\mathbf{x}_n - \mathbf{m})^{\mathrm{T}} \tag{4.43}$$

where $\mathbf{m}$ is the mean of the total data set

$$\mathbf{m} = \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n = \frac{1}{N} \sum_{k=1}^{K} N_k \mathbf{m}_k \tag{4.44}$$

and $N = \sum_k N_k$ is the total number of data points. The total covariance matrix can be decomposed into the sum of the within-class covariance matrix, given by (4.40) and (4.41), plus an additional matrix $\mathbf{S}_{\mathrm{B}}$, which we identify as a measure of the between-class covariance

$$\mathbf{S}_{\mathrm{T}} = \mathbf{S}_{\mathrm{W}} + \mathbf{S}_{\mathrm{B}} \tag{4.45}$$

where

$$\mathbf{S}_{\mathrm{B}} = \sum_{k=1}^{K} N_k (\mathbf{m}_k - \mathbf{m})(\mathbf{m}_k - \mathbf{m})^{\mathrm{T}}. \tag{4.46}$$

These covariance matrices have been defined in the original $\mathbf{x}$-space. We can now define similar matrices in the projected D'-dimensional $\mathbf{y}$-space

$$\mathbf{s}_{\mathrm{W}} = \sum_{k=1}^{K} \sum_{n \in \mathcal{C}_k} (\mathbf{y}_n - \boldsymbol{\mu}_k)(\mathbf{y}_n - \boldsymbol{\mu}_k)^{\mathrm{T}} \tag{4.47}$$

and

$$\mathbf{s}_{\mathrm{B}} = \sum_{k=1}^{K} N_k (\boldsymbol{\mu}_k - \boldsymbol{\mu})(\boldsymbol{\mu}_k - \boldsymbol{\mu})^{\mathrm{T}} \tag{4.48}$$

where

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n \in \mathcal{C}_k} \mathbf{y}_n, \qquad \boldsymbol{\mu} = \frac{1}{N} \sum_{k=1}^{K} N_k \boldsymbol{\mu}_k. \tag{4.49}$$

Again we wish to construct a scalar that is large when the between-class covariance is large and when the within-class covariance is small. There are now many possible choices of criterion (Fukunaga, 1990). One example is given by

$$J(\mathbf{W}) = \mathrm{Tr}\left\{\mathbf{s}_{\mathrm{W}}^{-1} \mathbf{s}_{\mathrm{B}}\right\}. \tag{4.50}$$

This criterion can then be rewritten as an explicit function of the projection matrix $\mathbf{W}$ in the form

$$J(\mathbf{w}) = \mathrm{Tr}\left\{(\mathbf{W}\mathbf{S}_{\mathrm{W}}\mathbf{W}^{\mathrm{T}})^{-1}(\mathbf{W}\mathbf{S}_{\mathrm{B}}\mathbf{W}^{\mathrm{T}})\right\}. \tag{4.51}$$

Maximization of such criteria is straightforward, though somewhat involved, and is discussed at length in Fukunaga (1990). The weight values are determined by those eigenvectors of $\mathbf{S}_{\mathrm{W}}^{-1}\mathbf{S}_{\mathrm{B}}$ that correspond to the D' largest eigenvalues.

There is one important result that is common to all such criteria, which is worth emphasizing. We first note from (4.46) that $\mathbf{S}_{\mathrm{B}}$ is composed of the sum of K matrices, each of which is an outer product of two vectors and therefore of rank 1. In addition, only $(K-1)$ of these matrices are independent as a result of the constraint (4.44). Thus, $\mathbf{S}_{\mathrm{B}}$ has rank at most equal to $(K-1)$ and so there are at most $(K-1)$ nonzero eigenvalues. This shows that the projection onto the $(K-1)$-dimensional subspace spanned by the eigenvectors of $\mathbf{S}_{\mathrm{B}}$ does not alter the value of $J(\mathbf{w})$, and so we are therefore unable to find more than $(K-1)$ linear 'features' by this means (Fukunaga, 1990).

4.1.7 The perceptron algorithm

Another example of a linear discriminant model is the perceptron of Rosenblatt (1962), which occupies an important place in the history of pattern recognition algorithms. It corresponds to a two-class model in which the input vector $\mathbf{x}$ is first transformed using a fixed nonlinear transformation to give a feature vector $\boldsymbol{\phi}(\mathbf{x})$, and this is then used to construct a generalized linear model of the form

$$y(\mathbf{x}) = f\left(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})\right) \tag{4.52}$$

where the nonlinear activation function $f(\cdot)$ is given by a step function of the form

$$f(a) = \begin{cases} +1, & a \geqslant 0 \\ -1, & a < 0. \end{cases} \tag{4.53}$$

The vector $\boldsymbol{\phi}(\mathbf{x})$ will typically include a bias component $\phi_0(\mathbf{x}) = 1$. In earlier discussions of two-class classification problems, we have focussed on a target coding scheme in which $t \in \{0, 1\}$, which is appropriate in the context of probabilistic models. For the perceptron, however, it is more convenient to use target values $t = +1$ for class $\mathcal{C}_1$ and $t = -1$ for class $\mathcal{C}_2$, which matches the choice of activation function.

The algorithm used to determine the parameters $\mathbf{w}$ of the perceptron can most easily be motivated by error function minimization. A natural choice of error function would be the total number of misclassified patterns. However, this does not lead to a simple learning algorithm because the error is a piecewise constant function of $\mathbf{w}$, with discontinuities wherever a change in $\mathbf{w}$ causes the decision boundary to move across one of the data points. Methods based on changing $\mathbf{w}$ using the gradient of the error function cannot then be applied, because the gradient is zero almost everywhere.

We therefore consider an alternative error function known as the *perceptron criterion*. To derive this, we note that we are seeking a weight vector $\mathbf{w}$ such that patterns $\mathbf{x}_n$ in class $\mathcal{C}_1$ will have $\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n) > 0$, whereas patterns $\mathbf{x}_n$ in class $\mathcal{C}_2$ have $\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n) < 0$. Using the $t \in \{-1, +1\}$ target coding scheme it follows that we would like all patterns to satisfy $\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)t_n > 0$. The perceptron criterion associates zero error with any pattern that is correctly classified, whereas for a misclassified pattern $\mathbf{x}_n$ it tries to minimize the quantity $-\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)t_n$. The perceptron criterion is therefore given by

$$E_{\mathrm{P}}(\mathbf{w}) = -\sum_{n \in \mathcal{M}} \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n t_n \tag{4.54}$$

Frank Rosenblatt
1928–1969

Rosenblatt's perceptron played an important role in the history of machine learning. Initially, Rosenblatt simulated the perceptron on an IBM 704 computer at Cornell in 1957, but by the early 1960s he had built special-purpose hardware that provided a direct, parallel implementation of perceptron learning. Many of his ideas were encapsulated in "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms" published in 1962. Rosenblatt's work was criticized by Marvin Minksy, whose objections were published in the book "Perceptrons", co-authored with Seymour Papert. This book was widely misinterpreted at the time as showing that neural networks were fatally flawed and could only learn solutions for linearly separable problems. In fact, it only proved such limitations in the case of single-layer networks such as the perceptron and merely conjectured (incorrectly) that they applied to more general network models. Unfortunately, however, this book contributed to the substantial decline in research funding for neural computing, a situation that was not reversed until the mid-1980s. Today, there are many hundreds, if not thousands, of applications of neural networks in widespread use, with examples in areas such as handwriting recognition and information retrieval being used routinely by millions of people.

where $\mathcal{M}$ denotes the set of all misclassified patterns. The contribution to the error associated with a particular misclassified pattern is a linear function of $\mathbf{w}$ in regions of $\mathbf{w}$ space where the pattern is misclassified and zero in regions where it is correctly classified. The total error function is therefore piecewise linear.

Section 3.1.3

We now apply the stochastic gradient descent algorithm to this error function. The change in the weight vector $\mathbf{w}$ is then given by

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_{\mathrm{P}}(\mathbf{w}) = \mathbf{w}^{(\tau)} + \eta \boldsymbol{\phi}_n t_n \tag{4.55}$$

where η is the learning rate parameter and τ is an integer that indexes the steps of the algorithm. Because the perceptron function $y(\mathbf{x}, \mathbf{w})$ is unchanged if we multiply $\mathbf{w}$ by a constant, we can set the learning rate parameter η equal to 1 without of generality. Note that, as the weight vector evolves during training, the set of patterns that are misclassified will change.

The perceptron learning algorithm has a simple interpretation, as follows. We cycle through the training patterns in turn, and for each pattern $\mathbf{x}_n$ we evaluate the perceptron function (4.52). If the pattern is correctly classified, then the weight vector remains unchanged, whereas if it is incorrectly classified, then for class $\mathcal{C}_1$ we add the vector $\boldsymbol{\phi}(\mathbf{x}_n)$ onto the current estimate of weight vector $\mathbf{w}$ while for class $\mathcal{C}_2$ we subtract the vector $\boldsymbol{\phi}(\mathbf{x}_n)$ from $\mathbf{w}$. The perceptron learning algorithm is illustrated in Figure 4.7.

If we consider the effect of a single update in the perceptron learning algorithm, we see that the contribution to the error from a misclassified pattern will be reduced because from (4.55) we have

$$-\mathbf{w}^{(\tau+1)\mathrm{T}} \boldsymbol{\phi}_n t_n = -\mathbf{w}^{(\tau)\mathrm{T}} \boldsymbol{\phi}_n t_n - (\boldsymbol{\phi}_n t_n)^{\mathrm{T}} \boldsymbol{\phi}_n t_n < -\mathbf{w}^{(\tau)\mathrm{T}} \boldsymbol{\phi}_n t_n \tag{4.56}$$

where we have set $\eta = 1$, and made use of $\|\boldsymbol{\phi}_n t_n\|^2 > 0$. Of course, this does not imply that the contribution to the error function from the other misclassified patterns will have been reduced. Furthermore, the change in weight vector may have caused some previously correctly classified patterns to become misclassified. Thus the perceptron learning rule is not guaranteed to reduce the total error function at each stage.

However, the *perceptron convergence theorem* states that if there exists an exact solution (in other words, if the training data set is linearly separable), then the perceptron learning algorithm is guaranteed to find an exact solution in a finite number of steps. Proofs of this theorem can be found for example in Rosenblatt (1962), Block (1962), Nilsson (1965), Minsky and Papert (1969), Hertz *et al.* (1991), and Bishop (1995a). Note, however, that the number of steps required to achieve convergence could still be substantial, and in practice, until convergence is achieved, we will not be able to distinguish between a nonseparable problem and one that is simply slow to converge.

Even when the data set is linearly separable, there may be many solutions, and which one is found will depend on the initialization of the parameters and on the order of presentation of the data points. Furthermore, for data sets that are not linearly separable, the perceptron learning algorithm will never converge.

Figure 4.7 Illustration of the convergence of the perceptron learning algorithm, showing data points from two classes (red and blue) in a two-dimensional feature space (ϕ_1, ϕ_2). The top left plot shows the initial parameter vector $\mathbf{w}$ shown as a black arrow together with the corresponding decision boundary (black line), in which the arrow points towards the decision region which classified as belonging to the red class. The data point circled in green is misclassified and so its feature vector is added to the current weight vector, giving the new decision boundary shown in the top right plot. The bottom left plot shows the next misclassified point to be considered, indicated by the green circle, and its feature vector is again added to the weight vector giving the decision boundary shown in the bottom right plot for which all data points are correctly classified.

Figure 4.8 Illustration of the Mark 1 perceptron hardware. The photograph on the left shows how the inputs were obtained using a simple camera system in which an input scene, in this case a printed character, was illuminated by powerful lights, and an image focussed onto a 20×20 array of cadmium sulphide photocells, giving a primitive 400 pixel image. The perceptron also had a patch board, shown in the middle photograph, which allowed different configurations of input features to be tried. Often these were wired up at random to demonstrate the ability of the perceptron to learn without the need for precise wiring, in contrast to a modern digital computer. The photograph on the right shows one of the racks of adaptive weights. Each weight was implemented using a rotary variable resistor, also called a potentiometer, driven by an electric motor thereby allowing the value of the weight to be adjusted automatically by the learning algorithm.

Aside from difficulties with the learning algorithm, the perceptron does not provide probabilistic outputs, nor does it generalize readily to $K > 2$ classes. The most important limitation, however, arises from the fact that (in common with all of the models discussed in this chapter and the previous one) it is based on linear combinations of fixed basis functions. More detailed discussions of the limitations of perceptrons can be found in Minsky and Papert (1969) and Bishop (1995a).

Analogue hardware implementations of the perceptron were built by Rosenblatt, based on motor-driven variable resistors to implement the adaptive parameters w_j. These are illustrated in Figure 4.8. The inputs were obtained from a simple camera system based on an array of photo-sensors, while the basis functions ϕ could be chosen in a variety of ways, for example based on simple fixed functions of randomly chosen subsets of pixels from the input image. Typical applications involved learning to discriminate simple shapes or characters.

At the same time that the perceptron was being developed, a closely related system called the *adaline*, which is short for 'adaptive linear element', was being explored by Widrow and co-workers. The functional form of the model was the same as for the perceptron, but a different approach to training was adopted (Widrow and Hoff, 1960; Widrow and Lehr, 1990).

4.2. Probabilistic Generative Models

We turn next to a probabilistic view of classification and show how models with linear decision boundaries arise from simple assumptions about the distribution of the data. In Section 1.5.4, we discussed the distinction between the discriminative and the generative approaches to classification. Here we shall adopt a generative

Figure 4.9 Plot of the logistic sigmoid function $\sigma(a)$ defined by (4.59), shown in red, together with the scaled probit function $\Phi(\lambda a)$, for $\lambda^2 = \pi/8$, shown in dashed blue, where $\Phi(a)$ is defined by (4.114). The scaling factor $\pi/8$ is chosen so that the derivatives of the two curves are equal for $a = 0$.

approach in which we model the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$, as well as the class priors $p(\mathcal{C}_k)$, and then use these to compute posterior probabilities $p(\mathcal{C}_k|\mathbf{x})$ through Bayes' theorem.

Consider first of all the case of two classes. The posterior probability for class $\mathcal{C}_1$ can be written as

$$\begin{aligned} p(\mathcal{C}_1|\mathbf{x}) &= \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1) + p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)} \\ &= \frac{1}{1+\exp(-a)} = \sigma(a) \end{aligned} \tag{4.57}$$

where we have defined

$$a = \ln \frac{p(\mathbf{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_2)p(\mathcal{C}_2)} \tag{4.58}$$

and $\sigma(a)$ is the *logistic sigmoid* function defined by

$$\sigma(a) = \frac{1}{1+\exp(-a)} \tag{4.59}$$

which is plotted in Figure 4.9. The term 'sigmoid' means S-shaped. This type of function is sometimes also called a 'squashing function' because it maps the whole real axis into a finite interval. The logistic sigmoid has been encountered already in earlier chapters and plays an important role in many classification algorithms. It satisfies the following symmetry property

$$\sigma(-a) = 1 - \sigma(a) \tag{4.60}$$

as is easily verified. The inverse of the logistic sigmoid is given by

$$a = \ln\left(\frac{\sigma}{1-\sigma}\right) \tag{4.61}$$

and is known as the *logit* function. It represents the log of the ratio of probabilities $\ln\left[p(\mathcal{C}_1|\mathbf{x})/p(\mathcal{C}_2|\mathbf{x})\right]$ for the two classes, also known as the *log odds*.

Note that in (4.57) we have simply rewritten the posterior probabilities in an equivalent form, and so the appearance of the logistic sigmoid may seem rather vacuous. However, it will have significance provided $a(\mathbf{x})$ takes a simple functional form. We shall shortly consider situations in which $a(\mathbf{x})$ is a linear function of $\mathbf{x}$, in which case the posterior probability is governed by a generalized linear model.

For the case of $K > 2$ classes, we have

$$\begin{aligned} p(\mathcal{C}_k|\mathbf{x}) &= \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{\sum_j p(\mathbf{x}|\mathcal{C}_j)p(\mathcal{C}_j)} \\ &= \frac{\exp(a_k)}{\sum_j \exp(a_j)} \end{aligned} \tag{4.62}$$

which is known as the *normalized exponential* and can be regarded as a multiclass generalization of the logistic sigmoid. Here the quantities a_k are defined by

$$a_k = \ln p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k). \tag{4.63}$$

The normalized exponential is also known as the *softmax function*, as it represents a smoothed version of the 'max' function because, if $a_k \gg a_j$ for all $j \neq k$, then $p(\mathcal{C}_k|\mathbf{x}) \simeq 1$, and $p(\mathcal{C}_j|\mathbf{x}) \simeq 0$.

We now investigate the consequences of choosing specific forms for the class-conditional densities, looking first at continuous input variables $\mathbf{x}$ and then discussing briefly the case of discrete inputs.

4.2.1 Continuous inputs

Let us assume that the class-conditional densities are Gaussian and then explore the resulting form for the posterior probabilities. To start with, we shall assume that all classes share the same covariance matrix. Thus the density for class $\mathcal{C}_k$ is given by

$$p(\mathbf{x}|\mathcal{C}_k) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} \exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_k)^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_k)\right\}. \tag{4.64}$$

Consider first the case of two classes. From (4.57) and (4.58), we have

$$p(\mathcal{C}_1|\mathbf{x}) = \sigma(\mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0) \tag{4.65}$$

where we have defined

$$\mathbf{w} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \tag{4.66}$$

$$w_0 = -\frac{1}{2}\boldsymbol{\mu}_1^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_1 + \frac{1}{2}\boldsymbol{\mu}_2^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_2 + \ln\frac{p(\mathcal{C}_1)}{p(\mathcal{C}_2)}. \tag{4.67}$$

We see that the quadratic terms in $\mathbf{x}$ from the exponents of the Gaussian densities have cancelled (due to the assumption of common covariance matrices) leading to a linear function of $\mathbf{x}$ in the argument of the logistic sigmoid. This result is illustrated for the case of a two-dimensional input space $\mathbf{x}$ in Figure 4.10. The resulting

Figure 4.10 The left-hand plot shows the class-conditional densities for two classes, denoted red and blue. On the right is the corresponding posterior probability $p(\mathcal{C}_1|\mathbf{x})$, which is given by a logistic sigmoid of a linear function of $\mathbf{x}$. The surface in the right-hand plot is coloured using a proportion of red ink given by $p(\mathcal{C}_1|\mathbf{x})$ and a proportion of blue ink given by $p(\mathcal{C}_2|\mathbf{x}) = 1 - p(\mathcal{C}_1|\mathbf{x})$.

decision boundaries correspond to surfaces along which the posterior probabilities $p(\mathcal{C}_k|\mathbf{x})$ are constant and so will be given by linear functions of $\mathbf{x}$, and therefore the decision boundaries are linear in input space. The prior probabilities $p(\mathcal{C}_k)$ enter only through the bias parameter w_0 so that changes in the priors have the effect of making parallel shifts of the decision boundary and more generally of the parallel contours of constant posterior probability.

For the general case of K classes we have, from (4.62) and (4.63),

$$a_k(\mathbf{x}) = \mathbf{w}_k^{\mathrm{T}}\mathbf{x} + w_{k0} \tag{4.68}$$

where we have defined

$$\mathbf{w}_k = \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_k \tag{4.69}$$

$$w_{k0} = -\frac{1}{2}\boldsymbol{\mu}_k^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_k + \ln p(\mathcal{C}_k). \tag{4.70}$$

We see that the $a_k(\mathbf{x})$ are again linear functions of $\mathbf{x}$ as a consequence of the cancellation of the quadratic terms due to the shared covariances. The resulting decision boundaries, corresponding to the minimum misclassification rate, will occur when two of the posterior probabilities (the two largest) are equal, and so will be defined by linear functions of $\mathbf{x}$, and so again we have a generalized linear model.

If we relax the assumption of a shared covariance matrix and allow each class-conditional density $p(\mathbf{x}|\mathcal{C}_k)$ to have its own covariance matrix $\boldsymbol{\Sigma}_k$, then the earlier cancellations will no longer occur, and we will obtain quadratic functions of $\mathbf{x}$, giving rise to a *quadratic discriminant*. The linear and quadratic decision boundaries are illustrated in Figure 4.11.

Figure 4.11 The left-hand plot shows the class-conditional densities for three classes each having a Gaussian distribution, coloured red, green, and blue, in which the red and green classes have the same covariance matrix. The right-hand plot shows the corresponding posterior probabilities, in which the RGB colour vector represents the posterior probabilities for the respective three classes. The decision boundaries are also shown. Notice that the boundary between the red and green classes, which have the same covariance matrix, is linear, whereas those between the other pairs of classes are quadratic.

4.2.2 Maximum likelihood solution

Once we have specified a parametric functional form for the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$, we can then determine the values of the parameters, together with the prior class probabilities $p(\mathcal{C}_k)$, using maximum likelihood. This requires a data set comprising observations of $\mathbf{x}$ along with their corresponding class labels.

Consider first the case of two classes, each having a Gaussian class-conditional density with a shared covariance matrix, and suppose we have a data set $\{\mathbf{x}_n, t_n\}$ where $n = 1, \ldots, N$. Here $t_n = 1$ denotes class $\mathcal{C}_1$ and $t_n = 0$ denotes class $\mathcal{C}_2$. We denote the prior class probability $p(\mathcal{C}_1) = \pi$, so that $p(\mathcal{C}_2) = 1 - \pi$. For a data point $\mathbf{x}_n$ from class $\mathcal{C}_1$, we have $t_n = 1$ and hence

$$p(\mathbf{x}_n, \mathcal{C}_1) = p(\mathcal{C}_1)p(\mathbf{x}_n|\mathcal{C}_1) = \pi\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_1, \boldsymbol{\Sigma}).$$

Similarly for class $\mathcal{C}_2$, we have $t_n = 0$ and hence

$$p(\mathbf{x}_n, \mathcal{C}_2) = p(\mathcal{C}_2)p(\mathbf{x}_n|\mathcal{C}_2) = (1 - \pi)\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_2, \boldsymbol{\Sigma}).$$

Thus the likelihood function is given by

$$p(\mathbf{t}|\pi, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \boldsymbol{\Sigma}) = \prod_{n=1}^{N} \left[\pi\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_1, \boldsymbol{\Sigma})\right]^{t_n} \left[(1 - \pi)\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_2, \boldsymbol{\Sigma})\right]^{1-t_n} \tag{4.71}$$

where $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$. As usual, it is convenient to maximize the log of the likelihood function. Consider first the maximization with respect to π. The terms in

the log likelihood function that depend on π are

$$\sum_{n=1}^{N} \{t_n \ln \pi + (1 - t_n) \ln(1 - \pi)\} . \tag{4.72}$$

Setting the derivative with respect to π equal to zero and rearranging, we obtain

$$\pi = \frac{1}{N} \sum_{n=1}^{N} t_n = \frac{N_1}{N} = \frac{N_1}{N_1 + N_2} \tag{4.73}$$

where N_1 denotes the total number of data points in class $\mathcal{C}_1$, and N_2 denotes the total number of data points in class $\mathcal{C}_2$. Thus the maximum likelihood estimate for π is simply the fraction of points in class $\mathcal{C}_1$ as expected. This result is easily generalized to the multiclass case where again the maximum likelihood estimate of the prior probability associated with class $\mathcal{C}_k$ is given by the fraction of the training set points assigned to that class.

Exercise 4.9

Now consider the maximization with respect to $\boldsymbol{\mu}_1$. Again we can pick out of the log likelihood function those terms that depend on $\boldsymbol{\mu}_1$ giving

$$\sum_{n=1}^{N} t_n \ln \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_1, \boldsymbol{\Sigma}) = -\frac{1}{2} \sum_{n=1}^{N} t_n (\mathbf{x}_n - \boldsymbol{\mu}_1)^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_1) + \text{const.} \tag{4.74}$$

Setting the derivative with respect to $\boldsymbol{\mu}_1$ to zero and rearranging, we obtain

$$\boldsymbol{\mu}_1 = \frac{1}{N_1} \sum_{n=1}^{N} t_n \mathbf{x}_n \tag{4.75}$$

which is simply the mean of all the input vectors $\mathbf{x}_n$ assigned to class $\mathcal{C}_1$. By a similar argument, the corresponding result for $\boldsymbol{\mu}_2$ is given by

$$\boldsymbol{\mu}_2 = \frac{1}{N_2} \sum_{n=1}^{N} (1 - t_n) \mathbf{x}_n \tag{4.76}$$

which again is the mean of all the input vectors $\mathbf{x}_n$ assigned to class $\mathcal{C}_2$.

Finally, consider the maximum likelihood solution for the shared covariance matrix $\boldsymbol{\Sigma}$. Picking out the terms in the log likelihood function that depend on $\boldsymbol{\Sigma}$, we have

$$\begin{aligned} &-\frac{1}{2} \sum_{n=1}^{N} t_n \ln |\boldsymbol{\Sigma}| - \frac{1}{2} \sum_{n=1}^{N} t_n (\mathbf{x}_n - \boldsymbol{\mu}_1)^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_1) \\ &-\frac{1}{2} \sum_{n=1}^{N} (1 - t_n) \ln |\boldsymbol{\Sigma}| - \frac{1}{2} \sum_{n=1}^{N} (1 - t_n) (\mathbf{x}_n - \boldsymbol{\mu}_2)^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_2) \\ &= -\frac{N}{2} \ln |\boldsymbol{\Sigma}| - \frac{N}{2} \mathrm{Tr} \left\{ \boldsymbol{\Sigma}^{-1} \mathbf{S} \right\} \end{aligned} \tag{4.77}$$

where we have defined

$$\mathbf{S} = \frac{N_1}{N}\mathbf{S}_1 + \frac{N_2}{N}\mathbf{S}_2 \tag{4.78}$$

$$\mathbf{S}_1 = \frac{1}{N_1}\sum_{n\in\mathcal{C}_1}(\mathbf{x}_n - \boldsymbol{\mu}_1)(\mathbf{x}_n - \boldsymbol{\mu}_1)^{\mathrm{T}} \tag{4.79}$$

$$\mathbf{S}_2 = \frac{1}{N_2}\sum_{n\in\mathcal{C}_2}(\mathbf{x}_n - \boldsymbol{\mu}_2)(\mathbf{x}_n - \boldsymbol{\mu}_2)^{\mathrm{T}}. \tag{4.80}$$

Using the standard result for the maximum likelihood solution for a Gaussian distribution, we see that $\boldsymbol{\Sigma} = \mathbf{S}$, which represents a weighted average of the covariance matrices associated with each of the two classes separately.

This result is easily extended to the K class problem to obtain the corresponding maximum likelihood solutions for the parameters in which each class-conditional density is Gaussian with a shared covariance matrix. Note that the approach of fitting Gaussian distributions to the classes is not robust to outliers, because the maximum likelihood estimation of a Gaussian is not robust.

Exercise 4.10

Section 2.3.7

4.2.3 Discrete features

Let us now consider the case of discrete feature values x_i. For simplicity, we begin by looking at binary feature values $x_i \in \{0, 1\}$ and discuss the extension to more general discrete features shortly. If there are D inputs, then a general distribution would correspond to a table of 2^D numbers for each class, containing $2^D - 1$ independent variables (due to the summation constraint). Because this grows exponentially with the number of features, we might seek a more restricted representation. Here we will make the *naive Bayes* assumption in which the feature values are treated as independent, conditioned on the class $\mathcal{C}_k$. Thus we have class-conditional distributions of the form

Section 8.2.2

$$p(\mathbf{x}|\mathcal{C}_k) = \prod_{i=1}^{D} \mu_{ki}^{x_i}(1 - \mu_{ki})^{1-x_i} \tag{4.81}$$

which contain D independent parameters for each class. Substituting into (4.63) then gives

$$a_k(\mathbf{x}) = \sum_{i=1}^{D}\{x_i \ln \mu_{ki} + (1 - x_i)\ln(1 - \mu_{ki})\} + \ln p(\mathcal{C}_k) \tag{4.82}$$

which again are linear functions of the input values x_i. For the case of $K = 2$ classes, we can alternatively consider the logistic sigmoid formulation given by (4.57). Analogous results are obtained for discrete variables each of which can take $M > 2$ states.

Exercise 4.11

4.2.4 Exponential family

As we have seen, for both Gaussian distributed and discrete inputs, the posterior class probabilities are given by generalized linear models with logistic sigmoid ($K =$

2 classes) or softmax ($K \geqslant 2$ classes) activation functions. These are particular cases of a more general result obtained by assuming that the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$ are members of the exponential family of distributions.

Using the form (2.194) for members of the exponential family, we see that the distribution of $\mathbf{x}$ can be written in the form

$$p(\mathbf{x}|\boldsymbol{\lambda}_k) = h(\mathbf{x})g(\boldsymbol{\lambda}_k)\exp\left\{\boldsymbol{\lambda}_k^{\mathrm{T}}\mathbf{u}(\mathbf{x})\right\}. \tag{4.83}$$

We now restrict attention to the subclass of such distributions for which $\mathbf{u}(\mathbf{x}) = \mathbf{x}$. Then we make use of (2.236) to introduce a scaling parameter s, so that we obtain the restricted set of exponential family class-conditional densities of the form

$$p(\mathbf{x}|\boldsymbol{\lambda}_k, s) = \frac{1}{s}h\left(\frac{1}{s}\mathbf{x}\right)g(\boldsymbol{\lambda}_k)\exp\left\{\frac{1}{s}\boldsymbol{\lambda}_k^{\mathrm{T}}\mathbf{x}\right\}. \tag{4.84}$$

Note that we are allowing each class to have its own parameter vector $\boldsymbol{\lambda}_k$ but we are assuming that the classes share the same scale parameter s.

For the two-class problem, we substitute this expression for the class-conditional densities into (4.58) and we see that the posterior class probability is again given by a logistic sigmoid acting on a linear function $a(\mathbf{x})$ which is given by

$$a(\mathbf{x}) = (\boldsymbol{\lambda}_1 - \boldsymbol{\lambda}_2)^{\mathrm{T}}\mathbf{x} + \ln g(\boldsymbol{\lambda}_1) - \ln g(\boldsymbol{\lambda}_2) + \ln p(\mathcal{C}_1) - \ln p(\mathcal{C}_2). \tag{4.85}$$

Similarly, for the K-class problem, we substitute the class-conditional density expression into (4.63) to give

$$a_k(\mathbf{x}) = \boldsymbol{\lambda}_k^{\mathrm{T}}\mathbf{x} + \ln g(\boldsymbol{\lambda}_k) + \ln p(\mathcal{C}_k) \tag{4.86}$$

and so again is a linear function of $\mathbf{x}$.

4.3. Probabilistic Discriminative Models

For the two-class classification problem, we have seen that the posterior probability of class $\mathcal{C}_1$ can be written as a logistic sigmoid acting on a linear function of $\mathbf{x}$, for a wide choice of class-conditional distributions $p(\mathbf{x}|\mathcal{C}_k)$. Similarly, for the multiclass case, the posterior probability of class $\mathcal{C}_k$ is given by a softmax transformation of a linear function of $\mathbf{x}$. For specific choices of the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$, we have used maximum likelihood to determine the parameters of the densities as well as the class priors $p(\mathcal{C}_k)$ and then used Bayes' theorem to find the posterior class probabilities.

However, an alternative approach is to use the functional form of the generalized linear model explicitly and to determine its parameters directly by using maximum likelihood. We shall see that there is an efficient algorithm finding such solutions known as *iterative reweighted least squares*, or *IRLS*.

The indirect approach to finding the parameters of a generalized linear model, by fitting class-conditional densities and class priors separately and then applying

Figure 4.12 Illustration of the role of nonlinear basis functions in linear classification models. The left plot shows the original input space (x_1, x_2) together with data points from two classes labelled red and blue. Two 'Gaussian' basis functions $\phi_1(\mathbf{x})$ and $\phi_2(\mathbf{x})$ are defined in this space with centres shown by the green crosses and with contours shown by the green circles. The right-hand plot shows the corresponding feature space (ϕ_1, ϕ_2) together with the linear decision boundary obtained given by a logistic regression model of the form discussed in Section 4.3.2. This corresponds to a nonlinear decision boundary in the original input space, shown by the black curve in the left-hand plot.

Bayes' theorem, represents an example of *generative* modelling, because we could take such a model and generate synthetic data by drawing values of $\mathbf{x}$ from the marginal distribution $p(\mathbf{x})$. In the direct approach, we are maximizing a likelihood function defined through the conditional distribution $p(\mathcal{C}_k|\mathbf{x})$, which represents a form of *discriminative* training. One advantage of the discriminative approach is that there will typically be fewer adaptive parameters to be determined, as we shall see shortly. It may also lead to improved predictive performance, particularly when the class-conditional density assumptions give a poor approximation to the true distributions.

4.3.1 Fixed basis functions

So far in this chapter, we have considered classification models that work directly with the original input vector $\mathbf{x}$. However, all of the algorithms are equally applicable if we first make a fixed nonlinear transformation of the inputs using a vector of basis functions $\boldsymbol{\phi}(\mathbf{x})$. The resulting decision boundaries will be linear in the feature space $\boldsymbol{\phi}$, and these correspond to nonlinear decision boundaries in the original $\mathbf{x}$ space, as illustrated in Figure 4.12. Classes that are linearly separable in the feature space $\boldsymbol{\phi}(\mathbf{x})$ need not be linearly separable in the original observation space $\mathbf{x}$. Note that as in our discussion of linear models for regression, one of the

basis functions is typically set to a constant, say $\phi_0(\mathbf{x}) = 1$, so that the corresponding parameter w_0 plays the role of a bias. For the remainder of this chapter, we shall include a fixed basis function transformation $\boldsymbol{\phi}(\mathbf{x})$, as this will highlight some useful similarities to the regression models discussed in Chapter 3.

For many problems of practical interest, there is significant overlap between the class-conditional densities $p(\mathbf{x}|\mathcal{C}_k)$. This corresponds to posterior probabilities $p(\mathcal{C}_k|\mathbf{x})$, which, for at least some values of $\mathbf{x}$, are not 0 or 1. In such cases, the optimal solution is obtained by modelling the posterior probabilities accurately and then applying standard decision theory, as discussed in Chapter 1. Note that nonlinear transformations $\boldsymbol{\phi}(\mathbf{x})$ cannot remove such class overlap. Indeed, they can increase the level of overlap, or create overlap where none existed in the original observation space. However, suitable choices of nonlinearity can make the process of modelling the posterior probabilities easier.

Section 3.6

Such fixed basis function models have important limitations, and these will be resolved in later chapters by allowing the basis functions themselves to adapt to the data. Notwithstanding these limitations, models with fixed nonlinear basis functions play an important role in applications, and a discussion of such models will introduce many of the key concepts needed for an understanding of their more complex counterparts.

4.3.2 Logistic regression

We begin our treatment of generalized linear models by considering the problem of two-class classification. In our discussion of generative approaches in Section 4.2, we saw that under rather general assumptions, the posterior probability of class $\mathcal{C}_1$ can be written as a logistic sigmoid acting on a linear function of the feature vector $\boldsymbol{\phi}$ so that

$$p(\mathcal{C}_1|\boldsymbol{\phi}) = y(\boldsymbol{\phi}) = \sigma\left(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}\right) \tag{4.87}$$

with $p(\mathcal{C}_2|\boldsymbol{\phi}) = 1 - p(\mathcal{C}_1|\boldsymbol{\phi})$. Here $\sigma(\cdot)$ is the *logistic sigmoid* function defined by (4.59). In the terminology of statistics, this model is known as *logistic regression*, although it should be emphasized that this is a model for classification rather than regression.

For an M-dimensional feature space $\boldsymbol{\phi}$, this model has M adjustable parameters. By contrast, if we had fitted Gaussian class conditional densities using maximum likelihood, we would have used $2M$ parameters for the means and $M(M+1)/2$ parameters for the (shared) covariance matrix. Together with the class prior $p(\mathcal{C}_1)$, this gives a total of $M(M+5)/2+1$ parameters, which grows quadratically with M, in contrast to the linear dependence on M of the number of parameters in logistic regression. For large values of M, there is a clear advantage in working with the logistic regression model directly.

We now use maximum likelihood to determine the parameters of the logistic regression model. To do this, we shall make use of the derivative of the logistic sigmoid function, which can conveniently be expressed in terms of the sigmoid function itself

Exercise 4.12

$$\frac{d\sigma}{da} = \sigma(1-\sigma). \tag{4.88}$$

For a data set $\{\boldsymbol{\phi}_n, t_n\}$, where $t_n \in \{0, 1\}$ and $\boldsymbol{\phi}_n = \boldsymbol{\phi}(\mathbf{x}_n)$, with $n = 1, \ldots, N$, the likelihood function can be written

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n} \{1 - y_n\}^{1-t_n} \tag{4.89}$$

where $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$ and $y_n = p(\mathcal{C}_1|\boldsymbol{\phi}_n)$. As usual, we can define an error function by taking the negative logarithm of the likelihood, which gives the *cross-entropy* error function in the form

$$E(\mathbf{w}) = -\ln p(\mathbf{t}|\mathbf{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\} \tag{4.90}$$

where $y_n = \sigma(a_n)$ and $a_n = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n$. Taking the gradient of the error function with respect to $\mathbf{w}$, we obtain

Exercise 4.13

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N} (y_n - t_n)\boldsymbol{\phi}_n \tag{4.91}$$

where we have made use of (4.88). We see that the factor involving the derivative of the logistic sigmoid has cancelled, leading to a simplified form for the gradient of the log likelihood. In particular, the contribution to the gradient from data point n is given by the 'error' $y_n - t_n$ between the target value and the prediction of the model, times the basis function vector $\boldsymbol{\phi}_n$. Furthermore, comparison with (3.13) shows that this takes precisely the same form as the gradient of the sum-of-squares error function for the linear regression model.

Section 3.1.1

If desired, we could make use of the result (4.91) to give a sequential algorithm in which patterns are presented one at a time, in which each of the weight vectors is updated using (3.22) in which ∇E_n is the nth term in (4.91).

It is worth noting that maximum likelihood can exhibit severe over-fitting for data sets that are linearly separable. This arises because the maximum likelihood solution occurs when the hyperplane corresponding to $\sigma = 0.5$, equivalent to $\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi} = 0$, separates the two classes and the magnitude of $\mathbf{w}$ goes to infinity. In this case, the logistic sigmoid function becomes infinitely steep in feature space, corresponding to a Heaviside step function, so that every training point from each class k is assigned a posterior probability $p(\mathcal{C}_k|\mathbf{x}) = 1$. Furthermore, there is typically a continuum of such solutions because any separating hyperplane will give rise to the same posterior probabilities at the training data points, as will be seen later in Figure 10.13. Maximum likelihood provides no way to favour one such solution over another, and which solution is found in practice will depend on the choice of optimization algorithm and on the parameter initialization. Note that the problem will arise even if the number of data points is large compared with the number of parameters in the model, so long as the training data set is linearly separable. The singularity can be avoided by inclusion of a prior and finding a MAP solution for $\mathbf{w}$, or equivalently by adding a regularization term to the error function.

Exercise 4.14

4.3.3 Iterative reweighted least squares

In the case of the linear regression models discussed in Chapter 3, the maximum likelihood solution, on the assumption of a Gaussian noise model, leads to a closed-form solution. This was a consequence of the quadratic dependence of the log likelihood function on the parameter vector $\mathbf{w}$. For logistic regression, there is no longer a closed-form solution, due to the nonlinearity of the logistic sigmoid function. However, the departure from a quadratic form is not substantial. To be precise, the error function is concave, as we shall see shortly, and hence has a unique minimum. Furthermore, the error function can be minimized by an efficient iterative technique based on the *Newton-Raphson* iterative optimization scheme, which uses a local quadratic approximation to the log likelihood function. The Newton-Raphson update, for minimizing a function $E(\mathbf{w})$, takes the form (Fletcher, 1987; Bishop and Nabney, 2008)

$$\mathbf{w}^{(\text{new})} = \mathbf{w}^{(\text{old})} - \mathbf{H}^{-1}\nabla E(\mathbf{w}). \tag{4.92}$$

where $\mathbf{H}$ is the Hessian matrix whose elements comprise the second derivatives of $E(\mathbf{w})$ with respect to the components of $\mathbf{w}$.

Let us first of all apply the Newton-Raphson method to the linear regression model (3.3) with the sum-of-squares error function (3.12). The gradient and Hessian of this error function are given by

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N}(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n - t_n)\boldsymbol{\phi}_n = \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\mathbf{w} - \boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t} \tag{4.93}$$

$$\mathbf{H} = \nabla\nabla E(\mathbf{w}) = \sum_{n=1}^{N}\boldsymbol{\phi}_n\boldsymbol{\phi}_n^{\mathrm{T}} = \boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} \tag{4.94}$$

Section 3.1.1

where $\boldsymbol{\Phi}$ is the $N \times M$ design matrix, whose n^{th} row is given by $\boldsymbol{\phi}_n^{\mathrm{T}}$. The Newton-Raphson update then takes the form

$$\begin{aligned}\mathbf{w}^{(\text{new})} &= \mathbf{w}^{(\text{old})} - (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi})^{-1}\left\{\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\mathbf{w}^{(\text{old})} - \boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t}\right\} \\ &= (\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t}\end{aligned} \tag{4.95}$$

which we recognize as the standard least-squares solution. Note that the error function in this case is quadratic and hence the Newton-Raphson formula gives the exact solution in one step.

Now let us apply the Newton-Raphson update to the cross-entropy error function (4.90) for the logistic regression model. From (4.91) we see that the gradient and Hessian of this error function are given by

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N}(y_n - t_n)\boldsymbol{\phi}_n = \boldsymbol{\Phi}^{\mathrm{T}}(\mathbf{y} - \mathbf{t}) \tag{4.96}$$

$$\mathbf{H} = \nabla\nabla E(\mathbf{w}) = \sum_{n=1}^{N} y_n(1 - y_n)\boldsymbol{\phi}_n\boldsymbol{\phi}_n^{\mathrm{T}} = \boldsymbol{\Phi}^{\mathrm{T}}\mathbf{R}\boldsymbol{\Phi} \tag{4.97}$$

where we have made use of (4.88). Also, we have introduced the $N \times N$ diagonal matrix $\mathbf{R}$ with elements

$$R_{nn} = y_n(1 - y_n). \tag{4.98}$$

We see that the Hessian is no longer constant but depends on $\mathbf{w}$ through the weighting matrix $\mathbf{R}$, corresponding to the fact that the error function is no longer quadratic. Using the property $0 < y_n < 1$, which follows from the form of the logistic sigmoid function, we see that $\mathbf{u}^{\mathrm{T}}\mathbf{H}\mathbf{u} > 0$ for an arbitrary vector $\mathbf{u}$, and so the Hessian matrix $\mathbf{H}$ is positive definite. It follows that the error function is a concave function of $\mathbf{w}$ and hence has a unique minimum.

Exercise 4.15

The Newton-Raphson update formula for the logistic regression model then becomes

$$\begin{aligned} \mathbf{w}^{(\mathrm{new})} &= \mathbf{w}^{(\mathrm{old})} - (\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{R}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^{\mathrm{T}}(\mathbf{y} - \mathbf{t}) \\ &= (\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{R}\boldsymbol{\Phi})^{-1}\left\{\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{R}\boldsymbol{\Phi}\mathbf{w}^{(\mathrm{old})} - \boldsymbol{\Phi}^{\mathrm{T}}(\mathbf{y} - \mathbf{t})\right\} \\ &= (\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{R}\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{R}\mathbf{z} \end{aligned} \tag{4.99}$$

where $\mathbf{z}$ is an N-dimensional vector with elements

$$\mathbf{z} = \boldsymbol{\Phi}\mathbf{w}^{(\mathrm{old})} - \mathbf{R}^{-1}(\mathbf{y} - \mathbf{t}). \tag{4.100}$$

We see that the update formula (4.99) takes the form of a set of normal equations for a weighted least-squares problem. Because the weighing matrix $\mathbf{R}$ is not constant but depends on the parameter vector $\mathbf{w}$, we must apply the normal equations iteratively, each time using the new weight vector $\mathbf{w}$ to compute a revised weighing matrix $\mathbf{R}$. For this reason, the algorithm is known as *iterative reweighted least squares*, or *IRLS* (Rubin, 1983). As in the weighted least-squares problem, the elements of the diagonal weighting matrix $\mathbf{R}$ can be interpreted as variances because the mean and variance of t in the logistic regression model are given by

$$\mathbb{E}[t] = \sigma(\mathbf{x}) = y \tag{4.101}$$

$$\mathrm{var}[t] = \mathbb{E}[t^2] - \mathbb{E}[t]^2 = \sigma(\mathbf{x}) - \sigma(\mathbf{x})^2 = y(1 - y) \tag{4.102}$$

where we have used the property $t^2 = t$ for $t \in \{0, 1\}$. In fact, we can interpret IRLS as the solution to a linearized problem in the space of the variable $a = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}$. The quantity z_n, which corresponds to the nth element of $\mathbf{z}$, can then be given a simple interpretation as an effective target value in this space obtained by making a local linear approximation to the logistic sigmoid function around the current operating point $\mathbf{w}^{(\mathrm{old})}$

$$\begin{aligned} a_n(\mathbf{w}) &\simeq a_n(\mathbf{w}^{(\mathrm{old})}) + \left.\frac{\mathrm{d}a_n}{\mathrm{d}y_n}\right|_{\mathbf{w}^{(\mathrm{old})}}(t_n - y_n) \\ &= \boldsymbol{\phi}_n^{\mathrm{T}}\mathbf{w}^{(\mathrm{old})} - \frac{(y_n - t_n)}{y_n(1 - y_n)} = z_n. \end{aligned} \tag{4.103}$$

4.3.4 Multiclass logistic regression

Section 4.2

In our discussion of generative models for multiclass classification, we have seen that for a large class of distributions, the posterior probabilities are given by a softmax transformation of linear functions of the feature variables, so that

$$p(\mathcal{C}_k|\boldsymbol{\phi}) = y_k(\boldsymbol{\phi}) = \frac{\exp(a_k)}{\sum_j \exp(a_j)} \tag{4.104}$$

where the 'activations' a_k are given by

$$a_k = \mathbf{w}_k^{\mathrm{T}}\boldsymbol{\phi}. \tag{4.105}$$

There we used maximum likelihood to determine separately the class-conditional densities and the class priors and then found the corresponding posterior probabilities using Bayes' theorem, thereby implicitly determining the parameters $\{\mathbf{w}_k\}$. Here we consider the use of maximum likelihood to determine the parameters $\{\mathbf{w}_k\}$ of this model directly. To do this, we will require the derivatives of y_k with respect to all of the activations a_j. These are given by

Exercise 4.17

$$\frac{\partial y_k}{\partial a_j} = y_k(I_{kj} - y_j) \tag{4.106}$$

where I_{kj} are the elements of the identity matrix.

Next we write down the likelihood function. This is most easily done using the 1-of-K coding scheme in which the target vector $\mathbf{t}_n$ for a feature vector $\boldsymbol{\phi}_n$ belonging to class $\mathcal{C}_k$ is a binary vector with all elements zero except for element k, which equals one. The likelihood function is then given by

$$p(\mathbf{T}|\mathbf{w}_1, \ldots, \mathbf{w}_K) = \prod_{n=1}^{N}\prod_{k=1}^{K} p(\mathcal{C}_k|\boldsymbol{\phi}_n)^{t_{nk}} = \prod_{n=1}^{N}\prod_{k=1}^{K} y_{nk}^{t_{nk}} \tag{4.107}$$

where $y_{nk} = y_k(\boldsymbol{\phi}_n)$, and $\mathbf{T}$ is an $N \times K$ matrix of target variables with elements t_{nk}. Taking the negative logarithm then gives

$$E(\mathbf{w}_1, \ldots, \mathbf{w}_K) = -\ln p(\mathbf{T}|\mathbf{w}_1, \ldots, \mathbf{w}_K) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln y_{nk} \tag{4.108}$$

which is known as the *cross-entropy* error function for the multiclass classification problem.

We now take the gradient of the error function with respect to one of the parameter vectors $\mathbf{w}_j$. Making use of the result (4.106) for the derivatives of the softmax function, we obtain

Exercise 4.18

$$\nabla_{\mathbf{w}_j} E(\mathbf{w}_1, \ldots, \mathbf{w}_K) = \sum_{n=1}^{N} (y_{nj} - t_{nj})\, \boldsymbol{\phi}_n \tag{4.109}$$

where we have made use of $\sum_k t_{nk} = 1$. Once again, we see the same form arising for the gradient as was found for the sum-of-squares error function with the linear model and the cross-entropy error for the logistic regression model, namely the product of the error $(y_{nj} - t_{nj})$ times the basis function $\boldsymbol{\phi}_n$. Again, we could use this to formulate a sequential algorithm in which patterns are presented one at a time, in which each of the weight vectors is updated using (3.22).

We have seen that the derivative of the log likelihood function for a linear regression model with respect to the parameter vector $\mathbf{w}$ for a data point n took the form of the 'error' $y_n - t_n$ times the feature vector $\boldsymbol{\phi}_n$. Similarly, for the combination of logistic sigmoid activation function and cross-entropy error function (4.90), and for the softmax activation function with the multiclass cross-entropy error function (4.108), we again obtain this same simple form. This is an example of a more general result, as we shall see in Section 4.3.6.

To find a batch algorithm, we again appeal to the Newton-Raphson update to obtain the corresponding IRLS algorithm for the multiclass problem. This requires evaluation of the Hessian matrix that comprises blocks of size $M \times M$ in which block j, k is given by

$$\nabla_{\mathbf{w}_k} \nabla_{\mathbf{w}_j} E(\mathbf{w}_1, \ldots, \mathbf{w}_K) = -\sum_{n=1}^{N} y_{nk}(I_{kj} - y_{nj})\boldsymbol{\phi}_n \boldsymbol{\phi}_n^{\mathrm{T}}. \tag{4.110}$$

Exercise 4.20

As with the two-class problem, the Hessian matrix for the multiclass logistic regression model is positive definite and so the error function again has a unique minimum. Practical details of IRLS for the multiclass case can be found in Bishop and Nabney (2008).

4.3.5 Probit regression

We have seen that, for a broad range of class-conditional distributions, described by the exponential family, the resulting posterior class probabilities are given by a logistic (or softmax) transformation acting on a linear function of the feature variables. However, not all choices of class-conditional density give rise to such a simple form for the posterior probabilities (for instance, if the class-conditional densities are modelled using Gaussian mixtures). This suggests that it might be worth exploring other types of discriminative probabilistic model. For the purposes of this chapter, however, we shall return to the two-class case, and again remain within the framework of generalized linear models so that

$$p(t = 1|a) = f(a) \tag{4.111}$$

where $a = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}$, and $f(\cdot)$ is the activation function.

One way to motivate an alternative choice for the link function is to consider a noisy threshold model, as follows. For each input $\boldsymbol{\phi}_n$, we evaluate $a_n = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n$ and then we set the target value according to

$$\begin{cases} t_n = 1 & \text{if } a_n \geqslant \theta \\ t_n = 0 & \text{otherwise.} \end{cases} \tag{4.112}$$

Figure 4.13 Schematic example of a probability density $p(\theta)$ shown by the blue curve, given in this example by a mixture of two Gaussians, along with its cumulative distribution function $f(a)$, shown by the red curve. Note that the value of the blue curve at any point, such as that indicated by the vertical green line, corresponds to the slope of the red curve at the same point. Conversely, the value of the red curve at this point corresponds to the area under the blue curve indicated by the shaded green region. In the stochastic threshold model, the class label takes the value $t = 1$ if the value of $a = \mathbf{w}^{\mathrm{T}}\phi$ exceeds a threshold, otherwise it takes the value $t = 0$. This is equivalent to an activation function given by the cumulative distribution function $f(a)$.

If the value of θ is drawn from a probability density $p(\theta)$, then the corresponding activation function will be given by the cumulative distribution function

$$f(a) = \int_{-\infty}^{a} p(\theta)\,\mathrm{d}\theta \tag{4.113}$$

as illustrated in Figure 4.13.

As a specific example, suppose that the density $p(\theta)$ is given by a zero mean, unit variance Gaussian. The corresponding cumulative distribution function is given by

$$\Phi(a) = \int_{-\infty}^{a} \mathcal{N}(\theta|0,1)\,\mathrm{d}\theta \tag{4.114}$$

which is known as the *probit* function. It has a sigmoidal shape and is compared with the logistic sigmoid function in Figure 4.9. Note that the use of a more general Gaussian distribution does not change the model because this is equivalent to a re-scaling of the linear coefficients $\mathbf{w}$. Many numerical packages provide for the evaluation of a closely related function defined by

$$\mathrm{erf}(a) = \frac{2}{\sqrt{\pi}} \int_{0}^{a} \exp(-\theta^2/2)\,\mathrm{d}\theta \tag{4.115}$$

and known as the *erf function* or *error function* (not to be confused with the error function of a machine learning model). It is related to the probit function by

Exercise 4.21

$$\mathbf{\Phi}(a) = \frac{1}{2}\left\{1 + \frac{1}{\sqrt{2}}\mathrm{erf}(a)\right\}. \tag{4.116}$$

The generalized linear model based on a probit activation function is known as *probit regression*.

We can determine the parameters of this model using maximum likelihood, by a straightforward extension of the ideas discussed earlier. In practice, the results found using probit regression tend to be similar to those of logistic regression. We shall,

however, find another use for the probit model when we discuss Bayesian treatments of logistic regression in Section 4.5.

One issue that can occur in practical applications is that of *outliers*, which can arise for instance through errors in measuring the input vector $\mathbf{x}$ or through mislabelling of the target value t. Because such points can lie a long way to the wrong side of the ideal decision boundary, they can seriously distort the classifier. Note that the logistic and probit regression models behave differently in this respect because the tails of the logistic sigmoid decay asymptotically like $\exp(-x)$ for $x \to \infty$, whereas for the probit activation function they decay like $\exp(-x^2)$, and so the probit model can be significantly more sensitive to outliers.

However, both the logistic and the probit models assume the data is correctly labelled. The effect of mislabelling is easily incorporated into a probabilistic model by introducing a probability ϵ that the target value t has been flipped to the wrong value (Opper and Winther, 2000a), leading to a target value distribution for data point $\mathbf{x}$ of the form

$$\begin{aligned} p(t|\mathbf{x}) &= (1-\epsilon)\sigma(\mathbf{x}) + \epsilon(1-\sigma(\mathbf{x})) \\ &= \epsilon + (1-2\epsilon)\sigma(\mathbf{x}) \end{aligned} \tag{4.117}$$

where $\sigma(\mathbf{x})$ is the activation function with input vector $\mathbf{x}$. Here ϵ may be set in advance, or it may be treated as a hyperparameter whose value is inferred from the data.

4.3.6 Canonical link functions

For the linear regression model with a Gaussian noise distribution, the error function, corresponding to the negative log likelihood, is given by (3.12). If we take the derivative with respect to the parameter vector $\mathbf{w}$ of the contribution to the error function from a data point n, this takes the form of the 'error' $y_n - t_n$ times the feature vector $\boldsymbol{\phi}_n$, where $y_n = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n$. Similarly, for the combination of the logistic sigmoid activation function and the cross-entropy error function (4.90), and for the softmax activation function with the multiclass cross-entropy error function (4.108), we again obtain this same simple form. We now show that this is a general result of assuming a conditional distribution for the target variable from the exponential family, along with a corresponding choice for the activation function known as the *canonical link function*.

We again make use of the restricted form (4.84) of exponential family distributions. Note that here we are applying the assumption of exponential family distribution to the target variable t, in contrast to Section 4.2.4 where we applied it to the input vector $\mathbf{x}$. We therefore consider conditional distributions of the target variable of the form

$$p(t|\eta, s) = \frac{1}{s} h\left(\frac{t}{s}\right) g(\eta) \exp\left\{\frac{\eta t}{s}\right\}. \tag{4.118}$$

Using the same line of argument as led to the derivation of the result (2.226), we see that the conditional mean of t, which we denote by y, is given by

$$y \equiv \mathbb{E}[t|\eta] = -s\frac{d}{d\eta}\ln g(\eta). \tag{4.119}$$

Thus y and η must related, and we denote this relation through $\eta = \psi(y)$.

Following Nelder and Wedderburn (1972), we define a *generalized linear model* to be one for which y is a nonlinear function of a linear combination of the input (or feature) variables so that

$$y = f(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}) \tag{4.120}$$

where $f(\cdot)$ is known as the *activation function* in the machine learning literature, and $f^{-1}(\cdot)$ is known as the *link function* in statistics.

Now consider the log likelihood function for this model, which, as a function of η, is given by

$$\ln p(\mathbf{t}|\eta, s) = \sum_{n=1}^{N} \ln p(t_n|\eta, s) = \sum_{n=1}^{N} \left\{ \ln g(\eta_n) + \frac{\eta_n t_n}{s} \right\} + \text{const} \tag{4.121}$$

where we are assuming that all observations share a common scale parameter (which corresponds to the noise variance for a Gaussian distribution for instance) and so s is independent of n. The derivative of the log likelihood with respect to the model parameters $\mathbf{w}$ is then given by

$$\begin{aligned} \nabla_{\mathbf{w}} \ln p(\mathbf{t}|\eta, s) &= \sum_{n=1}^{N} \left\{ \frac{d}{d\eta_n} \ln g(\eta_n) + \frac{t_n}{s} \right\} \frac{d\eta_n}{dy_n} \frac{dy_n}{da_n} \nabla a_n \\ &= \sum_{n=1}^{N} \frac{1}{s} \{t_n - y_n\} \psi'(y_n) f'(a_n) \boldsymbol{\phi}_n \end{aligned} \tag{4.122}$$

where $a_n = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n$, and we have used $y_n = f(a_n)$ together with the result (4.119) for $\mathbb{E}[t|\eta]$. We now see that there is a considerable simplification if we choose a particular form for the link function $f^{-1}(y)$ given by

$$f^{-1}(y) = \psi(y) \tag{4.123}$$

which gives $f(\psi(y)) = y$ and hence $f'(\psi)\psi'(y) = 1$. Also, because $a = f^{-1}(y)$, we have $a = \psi$ and hence $f'(a)\psi'(y) = 1$. In this case, the gradient of the error function reduces to

$$\nabla \ln E(\mathbf{w}) = \frac{1}{s} \sum_{n=1}^{N} \{y_n - t_n\} \boldsymbol{\phi}_n. \tag{4.124}$$

For the Gaussian $s = \beta^{-1}$, whereas for the logistic model $s = 1$.

4.4. The Laplace Approximation

In Section 4.5 we shall discuss the Bayesian treatment of logistic regression. As we shall see, this is more complex than the Bayesian treatment of linear regression models, discussed in Sections 3.3 and 3.5. In particular, we cannot integrate exactly

over the parameter vector $\mathbf{w}$ since the posterior distribution is no longer Gaussian. It is therefore necessary to introduce some form of approximation. Later in the book we shall consider a range of techniques based on analytical approximations and numerical sampling.

Chapter 10
Chapter 11

Here we introduce a simple, but widely used, framework called the Laplace approximation, that aims to find a Gaussian approximation to a probability density defined over a set of continuous variables. Consider first the case of a single continuous variable z, and suppose the distribution $p(z)$ is defined by

$$p(z) = \frac{1}{Z} f(z) \tag{4.125}$$

where $Z = \int f(z)\,\mathrm{d}z$ is the normalization coefficient. We shall suppose that the value of Z is unknown. In the Laplace method the goal is to find a Gaussian approximation $q(z)$ which is centred on a mode of the distribution $p(z)$. The first step is to find a mode of $p(z)$, in other words a point z_0 such that $p'(z_0) = 0$, or equivalently

$$\left.\frac{df(z)}{dz}\right|_{z=z_0} = 0. \tag{4.126}$$

A Gaussian distribution has the property that its logarithm is a quadratic function of the variables. We therefore consider a Taylor expansion of $\ln f(z)$ centred on the mode z_0 so that

$$\ln f(z) \simeq \ln f(z_0) - \frac{1}{2} A (z - z_0)^2 \tag{4.127}$$

where

$$A = -\left.\frac{d^2}{dz^2} \ln f(z)\right|_{z=z_0}. \tag{4.128}$$

Note that the first-order term in the Taylor expansion does not appear since z_0 is a local maximum of the distribution. Taking the exponential we obtain

$$f(z) \simeq f(z_0) \exp\left\{-\frac{A}{2}(z - z_0)^2\right\}. \tag{4.129}$$

We can then obtain a normalized distribution $q(z)$ by making use of the standard result for the normalization of a Gaussian, so that

$$q(z) = \left(\frac{A}{2\pi}\right)^{1/2} \exp\left\{-\frac{A}{2}(z - z_0)^2\right\}. \tag{4.130}$$

The Laplace approximation is illustrated in Figure 4.14. Note that the Gaussian approximation will only be well defined if its precision $A > 0$, in other words the stationary point z_0 must be a local maximum, so that the second derivative of $f(z)$ at the point z_0 is negative.

Figure 4.14 Illustration of the Laplace approximation applied to the distribution $p(z) \propto \exp(-z^2/2)\sigma(20z+4)$ where $\sigma(z)$ is the logistic sigmoid function defined by $\sigma(z) = (1+e^{-z})^{-1}$. The left plot shows the normalized distribution $p(z)$ in yellow, together with the Laplace approximation centred on the mode z_0 of $p(z)$ in red. The right plot shows the negative logarithms of the corresponding curves.

We can extend the Laplace method to approximate a distribution $p(\mathbf{z}) = f(\mathbf{z})/Z$ defined over an M-dimensional space $\mathbf{z}$. At a stationary point $\mathbf{z}_0$ the gradient $\nabla f(\mathbf{z})$ will vanish. Expanding around this stationary point we have

$$\ln f(\mathbf{z}) \simeq \ln f(\mathbf{z}_0) - \frac{1}{2}(\mathbf{z}-\mathbf{z}_0)^{\mathrm{T}}\mathbf{A}(\mathbf{z}-\mathbf{z}_0) \tag{4.131}$$

where the $M \times M$ Hessian matrix $\mathbf{A}$ is defined by

$$\mathbf{A} = -\nabla\nabla \ln f(\mathbf{z})|_{\mathbf{z}=\mathbf{z}_0} \tag{4.132}$$

and ∇ is the gradient operator. Taking the exponential of both sides we obtain

$$f(\mathbf{z}) \simeq f(\mathbf{z}_0)\exp\left\{-\frac{1}{2}(\mathbf{z}-\mathbf{z}_0)^{\mathrm{T}}\mathbf{A}(\mathbf{z}-\mathbf{z}_0)\right\}. \tag{4.133}$$

The distribution $q(\mathbf{z})$ is proportional to $f(\mathbf{z})$ and the appropriate normalization coefficient can be found by inspection, using the standard result (2.43) for a normalized multivariate Gaussian, giving

$$q(\mathbf{z}) = \frac{|\mathbf{A}|^{1/2}}{(2\pi)^{M/2}}\exp\left\{-\frac{1}{2}(\mathbf{z}-\mathbf{z}_0)^{\mathrm{T}}\mathbf{A}(\mathbf{z}-\mathbf{z}_0)\right\} = \mathcal{N}(\mathbf{z}|\mathbf{z}_0, \mathbf{A}^{-1}) \tag{4.134}$$

where $|\mathbf{A}|$ denotes the determinant of $\mathbf{A}$. This Gaussian distribution will be well defined provided its precision matrix, given by $\mathbf{A}$, is positive definite, which implies that the stationary point $\mathbf{z}_0$ must be a local maximum, not a minimum or a saddle point.

In order to apply the Laplace approximation we first need to find the mode $\mathbf{z}_0$, and then evaluate the Hessian matrix at that mode. In practice a mode will typically be found by running some form of numerical optimization algorithm (Bishop

and Nabney, 2008). Many of the distributions encountered in practice will be multimodal and so there will be different Laplace approximations according to which mode is being considered. Note that the normalization constant Z of the true distribution does not need to be known in order to apply the Laplace method. As a result of the central limit theorem, the posterior distribution for a model is expected to become increasingly better approximated by a Gaussian as the number of observed data points is increased, and so we would expect the Laplace approximation to be most useful in situations where the number of data points is relatively large.

One major weakness of the Laplace approximation is that, since it is based on a Gaussian distribution, it is only directly applicable to real variables. In other cases it may be possible to apply the Laplace approximation to a transformation of the variable. For instance if $0 \leqslant \tau < \infty$ then we can consider a Laplace approximation of $\ln \tau$. The most serious limitation of the Laplace framework, however, is that it is based purely on the aspects of the true distribution at a specific value of the variable, and so can fail to capture important global properties. In Chapter 10 we shall consider alternative approaches which adopt a more global perspective.

4.4.1 Model comparison and BIC

As well as approximating the distribution $p(\mathbf{z})$ we can also obtain an approximation to the normalization constant Z. Using the approximation (4.133) we have

$$\begin{aligned} Z &= \int f(\mathbf{z})\,\mathrm{d}\mathbf{z} \\ &\simeq f(\mathbf{z}_0) \int \exp\left\{-\frac{1}{2}(\mathbf{z}-\mathbf{z}_0)^{\mathrm{T}}\mathbf{A}(\mathbf{z}-\mathbf{z}_0)\right\}\mathrm{d}\mathbf{z} \\ &= f(\mathbf{z}_0)\frac{(2\pi)^{M/2}}{|\mathbf{A}|^{1/2}} \end{aligned} \tag{4.135}$$

where we have noted that the integrand is Gaussian and made use of the standard result (2.43) for a normalized Gaussian distribution. We can use the result (4.135) to obtain an approximation to the model evidence which, as discussed in Section 3.4, plays a central role in Bayesian model comparison.

Consider a data set $\mathcal{D}$ and a set of models $\{\mathcal{M}_i\}$ having parameters $\{\boldsymbol{\theta}_i\}$. For each model we define a likelihood function $p(\mathcal{D}|\boldsymbol{\theta}_i, \mathcal{M}_i)$. If we introduce a prior $p(\boldsymbol{\theta}_i|\mathcal{M}_i)$ over the parameters, then we are interested in computing the model evidence $p(\mathcal{D}|\mathcal{M}_i)$ for the various models. From now on we omit the conditioning on $\mathcal{M}_i$ to keep the notation uncluttered. From Bayes' theorem the model evidence is given by

$$p(\mathcal{D}) = \int p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})\,\mathrm{d}\boldsymbol{\theta}. \tag{4.136}$$

Identifying $f(\boldsymbol{\theta}) = p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})$ and $Z = p(\mathcal{D})$, and applying the result (4.135), we obtain

Exercise 4.22

$$\ln p(\mathcal{D}) \simeq \ln p(\mathcal{D}|\boldsymbol{\theta}_{\mathrm{MAP}}) + \underbrace{\ln p(\boldsymbol{\theta}_{\mathrm{MAP}}) + \frac{M}{2}\ln(2\pi) - \frac{1}{2}\ln|\mathbf{A}|}_{\text{Occam factor}} \tag{4.137}$$

where $\boldsymbol{\theta}_{\mathrm{MAP}}$ is the value of $\boldsymbol{\theta}$ at the mode of the posterior distribution, and $\mathbf{A}$ is the *Hessian* matrix of second derivatives of the negative log posterior

$$\mathbf{A} = -\nabla\nabla \ln p(\mathcal{D}|\boldsymbol{\theta}_{\mathrm{MAP}})p(\boldsymbol{\theta}_{\mathrm{MAP}}) = -\nabla\nabla \ln p(\boldsymbol{\theta}_{\mathrm{MAP}}|\mathcal{D}). \tag{4.138}$$

The first term on the right hand side of (4.137) represents the log likelihood evaluated using the optimized parameters, while the remaining three terms comprise the 'Occam factor' which penalizes model complexity.

If we assume that the Gaussian prior distribution over parameters is broad, and that the Hessian has full rank, then we can approximate (4.137) very roughly using

Exercise 4.23

$$\ln p(\mathcal{D}) \simeq \ln p(\mathcal{D}|\boldsymbol{\theta}_{\mathrm{MAP}}) - \frac{1}{2}M \ln N \tag{4.139}$$

where N is the number of data points, M is the number of parameters in $\boldsymbol{\theta}$ and we have omitted additive constants. This is known as the *Bayesian Information Criterion* (BIC) or the *Schwarz criterion* (Schwarz, 1978). Note that, compared to AIC given by (1.73), this penalizes model complexity more heavily.

Complexity measures such as AIC and BIC have the virtue of being easy to evaluate, but can also give misleading results. In particular, the assumption that the Hessian matrix has full rank is often not valid since many of the parameters are not 'well-determined'. We can use the result (4.137) to obtain a more accurate estimate of the model evidence starting from the Laplace approximation, as we illustrate in the context of neural networks in Section 5.7.

Section 3.5.3

4.5. Bayesian Logistic Regression

We now turn to a Bayesian treatment of logistic regression. Exact Bayesian inference for logistic regression is intractable. In particular, evaluation of the posterior distribution would require normalization of the product of a prior distribution and a likelihood function that itself comprises a product of logistic sigmoid functions, one for every data point. Evaluation of the predictive distribution is similarly intractable. Here we consider the application of the Laplace approximation to the problem of Bayesian logistic regression (Spiegelhalter and Lauritzen, 1990; MacKay, 1992b).

4.5.1 Laplace approximation

Recall from Section 4.4 that the Laplace approximation is obtained by finding the mode of the posterior distribution and then fitting a Gaussian centred at that mode. This requires evaluation of the second derivatives of the log posterior, which is equivalent to finding the Hessian matrix.

Because we seek a Gaussian representation for the posterior distribution, it is natural to begin with a Gaussian prior, which we write in the general form

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{m}_0, \mathbf{S}_0) \tag{4.140}$$

where $\mathbf{m}_0$ and $\mathbf{S}_0$ are fixed hyperparameters. The posterior distribution over $\mathbf{w}$ is given by

$$p(\mathbf{w}|\mathbf{t}) \propto p(\mathbf{w})p(\mathbf{t}|\mathbf{w}) \tag{4.141}$$

where $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$. Taking the log of both sides, and substituting for the prior distribution using (4.140), and for the likelihood function using (4.89), we obtain

$$\begin{aligned} \ln p(\mathbf{w}|\mathbf{t}) &= -\frac{1}{2}(\mathbf{w}-\mathbf{m}_0)^{\mathrm{T}}\mathbf{S}_0^{-1}(\mathbf{w}-\mathbf{m}_0) \\ &\quad + \sum_{n=1}^{N} \{t_n \ln y_n + (1-t_n)\ln(1-y_n)\} + \text{const} \end{aligned} \tag{4.142}$$

where $y_n = \sigma(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n)$. To obtain a Gaussian approximation to the posterior distribution, we first maximize the posterior distribution to give the MAP (maximum posterior) solution $\mathbf{w}_{\mathrm{MAP}}$, which defines the mean of the Gaussian. The covariance is then given by the inverse of the matrix of second derivatives of the negative log likelihood, which takes the form

$$\mathbf{S}_N = -\nabla\nabla \ln p(\mathbf{w}|\mathbf{t}) = \mathbf{S}_0^{-1} + \sum_{n=1}^{N} y_n(1-y_n)\boldsymbol{\phi}_n\boldsymbol{\phi}_n^{\mathrm{T}}. \tag{4.143}$$

The Gaussian approximation to the posterior distribution therefore takes the form

$$q(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{w}_{\mathrm{MAP}}, \mathbf{S}_N). \tag{4.144}$$

Having obtained a Gaussian approximation to the posterior distribution, there remains the task of marginalizing with respect to this distribution in order to make predictions.

4.5.2 Predictive distribution

The predictive distribution for class $\mathcal{C}_1$, given a new feature vector $\boldsymbol{\phi}(\mathbf{x})$, is obtained by marginalizing with respect to the posterior distribution $p(\mathbf{w}|\mathbf{t})$, which is itself approximated by a Gaussian distribution $q(\mathbf{w})$ so that

$$p(\mathcal{C}_1|\boldsymbol{\phi}, \mathbf{t}) = \int p(\mathcal{C}_1|\boldsymbol{\phi}, \mathbf{w})p(\mathbf{w}|\mathbf{t})\,\mathrm{d}\mathbf{w} \simeq \int \sigma(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi})q(\mathbf{w})\,\mathrm{d}\mathbf{w} \tag{4.145}$$

with the corresponding probability for class $\mathcal{C}_2$ given by $p(\mathcal{C}_2|\boldsymbol{\phi}, \mathbf{t}) = 1 - p(\mathcal{C}_1|\boldsymbol{\phi}, \mathbf{t})$. To evaluate the predictive distribution, we first note that the function $\sigma(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi})$ depends on $\mathbf{w}$ only through its projection onto $\boldsymbol{\phi}$. Denoting $a = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}$, we have

$$\sigma(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}) = \int \delta(a - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi})\sigma(a)\,\mathrm{d}a \tag{4.146}$$

where $\delta(\cdot)$ is the Dirac delta function. From this we obtain

$$\int \sigma(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi})q(\mathbf{w})\,\mathrm{d}\mathbf{w} = \int \sigma(a)p(a)\,\mathrm{d}a \tag{4.147}$$

where

$$p(a) = \int \delta(a - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi})q(\mathbf{w})\,\mathrm{d}\mathbf{w}. \tag{4.148}$$

We can evaluate $p(a)$ by noting that the delta function imposes a linear constraint on $\mathbf{w}$ and so forms a marginal distribution from the joint distribution $q(\mathbf{w})$ by integrating out all directions orthogonal to $\boldsymbol{\phi}$. Because $q(\mathbf{w})$ is Gaussian, we know from Section 2.3.2 that the marginal distribution will also be Gaussian. We can evaluate the mean and covariance of this distribution by taking moments, and interchanging the order of integration over a and $\mathbf{w}$, so that

$$\mu_a = \mathbb{E}[a] = \int p(a)a\,\mathrm{d}a = \int q(\mathbf{w})\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}\,\mathrm{d}\mathbf{w} = \mathbf{w}_{\mathrm{MAP}}^{\mathrm{T}}\boldsymbol{\phi} \tag{4.149}$$

where we have used the result (4.144) for the variational posterior distribution $q(\mathbf{w})$. Similarly

$$\begin{aligned}\sigma_a^2 &= \mathrm{var}[a] = \int p(a)\left\{a^2 - \mathbb{E}[a]^2\right\}\,\mathrm{d}a \\ &= \int q(\mathbf{w})\left\{(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi})^2 - (\mathbf{m}_N^{\mathrm{T}}\boldsymbol{\phi})^2\right\}\,\mathrm{d}\mathbf{w} = \boldsymbol{\phi}^{\mathrm{T}}\mathbf{S}_N\boldsymbol{\phi}. \end{aligned} \tag{4.150}$$

Note that the distribution of a takes the same form as the predictive distribution (3.58) for the linear regression model, with the noise variance set to zero. Thus our variational approximation to the predictive distribution becomes

$$p(\mathcal{C}_1|\mathbf{t}) = \int \sigma(a)p(a)\,\mathrm{d}a = \int \sigma(a)\mathcal{N}(a|\mu_a, \sigma_a^2)\,\mathrm{d}a. \tag{4.151}$$

This result can also be derived directly by making use of the results for the marginal of a Gaussian distribution given in Section 2.3.2.

Exercise 4.24

The integral over a represents the convolution of a Gaussian with a logistic sigmoid, and cannot be evaluated analytically. We can, however, obtain a good approximation (Spiegelhalter and Lauritzen, 1990; MacKay, 1992b; Barber and Bishop, 1998a) by making use of the close similarity between the logistic sigmoid function $\sigma(a)$ defined by (4.59) and the probit function $\Phi(a)$ defined by (4.114). In order to obtain the best approximation to the logistic function we need to re-scale the horizontal axis, so that we approximate $\sigma(a)$ by $\Phi(\lambda a)$. We can find a suitable value of λ by requiring that the two functions have the same slope at the origin, which gives $\lambda^2 = \pi/8$. The similarity of the logistic sigmoid and the probit function, for this choice of λ, is illustrated in Figure 4.9.

Exercise 4.25

The advantage of using a probit function is that its convolution with a Gaussian can be expressed analytically in terms of another probit function. Specifically we can show that

Exercise 4.26

$$\int \Phi(\lambda a)\mathcal{N}(a|\mu, \sigma^2)\,\mathrm{d}a = \Phi\left(\frac{\mu}{(\lambda^{-2} + \sigma^2)^{1/2}}\right). \tag{4.152}$$

We now apply the approximation $\sigma(a) \simeq \Phi(\lambda a)$ to the probit functions appearing on both sides of this equation, leading to the following approximation for the convolution of a logistic sigmoid with a Gaussian

$$\int \sigma(a)\mathcal{N}(a|\mu, \sigma^2)\,\mathrm{d}a \simeq \sigma\left(\kappa(\sigma^2)\mu\right) \tag{4.153}$$

where we have defined

$$\kappa(\sigma^2) = (1 + \pi\sigma^2/8)^{-1/2}. \tag{4.154}$$

Applying this result to (4.151) we obtain the approximate predictive distribution in the form

$$p(\mathcal{C}_1|\boldsymbol{\phi}, \mathbf{t}) = \sigma\left(\kappa(\sigma_a^2)\mu_a\right) \tag{4.155}$$

where μ_a and σ_a^2 are defined by (4.149) and (4.150), respectively, and $\kappa(\sigma_a^2)$ is defined by (4.154).

Note that the decision boundary corresponding to $p(\mathcal{C}_1|\boldsymbol{\phi}, \mathbf{t}) = 0.5$ is given by $\mu_a = 0$, which is the same as the decision boundary obtained by using the MAP value for $\mathbf{w}$. Thus if the decision criterion is based on minimizing misclassification rate, with equal prior probabilities, then the marginalization over $\mathbf{w}$ has no effect. However, for more complex decision criteria it will play an important role. Marginalization of the logistic sigmoid model under a Gaussian approximation to the posterior distribution will be illustrated in the context of variational inference in Figure 10.13.

Exercises

4.1 ($\star\star$) Given a set of data points $\{\mathbf{x}_n\}$, we can define the *convex hull* to be the set of all points $\mathbf{x}$ given by

$$\mathbf{x} = \sum_n \alpha_n \mathbf{x}_n \tag{4.156}$$

where $\alpha_n \geqslant 0$ and $\sum_n \alpha_n = 1$. Consider a second set of points $\{\mathbf{y}_n\}$ together with their corresponding convex hull. By definition, the two sets of points will be linearly separable if there exists a vector $\widehat{\mathbf{w}}$ and a scalar w_0 such that $\widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{x}_n + w_0 > 0$ for all $\mathbf{x}_n$, and $\widehat{\mathbf{w}}^{\mathrm{T}}\mathbf{y}_n + w_0 < 0$ for all $\mathbf{y}_n$. Show that if their convex hulls intersect, the two sets of points cannot be linearly separable, and conversely that if they are linearly separable, their convex hulls do not intersect.

4.2 ($\star\star$) **www** Consider the minimization of a sum-of-squares error function (4.15), and suppose that all of the target vectors in the training set satisfy a linear constraint

$$\mathbf{a}^{\mathrm{T}}\mathbf{t}_n + b = 0 \tag{4.157}$$

where $\mathbf{t}_n$ corresponds to the nth row of the matrix $\mathbf{T}$ in (4.15). Show that as a consequence of this constraint, the elements of the model prediction $\mathbf{y}(\mathbf{x})$ given by the least-squares solution (4.17) also satisfy this constraint, so that

$$\mathbf{a}^{\mathrm{T}}\mathbf{y}(\mathbf{x}) + b = 0. \tag{4.158}$$

To do so, assume that one of the basis functions $\phi_0(\mathbf{x}) = 1$ so that the corresponding parameter w_0 plays the role of a bias.

4.3 ($\star\star$) Extend the result of Exercise 4.2 to show that if multiple linear constraints are satisfied simultaneously by the target vectors, then the same constraints will also be satisfied by the least-squares prediction of a linear model.

4.4 ($\star$) **www** Show that maximization of the class separation criterion given by (4.23) with respect to $\mathbf{w}$, using a Lagrange multiplier to enforce the constraint $\mathbf{w}^{\mathrm{T}}\mathbf{w} = 1$, leads to the result that $\mathbf{w} \propto (\mathbf{m}_2 - \mathbf{m}_1)$.

4.5 ($\star$) By making use of (4.20), (4.23), and (4.24), show that the Fisher criterion (4.25) can be written in the form (4.26).

4.6 ($\star$) Using the definitions of the between-class and within-class covariance matrices given by (4.27) and (4.28), respectively, together with (4.34) and (4.36) and the choice of target values described in Section 4.1.5, show that the expression (4.33) that minimizes the sum-of-squares error function can be written in the form (4.37).

4.7 ($\star$) **www** Show that the logistic sigmoid function (4.59) satisfies the property $\sigma(-a) = 1 - \sigma(a)$ and that its inverse is given by $\sigma^{-1}(y) = \ln\{y/(1-y)\}$.

4.8 ($\star$) Using (4.57) and (4.58), derive the result (4.65) for the posterior class probability in the two-class generative model with Gaussian densities, and verify the results (4.66) and (4.67) for the parameters $\mathbf{w}$ and w_0.

4.9 ($\star$) **www** Consider a generative classification model for K classes defined by prior class probabilities $p(\mathcal{C}_k) = \pi_k$ and general class-conditional densities $p(\boldsymbol{\phi}|\mathcal{C}_k)$ where $\boldsymbol{\phi}$ is the input feature vector. Suppose we are given a training data set $\{\boldsymbol{\phi}_n, \mathbf{t}_n\}$ where $n = 1, \ldots, N$, and $\mathbf{t}_n$ is a binary target vector of length K that uses the 1-of-K coding scheme, so that it has components $t_{nj} = I_{jk}$ if pattern n is from class $\mathcal{C}_k$. Assuming that the data points are drawn independently from this model, show that the maximum-likelihood solution for the prior probabilities is given by

$$\pi_k = \frac{N_k}{N} \tag{4.159}$$

where N_k is the number of data points assigned to class $\mathcal{C}_k$.

4.10 ($\star\star$) Consider the classification model of Exercise 4.9 and now suppose that the class-conditional densities are given by Gaussian distributions with a shared covariance matrix, so that

$$p(\boldsymbol{\phi}|\mathcal{C}_k) = \mathcal{N}(\boldsymbol{\phi}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}). \tag{4.160}$$

Show that the maximum likelihood solution for the mean of the Gaussian distribution for class $\mathcal{C}_k$ is given by

$$\boldsymbol{\mu}_k = \frac{1}{N_k}\sum_{n=1}^{N} t_{nk}\boldsymbol{\phi}_n \tag{4.161}$$

which represents the mean of those feature vectors assigned to class $\mathcal{C}_k$. Similarly, show that the maximum likelihood solution for the shared covariance matrix is given by

$$\boldsymbol{\Sigma} = \sum_{k=1}^{K} \frac{N_k}{N} \mathbf{S}_k \tag{4.162}$$

where

$$\mathbf{S}_k = \frac{1}{N_k} \sum_{n=1}^{N} t_{nk} (\boldsymbol{\phi}_n - \boldsymbol{\mu}_k)(\boldsymbol{\phi}_n - \boldsymbol{\mu}_k)^{\mathrm{T}}. \tag{4.163}$$

Thus $\boldsymbol{\Sigma}$ is given by a weighted average of the covariances of the data associated with each class, in which the weighting coefficients are given by the prior probabilities of the classes.

4.11 ($\star\star$) Consider a classification problem with K classes for which the feature vector $\boldsymbol{\phi}$ has M components each of which can take L discrete states. Let the values of the components be represented by a 1-of-L binary coding scheme. Further suppose that, conditioned on the class $\mathcal{C}_k$, the M components of $\boldsymbol{\phi}$ are independent, so that the class-conditional density factorizes with respect to the feature vector components. Show that the quantities a_k given by (4.63), which appear in the argument to the softmax function describing the posterior class probabilities, are linear functions of the components of $\boldsymbol{\phi}$. Note that this represents an example of the naive Bayes model which is discussed in Section 8.2.2.

4.12 ($\star$) **www** Verify the relation (4.88) for the derivative of the logistic sigmoid function defined by (4.59).

4.13 ($\star$) **www** By making use of the result (4.88) for the derivative of the logistic sigmoid, show that the derivative of the error function (4.90) for the logistic regression model is given by (4.91).

4.14 ($\star$) Show that for a linearly separable data set, the maximum likelihood solution for the logistic regression model is obtained by finding a vector $\mathbf{w}$ whose decision boundary $\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}) = 0$ separates the classes and then taking the magnitude of $\mathbf{w}$ to infinity.

4.15 ($\star\star$) Show that the Hessian matrix $\mathbf{H}$ for the logistic regression model, given by (4.97), is positive definite. Here $\mathbf{R}$ is a diagonal matrix with elements $y_n(1 - y_n)$, and y_n is the output of the logistic regression model for input vector $\mathbf{x}_n$. Hence show that the error function is a concave function of $\mathbf{w}$ and that it has a unique minimum.

4.16 ($\star$) Consider a binary classification problem in which each observation $\mathbf{x}_n$ is known to belong to one of two classes, corresponding to $t = 0$ and $t = 1$, and suppose that the procedure for collecting training data is imperfect, so that training points are sometimes mislabelled. For every data point $\mathbf{x}_n$, instead of having a value t for the class label, we have instead a value π_n representing the probability that $t_n = 1$. Given a probabilistic model $p(t = 1|\boldsymbol{\phi})$, write down the log likelihood function appropriate to such a data set.

4.17 ($\star$) **www** Show that the derivatives of the softmax activation function (4.104), where the a_k are defined by (4.105), are given by (4.106).

4.18 ($\star$) Using the result (4.91) for the derivatives of the softmax activation function, show that the gradients of the cross-entropy error (4.108) are given by (4.109).

4.19 ($\star$) **www** Write down expressions for the gradient of the log likelihood, as well as the corresponding Hessian matrix, for the probit regression model defined in Section 4.3.5. These are the quantities that would be required to train such a model using IRLS.

4.20 ($\star\star$) Show that the Hessian matrix for the multiclass logistic regression problem, defined by (4.110), is positive semidefinite. Note that the full Hessian matrix for this problem is of size $MK \times MK$, where M is the number of parameters and K is the number of classes. To prove the positive semidefinite property, consider the product $\mathbf{u}^{\mathrm{T}}\mathbf{H}\mathbf{u}$ where $\mathbf{u}$ is an arbitrary vector of length MK, and then apply Jensen's inequality.

4.21 ($\star$) Show that the probit function (4.114) and the erf function (4.115) are related by (4.116).

4.22 ($\star$) Using the result (4.135), derive the expression (4.137) for the log model evidence under the Laplace approximation.

4.23 ($\star\star$) **www** In this exercise, we derive the BIC result (4.139) starting from the Laplace approximation to the model evidence given by (4.137). Show that if the prior over parameters is Gaussian of the form $p(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\mathbf{m}, \mathbf{V}_0)$, the log model evidence under the Laplace approximation takes the form

$$\ln p(\mathcal{D}) \simeq \ln p(\mathcal{D}|\boldsymbol{\theta}_{\mathrm{MAP}}) - \frac{1}{2}(\boldsymbol{\theta}_{\mathrm{MAP}} - \mathbf{m})^{\mathrm{T}}\mathbf{V}_0^{-1}(\boldsymbol{\theta}_{\mathrm{MAP}} - \mathbf{m}) - \frac{1}{2}\ln|\mathbf{H}| + \mathrm{const}$$

where $\mathbf{H}$ is the matrix of second derivatives of the log likelihood $\ln p(\mathcal{D}|\boldsymbol{\theta})$ evaluated at $\boldsymbol{\theta}_{\mathrm{MAP}}$. Now assume that the prior is broad so that $\mathbf{V}_0^{-1}$ is small and the second term on the right-hand side above can be neglected. Furthermore, consider the case of independent, identically distributed data so that $\mathbf{H}$ is the sum of terms one for each data point. Show that the log model evidence can then be written approximately in the form of the BIC expression (4.139).

4.24 ($\star\star$) Use the results from Section 2.3.2 to derive the result (4.151) for the marginalization of the logistic regression model with respect to a Gaussian posterior distribution over the parameters $\mathbf{w}$.

4.25 ($\star\star$) Suppose we wish to approximate the logistic sigmoid $\sigma(a)$ defined by (4.59) by a scaled probit function $\Phi(\lambda a)$, where $\Phi(a)$ is defined by (4.114). Show that if λ is chosen so that the derivatives of the two functions are equal at $a = 0$, then $\lambda^2 = \pi/8$.

4.26 ($\star\star$) In this exercise, we prove the relation (4.152) for the convolution of a probit function with a Gaussian distribution. To do this, show that the derivative of the left-hand side with respect to μ is equal to the derivative of the right-hand side, and then integrate both sides with respect to μ and then show that the constant of integration vanishes. Note that before differentiating the left-hand side, it is convenient first to introduce a change of variable given by $a = \mu + \sigma z$ so that the integral over a is replaced by an integral over z. When we differentiate the left-hand side of the relation (4.152), we will then obtain a Gaussian integral over z that can be evaluated analytically.

5 Neural Networks

In Chapters 3 and 4 we considered models for regression and classification that comprised linear combinations of fixed basis functions. We saw that such models have useful analytical and computational properties but that their practical applicability was limited by the curse of dimensionality. In order to apply such models to large-scale problems, it is necessary to adapt the basis functions to the data.

Support vector machines (SVMs), discussed in Chapter 7, address this by first defining basis functions that are centred on the training data points and then selecting a subset of these during training. One advantage of SVMs is that, although the training involves nonlinear optimization, the objective function is convex, and so the solution of the optimization problem is relatively straightforward. The number of basis functions in the resulting models is generally much smaller than the number of training points, although it is often still relatively large and typically increases with the size of the training set. The relevance vector machine, discussed in Section 7.2, also chooses a subset from a fixed set of basis functions and typically results in much

sparser models. Unlike the SVM it also produces probabilistic outputs, although this is at the expense of a nonconvex optimization during training.

An alternative approach is to fix the number of basis functions in advance but allow them to be adaptive, in other words to use parametric forms for the basis functions in which the parameter values are adapted during training. The most successful model of this type in the context of pattern recognition is the feed-forward neural network, also known as the *multilayer perceptron*, discussed in this chapter. In fact, 'multilayer perceptron' is really a misnomer, because the model comprises multiple layers of logistic regression models (with continuous nonlinearities) rather than multiple perceptrons (with discontinuous nonlinearities). For many applications, the resulting model can be significantly more compact, and hence faster to evaluate, than a support vector machine having the same generalization performance. The price to be paid for this compactness, as with the relevance vector machine, is that the likelihood function, which forms the basis for network training, is no longer a convex function of the model parameters. In practice, however, it is often worth investing substantial computational resources during the training phase in order to obtain a compact model that is fast at processing new data.

The term 'neural network' has its origins in attempts to find mathematical representations of information processing in biological systems (McCulloch and Pitts, 1943; Widrow and Hoff, 1960; Rosenblatt, 1962; Rumelhart *et al.*, 1986). Indeed, it has been used very broadly to cover a wide range of different models, many of which have been the subject of exaggerated claims regarding their biological plausibility. From the perspective of practical applications of pattern recognition, however, biological realism would impose entirely unnecessary constraints. Our focus in this chapter is therefore on neural networks as efficient models for statistical pattern recognition. In particular, we shall restrict our attention to the specific class of neural networks that have proven to be of greatest practical value, namely the multilayer perceptron.

We begin by considering the functional form of the network model, including the specific parameterization of the basis functions, and we then discuss the problem of determining the network parameters within a maximum likelihood framework, which involves the solution of a nonlinear optimization problem. This requires the evaluation of derivatives of the log likelihood function with respect to the network parameters, and we shall see how these can be obtained efficiently using the technique of *error backpropagation*. We shall also show how the backpropagation framework can be extended to allow other derivatives to be evaluated, such as the Jacobian and Hessian matrices. Next we discuss various approaches to regularization of neural network training and the relationships between them. We also consider some extensions to the neural network model, and in particular we describe a general framework for modelling conditional probability distributions known as *mixture density networks*. Finally, we discuss the use of Bayesian treatments of neural networks. Additional background on neural network models can be found in Bishop (1995a).

5.1. Feed-forward Network Functions

The linear models for regression and classification discussed in Chapters 3 and 4, respectively, are based on linear combinations of fixed nonlinear basis functions $\phi_j(\mathbf{x})$ and take the form

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right) \tag{5.1}$$

where $f(\cdot)$ is a nonlinear activation function in the case of classification and is the identity in the case of regression. Our goal is to extend this model by making the basis functions $\phi_j(\mathbf{x})$ depend on parameters and then to allow these parameters to be adjusted, along with the coefficients $\{w_j\}$, during training. There are, of course, many ways to construct parametric nonlinear basis functions. Neural networks use basis functions that follow the same form as (5.1), so that each basis function is itself a nonlinear function of a linear combination of the inputs, where the coefficients in the linear combination are adaptive parameters.

This leads to the basic neural network model, which can be described a series of functional transformations. First we construct M linear combinations of the input variables $x_1, \ldots, x_D$ in the form

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{5.2}$$

where $j = 1, \ldots, M$, and the superscript (1) indicates that the corresponding parameters are in the first 'layer' of the network. We shall refer to the parameters $w_{ji}^{(1)}$ as *weights* and the parameters $w_{j0}^{(1)}$ as *biases*, following the nomenclature of Chapter 3. The quantities a_j are known as *activations*. Each of them is then transformed using a differentiable, nonlinear *activation function* $h(\cdot)$ to give

$$z_j = h(a_j). \tag{5.3}$$

These quantities correspond to the outputs of the basis functions in (5.1) that, in the context of neural networks, are called *hidden units*. The nonlinear functions $h(\cdot)$ are generally chosen to be sigmoidal functions such as the logistic sigmoid or the '$\tanh$' function. Following (5.1), these values are again linearly combined to give *output unit activations*

Exercise 5.1

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{5.4}$$

where $k = 1, \ldots, K$, and K is the total number of outputs. This transformation corresponds to the second layer of the network, and again the $w_{k0}^{(2)}$ are bias parameters. Finally, the output unit activations are transformed using an appropriate activation function to give a set of network outputs y_k. The choice of activation function is determined by the nature of the data and the assumed distribution of target variables

Figure 5.1 Network diagram for the two-layer neural network corresponding to (5.7). The input, hidden, and output variables are represented by nodes, and the weight parameters are represented by links between the nodes, in which the bias parameters are denoted by links coming from additional input and hidden variables x_0 and z_0. Arrows denote the direction of information flow through the network during forward propagation.

and follows the same considerations as for linear models discussed in Chapters 3 and 4. Thus for standard regression problems, the activation function is the identity so that $y_k = a_k$. Similarly, for multiple binary classification problems, each output unit activation is transformed using a logistic sigmoid function so that

$$y_k = \sigma(a_k) \tag{5.5}$$

where

$$\sigma(a) = \frac{1}{1 + \exp(-a)}. \tag{5.6}$$

Finally, for multiclass problems, a softmax activation function of the form (4.62) is used. The choice of output unit activation function is discussed in detail in Section 5.2.

We can combine these various stages to give the overall network function that, for sigmoidal output unit activation functions, takes the form

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right) \tag{5.7}$$

where the set of all weight and bias parameters have been grouped together into a vector $\mathbf{w}$. Thus the neural network model is simply a nonlinear function from a set of input variables $\{x_i\}$ to a set of output variables $\{y_k\}$ controlled by a vector $\mathbf{w}$ of adjustable parameters.

This function can be represented in the form of a network diagram as shown in Figure 5.1. The process of evaluating (5.7) can then be interpreted as a *forward propagation* of information through the network. It should be emphasized that these diagrams do not represent probabilistic graphical models of the kind to be considered in Chapter 8 because the internal nodes represent deterministic variables rather than stochastic ones. For this reason, we have adopted a slightly different graphical

notation for the two kinds of model. We shall see later how to give a probabilistic interpretation to a neural network.

As discussed in Section 3.1, the bias parameters in (5.2) can be absorbed into the set of weight parameters by defining an additional input variable x_0 whose value is clamped at $x_0 = 1$, so that (5.2) takes the form

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i. \tag{5.8}$$

We can similarly absorb the second-layer biases into the second-layer weights, so that the overall network function becomes

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)\right). \tag{5.9}$$

As can be seen from Figure 5.1, the neural network model comprises two stages of processing, each of which resembles the perceptron model of Section 4.1.7, and for this reason the neural network is also known as the *multilayer perceptron*, or MLP. A key difference compared to the perceptron, however, is that the neural network uses continuous sigmoidal nonlinearities in the hidden units, whereas the perceptron uses step-function nonlinearities. This means that the neural network function is differentiable with respect to the network parameters, and this property will play a central role in network training.

If the activation functions of all the hidden units in a network are taken to be linear, then for any such network we can always find an equivalent network without hidden units. This follows from the fact that the composition of successive linear transformations is itself a linear transformation. However, if the number of hidden units is smaller than either the number of input or output units, then the transformations that the network can generate are not the most general possible linear transformations from inputs to outputs because information is lost in the dimensionality reduction at the hidden units. In Section 12.4.2, we show that networks of linear units give rise to principal component analysis. In general, however, there is little interest in multilayer networks of linear units.

The network architecture shown in Figure 5.1 is the most commonly used one in practice. However, it is easily generalized, for instance by considering additional layers of processing each consisting of a weighted linear combination of the form (5.4) followed by an element-wise transformation using a nonlinear activation function. Note that there is some confusion in the literature regarding the terminology for counting the number of layers in such networks. Thus the network in Figure 5.1 may be described as a 3-layer network (which counts the number of layers of units, and treats the inputs as units) or sometimes as a single-hidden-layer network (which counts the number of layers of hidden units). We recommend a terminology in which Figure 5.1 is called a two-layer network, because it is the number of layers of adaptive weights that is important for determining the network properties.

Another generalization of the network architecture is to include *skip-layer* connections, each of which is associated with a corresponding adaptive parameter. For

Figure 5.2 Example of a neural network having a general feed-forward topology. Note that each hidden and output unit has an associated bias parameter (omitted for clarity).

instance, in a two-layer network these would go directly from inputs to outputs. In principle, a network with sigmoidal hidden units can always mimic skip layer connections (for bounded input values) by using a sufficiently small first-layer weight that, over its operating range, the hidden unit is effectively linear, and then compensating with a large weight value from the hidden unit to the output. In practice, however, it may be advantageous to include skip-layer connections explicitly.

Furthermore, the network can be sparse, with not all possible connections within a layer being present. We shall see an example of a sparse network architecture when we consider convolutional neural networks in Section 5.5.6.

Because there is a direct correspondence between a network diagram and its mathematical function, we can develop more general network mappings by considering more complex network diagrams. However, these must be restricted to a *feed-forward* architecture, in other words to one having no closed directed cycles, to ensure that the outputs are deterministic functions of the inputs. This is illustrated with a simple example in Figure 5.2. Each (hidden or output) unit in such a network computes a function given by

$$z_k = h\left(\sum_j w_{kj} z_j\right) \tag{5.10}$$

where the sum runs over all units that send connections to unit k (and a bias parameter is included in the summation). For a given set of values applied to the inputs of the network, successive application of (5.10) allows the activations of all units in the network to be evaluated including those of the output units.

The approximation properties of feed-forward networks have been widely studied (Funahashi, 1989; Cybenko, 1989; Hornik *et al.*, 1989; Stinchecombe and White, 1989; Cotter, 1990; Ito, 1991; Hornik, 1991; Kreinovich, 1991; Ripley, 1996) and found to be very general. Neural networks are therefore said to be *universal approximators*. For example, a two-layer network with linear outputs can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy provided the network has a sufficiently large number of hidden units. This result holds for a wide range of hidden unit activation functions, but excluding polynomials. Although such theorems are reassuring, the key problem is how to find suitable parameter values given a set of training data, and in later sections of this chapter we

Figure 5.3 Illustration of the capability of a multilayer perceptron to approximate four different functions comprising (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, have been sampled uniformly in x over the interval $(-1, 1)$ and the corresponding values of $f(x)$ evaluated. These data points are then used to train a two-layer network having 3 hidden units with '$\tanh$' activation functions and linear output units. The resulting network functions are shown by the red curves, and the outputs of the three hidden units are shown by the three dashed curves.

will show that there exist effective solutions to this problem based on both maximum likelihood and Bayesian approaches.

The capability of a two-layer network to model a broad range of functions is illustrated in Figure 5.3. This figure also shows how individual hidden units work collaboratively to approximate the final function. The role of hidden units in a simple classification problem is illustrated in Figure 5.4 using the synthetic classification data set described in Appendix A.

5.1.1 Weight-space symmetries

One property of feed-forward networks, which will play a role when we consider Bayesian model comparison, is that multiple distinct choices for the weight vector $\mathbf{w}$ can all give rise to the same mapping function from inputs to outputs (Chen *et al.*, 1993). Consider a two-layer network of the form shown in Figure 5.1 with M hidden units having '$\tanh$' activation functions and full connectivity in both layers. If we change the sign of all of the weights and the bias feeding into a particular hidden unit, then, for a given input pattern, the sign of the activation of the hidden unit will be reversed, because '$\tanh$' is an odd function, so that $\tanh(-a) = -\tanh(a)$. This transformation can be exactly compensated by changing the sign of all of the weights leading out of that hidden unit. Thus, by changing the signs of a particular group of weights (and a bias), the input–output mapping function represented by the network is unchanged, and so we have found two different weight vectors that give rise to the same mapping function. For M hidden units, there will be M such 'sign-flip'

Figure 5.4 Example of the solution of a simple two-class classification problem involving synthetic data using a neural network having two inputs, two hidden units with '$\tanh$' activation functions, and a single output having a logistic sigmoid activation function. The dashed blue lines show the $z = 0.5$ contours for each of the hidden units, and the red line shows the $y = 0.5$ decision surface for the network. For comparison, the green line denotes the optimal decision boundary computed from the distributions used to generate the data.

symmetries, and thus any given weight vector will be one of a set 2^M equivalent weight vectors .

Similarly, imagine that we interchange the values of all of the weights (and the bias) leading both into and out of a particular hidden unit with the corresponding values of the weights (and bias) associated with a different hidden unit. Again, this clearly leaves the network input–output mapping function unchanged, but it corresponds to a different choice of weight vector. For M hidden units, any given weight vector will belong to a set of $M!$ equivalent weight vectors associated with this interchange symmetry, corresponding to the $M!$ different orderings of the hidden units. The network will therefore have an overall weight-space symmetry factor of $M!2^M$. For networks with more than two layers of weights, the total level of symmetry will be given by the product of such factors, one for each layer of hidden units.

It turns out that these factors account for all of the symmetries in weight space (except for possible accidental symmetries due to specific choices for the weight values). Furthermore, the existence of these symmetries is not a particular property of the '$\tanh$' function but applies to a wide range of activation functions (Kůrková and Kainen, 1994). In many cases, these symmetries in weight space are of little practical consequence, although in Section 5.7 we shall encounter a situation in which we need to take them into account.

5.2. Network Training

So far, we have viewed neural networks as a general class of parametric nonlinear functions from a vector $\mathbf{x}$ of input variables to a vector $\mathbf{y}$ of output variables. A simple approach to the problem of determining the network parameters is to make an analogy with the discussion of polynomial curve fitting in Section 1.1, and therefore to minimize a sum-of-squares error function. Given a training set comprising a set of input vectors $\{\mathbf{x}_n\}$, where $n = 1, \ldots, N$, together with a corresponding set of

target vectors $\{\mathbf{t}_n\}$, we minimize the error function

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2. \tag{5.11}$$

However, we can provide a much more general view of network training by first giving a probabilistic interpretation to the network outputs. We have already seen many advantages of using probabilistic predictions in Section 1.5.4. Here it will also provide us with a clearer motivation both for the choice of output unit nonlinearity and the choice of error function.

We start by discussing regression problems, and for the moment we consider a single target variable t that can take any real value. Following the discussions in Section 1.2.5 and 3.1, we assume that t has a Gaussian distribution with an $\mathbf{x}$-dependent mean, which is given by the output of the neural network, so that

$$p(t|\mathbf{x}, \mathbf{w}) = \mathcal{N}\left(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}\right) \tag{5.12}$$

where β is the precision (inverse variance) of the Gaussian noise. Of course this is a somewhat restrictive assumption, and in Section 5.6 we shall see how to extend this approach to allow for more general conditional distributions. For the conditional distribution given by (5.12), it is sufficient to take the output unit activation function to be the identity, because such a network can approximate any continuous function from $\mathbf{x}$ to y. Given a data set of N independent, identically distributed observations $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, along with corresponding target values $\mathbf{t} = \{t_1, \ldots, t_N\}$, we can construct the corresponding likelihood function

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} p(t_n|\mathbf{x}_n, \mathbf{w}, \beta).$$

Taking the negative logarithm, we obtain the error function

$$\frac{\beta}{2}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 - \frac{N}{2}\ln\beta + \frac{N}{2}\ln(2\pi) \tag{5.13}$$

which can be used to learn the parameters $\mathbf{w}$ and β. In Section 5.7, we shall discuss the Bayesian treatment of neural networks, while here we consider a maximum likelihood approach. Note that in the neural networks literature, it is usual to consider the minimization of an error function rather than the maximization of the (log) likelihood, and so here we shall follow this convention. Consider first the determination of $\mathbf{w}$. Maximizing the likelihood function is equivalent to minimizing the sum-of-squares error function given by

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 \tag{5.14}$$

where we have discarded additive and multiplicative constants. The value of $\mathbf{w}$ found by minimizing $E(\mathbf{w})$ will be denoted $\mathbf{w}_{\mathrm{ML}}$ because it corresponds to the maximum likelihood solution. In practice, the nonlinearity of the network function $y(\mathbf{x}_n, \mathbf{w})$ causes the error $E(\mathbf{w})$ to be nonconvex, and so in practice local maxima of the likelihood may be found, corresponding to local minima of the error function, as discussed in Section 5.2.1.

Having found $\mathbf{w}_{\mathrm{ML}}$, the value of β can be found by minimizing the negative log likelihood to give

$$\frac{1}{\beta_{\mathrm{ML}}} = \frac{1}{N}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}_{\mathrm{ML}}) - t_n\}^2. \tag{5.15}$$

Note that this can be evaluated once the iterative optimization required to find $\mathbf{w}_{\mathrm{ML}}$ is completed. If we have multiple target variables, and we assume that they are independent conditional on $\mathbf{x}$ and $\mathbf{w}$ with shared noise precision β, then the conditional distribution of the target values is given by

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \mathcal{N}\left(\mathbf{t}|\mathbf{y}(\mathbf{x}, \mathbf{w}), \beta^{-1}\mathbf{I}\right). \tag{5.16}$$

Following the same argument as for a single target variable, we see that the maximum likelihood weights are determined by minimizing the sum-of-squares error function (5.11). The noise precision is then given by

Exercise 5.2

$$\frac{1}{\beta_{\mathrm{ML}}} = \frac{1}{NK}\sum_{n=1}^{N}\|\mathbf{y}(\mathbf{x}_n, \mathbf{w}_{\mathrm{ML}}) - \mathbf{t}_n\|^2 \tag{5.17}$$

where K is the number of target variables. The assumption of independence can be dropped at the expense of a slightly more complex optimization problem.

Exercise 5.3

Recall from Section 4.3.6 that there is a natural pairing of the error function (given by the negative log likelihood) and the output unit activation function. In the regression case, we can view the network as having an output activation function that is the identity, so that $y_k = a_k$. The corresponding sum-of-squares error function has the property

$$\frac{\partial E}{\partial a_k} = y_k - t_k \tag{5.18}$$

which we shall make use of when discussing error backpropagation in Section 5.3.

Now consider the case of binary classification in which we have a single target variable t such that $t = 1$ denotes class $\mathcal{C}_1$ and $t = 0$ denotes class $\mathcal{C}_2$. Following the discussion of canonical link functions in Section 4.3.6, we consider a network having a single output whose activation function is a logistic sigmoid

$$y = \sigma(a) \equiv \frac{1}{1 + \exp(-a)} \tag{5.19}$$

so that $0 \leqslant y(\mathbf{x}, \mathbf{w}) \leqslant 1$. We can interpret $y(\mathbf{x}, \mathbf{w})$ as the conditional probability $p(\mathcal{C}_1|\mathbf{x})$, with $p(\mathcal{C}_2|\mathbf{x})$ given by $1 - y(\mathbf{x}, \mathbf{w})$. The conditional distribution of targets given inputs is then a Bernoulli distribution of the form

$$p(t|\mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w})^t \left\{1 - y(\mathbf{x}, \mathbf{w})\right\}^{1-t}. \tag{5.20}$$

If we consider a training set of independent observations, then the error function, which is given by the negative log likelihood, is then a *cross-entropy* error function of the form

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\} \tag{5.21}$$

where y_n denotes $y(\mathbf{x}_n, \mathbf{w})$. Note that there is no analogue of the noise precision β because the target values are assumed to be correctly labelled. However, the model is easily extended to allow for labelling errors. Simard *et al.* (2003) found that using the cross-entropy error function instead of the sum-of-squares for a classification problem leads to faster training as well as improved generalization.

Exercise 5.4

If we have K separate binary classifications to perform, then we can use a network having K outputs each of which has a logistic sigmoid activation function. Associated with each output is a binary class label $t_k \in \{0, 1\}$, where $k = 1, \ldots, K$. If we assume that the class labels are independent, given the input vector, then the conditional distribution of the targets is

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \prod_{k=1}^{K} y_k(\mathbf{x}, \mathbf{w})^{t_k} \left[1 - y_k(\mathbf{x}, \mathbf{w})\right]^{1-t_k}. \tag{5.22}$$

Taking the negative logarithm of the corresponding likelihood function then gives the following error function

Exercise 5.5

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} \{t_{nk} \ln y_{nk} + (1 - t_{nk}) \ln(1 - y_{nk})\} \tag{5.23}$$

where y_{nk} denotes $y_k(\mathbf{x}_n, \mathbf{w})$. Again, the derivative of the error function with respect to the activation for a particular output unit takes the form (5.18) just as in the regression case.

Exercise 5.6

It is interesting to contrast the neural network solution to this problem with the corresponding approach based on a linear classification model of the kind discussed in Chapter 4. Suppose that we are using a standard two-layer network of the kind shown in Figure 5.1. We see that the weight parameters in the first layer of the network are shared between the various outputs, whereas in the linear model each classification problem is solved independently. The first layer of the network can be viewed as performing a nonlinear feature extraction, and the sharing of features between the different outputs can save on computation and can also lead to improved generalization.

Finally, we consider the standard multiclass classification problem in which each input is assigned to one of K mutually exclusive classes. The binary target variables $t_k \in \{0, 1\}$ have a 1-of-K coding scheme indicating the class, and the network outputs are interpreted as $y_k(\mathbf{x}, \mathbf{w}) = p(t_k = 1|\mathbf{x})$, leading to the following error function

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{kn} \ln y_k(\mathbf{x}_n, \mathbf{w}). \tag{5.24}$$

Figure 5.5 Geometrical view of the error function $E(\mathbf{w})$ as a surface sitting over weight space. Point $\mathbf{w}_A$ is a local minimum and $\mathbf{w}_B$ is the global minimum. At any point $\mathbf{w}_C$, the local gradient of the error surface is given by the vector ∇E.

Following the discussion of Section 4.3.4, we see that the output unit activation function, which corresponds to the canonical link, is given by the softmax function

$$y_k(\mathbf{x}, \mathbf{w}) = \frac{\exp(a_k(\mathbf{x}, \mathbf{w}))}{\sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))} \tag{5.25}$$

which satisfies $0 \leqslant y_k \leqslant 1$ and $\sum_k y_k = 1$. Note that the $y_k(\mathbf{x}, \mathbf{w})$ are unchanged if a constant is added to all of the $a_k(\mathbf{x}, \mathbf{w})$, causing the error function to be constant for some directions in weight space. This degeneracy is removed if an appropriate regularization term (Section 5.5) is added to the error function.

Once again, the derivative of the error function with respect to the activation for a particular output unit takes the familiar form (5.18).

Exercise 5.7

In summary, there is a natural choice of both output unit activation function and matching error function, according to the type of problem being solved. For regression we use linear outputs and a sum-of-squares error, for (multiple independent) binary classifications we use logistic sigmoid outputs and a cross-entropy error function, and for multiclass classification we use softmax outputs with the corresponding multiclass cross-entropy error function. For classification problems involving two classes, we can use a single logistic sigmoid output, or alternatively we can use a network with two outputs having a softmax output activation function.

5.2.1 Parameter optimization

We turn next to the task of finding a weight vector $\mathbf{w}$ which minimizes the chosen function $E(\mathbf{w})$. At this point, it is useful to have a geometrical picture of the error function, which we can view as a surface sitting over weight space as shown in Figure 5.5. First note that if we make a small step in weight space from $\mathbf{w}$ to $\mathbf{w} + \delta\mathbf{w}$ then the change in the error function is $\delta E \simeq \delta\mathbf{w}^{\mathrm{T}} \nabla E(\mathbf{w})$, where the vector $\nabla E(\mathbf{w})$ points in the direction of greatest rate of increase of the error function. Because the error $E(\mathbf{w})$ is a smooth continuous function of $\mathbf{w}$, its smallest value will occur at a

point in weight space such that the gradient of the error function vanishes, so that

$$\nabla E(\mathbf{w}) = 0 \tag{5.26}$$

as otherwise we could make a small step in the direction of $-\nabla E(\mathbf{w})$ and thereby further reduce the error. Points at which the gradient vanishes are called stationary points, and may be further classified into minima, maxima, and saddle points.

Our goal is to find a vector $\mathbf{w}$ such that $E(\mathbf{w})$ takes its smallest value. However, the error function typically has a highly nonlinear dependence on the weights and bias parameters, and so there will be many points in weight space at which the gradient vanishes (or is numerically very small). Indeed, from the discussion in Section 5.1.1 we see that for any point $\mathbf{w}$ that is a local minimum, there will be other points in weight space that are equivalent minima. For instance, in a two-layer network of the kind shown in Figure 5.1, with M hidden units, each point in weight space is a member of a family of $M!2^M$ equivalent points.

Section 5.1.1

Furthermore, there will typically be multiple inequivalent stationary points and in particular multiple inequivalent minima. A minimum that corresponds to the smallest value of the error function for any weight vector is said to be a *global minimum*. Any other minima corresponding to higher values of the error function are said to be *local minima*. For a successful application of neural networks, it may not be necessary to find the global minimum (and in general it will not be known whether the global minimum has been found) but it may be necessary to compare several local minima in order to find a sufficiently good solution.

Because there is clearly no hope of finding an analytical solution to the equation $\nabla E(\mathbf{w}) = 0$ we resort to iterative numerical procedures. The optimization of continuous nonlinear functions is a widely studied problem and there exists an extensive literature on how to solve it efficiently. Most techniques involve choosing some initial value $\mathbf{w}^{(0)}$ for the weight vector and then moving through weight space in a succession of steps of the form

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)} \tag{5.27}$$

where τ labels the iteration step. Different algorithms involve different choices for the weight vector update $\Delta\mathbf{w}^{(\tau)}$. Many algorithms make use of gradient information and therefore require that, after each update, the value of $\nabla E(\mathbf{w})$ is evaluated at the new weight vector $\mathbf{w}^{(\tau+1)}$. In order to understand the importance of gradient information, it is useful to consider a local approximation to the error function based on a Taylor expansion.

5.2.2 Local quadratic approximation

Insight into the optimization problem, and into the various techniques for solving it, can be obtained by considering a local quadratic approximation to the error function.

Consider the Taylor expansion of $E(\mathbf{w})$ around some point $\widehat{\mathbf{w}}$ in weight space

$$E(\mathbf{w}) \simeq E(\widehat{\mathbf{w}}) + (\mathbf{w} - \widehat{\mathbf{w}})^{\mathrm{T}}\mathbf{b} + \frac{1}{2}(\mathbf{w} - \widehat{\mathbf{w}})^{\mathrm{T}}\mathbf{H}(\mathbf{w} - \widehat{\mathbf{w}}) \tag{5.28}$$

where cubic and higher terms have been omitted. Here $\mathbf{b}$ is defined to be the gradient of E evaluated at $\widehat{\mathbf{w}}$

$$\mathbf{b} \equiv \nabla E|_{\mathbf{w}=\widehat{\mathbf{w}}} \tag{5.29}$$

and the Hessian matrix $\mathbf{H} = \nabla\nabla E$ has elements

$$(\mathbf{H})_{ij} \equiv \left.\frac{\partial E}{\partial w_i \partial w_j}\right|_{\mathbf{w}=\widehat{\mathbf{w}}}. \tag{5.30}$$

From (5.28), the corresponding local approximation to the gradient is given by

$$\nabla E \simeq \mathbf{b} + \mathbf{H}(\mathbf{w} - \widehat{\mathbf{w}}). \tag{5.31}$$

For points $\mathbf{w}$ that are sufficiently close to $\widehat{\mathbf{w}}$, these expressions will give reasonable approximations for the error and its gradient.

Consider the particular case of a local quadratic approximation around a point $\mathbf{w}^\star$ that is a minimum of the error function. In this case there is no linear term, because $\nabla E = 0$ at $\mathbf{w}^\star$, and (5.28) becomes

$$E(\mathbf{w}) = E(\mathbf{w}^\star) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^\star)^{\mathrm{T}}\mathbf{H}(\mathbf{w} - \mathbf{w}^\star) \tag{5.32}$$

where the Hessian $\mathbf{H}$ is evaluated at $\mathbf{w}^\star$. In order to interpret this geometrically, consider the eigenvalue equation for the Hessian matrix

$$\mathbf{H}\mathbf{u}_i = \lambda_i \mathbf{u}_i \tag{5.33}$$

where the eigenvectors $\mathbf{u}_i$ form a complete orthonormal set (Appendix C) so that

$$\mathbf{u}_i^{\mathrm{T}}\mathbf{u}_j = \delta_{ij}. \tag{5.34}$$

We now expand $(\mathbf{w} - \mathbf{w}^\star)$ as a linear combination of the eigenvectors in the form

$$\mathbf{w} - \mathbf{w}^\star = \sum_i \alpha_i \mathbf{u}_i. \tag{5.35}$$

This can be regarded as a transformation of the coordinate system in which the origin is translated to the point $\mathbf{w}^\star$, and the axes are rotated to align with the eigenvectors (through the orthogonal matrix whose columns are the $\mathbf{u}_i$), and is discussed in more detail in Appendix C. Substituting (5.35) into (5.32), and using (5.33) and (5.34), allows the error function to be written in the form

$$E(\mathbf{w}) = E(\mathbf{w}^\star) + \frac{1}{2}\sum_i \lambda_i \alpha_i^2. \tag{5.36}$$

A matrix $\mathbf{H}$ is said to be *positive definite* if, and only if,

$$\mathbf{v}^{\mathrm{T}}\mathbf{H}\mathbf{v} > 0 \qquad \text{for all } \mathbf{v}. \tag{5.37}$$

Figure 5.6 In the neighbourhood of a minimum $\mathbf{w}^\star$, the error function can be approximated by a quadratic. Contours of constant error are then ellipses whose axes are aligned with the eigenvectors $\mathbf{u}_i$ of the Hessian matrix, with lengths that are inversely proportional to the square roots of the corresponding eigenvectors λ_i.

Because the eigenvectors $\{\mathbf{u}_i\}$ form a complete set, an arbitrary vector $\mathbf{v}$ can be written in the form

$$\mathbf{v} = \sum_i c_i \mathbf{u}_i. \tag{5.38}$$

From (5.33) and (5.34), we then have

$$\mathbf{v}^{\mathrm{T}}\mathbf{H}\mathbf{v} = \sum_i c_i^2 \lambda_i \tag{5.39}$$

Exercise 5.10 and so $\mathbf{H}$ will be positive definite if, and only if, all of its eigenvalues are positive. In the new coordinate system, whose basis vectors are given by the eigenvectors *Exercise 5.11* $\{\mathbf{u}_i\}$, the contours of constant E are ellipses centred on the origin, as illustrated in Figure 5.6. For a one-dimensional weight space, a stationary point $w^\star$ will be a minimum if

$$\left.\frac{\partial^2 E}{\partial w^2}\right|_{w^\star} > 0. \tag{5.40}$$

The corresponding result in D-dimensions is that the Hessian matrix, evaluated at *Exercise 5.12* $\mathbf{w}^\star$, should be positive definite.

5.2.3 Use of gradient information

As we shall see in Section 5.3, it is possible to evaluate the gradient of an error function efficiently by means of the backpropagation procedure. The use of this gradient information can lead to significant improvements in the speed with which the minima of the error function can be located. We can see why this is so, as follows.

In the quadratic approximation to the error function, given in (5.28), the error surface is specified by the quantities $\mathbf{b}$ and $\mathbf{H}$, which contain a total of $W(W+$ *Exercise 5.13* $3)/2$ independent elements (because the matrix $\mathbf{H}$ is symmetric), where W is the dimensionality of $\mathbf{w}$ (i.e., the total number of adaptive parameters in the network). The location of the minimum of this quadratic approximation therefore depends on $O(W^2)$ parameters, and we should not expect to be able to locate the minimum until we have gathered $O(W^2)$ independent pieces of information. If we do not make use of gradient information, we would expect to have to perform $O(W^2)$ function

evaluations, each of which would require $O(W)$ steps. Thus, the computational effort needed to find the minimum using such an approach would be $O(W^3)$.

Now compare this with an algorithm that makes use of the gradient information. Because each evaluation of ∇E brings W items of information, we might hope to find the minimum of the function in $O(W)$ gradient evaluations. As we shall see, by using error backpropagation, each such evaluation takes only $O(W)$ steps and so the minimum can now be found in $O(W^2)$ steps. For this reason, the use of gradient information forms the basis of practical algorithms for training neural networks.

5.2.4 Gradient descent optimization

The simplest approach to using gradient information is to choose the weight update in (5.27) to comprise a small step in the direction of the negative gradient, so that

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)}) \tag{5.41}$$

where the parameter $\eta > 0$ is known as the *learning rate*. After each such update, the gradient is re-evaluated for the new weight vector and the process repeated. Note that the error function is defined with respect to a training set, and so each step requires that the entire training set be processed in order to evaluate ∇E. Techniques that use the whole data set at once are called *batch* methods. At each step the weight vector is moved in the direction of the greatest rate of decrease of the error function, and so this approach is known as *gradient descent* or *steepest descent*. Although such an approach might intuitively seem reasonable, in fact it turns out to be a poor algorithm, for reasons discussed in Bishop and Nabney (2008).

For batch optimization, there are more efficient methods, such as *conjugate gradients* and *quasi-Newton* methods, which are much more robust and much faster than simple gradient descent (Gill *et al.*, 1981; Fletcher, 1987; Nocedal and Wright, 1999). Unlike gradient descent, these algorithms have the property that the error function always decreases at each iteration unless the weight vector has arrived at a local or global minimum.

In order to find a sufficiently good minimum, it may be necessary to run a gradient-based algorithm multiple times, each time using a different randomly chosen starting point, and comparing the resulting performance on an independent validation set.

There is, however, an on-line version of gradient descent that has proved useful in practice for training neural networks on large data sets (Le Cun *et al.*, 1989). Error functions based on maximum likelihood for a set of independent observations comprise a sum of terms, one for each data point

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}). \tag{5.42}$$

On-line gradient descent, also known as *sequential gradient descent* or *stochastic gradient descent*, makes an update to the weight vector based on one data point at a time, so that

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)}). \tag{5.43}$$

This update is repeated by cycling through the data either in sequence or by selecting points at random with replacement. There are of course intermediate scenarios in which the updates are based on batches of data points.

One advantage of on-line methods compared to batch methods is that the former handle redundancy in the data much more efficiently. To see, this consider an extreme example in which we take a data set and double its size by duplicating every data point. Note that this simply multiplies the error function by a factor of 2 and so is equivalent to using the original error function. Batch methods will require double the computational effort to evaluate the batch error function gradient, whereas on-line methods will be unaffected. Another property of on-line gradient descent is the possibility of escaping from local minima, since a stationary point with respect to the error function for the whole data set will generally not be a stationary point for each data point individually.

Nonlinear optimization algorithms, and their practical application to neural network training, are discussed in detail in Bishop and Nabney (2008).

5.3. Error Backpropagation

Our goal in this section is to find an efficient technique for evaluating the gradient of an error function $E(\mathbf{w})$ for a feed-forward neural network. We shall see that this can be achieved using a local message passing scheme in which information is sent alternately forwards and backwards through the network and is known as *error backpropagation*, or sometimes simply as *backprop*.

It should be noted that the term backpropagation is used in the neural computing literature to mean a variety of different things. For instance, the multilayer perceptron architecture is sometimes called a backpropagation network. The term backpropagation is also used to describe the training of a multilayer perceptron using gradient descent applied to a sum-of-squares error function. In order to clarify the terminology, it is useful to consider the nature of the training process more carefully. Most training algorithms involve an iterative procedure for minimization of an error function, with adjustments to the weights being made in a sequence of steps. At each such step, we can distinguish between two distinct stages. In the first stage, the derivatives of the error function with respect to the weights must be evaluated. As we shall see, the important contribution of the backpropagation technique is in providing a computationally efficient method for evaluating such derivatives. Because it is at this stage that errors are propagated backwards through the network, we shall use the term backpropagation specifically to describe the evaluation of derivatives. In the second stage, the derivatives are then used to compute the adjustments to be made to the weights. The simplest such technique, and the one originally considered by Rumelhart *et al.* (1986), involves gradient descent. It is important to recognize that the two stages are distinct. Thus, the first stage, namely the propagation of errors backwards through the network in order to evaluate derivatives, can be applied to many other kinds of network and not just the multilayer perceptron. It can also be applied to error functions other that just the simple sum-of-squares, and to the eval-

uation of other derivatives such as the Jacobian and Hessian matrices, as we shall see later in this chapter. Similarly, the second stage of weight adjustment using the calculated derivatives can be tackled using a variety of optimization schemes, many of which are substantially more powerful than simple gradient descent.

5.3.1 Evaluation of error-function derivatives

We now derive the backpropagation algorithm for a general network having arbitrary feed-forward topology, arbitrary differentiable nonlinear activation functions, and a broad class of error function. The resulting formulae will then be illustrated using a simple layered network structure having a single layer of sigmoidal hidden units together with a sum-of-squares error.

Many error functions of practical interest, for instance those defined by maximum likelihood for a set of i.i.d. data, comprise a sum of terms, one for each data point in the training set, so that

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}). \tag{5.44}$$

Here we shall consider the problem of evaluating $\nabla E_n(\mathbf{w})$ for one such term in the error function. This may be used directly for sequential optimization, or the results can be accumulated over the training set in the case of batch methods.

Consider first a simple linear model in which the outputs y_k are linear combinations of the input variables x_i so that

$$y_k = \sum_i w_{ki} x_i \tag{5.45}$$

together with an error function that, for a particular input pattern n, takes the form

$$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2 \tag{5.46}$$

where $y_{nk} = y_k(\mathbf{x}_n, \mathbf{w})$. The gradient of this error function with respect to a weight w_{ji} is given by

$$\frac{\partial E_n}{\partial w_{ji}} = (y_{nj} - t_{nj}) x_{ni} \tag{5.47}$$

which can be interpreted as a 'local' computation involving the product of an 'error signal' $y_{nj} - t_{nj}$ associated with the output end of the link w_{ji} and the variable x_{ni} associated with the input end of the link. In Section 4.3.2, we saw how a similar formula arises with the logistic sigmoid activation function together with the cross entropy error function, and similarly for the softmax activation function together with its matching cross-entropy error function. We shall now see how this simple result extends to the more complex setting of multilayer feed-forward networks.

In a general feed-forward network, each unit computes a weighted sum of its inputs of the form

$$a_j = \sum_i w_{ji} z_i \tag{5.48}$$

where z_i is the activation of a unit, or input, that sends a connection to unit j, and w_{ji} is the weight associated with that connection. In Section 5.1, we saw that biases can be included in this sum by introducing an extra unit, or input, with activation fixed at $+1$. We therefore do not need to deal with biases explicitly. The sum in (5.48) is transformed by a nonlinear activation function $h(\cdot)$ to give the activation z_j of unit j in the form

$$z_j = h(a_j). \tag{5.49}$$

Note that one or more of the variables z_i in the sum in (5.48) could be an input, and similarly, the unit j in (5.49) could be an output.

For each pattern in the training set, we shall suppose that we have supplied the corresponding input vector to the network and calculated the activations of all of the hidden and output units in the network by successive application of (5.48) and (5.49). This process is often called *forward propagation* because it can be regarded as a forward flow of information through the network.

Now consider the evaluation of the derivative of E_n with respect to a weight w_{ji}. The outputs of the various units will depend on the particular input pattern n. However, in order to keep the notation uncluttered, we shall omit the subscript n from the network variables. First we note that E_n depends on the weight w_{ji} only via the summed input a_j to unit j. We can therefore apply the chain rule for partial derivatives to give

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}. \tag{5.50}$$

We now introduce a useful notation

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} \tag{5.51}$$

where the δ's are often referred to as *errors* for reasons we shall see shortly. Using (5.48), we can write

$$\frac{\partial a_j}{\partial w_{ji}} = z_i. \tag{5.52}$$

Substituting (5.51) and (5.52) into (5.50), we then obtain

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i. \tag{5.53}$$

Equation (5.53) tells us that the required derivative is obtained simply by multiplying the value of δ for the unit at the output end of the weight by the value of z for the unit at the input end of the weight (where $z = 1$ in the case of a bias). Note that this takes the same form as for the simple linear model considered at the start of this section. Thus, in order to evaluate the derivatives, we need only to calculate the value of δ_j for each hidden and output unit in the network, and then apply (5.53).

As we have seen already, for the output units, we have

$$\delta_k = y_k - t_k \tag{5.54}$$

Figure 5.7 Illustration of the calculation of δ_j for hidden unit j by backpropagation of the δ's from those units k to which unit j sends connections. The blue arrow denotes the direction of information flow during forward propagation, and the red arrows indicate the backward propagation of error information.

provided we are using the canonical link as the output-unit activation function. To evaluate the δ's for hidden units, we again make use of the chain rule for partial derivatives,

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \tag{5.55}$$

where the sum runs over all units k to which unit j sends connections. The arrangement of units and weights is illustrated in Figure 5.7. Note that the units labelled k could include other hidden units and/or output units. In writing down (5.55), we are making use of the fact that variations in a_j give rise to variations in the error function only through variations in the variables a_k. If we now substitute the definition of δ given by (5.51) into (5.55), and make use of (5.48) and (5.49), we obtain the following *backpropagation* formula

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \tag{5.56}$$

which tells us that the value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network, as illustrated in Figure 5.7. Note that the summation in (5.56) is taken over the first index on w_{kj} (corresponding to backward propagation of information through the network), whereas in the forward propagation equation (5.10) it is taken over the second index. Because we already know the values of the δ's for the output units, it follows that by recursively applying (5.56) we can evaluate the δ's for all of the hidden units in a feed-forward network, regardless of its topology.

The backpropagation procedure can therefore be summarized as follows.

Error Backpropagation

1. Apply an input vector $\mathbf{x}_n$ to the network and forward propagate through the network using (5.48) and (5.49) to find the activations of all the hidden and output units.
2. Evaluate the δ_k for all the output units using (5.54).
3. Backpropagate the δ's using (5.56) to obtain δ_j for each hidden unit in the network.
4. Use (5.53) to evaluate the required derivatives.

For batch methods, the derivative of the total error E can then be obtained by repeating the above steps for each pattern in the training set and then summing over all patterns:

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}}. \tag{5.57}$$

In the above derivation we have implicitly assumed that each hidden or output unit in the network has the same activation function $h(\cdot)$. The derivation is easily generalized, however, to allow different units to have individual activation functions, simply by keeping track of which form of $h(\cdot)$ goes with which unit.

5.3.2 A simple example

The above derivation of the backpropagation procedure allowed for general forms for the error function, the activation functions, and the network topology. In order to illustrate the application of this algorithm, we shall consider a particular example. This is chosen both for its simplicity and for its practical importance, because many applications of neural networks reported in the literature make use of this type of network. Specifically, we shall consider a two-layer network of the form illustrated in Figure 5.1, together with a sum-of-squares error, in which the output units have linear activation functions, so that $y_k = a_k$, while the hidden units have logistic sigmoid activation functions given by

$$h(a) \equiv \tanh(a) \tag{5.58}$$

where

$$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}. \tag{5.59}$$

A useful feature of this function is that its derivative can be expressed in a particularly simple form:

$$h'(a) = 1 - h(a)^2. \tag{5.60}$$

We also consider a standard sum-of-squares error function, so that for pattern n the error is given by

$$E_n = \frac{1}{2}\sum_{k=1}^{K}(y_k - t_k)^2 \tag{5.61}$$

where y_k is the activation of output unit k, and t_k is the corresponding target, for a particular input pattern $\mathbf{x}_n$.

For each pattern in the training set in turn, we first perform a forward propagation using

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i \tag{5.62}$$

$$z_j = \tanh(a_j) \tag{5.63}$$

$$y_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j. \tag{5.64}$$

Next we compute the δ's for each output unit using

$$\delta_k = y_k - t_k. \tag{5.65}$$

Then we backpropagate these to obtain δs for the hidden units using

$$\delta_j = (1 - z_j^2) \sum_{k=1}^{K} w_{kj} \delta_k. \tag{5.66}$$

Finally, the derivatives with respect to the first-layer and second-layer weights are given by

$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \delta_j x_i, \qquad \frac{\partial E_n}{\partial w_{kj}^{(2)}} = \delta_k z_j. \tag{5.67}$$

5.3.3 Efficiency of backpropagation

One of the most important aspects of backpropagation is its computational efficiency. To understand this, let us examine how the number of computer operations required to evaluate the derivatives of the error function scales with the total number W of weights and biases in the network. A single evaluation of the error function (for a given input pattern) would require $O(W)$ operations, for sufficiently large W. This follows from the fact that, except for a network with very sparse connections, the number of weights is typically much greater than the number of units, and so the bulk of the computational effort in forward propagation is concerned with evaluating the sums in (5.48), with the evaluation of the activation functions representing a small overhead. Each term in the sum in (5.48) requires one multiplication and one addition, leading to an overall computational cost that is $O(W)$.

An alternative approach to backpropagation for computing the derivatives of the error function is to use finite differences. This can be done by perturbing each weight in turn, and approximating the derivatives by the expression

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji})}{\epsilon} + O(\epsilon) \tag{5.68}$$

where $\epsilon \ll 1$. In a software simulation, the accuracy of the approximation to the derivatives can be improved by making ϵ smaller, until numerical roundoff problems arise. The accuracy of the finite differences method can be improved significantly by using symmetrical *central differences* of the form

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{E_n(w_{ji} + \epsilon) - E_n(w_{ji} - \epsilon)}{2\epsilon} + O(\epsilon^2). \tag{5.69}$$

Exercise 5.14 In this case, the $O(\epsilon)$ corrections cancel, as can be verified by Taylor expansion on the right-hand side of (5.69), and so the residual corrections are $O(\epsilon^2)$. The number of computational steps is, however, roughly doubled compared with (5.68).

The main problem with numerical differentiation is that the highly desirable $O(W)$ scaling has been lost. Each forward propagation requires $O(W)$ steps, and

Figure 5.8 Illustration of a modular pattern recognition system in which the Jacobian matrix can be used to backpropagate error signals from the outputs through to earlier modules in the system.

there are W weights in the network each of which must be perturbed individually, so that the overall scaling is $O(W^2)$.

However, numerical differentiation plays an important role in practice, because a comparison of the derivatives calculated by backpropagation with those obtained using central differences provides a powerful check on the correctness of any software implementation of the backpropagation algorithm. When training networks in practice, derivatives should be evaluated using backpropagation, because this gives the greatest accuracy and numerical efficiency. However, the results should be compared with numerical differentiation using (5.69) for some test cases in order to check the correctness of the implementation.

5.3.4 The Jacobian matrix

We have seen how the derivatives of an error function with respect to the weights can be obtained by the propagation of errors backwards through the network. The technique of backpropagation can also be applied to the calculation of other derivatives. Here we consider the evaluation of the *Jacobian* matrix, whose elements are given by the derivatives of the network outputs with respect to the inputs

$$J_{ki} \equiv \frac{\partial y_k}{\partial x_i} \tag{5.70}$$

where each such derivative is evaluated with all other inputs held fixed. Jacobian matrices play a useful role in systems built from a number of distinct modules, as illustrated in Figure 5.8. Each module can comprise a fixed or adaptive function, which can be linear or nonlinear, so long as it is differentiable. Suppose we wish to minimize an error function E with respect to the parameter w in Figure 5.8. The derivative of the error function is given by

$$\frac{\partial E}{\partial w} = \sum_{k,j} \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial z_j} \frac{\partial z_j}{\partial w} \tag{5.71}$$

in which the Jacobian matrix for the red module in Figure 5.8 appears in the middle term.

Because the Jacobian matrix provides a measure of the local sensitivity of the outputs to changes in each of the input variables, it also allows any known errors Δx_i

associated with the inputs to be propagated through the trained network in order to estimate their contribution Δy_k to the errors at the outputs, through the relation

$$\Delta y_k \simeq \sum_i \frac{\partial y_k}{\partial x_i} \Delta x_i \tag{5.72}$$

which is valid provided the $|\Delta x_i|$ are small. In general, the network mapping represented by a trained neural network will be nonlinear, and so the elements of the Jacobian matrix will not be constants but will depend on the particular input vector used. Thus (5.72) is valid only for small perturbations of the inputs, and the Jacobian itself must be re-evaluated for each new input vector.

The Jacobian matrix can be evaluated using a backpropagation procedure that is similar to the one derived earlier for evaluating the derivatives of an error function with respect to the weights. We start by writing the element J_{ki} in the form

$$\begin{aligned} J_{ki} = \frac{\partial y_k}{\partial x_i} &= \sum_j \frac{\partial y_k}{\partial a_j} \frac{\partial a_j}{\partial x_i} \\ &= \sum_j w_{ji} \frac{\partial y_k}{\partial a_j} \end{aligned} \tag{5.73}$$

where we have made use of (5.48). The sum in (5.73) runs over all units j to which the input unit i sends connections (for example, over all units in the first hidden layer in the layered topology considered earlier). We now write down a recursive backpropagation formula to determine the derivatives $\partial y_k / \partial a_j$

$$\begin{aligned} \frac{\partial y_k}{\partial a_j} &= \sum_l \frac{\partial y_k}{\partial a_l} \frac{\partial a_l}{\partial a_j} \\ &= h'(a_j) \sum_l w_{lj} \frac{\partial y_k}{\partial a_l} \end{aligned} \tag{5.74}$$

where the sum runs over all units l to which unit j sends connections (corresponding to the first index of w_{lj}). Again, we have made use of (5.48) and (5.49). This backpropagation starts at the output units for which the required derivatives can be found directly from the functional form of the output-unit activation function. For instance, if we have individual sigmoidal activation functions at each output unit, then

$$\frac{\partial y_k}{\partial a_j} = \delta_{kj} \sigma'(a_j) \tag{5.75}$$

whereas for softmax outputs we have

$$\frac{\partial y_k}{\partial a_j} = \delta_{kj} y_k - y_k y_j. \tag{5.76}$$

We can summarize the procedure for evaluating the Jacobian matrix as follows. Apply the input vector corresponding to the point in input space at which the Jacobian matrix is to be found, and forward propagate in the usual way to obtain the

activations of all of the hidden and output units in the network. Next, for each row k of the Jacobian matrix, corresponding to the output unit k, backpropagate using the recursive relation (5.74), starting with (5.75) or (5.76), for all of the hidden units in the network. Finally, use (5.73) to do the backpropagation to the inputs. The Jacobian can also be evaluated using an alternative *forward* propagation formalism, which can be derived in an analogous way to the backpropagation approach given here.

Exercise 5.15

Again, the implementation of such algorithms can be checked by using numerical differentiation in the form

$$\frac{\partial y_k}{\partial x_i} = \frac{y_k(x_i + \epsilon) - y_k(x_i - \epsilon)}{2\epsilon} + O(\epsilon^2) \tag{5.77}$$

which involves $2D$ forward propagations for a network having D inputs.

5.4. The Hessian Matrix

We have shown how the technique of backpropagation can be used to obtain the first derivatives of an error function with respect to the weights in the network. Backpropagation can also be used to evaluate the second derivatives of the error, given by

$$\frac{\partial^2 E}{\partial w_{ji} \partial w_{lk}}. \tag{5.78}$$

Note that it is sometimes convenient to consider all of the weight and bias parameters as elements w_i of a single vector, denoted $\mathbf{w}$, in which case the second derivatives form the elements H_{ij} of the *Hessian* matrix $\mathbf{H}$, where $i, j \in \{1, \ldots, W\}$ and W is the total number of weights and biases. The Hessian plays an important role in many aspects of neural computing, including the following:

1. Several nonlinear optimization algorithms used for training neural networks are based on considerations of the second-order properties of the error surface, which are controlled by the Hessian matrix (Bishop and Nabney, 2008).

2. The Hessian forms the basis of a fast procedure for re-training a feed-forward network following a small change in the training data (Bishop, 1991).

3. The inverse of the Hessian has been used to identify the least significant weights in a network as part of network 'pruning' algorithms (Le Cun *et al.*, 1990).

4. The Hessian plays a central role in the Laplace approximation for a Bayesian neural network (see Section 5.7). Its inverse is used to determine the predictive distribution for a trained network, its eigenvalues determine the values of hyperparameters, and its determinant is used to evaluate the model evidence.

Various approximation schemes have been used to evaluate the Hessian matrix for a neural network. However, the Hessian can also be calculated exactly using an extension of the backpropagation technique.

An important consideration for many applications of the Hessian is the efficiency with which it can be evaluated. If there are W parameters (weights and biases) in the network, then the Hessian matrix has dimensions $W \times W$ and so the computational effort needed to evaluate the Hessian will scale like $O(W^2)$ for each pattern in the data set. As we shall see, there are efficient methods for evaluating the Hessian whose scaling is indeed $O(W^2)$.

5.4.1 Diagonal approximation

Some of the applications for the Hessian matrix discussed above require the inverse of the Hessian, rather than the Hessian itself. For this reason, there has been some interest in using a diagonal approximation to the Hessian, in other words one that simply replaces the off-diagonal elements with zeros, because its inverse is trivial to evaluate. Again, we shall consider an error function that consists of a sum of terms, one for each pattern in the data set, so that $E = \sum_n E_n$. The Hessian can then be obtained by considering one pattern at a time, and then summing the results over all patterns. From (5.48), the diagonal elements of the Hessian, for pattern n, can be written

$$\frac{\partial^2 E_n}{\partial w_{ji}^2} = \frac{\partial^2 E_n}{\partial a_j^2} z_i^2. \tag{5.79}$$

Using (5.48) and (5.49), the second derivatives on the right-hand side of (5.79) can be found recursively using the chain rule of differential calculus to give a backpropagation equation of the form

$$\frac{\partial^2 E_n}{\partial a_j^2} = h'(a_j)^2 \sum_k \sum_{k'} w_{kj} w_{k'j} \frac{\partial^2 E_n}{\partial a_k \partial a_{k'}} + h''(a_j) \sum_k w_{kj} \frac{\partial E^n}{\partial a_k}. \tag{5.80}$$

If we now neglect off-diagonal elements in the second-derivative terms, we obtain (Becker and Le Cun, 1989; Le Cun *et al.*, 1990)

$$\frac{\partial^2 E_n}{\partial a_j^2} = h'(a_j)^2 \sum_k w_{kj}^2 \frac{\partial^2 E_n}{\partial a_k^2} + h''(a_j) \sum_k w_{kj} \frac{\partial E_n}{\partial a_k}. \tag{5.81}$$

Note that the number of computational steps required to evaluate this approximation is $O(W)$, where W is the total number of weight and bias parameters in the network, compared with $O(W^2)$ for the full Hessian.

Ricotti *et al.* (1988) also used the diagonal approximation to the Hessian, but they retained all terms in the evaluation of $\partial^2 E_n / \partial a_j^2$ and so obtained exact expressions for the diagonal terms. Note that this no longer has $O(W)$ scaling. The major problem with diagonal approximations, however, is that in practice the Hessian is typically found to be strongly nondiagonal, and so these approximations, which are driven mainly be computational convenience, must be treated with care.

5.4.2 Outer product approximation

When neural networks are applied to regression problems, it is common to use a sum-of-squares error function of the form

$$E = \frac{1}{2}\sum_{n=1}^{N}(y_n - t_n)^2 \tag{5.82}$$

where we have considered the case of a single output in order to keep the notation simple (the extension to several outputs is straightforward). We can then write the Hessian matrix in the form

Exercise 5.16

$$\mathbf{H} = \nabla\nabla E = \sum_{n=1}^{N}\nabla y_n \nabla y_n + \sum_{n=1}^{N}(y_n - t_n)\nabla\nabla y_n. \tag{5.83}$$

If the network has been trained on the data set, and its outputs y_n happen to be very close to the target values t_n, then the second term in (5.83) will be small and can be neglected. More generally, however, it may be appropriate to neglect this term by the following argument. Recall from Section 1.5.5 that the optimal function that minimizes a sum-of-squares loss is the conditional average of the target data. The quantity $(y_n - t_n)$ is then a random variable with zero mean. If we assume that its value is uncorrelated with the value of the second derivative term on the right-hand side of (5.83), then the whole term will average to zero in the summation over n.

Exercise 5.17

By neglecting the second term in (5.83), we arrive at the *Levenberg–Marquardt* approximation or *outer product* approximation (because the Hessian matrix is built up from a sum of outer products of vectors), given by

$$\mathbf{H} \simeq \sum_{n=1}^{N}\mathbf{b}_n\mathbf{b}_n^{\mathrm{T}} \tag{5.84}$$

where $\mathbf{b}_n = \nabla y_n = \nabla a_n$ because the activation function for the output units is simply the identity. Evaluation of the outer product approximation for the Hessian is straightforward as it only involves first derivatives of the error function, which can be evaluated efficiently in $O(W)$ steps using standard backpropagation. The elements of the matrix can then be found in $O(W^2)$ steps by simple multiplication. It is important to emphasize that this approximation is only likely to be valid for a network that has been trained appropriately, and that for a general network mapping the second derivative terms on the right-hand side of (5.83) will typically not be negligible.

In the case of the cross-entropy error function for a network with logistic sigmoid output-unit activation functions, the corresponding approximation is given by

Exercise 5.19

$$\mathbf{H} \simeq \sum_{n=1}^{N} y_n(1 - y_n)\mathbf{b}_n\mathbf{b}_n^{\mathrm{T}}. \tag{5.85}$$

An analogous result can be obtained for multiclass networks having softmax output-unit activation functions.

Exercise 5.20

5.4.3 Inverse Hessian

We can use the outer-product approximation to develop a computationally efficient procedure for approximating the inverse of the Hessian (Hassibi and Stork, 1993). First we write the outer-product approximation in matrix notation as

$$\mathbf{H}_N = \sum_{n=1}^{N} \mathbf{b}_n \mathbf{b}_n^{\mathrm{T}} \tag{5.86}$$

where $\mathbf{b}_n \equiv \nabla_{\mathbf{w}} a_n$ is the contribution to the gradient of the output unit activation arising from data point n. We now derive a sequential procedure for building up the Hessian by including data points one at a time. Suppose we have already obtained the inverse Hessian using the first L data points. By separating off the contribution from data point $L+1$, we obtain

$$\mathbf{H}_{L+1} = \mathbf{H}_L + \mathbf{b}_{L+1}\mathbf{b}_{L+1}^{\mathrm{T}}. \tag{5.87}$$

In order to evaluate the inverse of the Hessian, we now consider the matrix identity

$$\left(\mathbf{M} + \mathbf{v}\mathbf{v}^{\mathrm{T}}\right)^{-1} = \mathbf{M}^{-1} - \frac{\left(\mathbf{M}^{-1}\mathbf{v}\right)\left(\mathbf{v}^{\mathrm{T}}\mathbf{M}^{-1}\right)}{1 + \mathbf{v}^{\mathrm{T}}\mathbf{M}^{-1}\mathbf{v}} \tag{5.88}$$

where $\mathbf{I}$ is the unit matrix, which is simply a special case of the Woodbury identity (C.7). If we now identify $\mathbf{H}_L$ with $\mathbf{M}$ and $\mathbf{b}_{L+1}$ with $\mathbf{v}$, we obtain

$$\mathbf{H}_{L+1}^{-1} = \mathbf{H}_L^{-1} - \frac{\mathbf{H}_L^{-1}\mathbf{b}_{L+1}\mathbf{b}_{L+1}^{\mathrm{T}}\mathbf{H}_L^{-1}}{1 + \mathbf{b}_{L+1}^{\mathrm{T}}\mathbf{H}_L^{-1}\mathbf{b}_{L+1}}. \tag{5.89}$$

In this way, data points are sequentially absorbed until $L+1 = N$ and the whole data set has been processed. This result therefore represents a procedure for evaluating the inverse of the Hessian using a single pass through the data set. The initial matrix $\mathbf{H}_0$ is chosen to be $\alpha\mathbf{I}$, where α is a small quantity, so that the algorithm actually finds the inverse of $\mathbf{H} + \alpha\mathbf{I}$. The results are not particularly sensitive to the precise value of α. Extension of this algorithm to networks having more than one output is straightforward.

Exercise 5.21

We note here that the Hessian matrix can sometimes be calculated indirectly as part of the network training algorithm. In particular, quasi-Newton nonlinear optimization algorithms gradually build up an approximation to the inverse of the Hessian during training. Such algorithms are discussed in detail in Bishop and Nabney (2008).

5.4.4 Finite differences

As in the case of the first derivatives of the error function, we can find the second derivatives by using finite differences, with accuracy limited by numerical precision. If we perturb each possible pair of weights in turn, we obtain

$$\begin{aligned}\frac{\partial^2 E}{\partial w_{ji}\partial w_{lk}} = \frac{1}{4\epsilon^2}\{&E(w_{ji}+\epsilon, w_{lk}+\epsilon) - E(w_{ji}+\epsilon, w_{lk}-\epsilon)\\ &-E(w_{ji}-\epsilon, w_{lk}+\epsilon) + E(w_{ji}-\epsilon, w_{lk}-\epsilon)\} + O(\epsilon^2).\end{aligned} \tag{5.90}$$

Again, by using a symmetrical central differences formulation, we ensure that the residual errors are $O(\epsilon^2)$ rather than $O(\epsilon)$. Because there are W^2 elements in the Hessian matrix, and because the evaluation of each element requires four forward propagations each needing $O(W)$ operations (per pattern), we see that this approach will require $O(W^3)$ operations to evaluate the complete Hessian. It therefore has poor scaling properties, although in practice it is very useful as a check on the software implementation of backpropagation methods.

A more efficient version of numerical differentiation can be found by applying central differences to the first derivatives of the error function, which are themselves calculated using backpropagation. This gives

$$\frac{\partial^2 E}{\partial w_{ji} \partial w_{lk}} = \frac{1}{2\epsilon} \left\{ \frac{\partial E}{\partial w_{ji}}(w_{lk} + \epsilon) - \frac{\partial E}{\partial w_{ji}}(w_{lk} - \epsilon) \right\} + O(\epsilon^2). \tag{5.91}$$

Because there are now only W weights to be perturbed, and because the gradients can be evaluated in $O(W)$ steps, we see that this method gives the Hessian in $O(W^2)$ operations.

5.4.5 Exact evaluation of the Hessian

So far, we have considered various approximation schemes for evaluating the Hessian matrix or its inverse. The Hessian can also be evaluated exactly, for a network of arbitrary feed-forward topology, using extension of the technique of backpropagation used to evaluate first derivatives, which shares many of its desirable features including computational efficiency (Bishop, 1991; Bishop, 1992). It can be applied to any differentiable error function that can be expressed as a function of the network outputs and to networks having arbitrary differentiable activation functions. The number of computational steps needed to evaluate the Hessian scales like $O(W^2)$. Similar algorithms have also been considered by Buntine and Weigend (1993).

Exercise 5.22

Here we consider the specific case of a network having two layers of weights, for which the required equations are easily derived. We shall use indices i and i' to denote inputs, indices j and j' to denoted hidden units, and indices k and k' to denote outputs. We first define

$$\delta_k = \frac{\partial E_n}{\partial a_k}, \qquad M_{kk'} \equiv \frac{\partial^2 E_n}{\partial a_k \partial a_{k'}} \tag{5.92}$$

where E_n is the contribution to the error from data point n. The Hessian matrix for this network can then be considered in three separate blocks as follows.

1. Both weights in the second layer:

$$\frac{\partial^2 E_n}{\partial w_{kj}^{(2)} \partial w_{k'j'}^{(2)}} = z_j z_{j'} M_{kk'}. \tag{5.93}$$

2. Both weights in the first layer:

$$\frac{\partial^2 E_n}{\partial w_{ji}^{(1)} \partial w_{j'i'}^{(1)}} = x_i x_{i'} h''(a_{j'}) I_{jj'} \sum_k w_{kj'}^{(2)} \delta_k + x_i x_{i'} h'(a_{j'}) h'(a_j) \sum_k \sum_{k'} w_{k'j'}^{(2)} w_{kj}^{(2)} M_{kk'}. \tag{5.94}$$

3. One weight in each layer:

$$\frac{\partial^2 E_n}{\partial w_{ji}^{(1)} \partial w_{kj'}^{(2)}} = x_i h'(a_{j'}) \left\{ \delta_k I_{jj'} + z_j \sum_{k'} w_{k'j'}^{(2)} H_{kk'} \right\}. \tag{5.95}$$

Here $I_{jj'}$ is the j, j' element of the identity matrix. If one or both of the weights is a bias term, then the corresponding expressions are obtained simply by setting the appropriate activation(s) to 1. Inclusion of skip-layer connections is straightforward.

Exercise 5.23

5.4.6 Fast multiplication by the Hessian

For many applications of the Hessian, the quantity of interest is not the Hessian matrix $\mathbf{H}$ itself but the product of $\mathbf{H}$ with some vector $\mathbf{v}$. We have seen that the evaluation of the Hessian takes $O(W^2)$ operations, and it also requires storage that is $O(W^2)$. The vector $\mathbf{v}^{\mathrm{T}}\mathbf{H}$ that we wish to calculate, however, has only W elements, so instead of computing the Hessian as an intermediate step, we can instead try to find an efficient approach to evaluating $\mathbf{v}^{\mathrm{T}}\mathbf{H}$ directly in a way that requires only $O(W)$ operations.

To do this, we first note that

$$\mathbf{v}^{\mathrm{T}}\mathbf{H} = \mathbf{v}^{\mathrm{T}} \nabla (\nabla E) \tag{5.96}$$

where ∇ denotes the gradient operator in weight space. We can then write down the standard forward-propagation and backpropagation equations for the evaluation of ∇E and apply (5.96) to these equations to give a set of forward-propagation and backpropagation equations for the evaluation of $\mathbf{v}^{\mathrm{T}}\mathbf{H}$ (Møller, 1993; Pearlmutter, 1994). This corresponds to acting on the original forward-propagation and back-propagation equations with a differential operator $\mathbf{v}^{\mathrm{T}}\nabla$. Pearlmutter (1994) used the notation $\mathcal{R}\{\cdot\}$ to denote the operator $\mathbf{v}^{\mathrm{T}}\nabla$, and we shall follow this convention. The analysis is straightforward and makes use of the usual rules of differential calculus, together with the result

$$\mathcal{R}\{\mathbf{w}\} = \mathbf{v}. \tag{5.97}$$

The technique is best illustrated with a simple example, and again we choose a two-layer network of the form shown in Figure 5.1, with linear output units and a sum-of-squares error function. As before, we consider the contribution to the error function from one pattern in the data set. The required vector is then obtained as

usual by summing over the contributions from each of the patterns separately. For the two-layer network, the forward-propagation equations are given by

$$a_j = \sum_i w_{ji} x_i \tag{5.98}$$

$$z_j = h(a_j) \tag{5.99}$$

$$y_k = \sum_j w_{kj} z_j. \tag{5.100}$$

We now act on these equations using the $\mathcal{R}\{\cdot\}$ operator to obtain a set of forward propagation equations in the form

$$\mathcal{R}\{a_j\} = \sum_i v_{ji} x_i \tag{5.101}$$

$$\mathcal{R}\{z_j\} = h'(a_j)\mathcal{R}\{a_j\} \tag{5.102}$$

$$\mathcal{R}\{y_k\} = \sum_j w_{kj}\mathcal{R}\{z_j\} + \sum_j v_{kj} z_j \tag{5.103}$$

where v_{ji} is the element of the vector $\mathbf{v}$ that corresponds to the weight w_{ji}. Quantities of the form $\mathcal{R}\{z_j\}$, $\mathcal{R}\{a_j\}$ and $\mathcal{R}\{y_k\}$ are to be regarded as new variables whose values are found using the above equations.

Because we are considering a sum-of-squares error function, we have the following standard backpropagation expressions:

$$\delta_k = y_k - t_k \tag{5.104}$$

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k. \tag{5.105}$$

Again, we act on these equations with the $\mathcal{R}\{\cdot\}$ operator to obtain a set of backpropagation equations in the form

$$\mathcal{R}\{\delta_k\} = \mathcal{R}\{y_k\} \tag{5.106}$$

$$\begin{aligned}\mathcal{R}\{\delta_j\} &= h''(a_j)\mathcal{R}\{a_j\} \sum_k w_{kj}\delta_k \\ &\quad + h'(a_j) \sum_k v_{kj}\delta_k + h'(a_j) \sum_k w_{kj}\mathcal{R}\{\delta_k\}.\end{aligned} \tag{5.107}$$

Finally, we have the usual equations for the first derivatives of the error

$$\frac{\partial E}{\partial w_{kj}} = \delta_k z_j \tag{5.108}$$

$$\frac{\partial E}{\partial w_{ji}} = \delta_j x_i \tag{5.109}$$

and acting on these with the $\mathcal{R}\{\cdot\}$ operator, we obtain expressions for the elements of the vector $\mathbf{v}^{\mathrm{T}}\mathbf{H}$

$$\mathcal{R}\left\{\frac{\partial E}{\partial w_{kj}}\right\} = \mathcal{R}\{\delta_k\}z_j + \delta_k\mathcal{R}\{z_j\} \tag{5.110}$$

$$\mathcal{R}\left\{\frac{\partial E}{\partial w_{ji}}\right\} = x_i\mathcal{R}\{\delta_j\}. \tag{5.111}$$

The implementation of this algorithm involves the introduction of additional variables $\mathcal{R}\{a_j\}$, $\mathcal{R}\{z_j\}$ and $\mathcal{R}\{\delta_j\}$ for the hidden units and $\mathcal{R}\{\delta_k\}$ and $\mathcal{R}\{y_k\}$ for the output units. For each input pattern, the values of these quantities can be found using the above results, and the elements of $\mathbf{v}^{\mathrm{T}}\mathbf{H}$ are then given by (5.110) and (5.111). An elegant aspect of this technique is that the equations for evaluating $\mathbf{v}^{\mathrm{T}}\mathbf{H}$ mirror closely those for standard forward and backward propagation, and so the extension of existing software to compute this product is typically straightforward.

If desired, the technique can be used to evaluate the full Hessian matrix by choosing the vector $\mathbf{v}$ to be given successively by a series of unit vectors of the form $(0, 0, \ldots, 1, \ldots, 0)$ each of which picks out one column of the Hessian. This leads to a formalism that is analytically equivalent to the backpropagation procedure of Bishop (1992), as described in Section 5.4.5, though with some loss of efficiency due to redundant calculations.

5.5. Regularization in Neural Networks

The number of input and outputs units in a neural network is generally determined by the dimensionality of the data set, whereas the number M of hidden units is a free parameter that can be adjusted to give the best predictive performance. Note that M controls the number of parameters (weights and biases) in the network, and so we might expect that in a maximum likelihood setting there will be an optimum value of M that gives the best generalization performance, corresponding to the optimum balance between under-fitting and over-fitting. Figure 5.9 shows an example of the effect of different values of M for the sinusoidal regression problem.

The generalization error, however, is not a simple function of M due to the presence of local minima in the error function, as illustrated in Figure 5.10. Here we see the effect of choosing multiple random initializations for the weight vector for a range of values of M. The overall best validation set performance in this case occurred for a particular solution having $M = 8$. In practice, one approach to choosing M is in fact to plot a graph of the kind shown in Figure 5.10 and then to choose the specific solution having the smallest validation set error.

There are, however, other ways to control the complexity of a neural network model in order to avoid over-fitting. From our discussion of polynomial curve fitting in Chapter 1, we see that an alternative approach is to choose a relatively large value for M and then to control complexity by the addition of a regularization term to the error function. The simplest regularizer is the quadratic, giving a regularized error

Figure 5.9 Examples of two-layer networks trained on 10 data points drawn from the sinusoidal data set. The graphs show the result of fitting networks having $M = 1$, 3 and 10 hidden units, respectively, by minimizing a sum-of-squares error function using a scaled conjugate-gradient algorithm.

of the form

$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}. \tag{5.112}$$

This regularizer is also known as *weight decay* and has been discussed at length in Chapter 3. The effective model complexity is then determined by the choice of the regularization coefficient λ. As we have seen previously, this regularizer can be interpreted as the negative logarithm of a zero-mean Gaussian prior distribution over the weight vector $\mathbf{w}$.

5.5.1 Consistent Gaussian priors

One of the limitations of simple weight decay in the form (5.112) is that is inconsistent with certain scaling properties of network mappings. To illustrate this, consider a multilayer perceptron network having two layers of weights and linear output units, which performs a mapping from a set of input variables $\{x_i\}$ to a set of output variables $\{y_k\}$. The activations of the hidden units in the first hidden layer

Figure 5.10 Plot of the sum-of-squares test-set error for the polynomial data set versus the number of hidden units in the network, with 30 random starts for each network size, showing the effect of local minima. For each new start, the weight vector was initialized by sampling from an isotropic Gaussian distribution having a mean of zero and a variance of 10.

take the form

$$z_j = h\left(\sum_i w_{ji}x_i + w_{j0}\right) \tag{5.113}$$

while the activations of the output units are given by

$$y_k = \sum_j w_{kj}z_j + w_{k0}. \tag{5.114}$$

Suppose we perform a linear transformation of the input data of the form

$$x_i \to \widetilde{x}_i = ax_i + b. \tag{5.115}$$

Then we can arrange for the mapping performed by the network to be unchanged by making a corresponding linear transformation of the weights and biases from the inputs to the units in the hidden layer of the form

Exercise 5.24

$$w_{ji} \to \widetilde{w}_{ji} = \frac{1}{a}w_{ji} \tag{5.116}$$

$$w_{j0} \to \widetilde{w}_{j0} = w_{j0} - \frac{b}{a}\sum_i w_{ji}. \tag{5.117}$$

Similarly, a linear transformation of the output variables of the network of the form

$$y_k \to \widetilde{y}_k = cy_k + d \tag{5.118}$$

can be achieved by making a transformation of the second-layer weights and biases using

$$w_{kj} \to \widetilde{w}_{kj} = cw_{kj} \tag{5.119}$$

$$w_{k0} \to \widetilde{w}_{k0} = cw_{k0} + d. \tag{5.120}$$

If we train one network using the original data and one network using data for which the input and/or target variables are transformed by one of the above linear transformations, then consistency requires that we should obtain equivalent networks that differ only by the linear transformation of the weights as given. Any regularizer should be consistent with this property, otherwise it arbitrarily favours one solution over another, equivalent one. Clearly, simple weight decay (5.112), that treats all weights and biases on an equal footing, does not satisfy this property.

We therefore look for a regularizer which is invariant under the linear transformations (5.116), (5.117), (5.119) and (5.120). These require that the regularizer should be invariant to re-scaling of the weights and to shifts of the biases. Such a regularizer is given by

$$\frac{\lambda_1}{2}\sum_{w\in\mathcal{W}_1} w^2 + \frac{\lambda_2}{2}\sum_{w\in\mathcal{W}_2} w^2 \tag{5.121}$$

where $\mathcal{W}_1$ denotes the set of weights in the first layer, $\mathcal{W}_2$ denotes the set of weights in the second layer, and biases are excluded from the summations. This regularizer

will remain unchanged under the weight transformations provided the regularization parameters are re-scaled using $\lambda_1 \to a^{1/2}\lambda_1$ and $\lambda_2 \to c^{-1/2}\lambda_2$.

The regularizer (5.121) corresponds to a prior of the form

$$p(\mathbf{w}|\alpha_1, \alpha_2) \propto \exp\left(-\frac{\alpha_1}{2}\sum_{w\in\mathcal{W}_1} w^2 - \frac{\alpha_2}{2}\sum_{w\in\mathcal{W}_2} w^2\right). \tag{5.122}$$

Note that priors of this form are *improper* (they cannot be normalized) because the bias parameters are unconstrained. The use of improper priors can lead to difficulties in selecting regularization coefficients and in model comparison within the Bayesian framework, because the corresponding evidence is zero. It is therefore common to include separate priors for the biases (which then break shift invariance) having their own hyperparameters. We can illustrate the effect of the resulting four hyperparameters by drawing samples from the prior and plotting the corresponding network functions, as shown in Figure 5.11.

More generally, we can consider priors in which the weights are divided into any number of groups $\mathcal{W}_k$ so that

$$p(\mathbf{w}) \propto \exp\left(-\frac{1}{2}\sum_k \alpha_k \|\mathbf{w}\|_k^2\right) \tag{5.123}$$

where

$$\|\mathbf{w}\|_k^2 = \sum_{j\in\mathcal{W}_k} w_j^2. \tag{5.124}$$

As a special case of this prior, if we choose the groups to correspond to the sets of weights associated with each of the input units, and we optimize the marginal likelihood with respect to the corresponding parameters α_k, we obtain *automatic relevance determination* as discussed in Section 7.2.2.

5.5.2 Early stopping

An alternative to regularization as a way of controlling the effective complexity of a network is the procedure of *early stopping*. The training of nonlinear network models corresponds to an iterative reduction of the error function defined with respect to a set of training data. For many of the optimization algorithms used for network training, such as conjugate gradients, the error is a nonincreasing function of the iteration index. However, the error measured with respect to independent data, generally called a validation set, often shows a decrease at first, followed by an increase as the network starts to over-fit. Training can therefore be stopped at the point of smallest error with respect to the validation data set, as indicated in Figure 5.12, in order to obtain a network having good generalization performance.

The behaviour of the network in this case is sometimes explained qualitatively in terms of the effective number of degrees of freedom in the network, in which this number starts out small and then to grows during the training process, corresponding to a steady increase in the effective complexity of the model. Halting training before

Figure 5.11 Illustration of the effect of the hyperparameters governing the prior distribution over weights and biases in a two-layer network having a single input, a single linear output, and 12 hidden units having '$\tanh$' activation functions. The priors are governed by four hyperparameters α_1^{b}, α_1^{w}, α_2^{b}, and α_2^{w}, which represent the precisions of the Gaussian distributions of the first-layer biases, first-layer weights, second-layer biases, and second-layer weights, respectively. We see that the parameter α_2^{w} governs the vertical scale of functions (note the different vertical axis ranges on the top two diagrams), α_1^{w} governs the horizontal scale of variations in the function values, and α_1^{b} governs the horizontal range over which variations occur. The parameter α_2^{b}, whose effect is not illustrated here, governs the range of vertical offsets of the functions.

a minimum of the training error has been reached then represents a way of limiting the effective network complexity.

In the case of a quadratic error function, we can verify this insight, and show that early stopping should exhibit similar behaviour to regularization using a simple weight-decay term. This can be understood from Figure 5.13, in which the axes in weight space have been rotated to be parallel to the eigenvectors of the Hessian matrix. If, in the absence of weight decay, the weight vector starts at the origin and proceeds during training along a path that follows the local negative gradient vector, then the weight vector will move initially parallel to the w_2 axis through a point corresponding roughly to $\widetilde{\mathbf{w}}$ and then move towards the minimum of the error function $\mathbf{w}_{\mathrm{ML}}$. This follows from the shape of the error surface and the widely differing eigenvalues of the Hessian. Stopping at a point near $\widetilde{\mathbf{w}}$ is therefore similar to weight decay. The relationship between early stopping and weight decay can be made quantitative, thereby showing that the quantity $\tau\eta$ (where τ is the iteration index, and η is the learning rate parameter) plays the role of the reciprocal of the regularization

Exercise 5.25

Figure 5.12 An illustration of the behaviour of training set error (left) and validation set error (right) during a typical training session, as a function of the iteration step, for the sinusoidal data set. The goal of achieving the best generalization performance suggests that training should be stopped at the point shown by the vertical dashed lines, corresponding to the minimum of the validation set error.

parameter λ. The effective number of parameters in the network therefore grows during the course of training.

5.5.3 Invariances

In many applications of pattern recognition, it is known that predictions should be unchanged, or *invariant*, under one or more transformations of the input variables. For example, in the classification of objects in two-dimensional images, such as handwritten digits, a particular object should be assigned the same classification irrespective of its position within the image (*translation invariance*) or of its size (*scale invariance*). Such transformations produce significant changes in the raw data, expressed in terms of the intensities at each of the pixels in the image, and yet should give rise to the same output from the classification system. Similarly in speech recognition, small levels of nonlinear warping along the time axis, which preserve temporal ordering, should not change the interpretation of the signal.

If sufficiently large numbers of training patterns are available, then an adaptive model such as a neural network can learn the invariance, at least approximately. This involves including within the training set a sufficiently large number of examples of the effects of the various transformations. Thus, for translation invariance in an image, the training set should include examples of objects at many different positions.

This approach may be impractical, however, if the number of training examples is limited, or if there are several invariants (because the number of combinations of transformations grows exponentially with the number of such transformations). We therefore seek alternative approaches for encouraging an adaptive model to exhibit the required invariances. These can broadly be divided into four categories:

1. The training set is augmented using replicas of the training patterns, transformed according to the desired invariances. For instance, in our digit recognition example, we could make multiple copies of each example in which the

Figure 5.13 A schematic illustration of why early stopping can give similar results to weight decay in the case of a quadratic error function. The ellipse shows a contour of constant error, and $\mathbf{w}_{\mathrm{ML}}$ denotes the minimum of the error function. If the weight vector starts at the origin and moves according to the local negative gradient direction, then it will follow the path shown by the curve. By stopping training early, a weight vector $\widetilde{\mathbf{w}}$ is found that is qualitatively similar to that obtained with a simple weight-decay regularizer and training to the minimum of the regularized error, as can be seen by comparing with Figure 3.15.

digit is shifted to a different position in each image.

2. A regularization term is added to the error function that penalizes changes in the model output when the input is transformed. This leads to the technique of *tangent propagation*, discussed in Section 5.5.4.

3. Invariance is built into the pre-processing by extracting features that are invariant under the required transformations. Any subsequent regression or classification system that uses such features as inputs will necessarily also respect these invariances.

4. The final option is to build the invariance properties into the structure of a neural network (or into the definition of a kernel function in the case of techniques such as the relevance vector machine). One way to achieve this is through the use of local receptive fields and shared weights, as discussed in the context of convolutional neural networks in Section 5.5.6.

Approach 1 is often relatively easy to implement and can be used to encourage complex invariances such as those illustrated in Figure 5.14. For sequential training algorithms, this can be done by transforming each input pattern before it is presented to the model so that, if the patterns are being recycled, a different transformation (drawn from an appropriate distribution) is added each time. For batch methods, a similar effect can be achieved by replicating each data point a number of times and transforming each copy independently. The use of such augmented data can lead to significant improvements in generalization (Simard *et al.*, 2003), although it can also be computationally costly.

Approach 2 leaves the data set unchanged but modifies the error function through the addition of a regularizer. In Section 5.5.5, we shall show that this approach is closely related to approach 2.

Figure 5.14 Illustration of the synthetic warping of a handwritten digit. The original image is shown on the left. On the right, the top row shows three examples of warped digits, with the corresponding displacement fields shown on the bottom row. These displacement fields are generated by sampling random displacements $\Delta x, \Delta y \in (0, 1)$ at each pixel and then smoothing by convolution with Gaussians of width 0.01, 30 and 60 respectively.

One advantage of approach 3 is that it can correctly extrapolate well beyond the range of transformations included in the training set. However, it can be difficult to find hand-crafted features with the required invariances that do not also discard information that can be useful for discrimination.

5.5.4 Tangent propagation

We can use regularization to encourage models to be invariant to transformations of the input through the technique of *tangent propagation* (Simard *et al.*, 1992). Consider the effect of a transformation on a particular input vector $\mathbf{x}_n$. Provided the transformation is continuous (such as translation or rotation, but not mirror reflection for instance), then the transformed pattern will sweep out a manifold $\mathcal{M}$ within the D-dimensional input space. This is illustrated in Figure 5.15, for the case of $D = 2$ for simplicity. Suppose the transformation is governed by a single parameter ξ (which might be rotation angle for instance). Then the subspace $\mathcal{M}$ swept out by $\mathbf{x}_n$

Figure 5.15 Illustration of a two-dimensional input space showing the effect of a continuous transformation on a particular input vector $\mathbf{x}_n$. A one-dimensional transformation, parameterized by the continuous variable ξ, applied to $\mathbf{x}_n$ causes it to sweep out a one-dimensional manifold $\mathcal{M}$. Locally, the effect of the transformation can be approximated by the tangent vector $\boldsymbol{\tau}_n$.

will be one-dimensional, and will be parameterized by ξ. Let the vector that results from acting on $\mathbf{x}_n$ by this transformation be denoted by $\mathbf{s}(\mathbf{x}_n, \xi)$, which is defined so that $\mathbf{s}(\mathbf{x}, 0) = \mathbf{x}$. Then the tangent to the curve $\mathcal{M}$ is given by the directional derivative $\boldsymbol{\tau} = \partial \mathbf{s}/\partial \xi$, and the tangent vector at the point $\mathbf{x}_n$ is given by

$$\boldsymbol{\tau}_n = \left.\frac{\partial \mathbf{s}(\mathbf{x}_n, \xi)}{\partial \xi}\right|_{\xi=0}. \tag{5.125}$$

Under a transformation of the input vector, the network output vector will, in general, change. The derivative of output k with respect to ξ is given by

$$\left.\frac{\partial y_k}{\partial \xi}\right|_{\xi=0} = \sum_{i=1}^{D} \left.\frac{\partial y_k}{\partial x_i}\frac{\partial x_i}{\partial \xi}\right|_{\xi=0} = \sum_{i=1}^{D} J_{ki}\tau_i \tag{5.126}$$

where J_{ki} is the (k, i) element of the Jacobian matrix $\mathbf{J}$, as discussed in Section 5.3.4. The result (5.126) can be used to modify the standard error function, so as to encourage local invariance in the neighbourhood of the data points, by the addition to the original error function E of a regularization function Ω to give a total error function of the form

$$\widetilde{E} = E + \lambda\Omega \tag{5.127}$$

where λ is a regularization coefficient and

$$\Omega = \frac{1}{2}\sum_n \sum_k \left(\left.\frac{\partial y_{nk}}{\partial \xi}\right|_{\xi=0}\right)^2 = \frac{1}{2}\sum_n \sum_k \left(\sum_{i=1}^{D} J_{nki}\tau_{ni}\right)^2. \tag{5.128}$$

The regularization function will be zero when the network mapping function is invariant under the transformation in the neighbourhood of each pattern vector, and the value of the parameter λ determines the balance between fitting the training data and learning the invariance property.

In a practical implementation, the tangent vector $\boldsymbol{\tau}_n$ can be approximated using finite differences, by subtracting the original vector $\mathbf{x}_n$ from the corresponding vector after transformation using a small value of ξ, and then dividing by ξ. This is illustrated in Figure 5.16.

The regularization function depends on the network weights through the Jacobian $\mathbf{J}$. A backpropagation formalism for computing the derivatives of the regularizer with respect to the network weights is easily obtained by extension of the techniques introduced in Section 5.3.

Exercise 5.26

If the transformation is governed by L parameters (e.g., $L = 3$ for the case of translations combined with in-plane rotations in a two-dimensional image), then the manifold $\mathcal{M}$ will have dimensionality L, and the corresponding regularizer is given by the sum of terms of the form (5.128), one for each transformation. If several transformations are considered at the same time, and the network mapping is made invariant to each separately, then it will be (locally) invariant to combinations of the transformations (Simard *et al.*, 1992).

Figure 5.16 Illustration showing (a) the original image $\mathbf{x}$ of a handwritten digit, (b) the tangent vector $\boldsymbol{\tau}$ corresponding to an infinitesimal clockwise rotation, (c) the result of adding a small contribution from the tangent vector to the original image giving $\mathbf{x} + \epsilon\boldsymbol{\tau}$ with $\epsilon = 15$ degrees, and (d) the true image rotated for comparison.

A related technique, called *tangent distance*, can be used to build invariance properties into distance-based methods such as nearest-neighbour classifiers (Simard *et al.*, 1993).

5.5.5 Training with transformed data

We have seen that one way to encourage invariance of a model to a set of transformations is to expand the training set using transformed versions of the original input patterns. Here we show that this approach is closely related to the technique of tangent propagation (Bishop, 1995b; Leen, 1995).

As in Section 5.5.4, we shall consider a transformation governed by a single parameter ξ and described by the function $\mathbf{s}(\mathbf{x}, \xi)$, with $\mathbf{s}(\mathbf{x}, 0) = \mathbf{x}$. We shall also consider a sum-of-squares error function. The error function for untransformed inputs can be written (in the infinite data set limit) in the form

$$E = \frac{1}{2} \iint \{y(\mathbf{x}) - t\}^2 p(t|\mathbf{x}) p(\mathbf{x}) \, \mathrm{d}\mathbf{x} \, \mathrm{d}t \tag{5.129}$$

as discussed in Section 1.5.5. Here we have considered a network having a single output, in order to keep the notation uncluttered. If we now consider an infinite number of copies of each data point, each of which is perturbed by the transformation

in which the parameter ξ is drawn from a distribution $p(\xi)$, then the error function defined over this expanded data set can be written as

$$\widetilde{E} = \frac{1}{2}\iiint \{y(\mathbf{s}(\mathbf{x},\xi)) - t\}^2 p(t|\mathbf{x})p(\mathbf{x})p(\xi)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t\,\mathrm{d}\xi. \tag{5.130}$$

We now assume that the distribution $p(\xi)$ has zero mean with small variance, so that we are only considering small transformations of the original input vectors. We can then expand the transformation function as a Taylor series in powers of ξ to give

$$\begin{aligned}\mathbf{s}(\mathbf{x},\xi) &= \mathbf{s}(\mathbf{x},0) + \xi\,\frac{\partial}{\partial\xi}\mathbf{s}(\mathbf{x},\xi)\bigg|_{\xi=0} + \frac{\xi^2}{2}\,\frac{\partial^2}{\partial\xi^2}\mathbf{s}(\mathbf{x},\xi)\bigg|_{\xi=0} + O(\xi^3)\\ &= \mathbf{x} + \xi\boldsymbol{\tau} + \frac{1}{2}\xi^2\boldsymbol{\tau}' + O(\xi^3)\end{aligned}$$

where $\boldsymbol{\tau}'$ denotes the second derivative of $\mathbf{s}(\mathbf{x},\xi)$ with respect to ξ evaluated at $\xi = 0$. This allows us to expand the model function to give

$$y(\mathbf{s}(\mathbf{x},\xi)) = y(\mathbf{x}) + \xi\boldsymbol{\tau}^{\mathrm{T}}\nabla y(\mathbf{x}) + \frac{\xi^2}{2}\left[(\boldsymbol{\tau}')^{\mathrm{T}}\nabla y(\mathbf{x}) + \boldsymbol{\tau}^{\mathrm{T}}\nabla\nabla y(\mathbf{x})\boldsymbol{\tau}\right] + O(\xi^3).$$

Substituting into the mean error function (5.130) and expanding, we then have

$$\begin{aligned}\widetilde{E} &= \frac{1}{2}\iint \{y(\mathbf{x}) - t\}^2 p(t|\mathbf{x})p(\mathbf{x})\,\mathrm{d}\mathbf{x}\,\mathrm{d}t\\ &+ \mathbb{E}[\xi]\iint \{y(\mathbf{x}) - t\}\boldsymbol{\tau}^{\mathrm{T}}\nabla y(\mathbf{x})p(t|\mathbf{x})p(\mathbf{x})\,\mathrm{d}\mathbf{x}\,\mathrm{d}t\\ &+ \mathbb{E}[\xi^2]\iint \Bigg[\{y(\mathbf{x}) - t\}\frac{1}{2}\left\{(\boldsymbol{\tau}')^{\mathrm{T}}\nabla y(\mathbf{x}) + \boldsymbol{\tau}^{\mathrm{T}}\nabla\nabla y(\mathbf{x})\boldsymbol{\tau}\right\}\\ &\quad + \left(\boldsymbol{\tau}^{\mathrm{T}}\nabla y(\mathbf{x})\right)^2\Bigg] p(t|\mathbf{x})p(\mathbf{x})\,\mathrm{d}\mathbf{x}\,\mathrm{d}t + O(\xi^3).\end{aligned}$$

Because the distribution of transformations has zero mean we have $\mathbb{E}[\xi] = 0$. Also, we shall denote $\mathbb{E}[\xi^2]$ by λ. Omitting terms of $O(\xi^3)$, the average error function then becomes

$$\widetilde{E} = E + \lambda\Omega \tag{5.131}$$

where E is the original sum-of-squares error, and the regularization term Ω takes the form

$$\begin{aligned}\Omega &= \int \Bigg[\{y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}]\}\frac{1}{2}\left\{(\boldsymbol{\tau}')^{\mathrm{T}}\nabla y(\mathbf{x}) + \boldsymbol{\tau}^{\mathrm{T}}\nabla\nabla y(\mathbf{x})\boldsymbol{\tau}\right\}\\ &\quad + \left(\boldsymbol{\tau}^{T}\nabla y(\mathbf{x})\right)^2\Bigg] p(\mathbf{x})\,\mathrm{d}\mathbf{x}\end{aligned} \tag{5.132}$$

in which we have performed the integration over t.

We can further simplify this regularization term as follows. In Section 1.5.5 we saw that the function that minimizes the sum-of-squares error is given by the conditional average $\mathbb{E}[t|\mathbf{x}]$ of the target values t. From (5.131) we see that the regularized error will equal the unregularized sum-of-squares plus terms which are $O(\xi)$, and so the network function that minimizes the total error will have the form

$$y(\mathbf{x}) = \mathbb{E}[t|\mathbf{x}] + O(\xi). \tag{5.133}$$

Thus, to leading order in ξ, the first term in the regularizer vanishes and we are left with

$$\Omega = \frac{1}{2} \int \left(\boldsymbol{\tau}^T \nabla y(\mathbf{x})\right)^2 p(\mathbf{x})\, \mathrm{d}\mathbf{x} \tag{5.134}$$

which is equivalent to the tangent propagation regularizer (5.128).

If we consider the special case in which the transformation of the inputs simply consists of the addition of random noise, so that $\mathbf{x} \to \mathbf{x} + \boldsymbol{\xi}$, then the regularizer takes the form

Exercise 5.27

$$\Omega = \frac{1}{2} \int \|\nabla y(\mathbf{x})\|^2 p(\mathbf{x})\, \mathrm{d}\mathbf{x} \tag{5.135}$$

which is known as *Tikhonov* regularization (Tikhonov and Arsenin, 1977; Bishop, 1995b). Derivatives of this regularizer with respect to the network weights can be found using an extended backpropagation algorithm (Bishop, 1993). We see that, for small noise amplitudes, Tikhonov regularization is related to the addition of random noise to the inputs, which has been shown to improve generalization in appropriate circumstances (Sietsma and Dow, 1991).

5.5.6 Convolutional networks

Another approach to creating models that are invariant to certain transformation of the inputs is to build the invariance properties into the structure of a neural network. This is the basis for the *convolutional neural network* (Le Cun *et al.*, 1989; LeCun *et al.*, 1998), which has been widely applied to image data.

Consider the specific task of recognizing handwritten digits. Each input image comprises a set of pixel intensity values, and the desired output is a posterior probability distribution over the ten digit classes. We know that the identity of the digit is invariant under translations and scaling as well as (small) rotations. Furthermore, the network must also exhibit invariance to more subtle transformations such as elastic deformations of the kind illustrated in Figure 5.14. One simple approach would be to treat the image as the input to a fully connected network, such as the kind shown in Figure 5.1. Given a sufficiently large training set, such a network could in principle yield a good solution to this problem and would learn the appropriate invariances by example.

However, this approach ignores a key property of images, which is that nearby pixels are more strongly correlated than more distant pixels. Many of the modern approaches to computer vision exploit this property by extracting *local* features that depend only on small subregions of the image. Information from such features can then be merged in later stages of processing in order to detect higher-order features

Figure 5.17 Diagram illustrating part of a convolutional neural network, showing a layer of convolutional units followed by a layer of subsampling units. Several successive pairs of such layers may be used.

and ultimately to yield information about the image as whole. Also, local features that are useful in one region of the image are likely to be useful in other regions of the image, for instance if the object of interest is translated.

These notions are incorporated into convolutional neural networks through three mechanisms: (i) local receptive fields, (ii) weight sharing, and (iii) subsampling. The structure of a convolutional network is illustrated in Figure 5.17. In the convolutional layer the units are organized into planes, each of which is called a *feature map*. Units in a feature map each take inputs only from a small subregion of the image, and all of the units in a feature map are constrained to share the same weight values. For instance, a feature map might consist of 100 units arranged in a 10×10 grid, with each unit taking inputs from a 5×5 pixel patch of the image. The whole feature map therefore has 25 adjustable weight parameters plus one adjustable bias parameter. Input values from a patch are linearly combined using the weights and the bias, and the result transformed by a sigmoidal nonlinearity using (5.1). If we think of the units as feature detectors, then all of the units in a feature map detect the same pattern but at different locations in the input image. Due to the weight sharing, the evaluation of the activations of these units is equivalent to a convolution of the image pixel intensities with a 'kernel' comprising the weight parameters. If the input image is shifted, the activations of the feature map will be shifted by the same amount but will otherwise be unchanged. This provides the basis for the (approximate) invariance of

the network outputs to translations and distortions of the input image. Because we will typically need to detect multiple features in order to build an effective model, there will generally be multiple feature maps in the convolutional layer, each having its own set of weight and bias parameters.

The outputs of the convolutional units form the inputs to the subsampling layer of the network. For each feature map in the convolutional layer, there is a plane of units in the subsampling layer and each unit takes inputs from a small receptive field in the corresponding feature map of the convolutional layer. These units perform subsampling. For instance, each subsampling unit might take inputs from a 2×2 unit region in the corresponding feature map and would compute the average of those inputs, multiplied by an adaptive weight with the addition of an adaptive bias parameter, and then transformed using a sigmoidal nonlinear activation function. The receptive fields are chosen to be contiguous and nonoverlapping so that there are half the number of rows and columns in the subsampling layer compared with the convolutional layer. In this way, the response of a unit in the subsampling layer will be relatively insensitive to small shifts of the image in the corresponding regions of the input space.

In a practical architecture, there may be several pairs of convolutional and subsampling layers. At each stage there is a larger degree of invariance to input transformations compared to the previous layer. There may be several feature maps in a given convolutional layer for each plane of units in the previous subsampling layer, so that the gradual reduction in spatial resolution is then compensated by an increasing number of features. The final layer of the network would typically be a fully connected, fully adaptive layer, with a softmax output nonlinearity in the case of multiclass classification.

The whole network can be trained by error minimization using backpropagation to evaluate the gradient of the error function. This involves a slight modification of the usual backpropagation algorithm to ensure that the shared-weight constraints are satisfied. Due to the use of local receptive fields, the number of weights in the network is smaller than if the network were fully connected. Furthermore, the number of independent parameters to be learned from the data is much smaller still, due to the substantial numbers of constraints on the weights.

Exercise 5.28

5.5.7 Soft weight sharing

One way to reduce the effective complexity of a network with a large number of weights is to constrain weights within certain groups to be equal. This is the technique of weight sharing that was discussed in Section 5.5.6 as a way of building translation invariance into networks used for image interpretation. It is only applicable, however, to particular problems in which the form of the constraints can be specified in advance. Here we consider a form of *soft weight sharing* (Nowlan and Hinton, 1992) in which the hard constraint of equal weights is replaced by a form of regularization in which groups of weights are encouraged to have similar values. Furthermore, the division of weights into groups, the mean weight value for each group, and the spread of values within the groups are all determined as part of the learning process.

Recall that the simple weight decay regularizer, given in (5.112), can be viewed as the negative log of a Gaussian prior distribution over the weights. We can encourage the weight values to form several groups, rather than just one group, by considering instead a probability distribution that is a *mixture* of Gaussians. The centres and variances of the Gaussian components, as well as the mixing coefficients, will be considered as adjustable parameters to be determined as part of the learning process. Thus, we have a probability density of the form

Section 2.3.9

$$p(\mathbf{w}) = \prod_i p(w_i) \tag{5.136}$$

where

$$p(w_i) = \sum_{j=1}^{M} \pi_j \mathcal{N}(w_i|\mu_j, \sigma_j^2) \tag{5.137}$$

and π_j are the mixing coefficients. Taking the negative logarithm then leads to a regularization function of the form

$$\Omega(\mathbf{w}) = -\sum_i \ln\left(\sum_{j=1}^{M} \pi_j \mathcal{N}(w_i|\mu_j, \sigma_j^2)\right). \tag{5.138}$$

The total error function is then given by

$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda\Omega(\mathbf{w}) \tag{5.139}$$

where λ is the regularization coefficient. This error is minimized both with respect to the weights w_i and with respect to the parameters $\{\pi_j, \mu_j, \sigma_j\}$ of the mixture model. If the weights were constant, then the parameters of the mixture model could be determined by using the EM algorithm discussed in Chapter 9. However, the distribution of weights is itself evolving during the learning process, and so to avoid numerical instability, a joint optimization is performed simultaneously over the weights and the mixture-model parameters. This can be done using a standard optimization algorithm such as conjugate gradients or quasi-Newton methods.

In order to minimize the total error function, it is necessary to be able to evaluate its derivatives with respect to the various adjustable parameters. To do this it is convenient to regard the $\{\pi_j\}$ as *prior* probabilities and to introduce the corresponding posterior probabilities which, following (2.192), are given by Bayes' theorem in the form

$$\gamma_j(w) = \frac{\pi_j \mathcal{N}(w|\mu_j, \sigma_j^2)}{\sum_k \pi_k \mathcal{N}(w|\mu_k, \sigma_k^2)}. \tag{5.140}$$

The derivatives of the total error function with respect to the weights are then given by

Exercise 5.29

$$\frac{\partial \widetilde{E}}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda \sum_j \gamma_j(w_i) \frac{(w_i - \mu_j)}{\sigma_j^2}. \tag{5.141}$$

The effect of the regularization term is therefore to pull each weight towards the centre of the jth Gaussian, with a force proportional to the posterior probability of that Gaussian for the given weight. This is precisely the kind of effect that we are seeking.

Exercise 5.30 Derivatives of the error with respect to the centres of the Gaussians are also easily computed to give

$$\frac{\partial \widetilde{E}}{\partial \mu_j} = \lambda \sum_i \gamma_j(w_i) \frac{(\mu_i - w_j)}{\sigma_j^2} \tag{5.142}$$

which has a simple intuitive interpretation, because it pushes μ_j towards an average of the weight values, weighted by the posterior probabilities that the respective weight parameters were generated by component j. Similarly, the derivatives with

Exercise 5.31 respect to the variances are given by

$$\frac{\partial \widetilde{E}}{\partial \sigma_j} = \lambda \sum_i \gamma_j(w_i) \left(\frac{1}{\sigma_j} - \frac{(w_i - \mu_j)^2}{\sigma_j^3} \right) \tag{5.143}$$

which drives σ_j towards the weighted average of the squared deviations of the weights around the corresponding centre μ_j, where the weighting coefficients are again given by the posterior probability that each weight is generated by component j. Note that in a practical implementation, new variables η_j defined by

$$\sigma_j^2 = \exp(\eta_j) \tag{5.144}$$

are introduced, and the minimization is performed with respect to the η_j. This ensures that the parameters σ_j remain positive. It also has the effect of discouraging pathological solutions in which one or more of the σ_j goes to zero, corresponding to a Gaussian component collapsing onto one of the weight parameter values. Such solutions are discussed in more detail in the context of Gaussian mixture models in Section 9.2.1.

For the derivatives with respect to the mixing coefficients π_j, we need to take account of the constraints

$$\sum_j \pi_j = 1, \qquad 0 \leqslant \pi_i \leqslant 1 \tag{5.145}$$

which follow from the interpretation of the π_j as prior probabilities. This can be done by expressing the mixing coefficients in terms of a set of auxiliary variables $\{\eta_j\}$ using the *softmax* function given by

$$\pi_j = \frac{\exp(\eta_j)}{\sum_{k=1}^{M} \exp(\eta_k)}. \tag{5.146}$$

The derivatives of the regularized error function with respect to the $\{\eta_j\}$ then take

Exercise 5.32 the form

Figure 5.18 The left figure shows a two-link robot arm, in which the Cartesian coordinates (x_1, x_2) of the end effector are determined uniquely by the two joint angles θ_1 and θ_2 and the (fixed) lengths L_1 and L_2 of the arms. This is know as the *forward kinematics* of the arm. In practice, we have to find the joint angles that will give rise to a desired end effector position and, as shown in the right figure, this *inverse kinematics* has two solutions corresponding to 'elbow up' and 'elbow down'.

$$\frac{\partial \widetilde{E}}{\partial \eta_j} = \sum_i \{\pi_j - \gamma_j(w_i)\}\,. \tag{5.147}$$

We see that π_j is therefore driven towards the average posterior probability for component j.

5.6. Mixture Density Networks

The goal of supervised learning is to model a conditional distribution $p(\mathbf{t}|\mathbf{x})$, which for many simple regression problems is chosen to be Gaussian. However, practical machine learning problems can often have significantly non-Gaussian distributions. These can arise, for example, with *inverse problems* in which the distribution can be multimodal, in which case the Gaussian assumption can lead to very poor predictions.

Exercise 5.33

As a simple example of an inverse problem, consider the kinematics of a robot arm, as illustrated in Figure 5.18. The *forward problem* involves finding the end effector position given the joint angles and has a unique solution. However, in practice we wish to move the end effector of the robot to a specific position, and to do this we must set appropriate joint angles. We therefore need to solve the inverse problem, which has two solutions as seen in Figure 5.18.

Forward problems often corresponds to causality in a physical system and generally have a unique solution. For instance, a specific pattern of symptoms in the human body may be caused by the presence of a particular disease. In pattern recognition, however, we typically have to solve an inverse problem, such as trying to predict the presence of a disease given a set of symptoms. If the forward problem involves a many-to-one mapping, then the inverse problem will have multiple solutions. For instance, several different diseases may result in the same symptoms.

In the robotics example, the kinematics is defined by geometrical equations, and the multimodality is readily apparent. However, in many machine learning problems the presence of multimodality, particularly in problems involving spaces of high dimensionality, can be less obvious. For tutorial purposes, however, we shall consider a simple toy problem for which we can easily visualize the multimodality. Data for this problem is generated by sampling a variable x uniformly over the interval $(0, 1)$, to give a set of values $\{x_n\}$, and the corresponding target values t_n are obtained

Figure 5.19 On the left is the data set for a simple 'forward problem' in which the red curve shows the result of fitting a two-layer neural network by minimizing the sum-of-squares error function. The corresponding inverse problem, shown on the right, is obtained by exchanging the roles of x and t. Here the same network trained again by minimizing the sum-of-squares error function gives a very poor fit to the data due to the multimodality of the data set.

by computing the function $x_n + 0.3\sin(2\pi x_n)$ and then adding uniform noise over the interval $(-0.1, 0.1)$. The inverse problem is then obtained by keeping the same data points but exchanging the roles of x and t. Figure 5.19 shows the data sets for the forward and inverse problems, along with the results of fitting two-layer neural networks having 6 hidden units and a single linear output unit by minimizing a sum-of-squares error function. Least squares corresponds to maximum likelihood under a Gaussian assumption. We see that this leads to a very poor model for the highly non-Gaussian inverse problem.

We therefore seek a general framework for modelling conditional probability distributions. This can be achieved by using a mixture model for $p(\mathbf{t}|\mathbf{x})$ in which both the mixing coefficients as well as the component densities are flexible functions of the input vector $\mathbf{x}$, giving rise to the *mixture density network*. For any given value of $\mathbf{x}$, the mixture model provides a general formalism for modelling an arbitrary conditional density function $p(\mathbf{t}|\mathbf{x})$. Provided we consider a sufficiently flexible network, we then have a framework for approximating arbitrary conditional distributions.

Here we shall develop the model explicitly for Gaussian components, so that

$$p(\mathbf{t}|\mathbf{x}) = \sum_{k=1}^{K} \pi_k(\mathbf{x}) \mathcal{N}\left(\mathbf{t}|\boldsymbol{\mu}_k(\mathbf{x}), \sigma_k^2(\mathbf{x})\right). \tag{5.148}$$

This is an example of a *heteroscedastic* model since the noise variance on the data is a function of the input vector $\mathbf{x}$. Instead of Gaussians, we can use other distributions for the components, such as Bernoulli distributions if the target variables are binary rather than continuous. We have also specialized to the case of isotropic covariances for the components, although the mixture density network can readily be extended to allow for general covariance matrices by representing the covariances using a Cholesky factorization (Williams, 1996). Even with isotropic components, the conditional distribution $p(\mathbf{t}|\mathbf{x})$ does not assume factorization with respect to the components of $\mathbf{t}$ (in contrast to the standard sum-of-squares regression model) as a consequence of the mixture distribution.

We now take the various parameters of the mixture model, namely the mixing coefficients $\pi_k(\mathbf{x})$, the means $\boldsymbol{\mu}_k(\mathbf{x})$, and the variances $\sigma_k^2(\mathbf{x})$, to be governed by

Figure 5.20 The *mixture density network* can represent general conditional probability densities $p(\mathbf{t}|\mathbf{x})$ by considering a parametric mixture model for the distribution of $\mathbf{t}$ whose parameters are determined by the outputs of a neural network that takes $\mathbf{x}$ as its input vector.

the outputs of a conventional neural network that takes $\mathbf{x}$ as its input. The structure of this mixture density network is illustrated in Figure 5.20. The mixture density network is closely related to the mixture of experts discussed in Section 14.5.3. The principle difference is that in the mixture density network the same function is used to predict the parameters of all of the component densities as well as the mixing coefficients, and so the nonlinear hidden units are shared amongst the input-dependent functions.

The neural network in Figure 5.20 can, for example, be a two-layer network having sigmoidal ('tanh') hidden units. If there are L components in the mixture model (5.148), and if $\mathbf{t}$ has K components, then the network will have L output unit activations denoted by a_k^π that determine the mixing coefficients $\pi_k(\mathbf{x})$, K outputs denoted by a_k^σ that determine the kernel widths $\sigma_k(\mathbf{x})$, and $L \times K$ outputs denoted by a_{kj}^μ that determine the components $\mu_{kj}(\mathbf{x})$ of the kernel centres $\boldsymbol{\mu}_k(\mathbf{x})$. The total number of network outputs is given by $(K+2)L$, as compared with the usual K outputs for a network, which simply predicts the conditional means of the target variables.

The mixing coefficients must satisfy the constraints

$$\sum_{k=1}^{K} \pi_k(\mathbf{x}) = 1, \qquad 0 \leqslant \pi_k(\mathbf{x}) \leqslant 1 \tag{5.149}$$

which can be achieved using a set of softmax outputs

$$\pi_k(\mathbf{x}) = \frac{\exp(a_k^\pi)}{\sum_{l=1}^{K} \exp(a_l^\pi)}. \tag{5.150}$$

Similarly, the variances must satisfy $\sigma_k^2(\mathbf{x}) \geqslant 0$ and so can be represented in terms of the exponentials of the corresponding network activations using

$$\sigma_k(\mathbf{x}) = \exp(a_k^\sigma). \tag{5.151}$$

Finally, because the means $\boldsymbol{\mu}_k(\mathbf{x})$ have real components, they can be represented

directly by the network output activations

$$\mu_{kj}(\mathbf{x}) = a_{kj}^{\mu}. \tag{5.152}$$

The adaptive parameters of the mixture density network comprise the vector $\mathbf{w}$ of weights and biases in the neural network, that can be set by maximum likelihood, or equivalently by minimizing an error function defined to be the negative logarithm of the likelihood. For independent data, this error function takes the form

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{k} \pi_k(\mathbf{x}_n, \mathbf{w}) \mathcal{N}\left(\mathbf{t}_n | \boldsymbol{\mu}_k(\mathbf{x}_n, \mathbf{w}), \sigma_k^2(\mathbf{x}_n, \mathbf{w})\right) \right\} \tag{5.153}$$

where we have made the dependencies on $\mathbf{w}$ explicit.

In order to minimize the error function, we need to calculate the derivatives of the error $E(\mathbf{w})$ with respect to the components of $\mathbf{w}$. These can be evaluated by using the standard backpropagation procedure, provided we obtain suitable expressions for the derivatives of the error with respect to the output-unit activations. These represent error signals δ for each pattern and for each output unit, and can be backpropagated to the hidden units and the error function derivatives evaluated in the usual way. Because the error function (5.153) is composed of a sum of terms, one for each training data point, we can consider the derivatives for a particular pattern n and then find the derivatives of E by summing over all patterns.

Because we are dealing with mixture distributions, it is convenient to view the mixing coefficients $\pi_k(\mathbf{x})$ as $\mathbf{x}$-dependent prior probabilities and to introduce the corresponding posterior probabilities given by

$$\gamma_k(\mathbf{t}|\mathbf{x}) = \frac{\pi_k \mathcal{N}_{nk}}{\sum_{l=1}^{K} \pi_l \mathcal{N}_{nl}} \tag{5.154}$$

where $\mathcal{N}_{nk}$ denotes $\mathcal{N}\left(\mathbf{t}_n | \boldsymbol{\mu}_k(\mathbf{x}_n), \sigma_k^2(\mathbf{x}_n)\right)$.

Exercise 5.34 The derivatives with respect to the network output activations governing the mixing coefficients are given by

$$\frac{\partial E_n}{\partial a_k^{\pi}} = \pi_k - \gamma_k. \tag{5.155}$$

Exercise 5.35 Similarly, the derivatives with respect to the output activations controlling the component means are given by

$$\frac{\partial E_n}{\partial a_{kl}^{\mu}} = \gamma_k \left\{ \frac{\mu_{kl} - t_l}{\sigma_k^2} \right\}. \tag{5.156}$$

Exercise 5.36 Finally, the derivatives with respect to the output activations controlling the component variances are given by

$$\frac{\partial E_n}{\partial a_k^{\sigma}} = -\gamma_k \left\{ \frac{\|\mathbf{t} - \boldsymbol{\mu}_k\|^2}{\sigma_k^3} - \frac{1}{\sigma_k} \right\}. \tag{5.157}$$

Figure 5.21 (a) Plot of the mixing coefficients $\pi_k(x)$ as a function of x for the three kernel functions in a mixture density network trained on the data shown in Figure 5.19. The model has three Gaussian components, and uses a two-layer multilayer perceptron with five '$\tanh$' sigmoidal units in the hidden layer, and nine outputs (corresponding to the 3 means and 3 variances of the Gaussian components and the 3 mixing coefficients). At both small and large values of x, where the conditional probability density of the target data is unimodal, only one of the kernels has a high value for its prior probability, while at intermediate values of x, where the conditional density is trimodal, the three mixing coefficients have comparable values. (b) Plots of the means $\mu_k(x)$ using the same colour coding as for the mixing coefficients. (c) Plot of the contours of the corresponding conditional probability density of the target data for the same mixture density network. (d) Plot of the approximate conditional mode, shown by the red points, of the conditional density.

We illustrate the use of a mixture density network by returning to the toy example of an inverse problem shown in Figure 5.19. Plots of the mixing coefficients $\pi_k(x)$, the means $\mu_k(x)$, and the conditional density contours corresponding to $p(t|x)$, are shown in Figure 5.21. The outputs of the neural network, and hence the parameters in the mixture model, are necessarily continuous single-valued functions of the input variables. However, we see from Figure 5.21(c) that the model is able to produce a conditional density that is unimodal for some values of x and trimodal for other values by modulating the amplitudes of the mixing components $\pi_k(\mathbf{x})$.

Once a mixture density network has been trained, it can predict the conditional density function of the target data for any given value of the input vector. This conditional density represents a complete description of the generator of the data, so far as the problem of predicting the value of the output vector is concerned. From this density function we can calculate more specific quantities that may be of interest in different applications. One of the simplest of these is the mean, corresponding to the conditional average of the target data, and is given by

$$\mathbb{E}\left[\mathbf{t}|\mathbf{x}\right] = \int \mathbf{t}p(\mathbf{t}|\mathbf{x})\,\mathrm{d}\mathbf{t} = \sum_{k=1}^{K} \pi_k(\mathbf{x})\boldsymbol{\mu}_k(\mathbf{x}) \tag{5.158}$$

where we have used (5.148). Because a standard network trained by least squares is approximating the conditional mean, we see that a mixture density network can reproduce the conventional least-squares result as a special case. Of course, as we have already noted, for a multimodal distribution the conditional mean is of limited value.

We can similarly evaluate the variance of the density function about the conditional average, to give

Exercise 5.37

$$s^2(\mathbf{x}) = \mathbb{E}\left[\|\mathbf{t} - \mathbb{E}[\mathbf{t}|\mathbf{x}]\|^2 \,|\mathbf{x}\right] \tag{5.159}$$

$$= \sum_{k=1}^{K} \pi_k(\mathbf{x}) \left\{ \sigma_k^2(\mathbf{x}) + \left\| \boldsymbol{\mu}_k(\mathbf{x}) - \sum_{l=1}^{K} \pi_l(\mathbf{x}) \boldsymbol{\mu}_l(\mathbf{x}) \right\|^2 \right\} \tag{5.160}$$

where we have used (5.148) and (5.158). This is more general than the corresponding least-squares result because the variance is a function of $\mathbf{x}$.

We have seen that for multimodal distributions, the conditional mean can give a poor representation of the data. For instance, in controlling the simple robot arm shown in Figure 5.18, we need to pick one of the two possible joint angle settings in order to achieve the desired end-effector location, whereas the average of the two solutions is not itself a solution. In such cases, the conditional mode may be of more value. Because the conditional mode for the mixture density network does not have a simple analytical solution, this would require numerical iteration. A simple alternative is to take the mean of the most probable component (i.e., the one with the largest mixing coefficient) at each value of $\mathbf{x}$. This is shown for the toy data set in Figure 5.21(d).

5.7. Bayesian Neural Networks

So far, our discussion of neural networks has focussed on the use of maximum likelihood to determine the network parameters (weights and biases). Regularized maximum likelihood can be interpreted as a MAP (maximum posterior) approach in which the regularizer can be viewed as the logarithm of a prior parameter distribution. However, in a Bayesian treatment we need to marginalize over the distribution of parameters in order to make predictions.

In Section 3.3, we developed a Bayesian solution for a simple linear regression model under the assumption of Gaussian noise. We saw that the posterior distribution, which is Gaussian, could be evaluated exactly and that the predictive distribution could also be found in closed form. In the case of a multilayered network, the highly nonlinear dependence of the network function on the parameter values means that an exact Bayesian treatment can no longer be found. In fact, the log of the posterior distribution will be nonconvex, corresponding to the multiple local minima in the error function.

The technique of variational inference, to be discussed in Chapter 10, has been applied to Bayesian neural networks using a factorized Gaussian approximation

to the posterior distribution (Hinton and van Camp, 1993) and also using a full-covariance Gaussian (Barber and Bishop, 1998a; Barber and Bishop, 1998b). The most complete treatment, however, has been based on the Laplace approximation (MacKay, 1992c; MacKay, 1992b) and forms the basis for the discussion given here. We will approximate the posterior distribution by a Gaussian, centred at a mode of the true posterior. Furthermore, we shall assume that the covariance of this Gaussian is small so that the network function is approximately linear with respect to the parameters over the region of parameter space for which the posterior probability is significantly nonzero. With these two approximations, we will obtain models that are analogous to the linear regression and classification models discussed in earlier chapters and so we can exploit the results obtained there. We can then make use of the evidence framework to provide point estimates for the hyperparameters and to compare alternative models (for example, networks having different numbers of hidden units). To start with, we shall discuss the regression case and then later consider the modifications needed for solving classification tasks.

5.7.1 Posterior parameter distribution

Consider the problem of predicting a single continuous target variable t from a vector $\mathbf{x}$ of inputs (the extension to multiple targets is straightforward). We shall suppose that the conditional distribution $p(t|\mathbf{x})$ is Gaussian, with an $\mathbf{x}$-dependent mean given by the output of a neural network model $y(\mathbf{x}, \mathbf{w})$, and with precision (inverse variance) β

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}). \tag{5.161}$$

Similarly, we shall choose a prior distribution over the weights $\mathbf{w}$ that is Gaussian of the form

$$p(\mathbf{w}|\alpha) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I}). \tag{5.162}$$

For an i.i.d. data set of N observations $\mathbf{x}_1, \ldots, \mathbf{x}_N$, with a corresponding set of target values $\mathcal{D} = \{t_1, \ldots, t_N\}$, the likelihood function is given by

$$p(\mathcal{D}|\mathbf{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}(t_n|y(\mathbf{x}_n, \mathbf{w}), \beta^{-1}) \tag{5.163}$$

and so the resulting posterior distribution is then

$$p(\mathbf{w}|\mathcal{D}, \alpha, \beta) \propto p(\mathbf{w}|\alpha)p(\mathcal{D}|\mathbf{w}, \beta). \tag{5.164}$$

which, as a consequence of the nonlinear dependence of $y(\mathbf{x}, \mathbf{w})$ on $\mathbf{w}$, will be non-Gaussian.

We can find a Gaussian approximation to the posterior distribution by using the Laplace approximation. To do this, we must first find a (local) maximum of the posterior, and this must be done using iterative numerical optimization. As usual, it is convenient to maximize the logarithm of the posterior, which can be written in the

form

$$\ln p(\mathbf{w}|\mathcal{D}) = -\frac{\alpha}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} - \frac{\beta}{2}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 + \text{const} \tag{5.165}$$

which corresponds to a regularized sum-of-squares error function. Assuming for the moment that α and β are fixed, we can find a maximum of the posterior, which we denote $\mathbf{w}_{\mathrm{MAP}}$, by standard nonlinear optimization algorithms such as conjugate gradients, using error backpropagation to evaluate the required derivatives.

Having found a mode $\mathbf{w}_{\mathrm{MAP}}$, we can then build a local Gaussian approximation by evaluating the matrix of second derivatives of the negative log posterior distribution. From (5.165), this is given by

$$\mathbf{A} = -\nabla\nabla \ln p(\mathbf{w}|\mathcal{D}, \alpha, \beta) = \alpha\mathbf{I} + \beta\mathbf{H} \tag{5.166}$$

where $\mathbf{H}$ is the Hessian matrix comprising the second derivatives of the sum-of-squares error function with respect to the components of $\mathbf{w}$. Algorithms for computing and approximating the Hessian were discussed in Section 5.4. The corresponding Gaussian approximation to the posterior is then given from (4.134) by

$$q(\mathbf{w}|\mathcal{D}) = \mathcal{N}(\mathbf{w}|\mathbf{w}_{\mathrm{MAP}}, \mathbf{A}^{-1}). \tag{5.167}$$

Similarly, the predictive distribution is obtained by marginalizing with respect to this posterior distribution

$$p(t|\mathbf{x}, \mathcal{D}) = \int p(t|\mathbf{x}, \mathbf{w})q(\mathbf{w}|\mathcal{D})\,\mathrm{d}\mathbf{w}. \tag{5.168}$$

However, even with the Gaussian approximation to the posterior, this integration is still analytically intractable due to the nonlinearity of the network function $y(\mathbf{x}, \mathbf{w})$ as a function of $\mathbf{w}$. To make progress, we now assume that the posterior distribution has small variance compared with the characteristic scales of $\mathbf{w}$ over which $y(\mathbf{x}, \mathbf{w})$ is varying. This allows us to make a Taylor series expansion of the network function around $\mathbf{w}_{\mathrm{MAP}}$ and retain only the linear terms

$$y(\mathbf{x}, \mathbf{w}) \simeq y(\mathbf{x}, \mathbf{w}_{\mathrm{MAP}}) + \mathbf{g}^{\mathbf{T}}(\mathbf{w} - \mathbf{w}_{\mathrm{MAP}}) \tag{5.169}$$

where we have defined

$$\mathbf{g} = \nabla_{\mathbf{w}} y(\mathbf{x}, \mathbf{w})|_{\mathbf{w}=\mathbf{w}_{\mathrm{MAP}}}. \tag{5.170}$$

With this approximation, we now have a linear-Gaussian model with a Gaussian distribution for $p(\mathbf{w})$ and a Gaussian for $p(t|\mathbf{w})$ whose mean is a linear function of $\mathbf{w}$ of the form

$$p(t|\mathbf{x}, \mathbf{w}, \beta) \simeq \mathcal{N}\left(t|y(\mathbf{x}, \mathbf{w}_{\mathrm{MAP}}) + \mathbf{g}^{\mathbf{T}}(\mathbf{w} - \mathbf{w}_{\mathrm{MAP}}), \beta^{-1}\right). \tag{5.171}$$

Exercise 5.38 We can therefore make use of the general result (2.115) for the marginal $p(t)$ to give

$$p(t|\mathbf{x}, \mathcal{D}, \alpha, \beta) = \mathcal{N}\left(t|y(\mathbf{x}, \mathbf{w}_{\mathrm{MAP}}), \sigma^2(\mathbf{x})\right) \tag{5.172}$$

where the input-dependent variance is given by

$$\sigma^2(\mathbf{x}) = \beta^{-1} + \mathbf{g}^{\mathrm{T}}\mathbf{A}^{-1}\mathbf{g}. \tag{5.173}$$

We see that the predictive distribution $p(t|\mathbf{x}, \mathcal{D})$ is a Gaussian whose mean is given by the network function $y(\mathbf{x}, \mathbf{w}_{\mathrm{MAP}})$ with the parameter set to their MAP value. The variance has two terms, the first of which arises from the intrinsic noise on the target variable, whereas the second is an $\mathbf{x}$-dependent term that expresses the uncertainty in the interpolant due to the uncertainty in the model parameters $\mathbf{w}$. This should be compared with the corresponding predictive distribution for the linear regression model, given by (3.58) and (3.59).

5.7.2 Hyperparameter optimization

So far, we have assumed that the hyperparameters α and β are fixed and known. We can make use of the evidence framework, discussed in Section 3.5, together with the Gaussian approximation to the posterior obtained using the Laplace approximation, to obtain a practical procedure for choosing the values of such hyperparameters.

The marginal likelihood, or evidence, for the hyperparameters is obtained by integrating over the network weights

$$p(\mathcal{D}|\alpha, \beta) = \int p(\mathcal{D}|\mathbf{w}, \beta)p(\mathbf{w}|\alpha)\,\mathrm{d}\mathbf{w}. \tag{5.174}$$

Exercise 5.39

This is easily evaluated by making use of the Laplace approximation result (4.135). Taking logarithms then gives

$$\ln p(\mathcal{D}|\alpha, \beta) \simeq -E(\mathbf{w}_{\mathrm{MAP}}) - \frac{1}{2}\ln|\mathbf{A}| + \frac{W}{2}\ln\alpha + \frac{N}{2}\ln\beta - \frac{N}{2}\ln(2\pi) \tag{5.175}$$

where W is the total number of parameters in $\mathbf{w}$, and the regularized error function is defined by

$$E(\mathbf{w}_{\mathrm{MAP}}) = \frac{\beta}{2}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}_{\mathrm{MAP}}) - t_n\}^2 + \frac{\alpha}{2}\mathbf{w}_{\mathrm{MAP}}^{\mathrm{T}}\mathbf{w}_{\mathrm{MAP}}. \tag{5.176}$$

We see that this takes the same form as the corresponding result (3.86) for the linear regression model.

In the evidence framework, we make point estimates for α and β by maximizing $\ln p(\mathcal{D}|\alpha, \beta)$. Consider first the maximization with respect to α, which can be done by analogy with the linear regression case discussed in Section 3.5.2. We first define the eigenvalue equation

$$\beta\mathbf{H}\mathbf{u}_i = \lambda_i\mathbf{u}_i \tag{5.177}$$

where $\mathbf{H}$ is the Hessian matrix comprising the second derivatives of the sum-of-squares error function, evaluated at $\mathbf{w} = \mathbf{w}_{\mathrm{MAP}}$. By analogy with (3.92), we obtain

$$\alpha = \frac{\gamma}{\mathbf{w}_{\mathrm{MAP}}^{\mathrm{T}}\mathbf{w}_{\mathrm{MAP}}} \tag{5.178}$$

Section 3.5.3

where γ represents the effective number of parameters and is defined by

$$\gamma = \sum_{i=1}^{W} \frac{\lambda_i}{\alpha + \lambda_i}. \tag{5.179}$$

Note that this result was exact for the linear regression case. For the nonlinear neural network, however, it ignores the fact that changes in α will cause changes in the Hessian $\mathbf{H}$, which in turn will change the eigenvalues. We have therefore implicitly ignored terms involving the derivatives of λ_i with respect to α.

Similarly, from (3.95) we see that maximizing the evidence with respect to β gives the re-estimation formula

$$\frac{1}{\beta} = \frac{1}{N - \gamma} \sum_{n=1}^{N} \{y(\mathbf{x}_n, \mathbf{w}_{\mathrm{MAP}}) - t_n\}^2. \tag{5.180}$$

As with the linear model, we need to alternate between re-estimation of the hyperparameters α and β and updating of the posterior distribution. The situation with a neural network model is more complex, however, due to the multimodality of the posterior distribution. As a consequence, the solution for $\mathbf{w}_{\mathrm{MAP}}$ found by maximizing the log posterior will depend on the initialization of $\mathbf{w}$. Solutions that differ only as a consequence of the interchange and sign reversal symmetries in the hidden units are identical so far as predictions are concerned, and it is irrelevant which of the equivalent solutions is found. However, there may be inequivalent solutions as well, and these will generally yield different values for the optimized hyperparameters.

Section 5.1.1

In order to compare different models, for example neural networks having different numbers of hidden units, we need to evaluate the model evidence $p(\mathcal{D})$. This can be approximated by taking (5.175) and substituting the values of α and β obtained from the iterative optimization of these hyperparameters. A more careful evaluation is obtained by marginalizing over α and β, again by making a Gaussian approximation (MacKay, 1992c; Bishop, 1995a). In either case, it is necessary to evaluate the determinant $|\mathbf{A}|$ of the Hessian matrix. This can be problematic in practice because the determinant, unlike the trace, is sensitive to the small eigenvalues that are often difficult to determine accurately.

The Laplace approximation is based on a local quadratic expansion around a mode of the posterior distribution over weights. We have seen in Section 5.1.1 that any given mode in a two-layer network is a member of a set of $M!2^M$ equivalent modes that differ by interchange and sign-change symmetries, where M is the number of hidden units. When comparing networks having different numbers of hidden units, this can be taken into account by multiplying the evidence by a factor of $M!2^M$.

5.7.3 Bayesian neural networks for classification

So far, we have used the Laplace approximation to develop a Bayesian treatment of neural network regression models. We now discuss the modifications to

this framework that arise when it is applied to classification. Here we shall consider a network having a single logistic sigmoid output corresponding to a two-class classification problem. The extension to networks with multiclass softmax outputs is straightforward. We shall build extensively on the analogous results for linear classification models discussed in Section 4.5, and so we encourage the reader to familiarize themselves with that material before studying this section.

Exercise 5.40

The log likelihood function for this model is given by

$$\ln p(\mathcal{D}|\mathbf{w}) = \sum_n = 1^N \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\} \tag{5.181}$$

where $t_n \in \{0, 1\}$ are the target values, and $y_n \equiv y(\mathbf{x}_n, \mathbf{w})$. Note that there is no hyperparameter β, because the data points are assumed to be correctly labelled. As before, the prior is taken to be an isotropic Gaussian of the form (5.162).

The first stage in applying the Laplace framework to this model is to initialize the hyperparameter α, and then to determine the parameter vector $\mathbf{w}$ by maximizing the log posterior distribution. This is equivalent to minimizing the regularized error function

$$E(\mathbf{w}) = -\ln p(\mathcal{D}|\mathbf{w}) + \frac{\alpha}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \tag{5.182}$$

and can be achieved using error backpropagation combined with standard optimization algorithms, as discussed in Section 5.3.

Having found a solution $\mathbf{w}_{\mathrm{MAP}}$ for the weight vector, the next step is to evaluate the Hessian matrix $\mathbf{H}$ comprising the second derivatives of the negative log likelihood function. This can be done, for instance, using the exact method of Section 5.4.5, or using the outer product approximation given by (5.85). The second derivatives of the negative log posterior can again be written in the form (5.166), and the Gaussian approximation to the posterior is then given by (5.167).

To optimize the hyperparameter α, we again maximize the marginal likelihood, which is easily shown to take the form

Exercise 5.41

$$\ln p(\mathcal{D}|\alpha) \simeq -E(\mathbf{w}_{\mathrm{MAP}}) - \frac{1}{2}\ln|\mathbf{A}| + \frac{W}{2}\ln\alpha + \mathrm{const} \tag{5.183}$$

where the regularized error function is defined by

$$E(\mathbf{w}_{\mathrm{MAP}}) = -\sum_{n=1}^{N}\{t_n \ln y_n + (1 - t_n)\ln(1 - y_n)\} + \frac{\alpha}{2}\mathbf{w}_{\mathrm{MAP}}^{\mathrm{T}}\mathbf{w}_{\mathrm{MAP}} \tag{5.184}$$

in which $y_n \equiv y(\mathbf{x}_n, \mathbf{w}_{\mathrm{MAP}})$. Maximizing this evidence function with respect to α again leads to the re-estimation equation given by (5.178).

The use of the evidence procedure to determine α is illustrated in Figure 5.22 for the synthetic two-dimensional data discussed in Appendix A.

Finally, we need the predictive distribution, which is defined by (5.168). Again, this integration is intractable due to the nonlinearity of the network function. The

Figure 5.22 Illustration of the evidence framework applied to a synthetic two-class data set. The green curve shows the optimal decision boundary, the black curve shows the result of fitting a two-layer network with 8 hidden units by maximum likelihood, and the red curve shows the result of including a regularizer in which α is optimized using the evidence procedure, starting from the initial value $\alpha = 0$. Note that the evidence procedure greatly reduces the over-fitting of the network.

simplest approximation is to assume that the posterior distribution is very narrow and hence make the approximation

$$p(t|\mathbf{x}, \mathcal{D}) \simeq p(t|\mathbf{x}, \mathbf{w}_{\mathrm{MAP}}). \tag{5.185}$$

We can improve on this, however, by taking account of the variance of the posterior distribution. In this case, a linear approximation for the network outputs, as was used in the case of regression, would be inappropriate due to the logistic sigmoid output-unit activation function that constrains the output to lie in the range $(0, 1)$. Instead, we make a linear approximation for the output unit activation in the form

$$a(\mathbf{x}, \mathbf{w}) \simeq a_{\mathrm{MAP}}(\mathbf{x}) + \mathbf{b}^{\mathrm{T}}(\mathbf{w} - \mathbf{w}_{\mathrm{MAP}}) \tag{5.186}$$

where $a_{\mathrm{MAP}}(\mathbf{x}) = a(\mathbf{x}, \mathbf{w}_{\mathrm{MAP}})$, and the vector $\mathbf{b} \equiv \nabla a(\mathbf{x}, \mathbf{w}_{\mathrm{MAP}})$ can be found by backpropagation.

Because we now have a Gaussian approximation for the posterior distribution over $\mathbf{w}$, and a model for a that is a linear function of $\mathbf{w}$, we can now appeal to the results of Section 4.5.2. The distribution of output unit activation values, induced by the distribution over network weights, is given by

$$p(a|\mathbf{x}, \mathcal{D}) = \int \delta\left(a - a_{\mathrm{MAP}}(\mathbf{x}) - \mathbf{b}^{\mathrm{T}}(\mathbf{x})(\mathbf{w} - \mathbf{w}_{\mathrm{MAP}})\right) q(\mathbf{w}|\mathcal{D})\, \mathrm{d}\mathbf{w} \tag{5.187}$$

where $q(\mathbf{w}|\mathcal{D})$ is the Gaussian approximation to the posterior distribution given by (5.167). From Section 4.5.2, we see that this distribution is Gaussian with mean $a_{\mathrm{MAP}} \equiv a(\mathbf{x}, \mathbf{w}_{\mathrm{MAP}})$, and variance

$$\sigma_a^2(\mathbf{x}) = \mathbf{b}^{\mathrm{T}}(\mathbf{x})\mathbf{A}^{-1}\mathbf{b}(\mathbf{x}). \tag{5.188}$$

Finally, to obtain the predictive distribution, we must marginalize over a using

$$p(t = 1|\mathbf{x}, \mathcal{D}) = \int \sigma(a) p(a|\mathbf{x}, \mathcal{D})\, \mathrm{d}a. \tag{5.189}$$

Figure 5.23 An illustration of the Laplace approximation for a Bayesian neural network having 8 hidden units with '$\tanh$' activation functions and a single logistic-sigmoid output unit. The weight parameters were found using scaled conjugate gradients, and the hyperparameter α was optimized using the evidence framework. On the left is the result of using the simple approximation (5.185) based on a point estimate $\mathbf{w}_{\mathrm{MAP}}$ of the parameters, in which the green curve shows the $y = 0.5$ decision boundary, and the other contours correspond to output probabilities of $y = 0.1$, 0.3, 0.7, and 0.9. On the right is the corresponding result obtained using (5.190). Note that the effect of marginalization is to spread out the contours and to make the predictions less confident, so that at each input point $\mathbf{x}$, the posterior probabilities are shifted towards 0.5, while the $y = 0.5$ contour itself is unaffected.

The convolution of a Gaussian with a logistic sigmoid is intractable. We therefore apply the approximation (4.153) to (5.189) giving

$$p(t = 1|\mathbf{x}, \mathcal{D}) = \sigma\left(\kappa(\sigma_a^2)\mathbf{b}^{\mathrm{T}}\mathbf{w}_{\mathrm{MAP}}\right) \tag{5.190}$$

where $\kappa(\cdot)$ is defined by (4.154). Recall that both σ_a^2 and $\mathbf{b}$ are functions of $\mathbf{x}$.

Figure 5.23 shows an example of this framework applied to the synthetic classification data set described in Appendix A.

Exercises

5.1 ($\star\star$) Consider a two-layer network function of the form (5.7) in which the hidden-unit nonlinear activation functions $g(\cdot)$ are given by logistic sigmoid functions of the form

$$\sigma(a) = \{1 + \exp(-a)\}^{-1}. \tag{5.191}$$

Show that there exists an equivalent network, which computes exactly the same function, but with hidden unit activation functions given by $\tanh(a)$ where the $\tanh$ function is defined by (5.59). Hint: first find the relation between $\sigma(a)$ and $\tanh(a)$, and then show that the parameters of the two networks differ by linear transformations.

5.2 ($\star$) **www** Show that maximizing the likelihood function under the conditional distribution (5.16) for a multioutput neural network is equivalent to minimizing the sum-of-squares error function (5.11).

5.3 ($\star\star$) Consider a regression problem involving multiple target variables in which it is assumed that the distribution of the targets, conditioned on the input vector $\mathbf{x}$, is a Gaussian of the form

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \mathcal{N}(\mathbf{t}|\mathbf{y}(\mathbf{x}, \mathbf{w}), \boldsymbol{\Sigma}) \tag{5.192}$$

where $\mathbf{y}(\mathbf{x}, \mathbf{w})$ is the output of a neural network with input vector $\mathbf{x}$ and weight vector $\mathbf{w}$, and $\boldsymbol{\Sigma}$ is the covariance of the assumed Gaussian noise on the targets. Given a set of independent observations of $\mathbf{x}$ and $\mathbf{t}$, write down the error function that must be minimized in order to find the maximum likelihood solution for $\mathbf{w}$, if we assume that $\boldsymbol{\Sigma}$ is fixed and known. Now assume that $\boldsymbol{\Sigma}$ is also to be determined from the data, and write down an expression for the maximum likelihood solution for $\boldsymbol{\Sigma}$. Note that the optimizations of $\mathbf{w}$ and $\boldsymbol{\Sigma}$ are now coupled, in contrast to the case of independent target variables discussed in Section 5.2.

5.4 ($\star\star$) Consider a binary classification problem in which the target values are $t \in \{0, 1\}$, with a network output $y(\mathbf{x}, \mathbf{w})$ that represents $p(t = 1|\mathbf{x})$, and suppose that there is a probability ϵ that the class label on a training data point has been incorrectly set. Assuming independent and identically distributed data, write down the error function corresponding to the negative log likelihood. Verify that the error function (5.21) is obtained when $\epsilon = 0$. Note that this error function makes the model robust to incorrectly labelled data, in contrast to the usual error function.

5.5 ($\star$) **www** Show that maximizing likelihood for a multiclass neural network model in which the network outputs have the interpretation $y_k(\mathbf{x}, \mathbf{w}) = p(t_k = 1|\mathbf{x})$ is equivalent to the minimization of the cross-entropy error function (5.24).

5.6 ($\star$) **www** Show the derivative of the error function (5.21) with respect to the activation a_k for an output unit having a logistic sigmoid activation function satisfies (5.18).

5.7 ($\star$) Show the derivative of the error function (5.24) with respect to the activation a_k for output units having a softmax activation function satisfies (5.18).

5.8 ($\star$) We saw in (4.88) that the derivative of the logistic sigmoid activation function can be expressed in terms of the function value itself. Derive the corresponding result for the '$\tanh$' activation function defined by (5.59).

5.9 ($\star$) **www** The error function (5.21) for binary classification problems was derived for a network having a logistic-sigmoid output activation function, so that $0 \leqslant y(\mathbf{x}, \mathbf{w}) \leqslant 1$, and data having target values $t \in \{0, 1\}$. Derive the corresponding error function if we consider a network having an output $-1 \leqslant y(\mathbf{x}, \mathbf{w}) \leqslant 1$ and target values $t = 1$ for class $\mathcal{C}_1$ and $t = -1$ for class $\mathcal{C}_2$. What would be the appropriate choice of output unit activation function?

5.10 ($\star$) **www** Consider a Hessian matrix $\mathbf{H}$ with eigenvector equation (5.33). By setting the vector $\mathbf{v}$ in (5.39) equal to each of the eigenvectors $\mathbf{u}_i$ in turn, show that $\mathbf{H}$ is positive definite if, and only if, all of its eigenvalues are positive.

5.11 ($\star\star$) **WWW** Consider a quadratic error function defined by (5.32), in which the Hessian matrix $\mathbf{H}$ has an eigenvalue equation given by (5.33). Show that the contours of constant error are ellipses whose axes are aligned with the eigenvectors $\mathbf{u}_i$, with lengths that are inversely proportional to the square root of the corresponding eigenvalues λ_i.

5.12 ($\star\star$) **WWW** By considering the local Taylor expansion (5.32) of an error function about a stationary point $\mathbf{w}^\star$, show that the necessary and sufficient condition for the stationary point to be a local minimum of the error function is that the Hessian matrix $\mathbf{H}$, defined by (5.30) with $\widehat{\mathbf{w}} = \mathbf{w}^\star$, be positive definite.

5.13 ($\star$) Show that as a consequence of the symmetry of the Hessian matrix $\mathbf{H}$, the number of independent elements in the quadratic error function (5.28) is given by $W(W+3)/2$.

5.14 ($\star$) By making a Taylor expansion, verify that the terms that are $O(\epsilon)$ cancel on the right-hand side of (5.69).

5.15 ($\star\star$) In Section 5.3.4, we derived a procedure for evaluating the Jacobian matrix of a neural network using a backpropagation procedure. Derive an alternative formalism for finding the Jacobian based on *forward propagation* equations.

5.16 ($\star$) The outer product approximation to the Hessian matrix for a neural network using a sum-of-squares error function is given by (5.84). Extend this result to the case of multiple outputs.

5.17 ($\star$) Consider a squared loss function of the form

$$E = \frac{1}{2} \iint \{y(\mathbf{x}, \mathbf{w}) - t\}^2 \, p(\mathbf{x}, t) \, \mathrm{d}\mathbf{x} \, \mathrm{d}t \tag{5.193}$$

where $y(\mathbf{x}, \mathbf{w})$ is a parametric function such as a neural network. The result (1.89) shows that the function $y(\mathbf{x}, \mathbf{w})$ that minimizes this error is given by the conditional expectation of t given $\mathbf{x}$. Use this result to show that the second derivative of E with respect to two elements w_r and w_s of the vector $\mathbf{w}$, is given by

$$\frac{\partial^2 E}{\partial w_r \partial w_s} = \int \frac{\partial y}{\partial w_r} \frac{\partial y}{\partial w_s} p(\mathbf{x}) \, \mathrm{d}\mathbf{x}. \tag{5.194}$$

Note that, for a finite sample from $p(\mathbf{x})$, we obtain (5.84).

5.18 ($\star$) Consider a two-layer network of the form shown in Figure 5.1 with the addition of extra parameters corresponding to skip-layer connections that go directly from the inputs to the outputs. By extending the discussion of Section 5.3.2, write down the equations for the derivatives of the error function with respect to these additional parameters.

5.19 ($\star$) **WWW** Derive the expression (5.85) for the outer product approximation to the Hessian matrix for a network having a single output with a logistic sigmoid output-unit activation function and a cross-entropy error function, corresponding to the result (5.84) for the sum-of-squares error function.

5.20 ($\star$) Derive an expression for the outer product approximation to the Hessian matrix for a network having K outputs with a softmax output-unit activation function and a cross-entropy error function, corresponding to the result (5.84) for the sum-of-squares error function.

5.21 ($\star\star\star$) Extend the expression (5.86) for the outer product approximation of the Hessian matrix to the case of $K > 1$ output units. Hence, derive a recursive expression analogous to (5.87) for incrementing the number N of patterns and a similar expression for incrementing the number K of outputs. Use these results, together with the identity (5.88), to find sequential update expressions analogous to (5.89) for finding the inverse of the Hessian by incrementally including both extra patterns and extra outputs.

5.22 ($\star\star$) Derive the results (5.93), (5.94), and (5.95) for the elements of the Hessian matrix of a two-layer feed-forward network by application of the chain rule of calculus.

5.23 ($\star\star$) Extend the results of Section 5.4.5 for the exact Hessian of a two-layer network to include skip-layer connections that go directly from inputs to outputs.

5.24 ($\star$) Verify that the network function defined by (5.113) and (5.114) is invariant under the transformation (5.115) applied to the inputs, provided the weights and biases are simultaneously transformed using (5.116) and (5.117). Similarly, show that the network outputs can be transformed according (5.118) by applying the transformation (5.119) and (5.120) to the second-layer weights and biases.

5.25 ($\star\star\star$) **WWW** Consider a quadratic error function of the form

$$E = E_0 + \frac{1}{2}(\mathbf{w} - \mathbf{w}^\star)^{\mathrm{T}}\mathbf{H}(\mathbf{w} - \mathbf{w}^\star) \tag{5.195}$$

where $\mathbf{w}^\star$ represents the minimum, and the Hessian matrix $\mathbf{H}$ is positive definite and constant. Suppose the initial weight vector $\mathbf{w}^{(0)}$ is chosen to be at the origin and is updated using simple gradient descent

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \rho\nabla E \tag{5.196}$$

where τ denotes the step number, and ρ is the learning rate (which is assumed to be small). Show that, after τ steps, the components of the weight vector parallel to the eigenvectors of $\mathbf{H}$ can be written

$$w_j^{(\tau)} = \{1 - (1 - \rho\eta_j)^\tau\}\, w_j^\star \tag{5.197}$$

where $w_j = \mathbf{w}^{\mathrm{T}}\mathbf{u}_j$, and $\mathbf{u}_j$ and η_j are the eigenvectors and eigenvalues, respectively, of $\mathbf{H}$ so that

$$\mathbf{H}\mathbf{u}_j = \eta_j\mathbf{u}_j. \tag{5.198}$$

Show that as $\tau \to \infty$, this gives $\mathbf{w}^{(\tau)} \to \mathbf{w}^\star$ as expected, provided $|1 - \rho\eta_j| < 1$. Now suppose that training is halted after a finite number τ of steps. Show that the

components of the weight vector parallel to the eigenvectors of the Hessian satisfy

$$w_j^{(\tau)} \simeq w_j^\star \quad \text{when} \quad \eta_j \gg (\rho\tau)^{-1} \tag{5.199}$$

$$|w_j^{(\tau)}| \ll |w_j^\star| \quad \text{when} \quad \eta_j \ll (\rho\tau)^{-1}. \tag{5.200}$$

Compare this result with the discussion in Section 3.5.3 of regularization with simple weight decay, and hence show that $(\rho\tau)^{-1}$ is analogous to the regularization parameter λ. The above results also show that the effective number of parameters in the network, as defined by (3.91), grows as the training progresses.

5.26 ($\star\star$) Consider a multilayer perceptron with arbitrary feed-forward topology, which is to be trained by minimizing the *tangent propagation* error function (5.127) in which the regularizing function is given by (5.128). Show that the regularization term Ω can be written as a sum over patterns of terms of the form

$$\Omega_n = \frac{1}{2}\sum_k (\mathcal{G}y_k)^2 \tag{5.201}$$

where $\mathcal{G}$ is a differential operator defined by

$$\mathcal{G} \equiv \sum_i \tau_i \frac{\partial}{\partial x_i}. \tag{5.202}$$

By acting on the forward propagation equations

$$z_j = h(a_j), \qquad a_j = \sum_i w_{ji} z_i \tag{5.203}$$

with the operator $\mathcal{G}$, show that Ω_n can be evaluated by forward propagation using the following equations:

$$\alpha_j = h'(a_j)\beta_j, \qquad \beta_j = \sum_i w_{ji}\alpha_i. \tag{5.204}$$

where we have defined the new variables

$$\alpha_j \equiv \mathcal{G}z_j, \qquad \beta_j \equiv \mathcal{G}a_j. \tag{5.205}$$

Now show that the derivatives of Ω_n with respect to a weight w_{rs} in the network can be written in the form

$$\frac{\partial \Omega_n}{\partial w_{rs}} = \sum_k \alpha_k \left\{\phi_{kr} z_s + \delta_{kr}\alpha_s\right\} \tag{5.206}$$

where we have defined

$$\delta_{kr} \equiv \frac{\partial y_k}{\partial a_r}, \qquad \phi_{kr} \equiv \mathcal{G}\delta_{kr}. \tag{5.207}$$

Write down the backpropagation equations for δ_{kr}, and hence derive a set of backpropagation equations for the evaluation of the ϕ_{kr}.

5.27 ($\star\star$) **www** Consider the framework for training with transformed data in the special case in which the transformation consists simply of the addition of random noise $\mathbf{x} \rightarrow \mathbf{x} + \boldsymbol{\xi}$ where $\boldsymbol{\xi}$ has a Gaussian distribution with zero mean and unit covariance. By following an argument analogous to that of Section 5.5.5, show that the resulting regularizer reduces to the Tikhonov form (5.135).

5.28 ($\star$) **www** Consider a neural network, such as the convolutional network discussed in Section 5.5.6, in which multiple weights are constrained to have the same value. Discuss how the standard backpropagation algorithm must be modified in order to ensure that such constraints are satisfied when evaluating the derivatives of an error function with respect to the adjustable parameters in the network.

5.29 ($\star$) **www** Verify the result (5.141).

5.30 ($\star$) Verify the result (5.142).

5.31 ($\star$) Verify the result (5.143).

5.32 ($\star\star$) Show that the derivatives of the mixing coefficients $\{\pi_k\}$, defined by (5.146), with respect to the auxiliary parameters $\{\eta_j\}$ are given by

$$\frac{\partial \pi_k}{\partial \eta_j} = \delta_{jk}\pi_j - \pi_j \pi_k. \tag{5.208}$$

Hence, by making use of the constraint $\sum_k \pi_k = 1$, derive the result (5.147).

5.33 ($\star$) Write down a pair of equations that express the Cartesian coordinates (x_1, x_2) for the robot arm shown in Figure 5.18 in terms of the joint angles θ_1 and θ_2 and the lengths L_1 and L_2 of the links. Assume the origin of the coordinate system is given by the attachment point of the lower arm. These equations define the 'forward kinematics' of the robot arm.

5.34 ($\star$) **www** Derive the result (5.155) for the derivative of the error function with respect to the network output activations controlling the mixing coefficients in the mixture density network.

5.35 ($\star$) Derive the result (5.156) for the derivative of the error function with respect to the network output activations controlling the component means in the mixture density network.

5.36 ($\star$) Derive the result (5.157) for the derivative of the error function with respect to the network output activations controlling the component variances in the mixture density network.

5.37 ($\star$) Verify the results (5.158) and (5.160) for the conditional mean and variance of the mixture density network model.

5.38 ($\star$) Using the general result (2.115), derive the predictive distribution (5.172) for the Laplace approximation to the Bayesian neural network model.

5.39 ($\star$) **www** Make use of the Laplace approximation result (4.135) to show that the evidence function for the hyperparameters α and β in the Bayesian neural network model can be approximated by (5.175).

5.40 ($\star$) **www** Outline the modifications needed to the framework for Bayesian neural networks, discussed in Section 5.7.3, to handle multiclass problems using networks having softmax output-unit activation functions.

5.41 ($\star\star$) By following analogous steps to those given in Section 5.7.1 for regression networks, derive the result (5.183) for the marginal likelihood in the case of a network having a cross-entropy error function and logistic-sigmoid output-unit activation function.

6 Kernel Methods

In Chapters 3 and 4, we considered linear parametric models for regression and classification in which the form of the mapping $y(\mathbf{x}, \mathbf{w})$ from input $\mathbf{x}$ to output y is governed by a vector $\mathbf{w}$ of adaptive parameters. During the learning phase, a set of training data is used either to obtain a point estimate of the parameter vector or to determine a posterior distribution over this vector. The training data is then discarded, and predictions for new inputs are based purely on the learned parameter vector $\mathbf{w}$. This approach is also used in nonlinear parametric models such as neural networks.

Chapter 5

However, there is a class of pattern recognition techniques, in which the training data points, or a subset of them, are kept and used also during the prediction phase. For instance, the Parzen probability density model comprised a linear combination of 'kernel' functions each one centred on one of the training data points. Similarly, in Section 2.5.2 we introduced a simple technique for classification called nearest neighbours, which involved assigning to each new test vector the same label as the

Section 2.5.1

closest example from the training set. These are examples of *memory-based* methods that involve storing the entire training set in order to make predictions for future data points. They typically require a metric to be defined that measures the similarity of any two vectors in input space, and are generally fast to 'train' but slow at making predictions for test data points.

Many linear parametric models can be re-cast into an equivalent 'dual representation' in which the predictions are also based on linear combinations of a *kernel function* evaluated at the training data points. As we shall see, for models which are based on a fixed nonlinear *feature space* mapping $\boldsymbol{\phi}(\mathbf{x})$, the kernel function is given by the relation

$$k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}'). \tag{6.1}$$

From this definition, we see that the kernel is a symmetric function of its arguments so that $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$. The kernel concept was introduced into the field of pattern recognition by Aizerman *et al.* (1964) in the context of the method of potential functions, so-called because of an analogy with electrostatics. Although neglected for many years, it was re-introduced into machine learning in the context of large-margin classifiers by Boser *et al.* (1992) giving rise to the technique of *support vector machines*. Since then, there has been considerable interest in this topic, both in terms of theory and applications. One of the most significant developments has been the extension of kernels to handle symbolic objects, thereby greatly expanding the range of problems that can be addressed.

Chapter 7

The simplest example of a kernel function is obtained by considering the identity mapping for the feature space in (6.1) so that $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$, in which case $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^{\mathrm{T}}\mathbf{x}'$. We shall refer to this as the linear kernel.

The concept of a kernel formulated as an inner product in a feature space allows us to build interesting extensions of many well-known algorithms by making use of the *kernel trick*, also known as *kernel substitution*. The general idea is that, if we have an algorithm formulated in such a way that the input vector $\mathbf{x}$ enters only in the form of scalar products, then we can replace that scalar product with some other choice of kernel. For instance, the technique of kernel substitution can be applied to principal component analysis in order to develop a nonlinear variant of PCA (Schölkopf *et al.*, 1998). Other examples of kernel substitution include nearest-neighbour classifiers and the kernel Fisher discriminant (Mika *et al.*, 1999; Roth and Steinhage, 2000; Baudat and Anouar, 2000).

Section 12.3

There are numerous forms of kernel functions in common use, and we shall encounter several examples in this chapter. Many have the property of being a function only of the difference between the arguments, so that $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x} - \mathbf{x}')$, which are known as *stationary* kernels because they are invariant to translations in input space. A further specialization involves *homogeneous* kernels, also known as *radial basis functions*, which depend only on the magnitude of the distance (typically Euclidean) between the arguments so that $k(\mathbf{x}, \mathbf{x}') = k(\|\mathbf{x} - \mathbf{x}'\|)$.

Section 6.3

For recent textbooks on kernel methods, see Schölkopf and Smola (2002), Herbrich (2002), and Shawe-Taylor and Cristianini (2004).

6.1. Dual Representations

Many linear models for regression and classification can be reformulated in terms of a dual representation in which the kernel function arises naturally. This concept will play an important role when we consider support vector machines in the next chapter. Here we consider a linear regression model whose parameters are determined by minimizing a regularized sum-of-squares error function given by

$$J(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\left\{\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n) - t_n\right\}^2 + \frac{\lambda}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \tag{6.2}$$

where $\lambda \geqslant 0$. If we set the gradient of $J(\mathbf{w})$ with respect to $\mathbf{w}$ equal to zero, we see that the solution for $\mathbf{w}$ takes the form of a linear combination of the vectors $\boldsymbol{\phi}(\mathbf{x}_n)$, with coefficients that are functions of $\mathbf{w}$, of the form

$$\mathbf{w} = -\frac{1}{\lambda}\sum_{n=1}^{N}\left\{\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n) - t_n\right\}\boldsymbol{\phi}(\mathbf{x}_n) = \sum_{n=1}^{N} a_n\boldsymbol{\phi}(\mathbf{x}_n) = \boldsymbol{\Phi}^{\mathrm{T}}\mathbf{a} \tag{6.3}$$

where $\boldsymbol{\Phi}$ is the design matrix, whose n^{th} row is given by $\boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}}$. Here the vector $\mathbf{a} = (a_1, \ldots, a_N)^{\mathrm{T}}$, and we have defined

$$a_n = -\frac{1}{\lambda}\left\{\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n) - t_n\right\}. \tag{6.4}$$

Instead of working with the parameter vector $\mathbf{w}$, we can now reformulate the least-squares algorithm in terms of the parameter vector $\mathbf{a}$, giving rise to a *dual representation*. If we substitute $\mathbf{w} = \boldsymbol{\Phi}^{\mathrm{T}}\mathbf{a}$ into $J(\mathbf{w})$, we obtain

$$J(\mathbf{a}) = \frac{1}{2}\mathbf{a}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{a} - \mathbf{a}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t} + \frac{1}{2}\mathbf{t}^{\mathrm{T}}\mathbf{t} + \frac{\lambda}{2}\mathbf{a}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{a} \tag{6.5}$$

where $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$. We now define the *Gram* matrix $\mathbf{K} = \boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}}$, which is an $N \times N$ symmetric matrix with elements

$$K_{nm} = \boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_m) = k(\mathbf{x}_n, \mathbf{x}_m) \tag{6.6}$$

where we have introduced the *kernel function* $k(\mathbf{x}, \mathbf{x}')$ defined by (6.1). In terms of the Gram matrix, the sum-of-squares error function can be written as

$$J(\mathbf{a}) = \frac{1}{2}\mathbf{a}^{\mathrm{T}}\mathbf{K}\mathbf{K}\mathbf{a} - \mathbf{a}^{\mathrm{T}}\mathbf{K}\mathbf{t} + \frac{1}{2}\mathbf{t}^{\mathrm{T}}\mathbf{t} + \frac{\lambda}{2}\mathbf{a}^{\mathrm{T}}\mathbf{K}\mathbf{a}. \tag{6.7}$$

Setting the gradient of $J(\mathbf{a})$ with respect to $\mathbf{a}$ to zero, we obtain the following solution

$$\mathbf{a} = (\mathbf{K} + \lambda\mathbf{I}_N)^{-1}\mathbf{t}. \tag{6.8}$$

If we substitute this back into the linear regression model, we obtain the following prediction for a new input $\mathbf{x}$

$$y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}) = \mathbf{a}^{\mathrm{T}}\boldsymbol{\Phi}\boldsymbol{\phi}(\mathbf{x}) = \mathbf{k}(\mathbf{x})^{\mathrm{T}}\left(\mathbf{K} + \lambda\mathbf{I}_N\right)^{-1}\mathbf{t} \tag{6.9}$$

where we have defined the vector $\mathbf{k}(\mathbf{x})$ with elements $k_n(\mathbf{x}) = k(\mathbf{x}_n, \mathbf{x})$. Thus we see that the dual formulation allows the solution to the least-squares problem to be expressed entirely in terms of the kernel function $k(\mathbf{x}, \mathbf{x}')$. This is known as a dual formulation because, by noting that the solution for $\mathbf{a}$ can be expressed as a linear combination of the elements of $\boldsymbol{\phi}(\mathbf{x})$, we recover the original formulation in terms of the parameter vector $\mathbf{w}$. Note that the prediction at $\mathbf{x}$ is given by a linear combination of the target values from the training set. In fact, we have already obtained this result, using a slightly different notation, in Section 3.3.3.

Exercise 6.1

In the dual formulation, we determine the parameter vector $\mathbf{a}$ by inverting an $N \times N$ matrix, whereas in the original parameter space formulation we had to invert an $M \times M$ matrix in order to determine $\mathbf{w}$. Because N is typically much larger than M, the dual formulation does not seem to be particularly useful. However, the advantage of the dual formulation, as we shall see, is that it is expressed entirely in terms of the kernel function $k(\mathbf{x}, \mathbf{x}')$. We can therefore work directly in terms of kernels and avoid the explicit introduction of the feature vector $\boldsymbol{\phi}(\mathbf{x})$, which allows us implicitly to use feature spaces of high, even infinite, dimensionality.

The existence of a dual representation based on the Gram matrix is a property of many linear models, including the perceptron. In Section 6.4, we will develop a duality between probabilistic linear models for regression and the technique of Gaussian processes. Duality will also play an important role when we discuss support vector machines in Chapter 7.

Exercise 6.2

6.2. Constructing Kernels

In order to exploit kernel substitution, we need to be able to construct valid kernel functions. One approach is to choose a feature space mapping $\boldsymbol{\phi}(\mathbf{x})$ and then use this to find the corresponding kernel, as is illustrated in Figure 6.1. Here the kernel function is defined for a one-dimensional input space by

$$k(x, x') = \boldsymbol{\phi}(x)^{\mathrm{T}}\boldsymbol{\phi}(x') = \sum_{i=1}^{M} \phi_i(x)\phi_i(x') \tag{6.10}$$

where $\phi_i(x)$ are the basis functions.

An alternative approach is to construct kernel functions directly. In this case, we must ensure that the function we choose is a valid kernel, in other words that it corresponds to a scalar product in some (perhaps infinite dimensional) feature space. As a simple example, consider a kernel function given by

$$k(\mathbf{x}, \mathbf{z}) = \left(\mathbf{x}^{\mathrm{T}}\mathbf{z}\right)^2. \tag{6.11}$$

Figure 6.1 Illustration of the construction of kernel functions starting from a corresponding set of basis functions. In each column the lower plot shows the kernel function $k(x, x')$ defined by (6.10) plotted as a function of x for $x' = 0$, while the upper plot shows the corresponding basis functions given by polynomials (left column), 'Gaussians' (centre column), and logistic sigmoids (right column).

If we take the particular case of a two-dimensional input space $\mathbf{x} = (x_1, x_2)$ we can expand out the terms and thereby identify the corresponding nonlinear feature mapping

$$\begin{aligned} k(\mathbf{x}, \mathbf{z}) &= \left(\mathbf{x}^{\mathrm{T}}\mathbf{z}\right)^2 = (x_1 z_1 + x_2 z_2)^2 \\ &= x_1^2 z_1^2 + 2x_1 z_1 x_2 z_2 + x_2^2 z_2^2 \\ &= (x_1^2, \sqrt{2}x_1 x_2, x_2^2)(z_1^2, \sqrt{2}z_1 z_2, z_2^2)^{\mathrm{T}} \\ &= \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{z}). \end{aligned} \tag{6.12}$$

We see that the feature mapping takes the form $\boldsymbol{\phi}(\mathbf{x}) = (x_1^2, \sqrt{2}x_1 x_2, x_2^2)^{\mathrm{T}}$ and therefore comprises all possible second order terms, with a specific weighting between them.

More generally, however, we need a simple way to test whether a function constitutes a valid kernel without having to construct the function $\boldsymbol{\phi}(\mathbf{x})$ explicitly. A necessary and sufficient condition for a function $k(\mathbf{x}, \mathbf{x}')$ to be a valid kernel (Shawe-Taylor and Cristianini, 2004) is that the Gram matrix $\mathbf{K}$, whose elements are given by $k(\mathbf{x}_n, \mathbf{x}_m)$, should be positive semidefinite for all possible choices of the set $\{\mathbf{x}_n\}$. Note that a positive semidefinite matrix is not the same thing as a matrix whose elements are nonnegative.

Appendix C

One powerful technique for constructing new kernels is to build them out of simpler kernels as building blocks. This can be done using the following properties:

Techniques for Constructing New Kernels.

Given valid kernels $k_1(\mathbf{x}, \mathbf{x}')$ and $k_2(\mathbf{x}, \mathbf{x}')$, the following new kernels will also be valid:

$$
\begin{aligned}
k(\mathbf{x}, \mathbf{x}') &= c k_1(\mathbf{x}, \mathbf{x}') && (6.13)\\
k(\mathbf{x}, \mathbf{x}') &= f(\mathbf{x}) k_1(\mathbf{x}, \mathbf{x}') f(\mathbf{x}') && (6.14)\\
k(\mathbf{x}, \mathbf{x}') &= q\left(k_1(\mathbf{x}, \mathbf{x}')\right) && (6.15)\\
k(\mathbf{x}, \mathbf{x}') &= \exp\left(k_1(\mathbf{x}, \mathbf{x}')\right) && (6.16)\\
k(\mathbf{x}, \mathbf{x}') &= k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}') && (6.17)\\
k(\mathbf{x}, \mathbf{x}') &= k_1(\mathbf{x}, \mathbf{x}') k_2(\mathbf{x}, \mathbf{x}') && (6.18)\\
k(\mathbf{x}, \mathbf{x}') &= k_3\left(\boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{x}')\right) && (6.19)\\
k(\mathbf{x}, \mathbf{x}') &= \mathbf{x}^{\mathrm{T}} \mathbf{A} \mathbf{x}' && (6.20)\\
k(\mathbf{x}, \mathbf{x}') &= k_a(\mathbf{x}_a, \mathbf{x}_a') + k_b(\mathbf{x}_b, \mathbf{x}_b') && (6.21)\\
k(\mathbf{x}, \mathbf{x}') &= k_a(\mathbf{x}_a, \mathbf{x}_a') k_b(\mathbf{x}_b, \mathbf{x}_b') && (6.22)
\end{aligned}
$$

where $c > 0$ is a constant, $f(\cdot)$ is any function, $q(\cdot)$ is a polynomial with nonnegative coefficients, $\boldsymbol{\phi}(\mathbf{x})$ is a function from $\mathbf{x}$ to $\mathbb{R}^M$, $k_3(\cdot, \cdot)$ is a valid kernel in $\mathbb{R}^M$, $\mathbf{A}$ is a symmetric positive semidefinite matrix, $\mathbf{x}_a$ and $\mathbf{x}_b$ are variables (not necessarily disjoint) with $\mathbf{x} = (\mathbf{x}_a, \mathbf{x}_b)$, and k_a and k_b are valid kernel functions over their respective spaces.

Equipped with these properties, we can now embark on the construction of more complex kernels appropriate to specific applications. We require that the kernel $k(\mathbf{x}, \mathbf{x}')$ be symmetric and positive semidefinite and that it expresses the appropriate form of similarity between $\mathbf{x}$ and $\mathbf{x}'$ according to the intended application. Here we consider a few common examples of kernel functions. For a more extensive discussion of 'kernel engineering', see Shawe-Taylor and Cristianini (2004).

We saw that the simple polynomial kernel $k(\mathbf{x}, \mathbf{x}') = \left(\mathbf{x}^{\mathrm{T}} \mathbf{x}'\right)^2$ contains only terms of degree two. If we consider the slightly generalized kernel $k(\mathbf{x}, \mathbf{x}') = \left(\mathbf{x}^{\mathrm{T}} \mathbf{x}' + c\right)^2$ with $c > 0$, then the corresponding feature mapping $\boldsymbol{\phi}(\mathbf{x})$ contains constant and linear terms as well as terms of order two. Similarly, $k(\mathbf{x}, \mathbf{x}') = \left(\mathbf{x}^{\mathrm{T}} \mathbf{x}'\right)^M$ contains all monomials of order M. For instance, if $\mathbf{x}$ and $\mathbf{x}'$ are two images, then the kernel represents a particular weighted sum of all possible products of M pixels in the first image with M pixels in the second image. This can similarly be generalized to include all terms up to degree M by considering $k(\mathbf{x}, \mathbf{x}') = \left(\mathbf{x}^{\mathrm{T}} \mathbf{x}' + c\right)^M$ with $c > 0$. Using the results (6.17) and (6.18) for combining kernels we see that these will all be valid kernel functions.

Another commonly used kernel takes the form

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\|\mathbf{x} - \mathbf{x}'\|^2 / 2\sigma^2\right) \tag{6.23}$$

and is often called a 'Gaussian' kernel. Note, however, that in this context it is not interpreted as a probability density, and hence the normalization coefficient is

omitted. We can see that this is a valid kernel by expanding the square

$$\|\mathbf{x} - \mathbf{x}'\|^2 = \mathbf{x}^{\mathrm{T}}\mathbf{x} + (\mathbf{x}')^{\mathrm{T}}\mathbf{x}' - 2\mathbf{x}^{\mathrm{T}}\mathbf{x}' \tag{6.24}$$

to give

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\mathbf{x}^{\mathrm{T}}\mathbf{x}/2\sigma^2\right) \exp\left(\mathbf{x}^{\mathrm{T}}\mathbf{x}'/\sigma^2\right) \exp\left(-(\mathbf{x}')^{\mathrm{T}}\mathbf{x}'/2\sigma^2\right) \tag{6.25}$$

and then making use of (6.14) and (6.16), together with the validity of the linear kernel $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^{\mathrm{T}}\mathbf{x}'$. Note that the feature vector that corresponds to the Gaussian kernel has infinite dimensionality.

Exercise 6.11

The Gaussian kernel is not restricted to the use of Euclidean distance. If we use kernel substitution in (6.24) to replace $\mathbf{x}^{\mathrm{T}}\mathbf{x}'$ with a nonlinear kernel $\kappa(\mathbf{x}, \mathbf{x}')$, we obtain

$$k(\mathbf{x}, \mathbf{x}') = \exp\left\{-\frac{1}{2\sigma^2}\left(\kappa(\mathbf{x}, \mathbf{x}) + \kappa(\mathbf{x}', \mathbf{x}') - 2\kappa(\mathbf{x}, \mathbf{x}')\right)\right\}. \tag{6.26}$$

An important contribution to arise from the kernel viewpoint has been the extension to inputs that are symbolic, rather than simply vectors of real numbers. Kernel functions can be defined over objects as diverse as graphs, sets, strings, and text documents. Consider, for instance, a fixed set and define a nonvectorial space consisting of all possible subsets of this set. If A_1 and A_2 are two such subsets then one simple choice of kernel would be

$$k(A_1, A_2) = 2^{|A_1 \cap A_2|} \tag{6.27}$$

where $A_1 \cap A_2$ denotes the intersection of sets A_1 and A_2, and $|A|$ denotes the number of subsets in A. This is a valid kernel function because it can be shown to correspond to an inner product in a feature space.

Exercise 6.12

One powerful approach to the construction of kernels starts from a probabilistic generative model (Haussler, 1999), which allows us to apply generative models in a discriminative setting. Generative models can deal naturally with missing data and in the case of hidden Markov models can handle sequences of varying length. By contrast, discriminative models generally give better performance on discriminative tasks than generative models. It is therefore of some interest to combine these two approaches (Lasserre *et al.*, 2006). One way to combine them is to use a generative model to define a kernel, and then use this kernel in a discriminative approach.

Given a generative model $p(\mathbf{x})$ we can define a kernel by

$$k(\mathbf{x}, \mathbf{x}') = p(\mathbf{x})p(\mathbf{x}'). \tag{6.28}$$

This is clearly a valid kernel function because we can interpret it as an inner product in the one-dimensional feature space defined by the mapping $p(\mathbf{x})$. It says that two inputs $\mathbf{x}$ and $\mathbf{x}'$ are similar if they both have high probabilities. We can use (6.13) and (6.17) to extend this class of kernels by considering sums over products of different probability distributions, with positive weighting coefficients $p(i)$, of the form

$$k(\mathbf{x}, \mathbf{x}') = \sum_i p(\mathbf{x}|i)p(\mathbf{x}'|i)p(i). \tag{6.29}$$

This is equivalent, up to an overall multiplicative constant, to a mixture distribution in which the components factorize, with the index i playing the role of a 'latent' variable. Two inputs $\mathbf{x}$ and $\mathbf{x}'$ will give a large value for the kernel function, and hence appear similar, if they have significant probability under a range of different components. Taking the limit of an infinite sum, we can also consider kernels of the form

Section 9.2

$$k(\mathbf{x}, \mathbf{x}') = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{x}'|\mathbf{z})p(\mathbf{z})\,\mathrm{d}\mathbf{z} \tag{6.30}$$

where $\mathbf{z}$ is a continuous latent variable.

Now suppose that our data consists of ordered sequences of length L so that an observation is given by $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_L\}$. A popular generative model for sequences is the hidden Markov model, which expresses the distribution $p(\mathbf{X})$ as a marginalization over a corresponding sequence of hidden states $\mathbf{Z} = \{\mathbf{z}_1, \ldots, \mathbf{z}_L\}$. We can use this approach to define a kernel function measuring the similarity of two sequences $\mathbf{X}$ and $\mathbf{X}'$ by extending the mixture representation (6.29) to give

Section 13.2

$$k(\mathbf{X}, \mathbf{X}') = \sum_{\mathbf{Z}} p(\mathbf{X}|\mathbf{Z})p(\mathbf{X}'|\mathbf{Z})p(\mathbf{Z}) \tag{6.31}$$

so that both observed sequences are generated by the same hidden sequence $\mathbf{Z}$. This model can easily be extended to allow sequences of differing length to be compared.

An alternative technique for using generative models to define kernel functions is known as the *Fisher kernel* (Jaakkola and Haussler, 1999). Consider a parametric generative model $p(\mathbf{x}|\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ denotes the vector of parameters. The goal is to find a kernel that measures the similarity of two input vectors $\mathbf{x}$ and $\mathbf{x}'$ induced by the generative model. Jaakkola and Haussler (1999) consider the gradient with respect to $\boldsymbol{\theta}$, which defines a vector in a 'feature' space having the same dimensionality as $\boldsymbol{\theta}$. In particular, they consider the *Fisher score*

$$\mathbf{g}(\boldsymbol{\theta}, \mathbf{x}) = \nabla_{\boldsymbol{\theta}} \ln p(\mathbf{x}|\boldsymbol{\theta}) \tag{6.32}$$

from which the Fisher kernel is defined by

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{g}(\boldsymbol{\theta}, \mathbf{x})^{\mathrm{T}}\mathbf{F}^{-1}\mathbf{g}(\boldsymbol{\theta}, \mathbf{x}'). \tag{6.33}$$

Here $\mathbf{F}$ is the *Fisher information matrix*, given by

$$\mathbf{F} = \mathbb{E}_{\mathbf{x}}\left[\mathbf{g}(\boldsymbol{\theta}, \mathbf{x})\mathbf{g}(\boldsymbol{\theta}, \mathbf{x})^{\mathrm{T}}\right] \tag{6.34}$$

where the expectation is with respect to $\mathbf{x}$ under the distribution $p(\mathbf{x}|\boldsymbol{\theta})$. This can be motivated from the perspective of *information geometry* (Amari, 1998), which considers the differential geometry of the space of model parameters. Here we simply note that the presence of the Fisher information matrix causes this kernel to be invariant under a nonlinear re-parameterization of the density model $\boldsymbol{\theta} \to \boldsymbol{\psi}(\boldsymbol{\theta})$.

Exercise 6.13

In practice, it is often infeasible to evaluate the Fisher information matrix. One approach is simply to replace the expectation in the definition of the Fisher information with the sample average, giving

$$\mathbf{F} \simeq \frac{1}{N}\sum_{n=1}^{N} \mathbf{g}(\boldsymbol{\theta}, \mathbf{x}_n)\mathbf{g}(\boldsymbol{\theta}, \mathbf{x}_n)^{\mathrm{T}}. \tag{6.35}$$

Section 12.1.3

This is the covariance matrix of the Fisher scores, and so the Fisher kernel corresponds to a whitening of these scores. More simply, we can just omit the Fisher information matrix altogether and use the noninvariant kernel

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{g}(\boldsymbol{\theta}, \mathbf{x})^{\mathrm{T}} \mathbf{g}(\boldsymbol{\theta}, \mathbf{x}'). \tag{6.36}$$

An application of Fisher kernels to document retrieval is given by Hofmann (2000).

A final example of a kernel function is the sigmoidal kernel given by

$$k(\mathbf{x}, \mathbf{x}') = \tanh\left(a\mathbf{x}^{\mathrm{T}}\mathbf{x}' + b\right) \tag{6.37}$$

whose Gram matrix in general is not positive semidefinite. This form of kernel has, however, been used in practice (Vapnik, 1995), possibly because it gives kernel expansions such as the support vector machine a superficial resemblance to neural network models. As we shall see, in the limit of an infinite number of basis functions, a Bayesian neural network with an appropriate prior reduces to a Gaussian process, thereby providing a deeper link between neural networks and kernel methods.

Section 6.4.7

6.3. Radial Basis Function Networks

In Chapter 3, we discussed regression models based on linear combinations of fixed basis functions, although we did not discuss in detail what form those basis functions might take. One choice that has been widely used is that of *radial basis functions*, which have the property that each basis function depends only on the radial distance (typically Euclidean) from a centre $\boldsymbol{\mu}_j$, so that $\phi_j(\mathbf{x}) = h(\|\mathbf{x} - \boldsymbol{\mu}_j\|)$.

Historically, radial basis functions were introduced for the purpose of exact function interpolation (Powell, 1987). Given a set of input vectors $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ along with corresponding target values $\{t_1, \ldots, t_N\}$, the goal is to find a smooth function $f(\mathbf{x})$ that fits every target value exactly, so that $f(\mathbf{x}_n) = t_n$ for $n = 1, \ldots, N$. This is achieved by expressing $f(\mathbf{x})$ as a linear combination of radial basis functions, one centred on every data point

$$f(\mathbf{x}) = \sum_{n=1}^{N} w_n h(\|\mathbf{x} - \mathbf{x}_n\|). \tag{6.38}$$

The values of the coefficients $\{w_n\}$ are found by least squares, and because there are the same number of coefficients as there are constraints, the result is a function that fits every target value exactly. In pattern recognition applications, however, the target values are generally noisy, and exact interpolation is undesirable because this corresponds to an over-fitted solution.

Expansions in radial basis functions also arise from regularization theory (Poggio and Girosi, 1990; Bishop, 1995a). For a sum-of-squares error function with a regularizer defined in terms of a differential operator, the optimal solution is given by an expansion in the *Green's functions* of the operator (which are analogous to the eigenvectors of a discrete matrix), again with one basis function centred on each data

point. If the differential operator is isotropic then the Green's functions depend only on the radial distance from the corresponding data point. Due to the presence of the regularizer, the solution no longer interpolates the training data exactly.

Another motivation for radial basis functions comes from a consideration of the interpolation problem when the input (rather than the target) variables are noisy (Webb, 1994; Bishop, 1995a). If the noise on the input variable $\mathbf{x}$ is described by a variable $\boldsymbol{\xi}$ having a distribution $\nu(\boldsymbol{\xi})$, then the sum-of-squares error function becomes

$$E = \frac{1}{2}\sum_{n=1}^{N}\int \{y(\mathbf{x}_n + \boldsymbol{\xi}) - t_n\}^2 \, \nu(\boldsymbol{\xi}) \, \mathrm{d}\boldsymbol{\xi}. \tag{6.39}$$

Appendix D
Exercise 6.17

Using the calculus of variations, we can optimize with respect to the function $f(\mathbf{x})$ to give

$$y(\mathbf{x}_n) = \sum_{n=1}^{N} t_n h(\mathbf{x} - \mathbf{x}_n) \tag{6.40}$$

where the basis functions are given by

$$h(\mathbf{x} - \mathbf{x}_n) = \frac{\nu(\mathbf{x} - \mathbf{x}_n)}{\displaystyle\sum_{n=1}^{N} \nu(\mathbf{x} - \mathbf{x}_n)}. \tag{6.41}$$

We see that there is one basis function centred on every data point. This is known as the *Nadaraya-Watson* model and will be derived again from a different perspective in Section 6.3.1. If the noise distribution $\nu(\boldsymbol{\xi})$ is isotropic, so that it is a function only of $\|\boldsymbol{\xi}\|$, then the basis functions will be radial.

Note that the basis functions (6.41) are normalized, so that $\sum_n h(\mathbf{x} - \mathbf{x}_n) = 1$ for any value of $\mathbf{x}$. The effect of such normalization is shown in Figure 6.2. Normalization is sometimes used in practice as it avoids having regions of input space where all of the basis functions take small values, which would necessarily lead to predictions in such regions that are either small or controlled purely by the bias parameter.

Another situation in which expansions in normalized radial basis functions arise is in the application of kernel density estimation to the problem of regression, as we shall discuss in Section 6.3.1.

Because there is one basis function associated with every data point, the corresponding model can be computationally costly to evaluate when making predictions for new data points. Models have therefore been proposed (Broomhead and Lowe, 1988; Moody and Darken, 1989; Poggio and Girosi, 1990), which retain the expansion in radial basis functions but where the number M of basis functions is smaller than the number N of data points. Typically, the number of basis functions, and the locations $\boldsymbol{\mu}_i$ of their centres, are determined based on the input data $\{\mathbf{x}_n\}$ alone. The basis functions are then kept fixed and the coefficients $\{w_i\}$ are determined by least squares by solving the usual set of linear equations, as discussed in Section 3.1.1.

Figure 6.2 Plot of a set of Gaussian basis functions on the left, together with the corresponding normalized basis functions on the right.

One of the simplest ways of choosing basis function centres is to use a randomly chosen subset of the data points. A more systematic approach is called *orthogonal least squares* (Chen *et al.*, 1991). This is a sequential selection process in which at each step the next data point to be chosen as a basis function centre corresponds to the one that gives the greatest reduction in the sum-of-squares error. Values for the expansion coefficients are determined as part of the algorithm. Clustering algorithms such as K-means have also been used, which give a set of basis function centres that no longer coincide with training data points.

Section 9.1

6.3.1 Nadaraya-Watson model

In Section 3.3.3, we saw that the prediction of a linear regression model for a new input $\mathbf{x}$ takes the form of a linear combination of the training set target values with coefficients given by the 'equivalent kernel' (3.62) where the equivalent kernel satisfies the summation constraint (3.64).

We can motivate the kernel regression model (3.61) from a different perspective, starting with kernel density estimation. Suppose we have a training set $\{\mathbf{x}_n, t_n\}$ and we use a Parzen density estimator to model the joint distribution $p(\mathbf{x}, t)$, so that

Section 2.5.1

$$p(\mathbf{x}, t) = \frac{1}{N} \sum_{n=1}^{N} f(\mathbf{x} - \mathbf{x}_n, t - t_n) \tag{6.42}$$

where $f(\mathbf{x}, t)$ is the component density function, and there is one such component centred on each data point. We now find an expression for the regression function $y(\mathbf{x})$, corresponding to the conditional average of the target variable conditioned on

the input variable, which is given by

$$\begin{aligned} y(\mathbf{x}) &= \mathbb{E}[t|\mathbf{x}] = \int_{-\infty}^{\infty} tp(t|\mathbf{x})\,\mathrm{d}t \\ &= \frac{\displaystyle\int tp(\mathbf{x},t)\,\mathrm{d}t}{\displaystyle\int p(\mathbf{x},t)\,\mathrm{d}t} \\ &= \frac{\displaystyle\sum_n \int tf(\mathbf{x}-\mathbf{x}_n, t-t_n)\,\mathrm{d}t}{\displaystyle\sum_m \int f(\mathbf{x}-\mathbf{x}_m, t-t_m)\,\mathrm{d}t}. \end{aligned} \tag{6.43}$$

We now assume for simplicity that the component density functions have zero mean so that

$$\int_{-\infty}^{\infty} f(\mathbf{x},t)t\,\mathrm{d}t = 0 \tag{6.44}$$

for all values of $\mathbf{x}$. Using a simple change of variable, we then obtain

$$\begin{aligned} y(\mathbf{x}) &= \frac{\displaystyle\sum_n g(\mathbf{x}-\mathbf{x}_n)t_n}{\displaystyle\sum_m g(\mathbf{x}-\mathbf{x}_m)} \\ &= \sum_n k(\mathbf{x},\mathbf{x}_n)t_n \end{aligned} \tag{6.45}$$

where $n, m = 1, \ldots, N$ and the kernel function $k(\mathbf{x}, \mathbf{x}_n)$ is given by

$$k(\mathbf{x},\mathbf{x}_n) = \frac{g(\mathbf{x}-\mathbf{x}_n)}{\displaystyle\sum_m g(\mathbf{x}-\mathbf{x}_m)} \tag{6.46}$$

and we have defined

$$g(\mathbf{x}) = \int_{-\infty}^{\infty} f(\mathbf{x},t)\,\mathrm{d}t. \tag{6.47}$$

The result (6.45) is known as the *Nadaraya-Watson* model, or *kernel regression* (Nadaraya, 1964; Watson, 1964). For a localized kernel function, it has the property of giving more weight to the data points $\mathbf{x}_n$ that are close to $\mathbf{x}$. Note that the kernel (6.46) satisfies the summation constraint

$$\sum_{n=1}^{N} k(\mathbf{x},\mathbf{x}_n) = 1.$$

Figure 6.3 Illustration of the Nadaraya-Watson kernel regression model using isotropic Gaussian kernels, for the sinusoidal data set. The original sine function is shown by the green curve, the data points are shown in blue, and each is the centre of an isotropic Gaussian kernel. The resulting regression function, given by the conditional mean, is shown by the red line, along with the two-standard-deviation region for the conditional distribution $p(t|x)$ shown by the red shading. The blue ellipse around each data point shows one standard deviation contour for the corresponding kernel. These appear noncircular due to the different scales on the horizontal and vertical axes.

In fact, this model defines not only a conditional expectation but also a full conditional distribution given by

$$p(t|\mathbf{x}) = \frac{p(t,\mathbf{x})}{\displaystyle\int p(t,\mathbf{x})\,\mathrm{d}t} = \frac{\displaystyle\sum_n f(\mathbf{x}-\mathbf{x}_n, t-t_n)}{\displaystyle\sum_m \int f(\mathbf{x}-\mathbf{x}_m, t-t_m)\,\mathrm{d}t} \tag{6.48}$$

from which other expectations can be evaluated.

As an illustration we consider the case of a single input variable x in which $f(x,t)$ is given by a zero-mean isotropic Gaussian over the variable $\mathbf{z} = (x,t)$ with variance σ^2. The corresponding conditional distribution (6.48) is given by a Gaussian mixture, and is shown, together with the conditional mean, for the sinusoidal synthetic data set in Figure 6.3.

Exercise 6.18

An obvious extension of this model is to allow for more flexible forms of Gaussian components, for instance having different variance parameters for the input and target variables. More generally, we could model the joint distribution $p(t,\mathbf{x})$ using a Gaussian mixture model, trained using techniques discussed in Chapter 9 (Ghahramani and Jordan, 1994), and then find the corresponding conditional distribution $p(t|\mathbf{x})$. In this latter case we no longer have a representation in terms of kernel functions evaluated at the training set data points. However, the number of components in the mixture model can be smaller than the number of training set points, resulting in a model that is faster to evaluate for test data points. We have thereby accepted an increased computational cost during the training phase in order to have a model that is faster at making predictions.

6.4. Gaussian Processes

In Section 6.1, we introduced kernels by applying the concept of duality to a non-probabilistic model for regression. Here we extend the role of kernels to probabilis-

tic discriminative models, leading to the framework of Gaussian processes. We shall thereby see how kernels arise naturally in a Bayesian setting.

In Chapter 3, we considered linear regression models of the form $y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x})$ in which $\mathbf{w}$ is a vector of parameters and $\boldsymbol{\phi}(\mathbf{x})$ is a vector of fixed nonlinear basis functions that depend on the input vector $\mathbf{x}$. We showed that a prior distribution over $\mathbf{w}$ induced a corresponding prior distribution over functions $y(\mathbf{x}, \mathbf{w})$. Given a training data set, we then evaluated the posterior distribution over $\mathbf{w}$ and thereby obtained the corresponding posterior distribution over regression functions, which in turn (with the addition of noise) implies a predictive distribution $p(t|\mathbf{x})$ for new input vectors $\mathbf{x}$.

In the Gaussian process viewpoint, we dispense with the parametric model and instead define a prior probability distribution over functions directly. At first sight, it might seem difficult to work with a distribution over the uncountably infinite space of functions. However, as we shall see, for a finite training set we only need to consider the values of the function at the discrete set of input values $\mathbf{x}_n$ corresponding to the training set and test set data points, and so in practice we can work in a finite space.

Models equivalent to Gaussian processes have been widely studied in many different fields. For instance, in the geostatistics literature Gaussian process regression is known as *kriging* (Cressie, 1993). Similarly, ARMA (autoregressive moving average) models, Kalman filters, and radial basis function networks can all be viewed as forms of Gaussian process models. Reviews of Gaussian processes from a machine learning perspective can be found in MacKay (1998), Williams (1999), and MacKay (2003), and a comparison of Gaussian process models with alternative approaches is given in Rasmussen (1996). See also Rasmussen and Williams (2006) for a recent textbook on Gaussian processes.

6.4.1 Linear regression revisited

In order to motivate the Gaussian process viewpoint, let us return to the linear regression example and re-derive the predictive distribution by working in terms of distributions over functions $y(\mathbf{x}, \mathbf{w})$. This will provide a specific example of a Gaussian process.

Consider a model defined in terms of a linear combination of M fixed basis functions given by the elements of the vector $\boldsymbol{\phi}(\mathbf{x})$ so that

$$y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}) \tag{6.49}$$

where $\mathbf{x}$ is the input vector and $\mathbf{w}$ is the M-dimensional weight vector. Now consider a prior distribution over $\mathbf{w}$ given by an isotropic Gaussian of the form

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I}) \tag{6.50}$$

governed by the hyperparameter α, which represents the precision (inverse variance) of the distribution. For any given value of $\mathbf{w}$, the definition (6.49) defines a particular function of $\mathbf{x}$. The probability distribution over $\mathbf{w}$ defined by (6.50) therefore induces a probability distribution over functions $y(\mathbf{x})$. In practice, we wish to evaluate this function at specific values of $\mathbf{x}$, for example at the training data points

$\mathbf{x}_1, \ldots, \mathbf{x}_N$. We are therefore interested in the joint distribution of the function values $y(\mathbf{x}_1), \ldots, y(\mathbf{x}_N)$, which we denote by the vector $\mathbf{y}$ with elements $y_n = y(\mathbf{x}_n)$ for $n = 1, \ldots, N$. From (6.49), this vector is given by

$$\mathbf{y} = \boldsymbol{\Phi}\mathbf{w} \tag{6.51}$$

where $\boldsymbol{\Phi}$ is the design matrix with elements $\Phi_{nk} = \phi_k(\mathbf{x}_n)$. We can find the probability distribution of $\mathbf{y}$ as follows. First of all we note that $\mathbf{y}$ is a linear combination of Gaussian distributed variables given by the elements of $\mathbf{w}$ and hence is itself Gaussian. We therefore need only to find its mean and covariance, which are given from (6.50) by

Exercise 2.31

$$\mathbb{E}[\mathbf{y}] = \boldsymbol{\Phi}\mathbb{E}[\mathbf{w}] = \mathbf{0} \tag{6.52}$$

$$\mathrm{cov}[\mathbf{y}] = \mathbb{E}\left[\mathbf{y}\mathbf{y}^{\mathrm{T}}\right] = \boldsymbol{\Phi}\mathbb{E}\left[\mathbf{w}\mathbf{w}^{\mathrm{T}}\right]\boldsymbol{\Phi}^{\mathrm{T}} = \frac{1}{\alpha}\boldsymbol{\Phi}\boldsymbol{\Phi}^{\mathrm{T}} = \mathbf{K} \tag{6.53}$$

where $\mathbf{K}$ is the Gram matrix with elements

$$K_{nm} = k(\mathbf{x}_n, \mathbf{x}_m) = \frac{1}{\alpha}\boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_m) \tag{6.54}$$

and $k(\mathbf{x}, \mathbf{x}')$ is the kernel function.

This model provides us with a particular example of a Gaussian process. In general, a Gaussian process is defined as a probability distribution over functions $y(\mathbf{x})$ such that the set of values of $y(\mathbf{x})$ evaluated at an arbitrary set of points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ jointly have a Gaussian distribution. In cases where the input vector $\mathbf{x}$ is two dimensional, this may also be known as a *Gaussian random field*. More generally, a *stochastic process* $y(\mathbf{x})$ is specified by giving the joint probability distribution for any finite set of values $y(\mathbf{x}_1), \ldots, y(\mathbf{x}_N)$ in a consistent manner.

A key point about Gaussian stochastic processes is that the joint distribution over N variables $y_1, \ldots, y_N$ is specified completely by the second-order statistics, namely the mean and the covariance. In most applications, we will not have any prior knowledge about the mean of $y(\mathbf{x})$ and so by symmetry we take it to be zero. This is equivalent to choosing the mean of the prior over weight values $p(\mathbf{w}|\alpha)$ to be zero in the basis function viewpoint. The specification of the Gaussian process is then completed by giving the covariance of $y(\mathbf{x})$ evaluated at any two values of $\mathbf{x}$, which is given by the kernel function

$$\mathbb{E}\left[y(\mathbf{x}_n)y(\mathbf{x}_m)\right] = k(\mathbf{x}_n, \mathbf{x}_m). \tag{6.55}$$

For the specific case of a Gaussian process defined by the linear regression model (6.49) with a weight prior (6.50), the kernel function is given by (6.54).

We can also define the kernel function directly, rather than indirectly through a choice of basis function. Figure 6.4 shows samples of functions drawn from Gaussian processes for two different choices of kernel function. The first of these is a 'Gaussian' kernel of the form (6.23), and the second is the exponential kernel given by

$$k(x, x') = \exp\left(-\theta\left|x - x'\right|\right) \tag{6.56}$$

which corresponds to the *Ornstein-Uhlenbeck process* originally introduced by Uhlenbeck and Ornstein (1930) to describe Brownian motion.

Figure 6.4 Samples from Gaussian processes for a 'Gaussian' kernel (left) and an exponential kernel (right).

6.4.2 Gaussian processes for regression

In order to apply Gaussian process models to the problem of regression, we need to take account of the noise on the observed target values, which are given by

$$t_n = y_n + \epsilon_n \tag{6.57}$$

where $y_n = y(\mathbf{x}_n)$, and ϵ_n is a random noise variable whose value is chosen independently for each observation n. Here we shall consider noise processes that have a Gaussian distribution, so that

$$p(t_n|y_n) = \mathcal{N}(t_n|y_n, \beta^{-1}) \tag{6.58}$$

where β is a hyperparameter representing the precision of the noise. Because the noise is independent for each data point, the joint distribution of the target values $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$ conditioned on the values of $\mathbf{y} = (y_1, \ldots, y_N)^{\mathrm{T}}$ is given by an isotropic Gaussian of the form

$$p(\mathbf{t}|\mathbf{y}) = \mathcal{N}(\mathbf{t}|\mathbf{y}, \beta^{-1}\mathbf{I}_N) \tag{6.59}$$

where $\mathbf{I}_N$ denotes the $N \times N$ unit matrix. From the definition of a Gaussian process, the marginal distribution $p(\mathbf{y})$ is given by a Gaussian whose mean is zero and whose covariance is defined by a Gram matrix $\mathbf{K}$ so that

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K}). \tag{6.60}$$

The kernel function that determines $\mathbf{K}$ is typically chosen to express the property that, for points $\mathbf{x}_n$ and $\mathbf{x}_m$ that are similar, the corresponding values $y(\mathbf{x}_n)$ and $y(\mathbf{x}_m)$ will be more strongly correlated than for dissimilar points. Here the notion of similarity will depend on the application.

In order to find the marginal distribution $p(\mathbf{t})$, conditioned on the input values $\mathbf{x}_1, \ldots, \mathbf{x}_N$, we need to integrate over $\mathbf{y}$. This can be done by making use of the results from Section 2.3.3 for the linear-Gaussian model. Using (2.115), we see that the marginal distribution of $\mathbf{t}$ is given by

$$p(\mathbf{t}) = \int p(\mathbf{t}|\mathbf{y})p(\mathbf{y})\,\mathrm{d}\mathbf{y} = \mathcal{N}(\mathbf{t}|\mathbf{0}, \mathbf{C}) \tag{6.61}$$

where the covariance matrix $\mathbf{C}$ has elements

$$C(\mathbf{x}_n, \mathbf{x}_m) = k(\mathbf{x}_n, \mathbf{x}_m) + \beta^{-1}\delta_{nm}. \tag{6.62}$$

This result reflects the fact that the two Gaussian sources of randomness, namely that associated with $y(\mathbf{x})$ and that associated with ϵ, are independent and so their covariances simply add.

One widely used kernel function for Gaussian process regression is given by the exponential of a quadratic form, with the addition of constant and linear terms to give

$$k(\mathbf{x}_n, \mathbf{x}_m) = \theta_0 \exp\left\{-\frac{\theta_1}{2}\|\mathbf{x}_n - \mathbf{x}_m\|^2\right\} + \theta_2 + \theta_3 \mathbf{x}_n^{\mathrm{T}}\mathbf{x}_m. \tag{6.63}$$

Note that the term involving θ_3 corresponds to a parametric model that is a linear function of the input variables. Samples from this prior are plotted for various values of the parameters $\theta_0, \ldots, \theta_3$ in Figure 6.5, and Figure 6.6 shows a set of points sampled from the joint distribution (6.60) along with the corresponding values defined by (6.61).

So far, we have used the Gaussian process viewpoint to build a model of the joint distribution over sets of data points. Our goal in regression, however, is to make predictions of the target variables for new inputs, given a set of training data. Let us suppose that $\mathbf{t}_N = (t_1, \ldots, t_N)^{\mathrm{T}}$, corresponding to input values $\mathbf{x}_1, \ldots, \mathbf{x}_N$, comprise the observed training set, and our goal is to predict the target variable t_{N+1} for a new input vector $\mathbf{x}_{N+1}$. This requires that we evaluate the predictive distribution $p(t_{N+1}|\mathbf{t}_N)$. Note that this distribution is conditioned also on the variables $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and $\mathbf{x}_{N+1}$. However, to keep the notation simple we will not show these conditioning variables explicitly.

To find the conditional distribution $p(t_{N+1}|\mathbf{t})$, we begin by writing down the joint distribution $p(\mathbf{t}_{N+1})$, where $\mathbf{t}_{N+1}$ denotes the vector $(t_1, \ldots, t_N, t_{N+1})^{\mathrm{T}}$. We then apply the results from Section 2.3.1 to obtain the required conditional distribution, as illustrated in Figure 6.7.

From (6.61), the joint distribution over $t_1, \ldots, t_{N+1}$ will be given by

$$p(\mathbf{t}_{N+1}) = \mathcal{N}(\mathbf{t}_{N+1}|\mathbf{0}, \mathbf{C}_{N+1}) \tag{6.64}$$

where $\mathbf{C}_{N+1}$ is an $(N+1) \times (N+1)$ covariance matrix with elements given by (6.62). Because this joint distribution is Gaussian, we can apply the results from Section 2.3.1 to find the conditional Gaussian distribution. To do this, we partition the covariance matrix as follows

$$\mathbf{C}_{N+1} = \begin{pmatrix} \mathbf{C}_N & \mathbf{k} \\ \mathbf{k}^{\mathrm{T}} & c \end{pmatrix} \tag{6.65}$$

where $\mathbf{C}_N$ is the $N \times N$ covariance matrix with elements given by (6.62) for $n, m = 1, \ldots, N$, the vector $\mathbf{k}$ has elements $k(\mathbf{x}_n, \mathbf{x}_{N+1})$ for $n = 1, \ldots, N$, and the scalar

Figure 6.5 Samples from a Gaussian process prior defined by the covariance function (6.63). The title above each plot denotes $(\theta_0, \theta_1, \theta_2, \theta_3)$.

$c = k(\mathbf{x}_{N+1}, \mathbf{x}_{N+1}) + \beta^{-1}$. Using the results (2.81) and (2.82), we see that the conditional distribution $p(t_{N+1}|\mathbf{t})$ is a Gaussian distribution with mean and covariance given by

$$m(\mathbf{x}_{N+1}) = \mathbf{k}^{\mathrm{T}}\mathbf{C}_N^{-1}\mathbf{t} \tag{6.66}$$

$$\sigma^2(\mathbf{x}_{N+1}) = c - \mathbf{k}^{\mathrm{T}}\mathbf{C}_N^{-1}\mathbf{k}. \tag{6.67}$$

These are the key results that define Gaussian process regression. Because the vector $\mathbf{k}$ is a function of the test point input value $\mathbf{x}_{N+1}$, we see that the predictive distribution is a Gaussian whose mean and variance both depend on $\mathbf{x}_{N+1}$. An example of Gaussian process regression is shown in Figure 6.8.

The only restriction on the kernel function is that the covariance matrix given by (6.62) must be positive definite. If λ_i is an eigenvalue of $\mathbf{K}$, then the corresponding eigenvalue of $\mathbf{C}$ will be $\lambda_i + \beta^{-1}$. It is therefore sufficient that the kernel matrix $k(\mathbf{x}_n, \mathbf{x}_m)$ be positive semidefinite for any pair of points $\mathbf{x}_n$ and $\mathbf{x}_m$, so that $\lambda_i \geqslant 0$, because any eigenvalue λ_i that is zero will still give rise to a positive eigenvalue for $\mathbf{C}$ because $\beta > 0$. This is the same restriction on the kernel function discussed earlier, and so we can again exploit all of the techniques in Section 6.2 to construct

Figure 6.6 Illustration of the sampling of data points $\{t_n\}$ from a Gaussian process. The blue curve shows a sample function from the Gaussian process prior over functions, and the red points show the values of y_n obtained by evaluating the function at a set of input values $\{x_n\}$. The corresponding values of $\{t_n\}$, shown in green, are obtained by adding independent Gaussian noise to each of the $\{y_n\}$.

suitable kernels.

Note that the mean (6.66) of the predictive distribution can be written, as a function of $\mathbf{x}_{N+1}$, in the form

$$m(\mathbf{x}_{N+1}) = \sum_{n=1}^{N} a_n k(\mathbf{x}_n, \mathbf{x}_{N+1}) \tag{6.68}$$

where a_n is the nth component of $\mathbf{C}_N^{-1}\mathbf{t}$. Thus, if the kernel function $k(\mathbf{x}_n, \mathbf{x}_m)$ depends only on the distance $\|\mathbf{x}_n - \mathbf{x}_m\|$, then we obtain an expansion in radial basis functions.

The results (6.66) and (6.67) define the predictive distribution for Gaussian process regression with an arbitrary kernel function $k(\mathbf{x}_n, \mathbf{x}_m)$. In the particular case in which the kernel function $k(\mathbf{x}, \mathbf{x}')$ is defined in terms of a finite set of basis functions, we can derive the results obtained previously in Section 3.3.2 for linear regression starting from the Gaussian process viewpoint.

Exercise 6.21

For such models, we can therefore obtain the predictive distribution either by taking a parameter space viewpoint and using the linear regression result or by taking a function space viewpoint and using the Gaussian process result.

The central computational operation in using Gaussian processes will involve the inversion of a matrix of size $N \times N$, for which standard methods require $O(N^3)$ computations. By contrast, in the basis function model we have to invert a matrix $\mathbf{S}_N$ of size $M \times M$, which has $O(M^3)$ computational complexity. Note that for both viewpoints, the matrix inversion must be performed once for the given training set. For each new test point, both methods require a vector-matrix multiply, which has cost $O(N^2)$ in the Gaussian process case and $O(M^2)$ for the linear basis function model. If the number M of basis functions is smaller than the number N of data points, it will be computationally more efficient to work in the basis function

Figure 6.7 Illustration of the mechanism of Gaussian process regression for the case of one training point and one test point, in which the red ellipses show contours of the joint distribution $p(t_1, t_2)$. Here t_1 is the training data point, and conditioning on the value of t_1, corresponding to the vertical blue line, we obtain $p(t_2|t_1)$ shown as a function of t_2 by the green curve.

framework. However, an advantage of a Gaussian processes viewpoint is that we can consider covariance functions that can only be expressed in terms of an infinite number of basis functions.

For large training data sets, however, the direct application of Gaussian process methods can become infeasible, and so a range of approximation schemes have been developed that have better scaling with training set size than the exact approach (Gibbs, 1997; Tresp, 2001; Smola and Bartlett, 2001; Williams and Seeger, 2001; Csató and Opper, 2002; Seeger *et al.*, 2003). Practical issues in the application of Gaussian processes are discussed in Bishop and Nabney (2008).

We have introduced Gaussian process regression for the case of a single target variable. The extension of this formalism to multiple target variables, known as co-kriging (Cressie, 1993), is straightforward. Various other extensions of Gaus-

Exercise 6.23

Figure 6.8 Illustration of Gaussian process regression applied to the sinusoidal data set in Figure A.6 in which the three right-most data points have been omitted. The green curve shows the sinusoidal function from which the data points, shown in blue, are obtained by sampling and addition of Gaussian noise. The red line shows the mean of the Gaussian process predictive distribution, and the shaded region corresponds to plus and minus two standard deviations. Notice how the uncertainty increases in the region to the right of the data points.

sian process regression have also been considered, for purposes such as modelling the distribution over low-dimensional manifolds for unsupervised learning (Bishop *et al.*, 1998a) and the solution of stochastic differential equations (Graepel, 2003).

6.4.3 Learning the hyperparameters

The predictions of a Gaussian process model will depend, in part, on the choice of covariance function. In practice, rather than fixing the covariance function, we may prefer to use a parametric family of functions and then infer the parameter values from the data. These parameters govern such things as the length scale of the correlations and the precision of the noise and correspond to the hyperparameters in a standard parametric model.

Techniques for learning the hyperparameters are based on the evaluation of the likelihood function $p(\mathbf{t}|\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ denotes the hyperparameters of the Gaussian process model. The simplest approach is to make a point estimate of $\boldsymbol{\theta}$ by maximizing the log likelihood function. Because $\boldsymbol{\theta}$ represents a set of hyperparameters for the regression problem, this can be viewed as analogous to the type 2 maximum likelihood procedure for linear regression models. Maximization of the log likelihood can be done using efficient gradient-based optimization algorithms such as conjugate gradients (Fletcher, 1987; Nocedal and Wright, 1999; Bishop and Nabney, 2008).

Section 3.5

The log likelihood function for a Gaussian process regression model is easily evaluated using the standard form for a multivariate Gaussian distribution, giving

$$\ln p(\mathbf{t}|\boldsymbol{\theta}) = -\frac{1}{2}\ln|\mathbf{C}_N| - \frac{1}{2}\mathbf{t}^{\mathrm{T}}\mathbf{C}_N^{-1}\mathbf{t} - \frac{N}{2}\ln(2\pi). \tag{6.69}$$

For nonlinear optimization, we also need the gradient of the log likelihood function with respect to the parameter vector $\boldsymbol{\theta}$. We shall assume that evaluation of the derivatives of $\mathbf{C}_N$ is straightforward, as would be the case for the covariance functions considered in this chapter. Making use of the result (C.21) for the derivative of $\mathbf{C}_N^{-1}$, together with the result (C.22) for the derivative of $\ln|\mathbf{C}_N|$, we obtain

$$\frac{\partial}{\partial\theta_i}\ln p(\mathbf{t}|\boldsymbol{\theta}) = -\frac{1}{2}\mathrm{Tr}\left(\mathbf{C}_N^{-1}\frac{\partial\mathbf{C}_N}{\partial\theta_i}\right) + \frac{1}{2}\mathbf{t}^{\mathrm{T}}\mathbf{C}_N^{-1}\frac{\partial\mathbf{C}_N}{\partial\theta_i}\mathbf{C}_N^{-1}\mathbf{t}. \tag{6.70}$$

Because $\ln p(\mathbf{t}|\boldsymbol{\theta})$ will in general be a nonconvex function, it can have multiple maxima.

It is straightforward to introduce a prior over $\boldsymbol{\theta}$ and to maximize the log posterior using gradient-based methods. In a fully Bayesian treatment, we need to evaluate marginals over $\boldsymbol{\theta}$ weighted by the product of the prior $p(\boldsymbol{\theta})$ and the likelihood function $p(\mathbf{t}|\boldsymbol{\theta})$. In general, however, exact marginalization will be intractable, and we must resort to approximations.

The Gaussian process regression model gives a predictive distribution whose mean and variance are functions of the input vector $\mathbf{x}$. However, we have assumed that the contribution to the predictive variance arising from the additive noise, governed by the parameter β, is a constant. For some problems, known as *heteroscedastic*, the noise variance itself will also depend on $\mathbf{x}$. To model this, we can extend the

Figure 6.9 Samples from the ARD prior for Gaussian processes, in which the kernel function is given by (6.71). The left plot corresponds to $\eta_1 = \eta_2 = 1$, and the right plot corresponds to $\eta_1 = 1$, $\eta_2 = 0.01$.

Gaussian process framework by introducing a second Gaussian process to represent the dependence of β on the input $\mathbf{x}$ (Goldberg *et al.*, 1998). Because β is a variance, and hence nonnegative, we use the Gaussian process to model $\ln \beta(\mathbf{x})$.

6.4.4 Automatic relevance determination

In the previous section, we saw how maximum likelihood could be used to determine a value for the correlation length-scale parameter in a Gaussian process. This technique can usefully be extended by incorporating a separate parameter for each input variable (Rasmussen and Williams, 2006). The result, as we shall see, is that the optimization of these parameters by maximum likelihood allows the relative importance of different inputs to be inferred from the data. This represents an example in the Gaussian process context of *automatic relevance determination*, or *ARD*, which was originally formulated in the framework of neural networks (MacKay, 1994; Neal, 1996). The mechanism by which appropriate inputs are preferred is discussed in Section 7.2.2.

Consider a Gaussian process with a two-dimensional input space $\mathbf{x} = (x_1, x_2)$, having a kernel function of the form

$$k(\mathbf{x}, \mathbf{x}') = \theta_0 \exp\left\{-\frac{1}{2}\sum_{i=1}^{2} \eta_i (x_i - x_i')^2\right\}. \tag{6.71}$$

Samples from the resulting prior over functions $y(\mathbf{x})$ are shown for two different settings of the precision parameters η_i in Figure 6.9. We see that, as a particular parameter η_i becomes small, the function becomes relatively insensitive to the corresponding input variable x_i. By adapting these parameters to a data set using maximum likelihood, it becomes possible to detect input variables that have little effect on the predictive distribution, because the corresponding values of η_i will be small. This can be useful in practice because it allows such inputs to be discarded. ARD is illustrated using a simple synthetic data set having three inputs x_1, x_2 and x_3 (Nabney, 2002) in Figure 6.10. The target variable t, is generated by sampling 100 values of x_1 from a Gaussian, evaluating the function $\sin(2\pi x_1)$, and then adding

Figure 6.10 Illustration of automatic relevance determination in a Gaussian process for a synthetic problem having three inputs x_1, x_2, and x_3, for which the curves show the corresponding values of the hyperparameters η_1 (red), η_2 (green), and η_3 (blue) as a function of the number of iterations when optimizing the marginal likelihood. Details are given in the text. Note the logarithmic scale on the vertical axis.

Gaussian noise. Values of x_2 are given by copying the corresponding values of x_1 and adding noise, and values of x_3 are sampled from an independent Gaussian distribution. Thus x_1 is a good predictor of t, x_2 is a more noisy predictor of t, and x_3 has only chance correlations with t. The marginal likelihood for a Gaussian process with ARD parameters η_1, η_2, η_3 is optimized using the scaled conjugate gradients algorithm. We see from Figure 6.10 that η_1 converges to a relatively large value, η_2 converges to a much smaller value, and η_3 becomes very small indicating that x_3 is irrelevant for predicting t.

The ARD framework is easily incorporated into the exponential-quadratic kernel (6.63) to give the following form of kernel function, which has been found useful for applications of Gaussian processes to a range of regression problems

$$k(\mathbf{x}_n, \mathbf{x}_m) = \theta_0 \exp\left\{-\frac{1}{2}\sum_{i=1}^{D} \eta_i (x_{ni} - x_{mi})^2\right\} + \theta_2 + \theta_3 \sum_{i=1}^{D} x_{ni} x_{mi} \tag{6.72}$$

where D is the dimensionality of the input space.

6.4.5 Gaussian processes for classification

In a probabilistic approach to classification, our goal is to model the posterior probabilities of the target variable for a new input vector, given a set of training data. These probabilities must lie in the interval $(0, 1)$, whereas a Gaussian process model makes predictions that lie on the entire real axis. However, we can easily adapt Gaussian processes to classification problems by transforming the output of the Gaussian process using an appropriate nonlinear activation function.

Consider first the two-class problem with a target variable $t \in \{0, 1\}$. If we define a Gaussian process over a function $a(\mathbf{x})$ and then transform the function using a logistic sigmoid $y = \sigma(a)$, given by (4.59), then we will obtain a non-Gaussian stochastic process over functions $y(\mathbf{x})$ where $y \in (0, 1)$. This is illustrated for the case of a one-dimensional input space in Figure 6.11 in which the probability distri-

Figure 6.11 The left plot shows a sample from a Gaussian process prior over functions $a(\mathbf{x})$, and the right plot shows the result of transforming this sample using a logistic sigmoid function.

bution over the target variable t is then given by the Bernoulli distribution

$$p(t|a) = \sigma(a)^t(1 - \sigma(a))^{1-t}. \tag{6.73}$$

As usual, we denote the training set inputs by $\mathbf{x}_1, \ldots, \mathbf{x}_N$ with corresponding observed target variables $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$. We also consider a single test point $\mathbf{x}_{N+1}$ with target value t_{N+1}. Our goal is to determine the predictive distribution $p(t_{N+1}|\mathbf{t})$, where we have left the conditioning on the input variables implicit. To do this we introduce a Gaussian process prior over the vector $\mathbf{a}_{N+1}$, which has components $a(\mathbf{x}_1), \ldots, a(\mathbf{x}_{N+1})$. This in turn defines a non-Gaussian process over $\mathbf{t}_{N+1}$, and by conditioning on the training data $\mathbf{t}_N$ we obtain the required predictive distribution. The Gaussian process prior for $\mathbf{a}_{N+1}$ takes the form

$$p(\mathbf{a}_{N+1}) = \mathcal{N}(\mathbf{a}_{N+1}|\mathbf{0}, \mathbf{C}_{N+1}). \tag{6.74}$$

Unlike the regression case, the covariance matrix no longer includes a noise term because we assume that all of the training data points are correctly labelled. However, for numerical reasons it is convenient to introduce a noise-like term governed by a parameter ν that ensures that the covariance matrix is positive definite. Thus the covariance matrix $\mathbf{C}_{N+1}$ has elements given by

$$C(\mathbf{x}_n, \mathbf{x}_m) = k(\mathbf{x}_n, \mathbf{x}_m) + \nu\delta_{nm} \tag{6.75}$$

where $k(\mathbf{x}_n, \mathbf{x}_m)$ is any positive semidefinite kernel function of the kind considered in Section 6.2, and the value of ν is typically fixed in advance. We shall assume that the kernel function $k(\mathbf{x}, \mathbf{x}')$ is governed by a vector $\boldsymbol{\theta}$ of parameters, and we shall later discuss how $\boldsymbol{\theta}$ may be learned from the training data.

For two-class problems, it is sufficient to predict $p(t_{N+1} = 1|\mathbf{t}_N)$ because the value of $p(t_{N+1} = 0|\mathbf{t}_N)$ is then given by $1 - p(t_{N+1} = 1|\mathbf{t}_N)$. The required

predictive distribution is given by

$$p(t_{N+1} = 1|\mathbf{t}_N) = \int p(t_{N+1} = 1|a_{N+1})p(a_{N+1}|\mathbf{t}_N)\,\mathrm{d}a_{N+1} \tag{6.76}$$

where $p(t_{N+1} = 1|a_{N+1}) = \sigma(a_{N+1})$.

This integral is analytically intractable, and so may be approximated using sampling methods (Neal, 1997). Alternatively, we can consider techniques based on an analytical approximation. In Section 4.5.2, we derived the approximate formula (4.153) for the convolution of a logistic sigmoid with a Gaussian distribution. We can use this result to evaluate the integral in (6.76) provided we have a Gaussian approximation to the posterior distribution $p(a_{N+1}|\mathbf{t}_N)$. The usual justification for a Gaussian approximation to a posterior distribution is that the true posterior will tend to a Gaussian as the number of data points increases as a consequence of the central limit theorem. In the case of Gaussian processes, the number of variables grows with the number of data points, and so this argument does not apply directly. However, if we consider increasing the number of data points falling in a fixed region of $\mathbf{x}$ space, then the corresponding uncertainty in the function $a(\mathbf{x})$ will decrease, again leading asymptotically to a Gaussian (Williams and Barber, 1998).

Section 2.3

Three different approaches to obtaining a Gaussian approximation have been considered. One technique is based on *variational inference* (Gibbs and MacKay, 2000) and makes use of the local variational bound (10.144) on the logistic sigmoid. This allows the product of sigmoid functions to be approximated by a product of Gaussians thereby allowing the marginalization over $\mathbf{a}_N$ to be performed analytically. The approach also yields a lower bound on the likelihood function $p(\mathbf{t}_N|\boldsymbol{\theta})$. The variational framework for Gaussian process classification can also be extended to multiclass ($K > 2$) problems by using a Gaussian approximation to the softmax function (Gibbs, 1997).

Section 10.1

A second approach uses *expectation propagation* (Opper and Winther, 2000b; Minka, 2001b; Seeger, 2003). Because the true posterior distribution is unimodal, as we shall see shortly, the expectation propagation approach can give good results.

Section 10.7

6.4.6 Laplace approximation

The third approach to Gaussian process classification is based on the Laplace approximation, which we now consider in detail. In order to evaluate the predictive distribution (6.76), we seek a Gaussian approximation to the posterior distribution over a_{N+1}, which, using Bayes' theorem, is given by

Section 4.4

$$\begin{aligned} p(a_{N+1}|\mathbf{t}_N) &= \int p(a_{N+1}, \mathbf{a}_N|\mathbf{t}_N)\,\mathrm{d}\mathbf{a}_N \\ &= \frac{1}{p(\mathbf{t}_N)} \int p(a_{N+1}, \mathbf{a}_N)p(\mathbf{t}_N|a_{N+1}, \mathbf{a}_N)\,\mathrm{d}\mathbf{a}_N \\ &= \frac{1}{p(\mathbf{t}_N)} \int p(a_{N+1}|\mathbf{a}_N)p(\mathbf{a}_N)p(\mathbf{t}_N|\mathbf{a}_N)\,\mathrm{d}\mathbf{a}_N \\ &= \int p(a_{N+1}|\mathbf{a}_N)p(\mathbf{a}_N|\mathbf{t}_N)\,\mathrm{d}\mathbf{a}_N \end{aligned} \tag{6.77}$$

where we have used $p(\mathbf{t}_N|a_{N+1}, \mathbf{a}_N) = p(\mathbf{t}_N|\mathbf{a}_N)$. The conditional distribution $p(a_{N+1}|\mathbf{a}_N)$ is obtained by invoking the results (6.66) and (6.67) for Gaussian process regression, to give

$$p(a_{N+1}|\mathbf{a}_N) = \mathcal{N}(a_{N+1}|\mathbf{k}^{\mathrm{T}}\mathbf{C}_N^{-1}\mathbf{a}_N, c - \mathbf{k}^{\mathrm{T}}\mathbf{C}_N^{-1}\mathbf{k}). \tag{6.78}$$

We can therefore evaluate the integral in (6.77) by finding a Laplace approximation for the posterior distribution $p(\mathbf{a}_N|\mathbf{t}_N)$, and then using the standard result for the convolution of two Gaussian distributions.

The prior $p(\mathbf{a}_N)$ is given by a zero-mean Gaussian process with covariance matrix $\mathbf{C}_N$, and the data term (assuming independence of the data points) is given by

$$p(\mathbf{t}_N|\mathbf{a}_N) = \prod_{n=1}^{N} \sigma(a_n)^{t_n}(1 - \sigma(a_n))^{1-t_n} = \prod_{n=1}^{N} e^{a_n t_n}\sigma(-a_n). \tag{6.79}$$

We then obtain the Laplace approximation by Taylor expanding the logarithm of $p(\mathbf{a}_N|\mathbf{t}_N)$, which up to an additive normalization constant is given by the quantity

$$\begin{aligned}\Psi(\mathbf{a}_N) &= \ln p(\mathbf{a}_N) + \ln p(\mathbf{t}_N|\mathbf{a}_N) \\ &= -\frac{1}{2}\mathbf{a}_N^{\mathrm{T}}\mathbf{C}_N^{-1}\mathbf{a}_N - \frac{N}{2}\ln(2\pi) - \frac{1}{2}\ln|\mathbf{C}_N| + \mathbf{t}_N^{\mathrm{T}}\mathbf{a}_N \\ &\quad - \sum_{n=1}^{N}\ln(1 + e^{a_n}) + \text{const.}\end{aligned} \tag{6.80}$$

First we need to find the mode of the posterior distribution, and this requires that we evaluate the gradient of $\Psi(\mathbf{a}_N)$, which is given by

$$\nabla\Psi(\mathbf{a}_N) = \mathbf{t}_N - \boldsymbol{\sigma}_N - \mathbf{C}_N^{-1}\mathbf{a}_N \tag{6.81}$$

where $\boldsymbol{\sigma}_N$ is a vector with elements $\sigma(a_n)$. We cannot simply find the mode by setting this gradient to zero, because $\boldsymbol{\sigma}_N$ depends nonlinearly on $\mathbf{a}_N$, and so we resort to an iterative scheme based on the Newton-Raphson method, which gives rise to an iterative reweighted least squares (IRLS) algorithm. This requires the second derivatives of $\Psi(\mathbf{a}_N)$, which we also require for the Laplace approximation anyway, and which are given by

Section 4.3.3

$$\nabla\nabla\Psi(\mathbf{a}_N) = -\mathbf{W}_N - \mathbf{C}_N^{-1} \tag{6.82}$$

where $\mathbf{W}_N$ is a diagonal matrix with elements $\sigma(a_n)(1 - \sigma(a_n))$, and we have used the result (4.88) for the derivative of the logistic sigmoid function. Note that these diagonal elements lie in the range $(0, 1/4)$, and hence $\mathbf{W}_N$ is a positive definite matrix. Because $\mathbf{C}_N$ (and hence its inverse) is positive definite by construction, and because the sum of two positive definite matrices is also positive definite, we see that the Hessian matrix $\mathbf{A} = -\nabla\nabla\Psi(\mathbf{a}_N)$ is positive definite and so the posterior distribution $p(\mathbf{a}_N|\mathbf{t}_N)$ is log convex and therefore has a single mode that is the global

Exercise 6.24

maximum. The posterior distribution is not Gaussian, however, because the Hessian is a function of $\mathbf{a}_N$.

Using the Newton-Raphson formula (4.92), the iterative update equation for $\mathbf{a}_N$ is given by

Exercise 6.25

$$\mathbf{a}_N^{\text{new}} = \mathbf{C}_N(\mathbf{I} + \mathbf{W}_N\mathbf{C}_N)^{-1}\left\{\mathbf{t}_N - \boldsymbol{\sigma}_N + \mathbf{W}_N\mathbf{a}_N\right\}. \tag{6.83}$$

These equations are iterated until they converge to the mode which we denote by $\mathbf{a}_N^\star$. At the mode, the gradient $\nabla\Psi(\mathbf{a}_N)$ will vanish, and hence $\mathbf{a}_N^\star$ will satisfy

$$\mathbf{a}_N^\star = \mathbf{C}_N(\mathbf{t}_N - \boldsymbol{\sigma}_N). \tag{6.84}$$

Once we have found the mode $\mathbf{a}_N^\star$ of the posterior, we can evaluate the Hessian matrix given by

$$\mathbf{H} = -\nabla\nabla\Psi(\mathbf{a}_N) = \mathbf{W}_N + \mathbf{C}_N^{-1} \tag{6.85}$$

where the elements of $\mathbf{W}_N$ are evaluated using $\mathbf{a}_N^\star$. This defines our Gaussian approximation to the posterior distribution $p(\mathbf{a}_N|\mathbf{t}_N)$ given by

$$q(\mathbf{a}_N) = \mathcal{N}(\mathbf{a}_N|\mathbf{a}_N^\star, \mathbf{H}^{-1}). \tag{6.86}$$

We can now combine this with (6.78) and hence evaluate the integral (6.77). Because this corresponds to a linear-Gaussian model, we can use the general result (2.115) to give

Exercise 6.26

$$\mathbb{E}[a_{N+1}|\mathbf{t}_N] = \mathbf{k}^{\mathrm{T}}(\mathbf{t}_N - \boldsymbol{\sigma}_N) \tag{6.87}$$

$$\mathrm{var}[a_{N+1}|\mathbf{t}_N] = c - \mathbf{k}^{\mathrm{T}}(\mathbf{W}_N^{-1} + \mathbf{C}_N)^{-1}\mathbf{k}. \tag{6.88}$$

Now that we have a Gaussian distribution for $p(a_{N+1}|\mathbf{t}_N)$, we can approximate the integral (6.76) using the result (4.153). As with the Bayesian logistic regression model of Section 4.5, if we are only interested in the decision boundary corresponding to $p(t_{N+1}|\mathbf{t}_N) = 0.5$, then we need only consider the mean and we can ignore the effect of the variance.

We also need to determine the parameters $\boldsymbol{\theta}$ of the covariance function. One approach is to maximize the likelihood function given by $p(\mathbf{t}_N|\boldsymbol{\theta})$ for which we need expressions for the log likelihood and its gradient. If desired, suitable regularization terms can also be added, leading to a penalized maximum likelihood solution. The likelihood function is defined by

$$p(\mathbf{t}_N|\boldsymbol{\theta}) = \int p(\mathbf{t}_N|\mathbf{a}_N)p(\mathbf{a}_N|\boldsymbol{\theta})\,\mathrm{d}\mathbf{a}_N. \tag{6.89}$$

This integral is analytically intractable, so again we make use of the Laplace approximation. Using the result (4.135), we obtain the following approximation for the log of the likelihood function

$$\ln p(\mathbf{t}_N|\boldsymbol{\theta}) = \Psi(\mathbf{a}_N^\star) - \frac{1}{2}\ln|\mathbf{W}_N + \mathbf{C}_N^{-1}| + \frac{N}{2}\ln(2\pi) \tag{6.90}$$

where $\Psi(\mathbf{a}_N^\star) = \ln p(\mathbf{a}_N^\star|\boldsymbol{\theta}) + \ln p(\mathbf{t}_N|\mathbf{a}_N^\star)$. We also need to evaluate the gradient of $\ln p(\mathbf{t}_N|\boldsymbol{\theta})$ with respect to the parameter vector $\boldsymbol{\theta}$. Note that changes in $\boldsymbol{\theta}$ will cause changes in $\mathbf{a}_N^\star$, leading to additional terms in the gradient. Thus, when we differentiate (6.90) with respect to $\boldsymbol{\theta}$, we obtain two sets of terms, the first arising from the dependence of the covariance matrix $\mathbf{C}_N$ on $\boldsymbol{\theta}$, and the rest arising from dependence of $\mathbf{a}_N^\star$ on $\boldsymbol{\theta}$.

The terms arising from the explicit dependence on $\boldsymbol{\theta}$ can be found by using (6.80) together with the results (C.21) and (C.22), and are given by

$$\begin{aligned}\frac{\partial \ln p(\mathbf{t}_N|\boldsymbol{\theta})}{\partial \theta_j} &= \frac{1}{2}\mathbf{a}_N^{\star\mathrm{T}}\mathbf{C}_N^{-1}\frac{\partial \mathbf{C}_N}{\partial \theta_j}\mathbf{C}_N^{-1}\mathbf{a}_N^\star \\ &\quad -\frac{1}{2}\mathrm{Tr}\left[(\mathbf{I}+\mathbf{C}_N\mathbf{W}_N)^{-1}\mathbf{W}_N\frac{\partial \mathbf{C}_N}{\partial \theta_j}\right]. \end{aligned} \tag{6.91}$$

To compute the terms arising from the dependence of $\mathbf{a}_N^\star$ on $\boldsymbol{\theta}$, we note that the Laplace approximation has been constructed such that $\Psi(\mathbf{a}_N)$ has zero gradient at $\mathbf{a}_N = \mathbf{a}_N^\star$, and so $\Psi(\mathbf{a}_N^\star)$ gives no contribution to the gradient as a result of its dependence on $\mathbf{a}_N^\star$. This leaves the following contribution to the derivative with respect to a component θ_j of $\boldsymbol{\theta}$

$$\begin{aligned}&-\frac{1}{2}\sum_{n=1}^{N}\frac{\partial \ln|\mathbf{W}_N+\mathbf{C}_N^{-1}|}{\partial a_n^\star}\frac{\partial a_n^\star}{\partial \theta_j} \\ &\quad = -\frac{1}{2}\sum_{n=1}^{N}\left[(\mathbf{I}+\mathbf{C}_N\mathbf{W}_N)^{-1}\mathbf{C}_N\right]_{nn}\sigma_n^\star(1-\sigma_n^\star)(1-2\sigma_n^\star)\frac{\partial a_n^\star}{\partial \theta_j}\end{aligned} \tag{6.92}$$

where $\sigma_n^\star = \sigma(a_n^\star)$, and again we have used the result (C.22) together with the definition of $\mathbf{W}_N$. We can evaluate the derivative of $a_N^\star$ with respect to θ_j by differentiating the relation (6.84) with respect to θ_j to give

$$\frac{\partial a_n^\star}{\partial \theta_j} = \frac{\partial \mathbf{C}_N}{\partial \theta_j}(\mathbf{t}_N - \boldsymbol{\sigma}_N) - \mathbf{C}_N\mathbf{W}_N\frac{\partial a_n^\star}{\partial \theta_j}. \tag{6.93}$$

Rearranging then gives

$$\frac{\partial a_n^\star}{\partial \theta_j} = (\mathbf{I}+\mathbf{W}_N\mathbf{C}_N)^{-1}\frac{\partial \mathbf{C}_N}{\partial \theta_j}(\mathbf{t}_N - \boldsymbol{\sigma}_N). \tag{6.94}$$

Combining (6.91), (6.92), and (6.94), we can evaluate the gradient of the log likelihood function, which can be used with standard nonlinear optimization algorithms in order to determine a value for $\boldsymbol{\theta}$.

Appendix A

We can illustrate the application of the Laplace approximation for Gaussian processes using the synthetic two-class data set shown in Figure 6.12. Extension of the Laplace approximation to Gaussian processes involving $K > 2$ classes, using the softmax activation function, is straightforward (Williams and Barber, 1998).

Figure 6.12 Illustration of the use of a Gaussian process for classification, showing the data on the left together with the optimal decision boundary from the true distribution in green, and the decision boundary from the Gaussian process classifier in black. On the right is the predicted posterior probability for the blue and red classes together with the Gaussian process decision boundary.

6.4.7 Connection to neural networks

We have seen that the range of functions which can be represented by a neural network is governed by the number M of hidden units, and that, for sufficiently large M, a two-layer network can approximate any given function with arbitrary accuracy. In the framework of maximum likelihood, the number of hidden units needs to be limited (to a level dependent on the size of the training set) in order to avoid over-fitting. However, from a Bayesian perspective it makes little sense to limit the number of parameters in the network according to the size of the training set.

In a Bayesian neural network, the prior distribution over the parameter vector $\mathbf{w}$, in conjunction with the network function $f(\mathbf{x}, \mathbf{w})$, produces a prior distribution over functions from $y(\mathbf{x})$ where $\mathbf{y}$ is the vector of network outputs. Neal (1996) has shown that, for a broad class of prior distributions over $\mathbf{w}$, the distribution of functions generated by a neural network will tend to a Gaussian process in the limit $M \to \infty$. It should be noted, however, that in this limit the output variables of the neural network become independent. One of the great merits of neural networks is that the outputs share the hidden units and so they can 'borrow statistical strength' from each other, that is, the weights associated with each hidden unit are influenced by all of the output variables not just by one of them. This property is therefore lost in the Gaussian process limit.

We have seen that a Gaussian process is determined by its covariance (kernel) function. Williams (1998) has given explicit forms for the covariance in the case of two specific choices for the hidden unit activation function (probit and Gaussian). These kernel functions $k(\mathbf{x}, \mathbf{x}')$ are nonstationary, i.e. they cannot be expressed as a function of the difference $\mathbf{x} - \mathbf{x}'$, as a consequence of the Gaussian weight prior being centred on zero which breaks translation invariance in weight space.

By working directly with the covariance function we have implicitly marginalized over the distribution of weights. If the weight prior is governed by hyperparameters, then their values will determine the length scales of the distribution over functions, as can be understood by studying the examples in Figure 5.11 for the case of a finite number of hidden units. Note that we cannot marginalize out the hyperparameters analytically, and must instead resort to techniques of the kind discussed in Section 6.4.

Exercises

6.1 ($\star\star$) **WWW** Consider the dual formulation of the least squares linear regression problem given in Section 6.1. Show that the solution for the components a_n of the vector $\mathbf{a}$ can be expressed as a linear combination of the elements of the vector $\boldsymbol{\phi}(\mathbf{x}_n)$. Denoting these coefficients by the vector $\mathbf{w}$, show that the dual of the dual formulation is given by the original representation in terms of the parameter vector $\mathbf{w}$.

6.2 ($\star\star$) In this exercise, we develop a dual formulation of the perceptron learning algorithm. Using the perceptron learning rule (4.55), show that the learned weight vector $\mathbf{w}$ can be written as a linear combination of the vectors $t_n\boldsymbol{\phi}(\mathbf{x}_n)$ where $t_n \in \{-1, +1\}$. Denote the coefficients of this linear combination by α_n and derive a formulation of the perceptron learning algorithm, and the predictive function for the perceptron, in terms of the α_n. Show that the feature vector $\boldsymbol{\phi}(\mathbf{x})$ enters only in the form of the kernel function $k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}')$.

6.3 ($\star$) The nearest-neighbour classifier (Section 2.5.2) assigns a new input vector $\mathbf{x}$ to the same class as that of the nearest input vector $\mathbf{x}_n$ from the training set, where in the simplest case, the distance is defined by the Euclidean metric $\|\mathbf{x} - \mathbf{x}_n\|^2$. By expressing this rule in terms of scalar products and then making use of kernel substitution, formulate the nearest-neighbour classifier for a general nonlinear kernel.

6.4 ($\star$) In Appendix C, we give an example of a matrix that has positive elements but that has a negative eigenvalue and hence that is not positive definite. Find an example of the converse property, namely a 2×2 matrix with positive eigenvalues yet that has at least one negative element.

6.5 ($\star$) **WWW** Verify the results (6.13) and (6.14) for constructing valid kernels.

6.6 ($\star$) Verify the results (6.15) and (6.16) for constructing valid kernels.

6.7 ($\star$) **WWW** Verify the results (6.17) and (6.18) for constructing valid kernels.

6.8 ($\star$) Verify the results (6.19) and (6.20) for constructing valid kernels.

6.9 ($\star$) Verify the results (6.21) and (6.22) for constructing valid kernels.

6.10 ($\star$) Show that an excellent choice of kernel for learning a function $f(\mathbf{x})$ is given by $k(\mathbf{x}, \mathbf{x}') = f(\mathbf{x})f(\mathbf{x}')$ by showing that a linear learning machine based on this kernel will always find a solution proportional to $f(\mathbf{x})$.

6.11 ($\star$) By making use of the expansion (6.25), and then expanding the middle factor as a power series, show that the Gaussian kernel (6.23) can be expressed as the inner product of an infinite-dimensional feature vector.

6.12 ($\star\star$) **www** Consider the space of all possible subsets A of a given fixed set D. Show that the kernel function (6.27) corresponds to an inner product in a feature space of dimensionality $2^{|D|}$ defined by the mapping $\boldsymbol{\phi}(A)$ where A is a subset of D and the element $\phi_U(A)$, indexed by the subset U, is given by

$$\phi_U(A) = \begin{cases} 1, & \text{if } U \subseteq A; \\ 0, & \text{otherwise.} \end{cases} \tag{6.95}$$

Here $U \subseteq A$ denotes that U is either a subset of A or is equal to A.

6.13 ($\star$) Show that the Fisher kernel, defined by (6.33), remains invariant if we make a nonlinear transformation of the parameter vector $\boldsymbol{\theta} \rightarrow \boldsymbol{\psi}(\boldsymbol{\theta})$, where the function $\boldsymbol{\psi}(\cdot)$ is invertible and differentiable.

6.14 ($\star$) **www** Write down the form of the Fisher kernel, defined by (6.33), for the case of a distribution $p(\mathbf{x}|\boldsymbol{\mu}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \mathbf{S})$ that is Gaussian with mean $\boldsymbol{\mu}$ and fixed covariance $\mathbf{S}$.

6.15 ($\star$) By considering the determinant of a 2×2 Gram matrix, show that a positive-definite kernel function $k(x, x')$ satisfies the Cauchy-Schwartz inequality

$$k(x_1, x_2)^2 \leqslant k(x_1, x_1)k(x_2, x_2). \tag{6.96}$$

6.16 ($\star\star$) Consider a parametric model governed by the parameter vector $\mathbf{w}$ together with a data set of input values $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and a nonlinear feature mapping $\boldsymbol{\phi}(\mathbf{x})$. Suppose that the dependence of the error function on $\mathbf{w}$ takes the form

$$J(\mathbf{w}) = f(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_1), \ldots, \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_N)) + g(\mathbf{w}^{\mathrm{T}}\mathbf{w}) \tag{6.97}$$

where $g(\cdot)$ is a monotonically increasing function. By writing $\mathbf{w}$ in the form

$$\mathbf{w} = \sum_{n=1}^{N} \alpha_n \boldsymbol{\phi}(\mathbf{x}_n) + \mathbf{w}_\perp \tag{6.98}$$

show that the value of $\mathbf{w}$ that minimizes $J(\mathbf{w})$ takes the form of a linear combination of the basis functions $\boldsymbol{\phi}(\mathbf{x}_n)$ for $n = 1, \ldots, N$.

6.17 ($\star\star$) **www** Consider the sum-of-squares error function (6.39) for data having noisy inputs, where $\nu(\boldsymbol{\xi})$ is the distribution of the noise. Use the calculus of variations to minimize this error function with respect to the function $y(\mathbf{x})$, and hence show that the optimal solution is given by an expansion of the form (6.40) in which the basis functions are given by (6.41).

6.18 ($\star$) Consider a Nadaraya-Watson model with one input variable x and one target variable t having Gaussian components with isotropic covariances, so that the covariance matrix is given by $\sigma^2\mathbf{I}$ where $\mathbf{I}$ is the unit matrix. Write down expressions for the conditional density $p(t|x)$ and for the conditional mean $\mathbb{E}[t|x]$ and variance $\mathrm{var}[t|x]$, in terms of the kernel function $k(x, x_n)$.

6.19 ($\star\star$) Another viewpoint on kernel regression comes from a consideration of regression problems in which the input variables as well as the target variables are corrupted with additive noise. Suppose each target value t_n is generated as usual by taking a function $y(\mathbf{z}_n)$ evaluated at a point $\mathbf{z}_n$, and adding Gaussian noise. The value of $\mathbf{z}_n$ is not directly observed, however, but only a noise corrupted version $\mathbf{x}_n = \mathbf{z}_n + \boldsymbol{\xi}_n$ where the random variable $\boldsymbol{\xi}$ is governed by some distribution $g(\boldsymbol{\xi})$. Consider a set of observations $\{\mathbf{x}_n, t_n\}$, where $n = 1, \ldots, N$, together with a corresponding sum-of-squares error function defined by averaging over the distribution of input noise to give

$$E = \frac{1}{2}\sum_{n=1}^{N}\int \{y(\mathbf{x}_n - \boldsymbol{\xi}_n) - t_n\}^2\, g(\boldsymbol{\xi}_n)\,\mathrm{d}\boldsymbol{\xi}_n. \tag{6.99}$$

By minimizing E with respect to the function $y(\mathbf{z})$ using the calculus of variations (Appendix D), show that optimal solution for $y(\mathbf{x})$ is given by a Nadaraya-Watson kernel regression solution of the form (6.45) with a kernel of the form (6.46).

6.20 ($\star\star$) **www** Verify the results (6.66) and (6.67).

6.21 ($\star\star$) **www** Consider a Gaussian process regression model in which the kernel function is defined in terms of a fixed set of nonlinear basis functions. Show that the predictive distribution is identical to the result (3.58) obtained in Section 3.3.2 for the Bayesian linear regression model. To do this, note that both models have Gaussian predictive distributions, and so it is only necessary to show that the conditional mean and variance are the same. For the mean, make use of the matrix identity (C.6), and for the variance, make use of the matrix identity (C.7).

6.22 ($\star\star$) Consider a regression problem with N training set input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and L test set input vectors $\mathbf{x}_{N+1}, \ldots, \mathbf{x}_{N+L}$, and suppose we define a Gaussian process prior over functions $t(\mathbf{x})$. Derive an expression for the joint predictive distribution for $t(\mathbf{x}_{N+1}), \ldots, t(\mathbf{x}_{N+L})$, given the values of $t(\mathbf{x}_1), \ldots, t(\mathbf{x}_N)$. Show the marginal of this distribution for one of the test observations t_j where $N + 1 \leqslant j \leqslant N + L$ is given by the usual Gaussian process regression result (6.66) and (6.67).

6.23 ($\star\star$) **www** Consider a Gaussian process regression model in which the target variable $\mathbf{t}$ has dimensionality D. Write down the conditional distribution of $\mathbf{t}_{N+1}$ for a test input vector $\mathbf{x}_{N+1}$, given a training set of input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_{N+1}$ and corresponding target observations $\mathbf{t}_1, \ldots, \mathbf{t}_N$.

6.24 ($\star$) Show that a diagonal matrix $\mathbf{W}$ whose elements satisfy $0 < W_{ii} < 1$ is positive definite. Show that the sum of two positive definite matrices is itself positive definite.

6.25 ($\star$) **www** Using the Newton-Raphson formula (4.92), derive the iterative update formula (6.83) for finding the mode $\mathbf{a}_N^\star$ of the posterior distribution in the Gaussian process classification model.

6.26 ($\star$) Using the result (2.115), derive the expressions (6.87) and (6.88) for the mean and variance of the posterior distribution $p(a_{N+1}|\mathbf{t}_N)$ in the Gaussian process classification model.

6.27 ($\star\star\star$) Derive the result (6.90) for the log likelihood function in the Laplace approximation framework for Gaussian process classification. Similarly, derive the results (6.91), (6.92), and (6.94) for the terms in the gradient of the log likelihood.

7

Sparse Kernel Machines

In the previous chapter, we explored a variety of learning algorithms based on nonlinear kernels. One of the significant limitations of many such algorithms is that the kernel function $k(\mathbf{x}_n, \mathbf{x}_m)$ must be evaluated for all possible pairs $\mathbf{x}_n$ and $\mathbf{x}_m$ of training points, which can be computationally infeasible during training and can lead to excessive computation times when making predictions for new data points. In this chapter we shall look at kernel-based algorithms that have *sparse* solutions, so that predictions for new inputs depend only on the kernel function evaluated at a subset of the training data points.

We begin by looking in some detail at the *support vector machine* (SVM), which became popular in some years ago for solving problems in classification, regression, and novelty detection. An important property of support vector machines is that the determination of the model parameters corresponds to a convex optimization problem, and so any local solution is also a global optimum. Because the discussion of support vector machines makes extensive use of Lagrange multipliers, the reader is

encouraged to review the key concepts covered in Appendix E. Additional information on support vector machines can be found in Vapnik (1995), Burges (1998), Cristianini and Shawe-Taylor (2000), Müller *et al.* (2001), Schölkopf and Smola (2002), and Herbrich (2002).

The SVM is a decision machine and so does not provide posterior probabilities. We have already discussed some of the benefits of determining probabilities in Section 1.5.4. An alternative sparse kernel technique, known as the *relevance vector machine* (RVM), is based on a Bayesian formulation and provides posterior probabilistic outputs, as well as having typically much sparser solutions than the SVM.

Section 7.2

7.1. Maximum Margin Classifiers

We begin our discussion of support vector machines by returning to the two-class classification problem using linear models of the form

$$y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}) + b \tag{7.1}$$

where $\boldsymbol{\phi}(\mathbf{x})$ denotes a fixed feature-space transformation, and we have made the bias parameter b explicit. Note that we shall shortly introduce a dual representation expressed in terms of kernel functions, which avoids having to work explicitly in feature space. The training data set comprises N input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$, with corresponding target values $t_1, \ldots, t_N$ where $t_n \in \{-1, 1\}$, and new data points $\mathbf{x}$ are classified according to the sign of $y(\mathbf{x})$.

We shall assume for the moment that the training data set is linearly separable in feature space, so that by definition there exists at least one choice of the parameters $\mathbf{w}$ and b such that a function of the form (7.1) satisfies $y(\mathbf{x}_n) > 0$ for points having $t_n = +1$ and $y(\mathbf{x}_n) < 0$ for points having $t_n = -1$, so that $t_n y(\mathbf{x}_n) > 0$ for all training data points.

There may of course exist many such solutions that separate the classes exactly. In Section 4.1.7, we described the perceptron algorithm that is guaranteed to find a solution in a finite number of steps. The solution that it finds, however, will be dependent on the (arbitrary) initial values chosen for $\mathbf{w}$ and b as well as on the order in which the data points are presented. If there are multiple solutions all of which classify the training data set exactly, then we should try to find the one that will give the smallest generalization error. The support vector machine approaches this problem through the concept of the *margin*, which is defined to be the smallest distance between the decision boundary and any of the samples, as illustrated in Figure 7.1.

In support vector machines the decision boundary is chosen to be the one for which the margin is maximized. The maximum margin solution can be motivated using *computational learning theory*, also known as *statistical learning theory*. However, a simple insight into the origins of maximum margin has been given by Tong and Koller (2000) who consider a framework for classification based on a hybrid of generative and discriminative approaches. They first model the distribution over input vectors $\mathbf{x}$ for each class using a Parzen density estimator with Gaussian kernels

Section 7.1.5

Figure 7.1 The margin is defined as the perpendicular distance between the decision boundary and the closest of the data points, as shown on the left figure. Maximizing the margin leads to a particular choice of decision boundary, as shown on the right. The location of this boundary is determined by a subset of the data points, known as support vectors, which are indicated by the circles.

having a common parameter σ^2. Together with the class priors, this defines an optimal misclassification-rate decision boundary. However, instead of using this optimal boundary, they determine the best hyperplane by minimizing the probability of error relative to the learned density model. In the limit $\sigma^2 \to 0$, the optimal hyperplane is shown to be the one having maximum margin. The intuition behind this result is that as σ^2 is reduced, the hyperplane is increasingly dominated by nearby data points relative to more distant ones. In the limit, the hyperplane becomes independent of data points that are not support vectors.

We shall see in Figure 10.13 that marginalization with respect to the prior distribution of the parameters in a Bayesian approach for a simple linearly separable data set leads to a decision boundary that lies in the middle of the region separating the data points. The large margin solution has similar behaviour.

Recall from Figure 4.1 that the perpendicular distance of a point $\mathbf{x}$ from a hyperplane defined by $y(\mathbf{x}) = 0$ where $y(\mathbf{x})$ takes the form (7.1) is given by $|y(\mathbf{x})|/\|\mathbf{w}\|$. Furthermore, we are only interested in solutions for which all data points are correctly classified, so that $t_n y(\mathbf{x}_n) > 0$ for all n. Thus the distance of a point $\mathbf{x}_n$ to the decision surface is given by

$$\frac{t_n y(\mathbf{x}_n)}{\|\mathbf{w}\|} = \frac{t_n(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n) + b)}{\|\mathbf{w}\|}. \tag{7.2}$$

The margin is given by the perpendicular distance to the closest point $\mathbf{x}_n$ from the data set, and we wish to optimize the parameters $\mathbf{w}$ and b in order to maximize this distance. Thus the maximum margin solution is found by solving

$$\arg\max_{\mathbf{w},b} \left\{ \frac{1}{\|\mathbf{w}\|} \min_n \left[t_n \left(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n) + b \right) \right] \right\} \tag{7.3}$$

where we have taken the factor $1/\|\mathbf{w}\|$ outside the optimization over n because $\mathbf{w}$

does not depend on n. Direct solution of this optimization problem would be very complex, and so we shall convert it into an equivalent problem that is much easier to solve. To do this we note that if we make the rescaling $\mathbf{w} \to \kappa\mathbf{w}$ and $b \to \kappa b$, then the distance from any point $\mathbf{x}_n$ to the decision surface, given by $t_n y(\mathbf{x}_n)/\|\mathbf{w}\|$, is unchanged. We can use this freedom to set

$$t_n \left(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n) + b\right) = 1 \tag{7.4}$$

for the point that is closest to the surface. In this case, all data points will satisfy the constraints

$$t_n \left(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n) + b\right) \geqslant 1, \qquad n = 1, \ldots, N. \tag{7.5}$$

This is known as the canonical representation of the decision hyperplane. In the case of data points for which the equality holds, the constraints are said to be *active*, whereas for the remainder they are said to be *inactive*. By definition, there will always be at least one active constraint, because there will always be a closest point, and once the margin has been maximized there will be at least two active constraints. The optimization problem then simply requires that we maximize $\|\mathbf{w}\|^{-1}$, which is equivalent to minimizing $\|\mathbf{w}\|^2$, and so we have to solve the optimization problem

$$\arg\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 \tag{7.6}$$

subject to the constraints given by (7.5). The factor of $1/2$ in (7.6) is included for later convenience. This is an example of a *quadratic programming* problem in which we are trying to minimize a quadratic function subject to a set of linear inequality constraints. It appears that the bias parameter b has disappeared from the optimization. However, it is determined implicitly via the constraints, because these require that changes to $\|\mathbf{w}\|$ be compensated by changes to b. We shall see how this works shortly.

Appendix E

In order to solve this constrained optimization problem, we introduce Lagrange multipliers $a_n \geqslant 0$, with one multiplier a_n for each of the constraints in (7.5), giving the Lagrangian function

$$L(\mathbf{w}, b, \mathbf{a}) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{n=1}^{N} a_n \left\{t_n(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n) + b) - 1\right\} \tag{7.7}$$

where $\mathbf{a} = (a_1, \ldots, a_N)^{\mathrm{T}}$. Note the minus sign in front of the Lagrange multiplier term, because we are minimizing with respect to $\mathbf{w}$ and b, and maximizing with respect to $\mathbf{a}$. Setting the derivatives of $L(\mathbf{w}, b, \mathbf{a})$ with respect to $\mathbf{w}$ and b equal to zero, we obtain the following two conditions

$$\mathbf{w} = \sum_{n=1}^{N} a_n t_n \boldsymbol{\phi}(\mathbf{x}_n) \tag{7.8}$$

$$0 = \sum_{n=1}^{N} a_n t_n. \tag{7.9}$$

Eliminating $\mathbf{w}$ and b from $L(\mathbf{w}, b, \mathbf{a})$ using these conditions then gives the *dual representation* of the maximum margin problem in which we maximize

$$\widetilde{L}(\mathbf{a}) = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m t_n t_m k(\mathbf{x}_n, \mathbf{x}_m) \tag{7.10}$$

with respect to $\mathbf{a}$ subject to the constraints

$$a_n \geqslant 0, \qquad n = 1, \ldots, N, \tag{7.11}$$

$$\sum_{n=1}^{N} a_n t_n = 0. \tag{7.12}$$

Here the kernel function is defined by $k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}')$. Again, this takes the form of a quadratic programming problem in which we optimize a quadratic function of $\mathbf{a}$ subject to a set of inequality constraints. We shall discuss techniques for solving such quadratic programming problems in Section 7.1.1.

The solution to a quadratic programming problem in M variables in general has computational complexity that is $O(M^3)$. In going to the dual formulation we have turned the original optimization problem, which involved minimizing (7.6) over M variables, into the dual problem (7.10), which has N variables. For a fixed set of basis functions whose number M is smaller than the number N of data points, the move to the dual problem appears disadvantageous. However, it allows the model to be reformulated using kernels, and so the maximum margin classifier can be applied efficiently to feature spaces whose dimensionality exceeds the number of data points, including infinite feature spaces. The kernel formulation also makes clear the role of the constraint that the kernel function $k(\mathbf{x}, \mathbf{x}')$ be positive definite, because this ensures that the Lagrangian function $\widetilde{L}(\mathbf{a})$ is bounded below, giving rise to a well-defined optimization problem.

In order to classify new data points using the trained model, we evaluate the sign of $y(\mathbf{x})$ defined by (7.1). This can be expressed in terms of the parameters $\{a_n\}$ and the kernel function by substituting for $\mathbf{w}$ using (7.8) to give

$$y(\mathbf{x}) = \sum_{n=1}^{N} a_n t_n k(\mathbf{x}, \mathbf{x}_n) + b. \tag{7.13}$$

Joseph-Louis Lagrange

1736–1813

Although widely considered to be a French mathematician, Lagrange was born in Turin in Italy. By the age of nineteen, he had already made important contributions mathematics and had been appointed as Professor at the Royal Artillery School in Turin. For many years, Euler worked hard to persuade Lagrange to move to Berlin, which he eventually did in 1766 where he succeeded Euler as Director of Mathematics at the Berlin Academy. Later he moved to Paris, narrowly escaping with his life during the French revolution thanks to the personal intervention of Lavoisier (the French chemist who discovered oxygen) who himself was later executed at the guillotine. Lagrange made key contributions to the calculus of variations and the foundations of dynamics.

In Appendix E, we show that a constrained optimization of this form satisfies the *Karush-Kuhn-Tucker* (KKT) conditions, which in this case require that the following three properties hold

$$a_n \geqslant 0 \tag{7.14}$$

$$t_n y(\mathbf{x}_n) - 1 \geqslant 0 \tag{7.15}$$

$$a_n \{t_n y(\mathbf{x}_n) - 1\} = 0. \tag{7.16}$$

Thus for every data point, either $a_n = 0$ or $t_n y(\mathbf{x}_n) = 1$. Any data point for which $a_n = 0$ will not appear in the sum in (7.13) and hence plays no role in making predictions for new data points. The remaining data points are called *support vectors*, and because they satisfy $t_n y(\mathbf{x}_n) = 1$, they correspond to points that lie on the maximum margin hyperplanes in feature space, as illustrated in Figure 7.1. This property is central to the practical applicability of support vector machines. Once the model is trained, a significant proportion of the data points can be discarded and only the support vectors retained.

Having solved the quadratic programming problem and found a value for $\mathbf{a}$, we can then determine the value of the threshold parameter b by noting that any support vector $\mathbf{x}_n$ satisfies $t_n y(\mathbf{x}_n) = 1$. Using (7.13) this gives

$$t_n \left(\sum_{m \in \mathcal{S}} a_m t_m k(\mathbf{x}_n, \mathbf{x}_m) + b \right) = 1 \tag{7.17}$$

where $\mathcal{S}$ denotes the set of indices of the support vectors. Although we can solve this equation for b using an arbitrarily chosen support vector $\mathbf{x}_n$, a numerically more stable solution is obtained by first multiplying through by t_n, making use of $t_n^2 = 1$, and then averaging these equations over all support vectors and solving for b to give

$$b = \frac{1}{N_{\mathcal{S}}} \sum_{n \in \mathcal{S}} \left(t_n - \sum_{m \in \mathcal{S}} a_m t_m k(\mathbf{x}_n, \mathbf{x}_m) \right) \tag{7.18}$$

where $N_{\mathcal{S}}$ is the total number of support vectors.

For later comparison with alternative models, we can express the maximum-margin classifier in terms of the minimization of an error function, with a simple quadratic regularizer, in the form

$$\sum_{n=1}^{N} E_\infty(y(\mathbf{x}_n) t_n - 1) + \lambda \|\mathbf{w}\|^2 \tag{7.19}$$

where $E_\infty(z)$ is a function that is zero if $z \geqslant 0$ and ∞ otherwise and ensures that the constraints (7.5) are satisfied. Note that as long as the regularization parameter satisfies $\lambda > 0$, its precise value plays no role.

Figure 7.2 shows an example of the classification resulting from training a support vector machine on a simple synthetic data set using a Gaussian kernel of the

Figure 7.2 Example of synthetic data from two classes in two dimensions showing contours of constant $y(\mathbf{x})$ obtained from a support vector machine having a Gaussian kernel function. Also shown are the decision boundary, the margin boundaries, and the support vectors.

form (6.23). Although the data set is not linearly separable in the two-dimensional data space $\mathbf{x}$, it is linearly separable in the nonlinear feature space defined implicitly by the nonlinear kernel function. Thus the training data points are perfectly separated in the original data space.

This example also provides a geometrical insight into the origin of sparsity in the SVM. The maximum margin hyperplane is defined by the location of the support vectors. Other data points can be moved around freely (so long as they remain outside the margin region) without changing the decision boundary, and so the solution will be independent of such data points.

7.1.1 Overlapping class distributions

So far, we have assumed that the training data points are linearly separable in the feature space $\phi(\mathbf{x})$. The resulting support vector machine will give exact separation of the training data in the original input space $\mathbf{x}$, although the corresponding decision boundary will be nonlinear. In practice, however, the class-conditional distributions may overlap, in which case exact separation of the training data can lead to poor generalization.

We therefore need a way to modify the support vector machine so as to allow some of the training points to be misclassified. From (7.19) we see that in the case of separable classes, we implicitly used an error function that gave infinite error if a data point was misclassified and zero error if it was classified correctly, and then optimized the model parameters to maximize the margin. We now modify this approach so that data points are allowed to be on the 'wrong side' of the margin boundary, but with a penalty that increases with the distance from that boundary. For the subsequent optimization problem, it is convenient to make this penalty a linear function of this distance. To do this, we introduce *slack variables*, $\xi_n \geqslant 0$ where $n = 1, \ldots, N$, with one slack variable for each training data point (Bennett, 1992; Cortes and Vapnik, 1995). These are defined by $\xi_n = 0$ for data points that are on or inside the correct margin boundary and $\xi_n = |t_n - y(\mathbf{x}_n)|$ for other points. Thus a data point that is on the decision boundary $y(\mathbf{x}_n) = 0$ will have $\xi_n = 1$, and points

Figure 7.3 Illustration of the slack variables $\xi_n \geqslant 0$. Data points with circles around them are support vectors.

with $\xi_n > 1$ will be misclassified. The exact classification constraints (7.5) are then replaced with

$$t_n y(\mathbf{x}_n) \geqslant 1 - \xi_n, \qquad n = 1, \ldots, N \tag{7.20}$$

in which the slack variables are constrained to satisfy $\xi_n \geqslant 0$. Data points for which $\xi_n = 0$ are correctly classified and are either on the margin or on the correct side of the margin. Points for which $0 < \xi_n \leqslant 1$ lie inside the margin, but on the correct side of the decision boundary, and those data points for which $\xi_n > 1$ lie on the wrong side of the decision boundary and are misclassified, as illustrated in Figure 7.3. This is sometimes described as relaxing the hard margin constraint to give a *soft margin* and allows some of the training set data points to be misclassified. Note that while slack variables allow for overlapping class distributions, this framework is still sensitive to outliers because the penalty for misclassification increases linearly with ξ.

Our goal is now to maximize the margin while softly penalizing points that lie on the wrong side of the margin boundary. We therefore minimize

$$C \sum_{n=1}^{N} \xi_n + \frac{1}{2}\|\mathbf{w}\|^2 \tag{7.21}$$

where the parameter $C > 0$ controls the trade-off between the slack variable penalty and the margin. Because any point that is misclassified has $\xi_n > 1$, it follows that $\sum_n \xi_n$ is an upper bound on the number of misclassified points. The parameter C is therefore analogous to (the inverse of) a regularization coefficient because it controls the trade-off between minimizing training errors and controlling model complexity. In the limit $C \to \infty$, we will recover the earlier support vector machine for separable data.

We now wish to minimize (7.21) subject to the constraints (7.20) together with $\xi_n \geqslant 0$. The corresponding Lagrangian is given by

$$L(\mathbf{w}, b, \mathbf{a}) = \frac{1}{2}\|\mathbf{w}\|^2 + C \sum_{n=1}^{N} \xi_n - \sum_{n=1}^{N} a_n \left\{ t_n y(\mathbf{x}_n) - 1 + \xi_n \right\} - \sum_{n=1}^{N} \mu_n \xi_n \tag{7.22}$$

Appendix E

where $\{a_n \geqslant 0\}$ and $\{\mu_n \geqslant 0\}$ are Lagrange multipliers. The corresponding set of KKT conditions are given by

$$a_n \geqslant 0 \tag{7.23}$$
$$t_n y(\mathbf{x}_n) - 1 + \xi_n \geqslant 0 \tag{7.24}$$
$$a_n \left(t_n y(\mathbf{x}_n) - 1 + \xi_n\right) = 0 \tag{7.25}$$
$$\mu_n \geqslant 0 \tag{7.26}$$
$$\xi_n \geqslant 0 \tag{7.27}$$
$$\mu_n \xi_n = 0 \tag{7.28}$$

where $n = 1, \ldots, N$.

We now optimize out $\mathbf{w}$, b, and $\{\xi_n\}$ making use of the definition (7.1) of $y(\mathbf{x})$ to give

$$\frac{\partial L}{\partial \mathbf{w}} = 0 \quad \Rightarrow \quad \mathbf{w} = \sum_{n=1}^{N} a_n t_n \boldsymbol{\phi}(\mathbf{x}_n) \tag{7.29}$$
$$\frac{\partial L}{\partial b} = 0 \quad \Rightarrow \quad \sum_{n=1}^{N} a_n t_n = 0 \tag{7.30}$$
$$\frac{\partial L}{\partial \xi_n} = 0 \quad \Rightarrow \quad a_n = C - \mu_n. \tag{7.31}$$

Using these results to eliminate $\mathbf{w}$, b, and $\{\xi_n\}$ from the Lagrangian, we obtain the dual Lagrangian in the form

$$\widetilde{L}(\mathbf{a}) = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m t_n t_m k(\mathbf{x}_n, \mathbf{x}_m) \tag{7.32}$$

which is identical to the separable case, except that the constraints are somewhat different. To see what these constraints are, we note that $a_n \geqslant 0$ is required because these are Lagrange multipliers. Furthermore, (7.31) together with $\mu_n \geqslant 0$ implies $a_n \leqslant C$. We therefore have to minimize (7.32) with respect to the dual variables $\{a_n\}$ subject to

$$0 \leqslant a_n \leqslant C \tag{7.33}$$
$$\sum_{n=1}^{N} a_n t_n = 0 \tag{7.34}$$

for $n = 1, \ldots, N$, where (7.33) are known as *box constraints*. This again represents a quadratic programming problem. If we substitute (7.29) into (7.1), we see that predictions for new data points are again made by using (7.13).

We can now interpret the resulting solution. As before, a subset of the data points may have $a_n = 0$, in which case they do not contribute to the predictive

model (7.13). The remaining data points constitute the support vectors. These have $a_n > 0$ and hence from (7.25) must satisfy

$$t_n y(\mathbf{x}_n) = 1 - \xi_n. \tag{7.35}$$

If $a_n < C$, then (7.31) implies that $\mu_n > 0$, which from (7.28) requires $\xi_n = 0$ and hence such points lie on the margin. Points with $a_n = C$ can lie inside the margin and can either be correctly classified if $\xi_n \leqslant 1$ or misclassified if $\xi_n > 1$.

To determine the parameter b in (7.1), we note that those support vectors for which $0 < a_n < C$ have $\xi_n = 0$ so that $t_n y(\mathbf{x}_n) = 1$ and hence will satisfy

$$t_n \left(\sum_{m \in \mathcal{S}} a_m t_m k(\mathbf{x}_n, \mathbf{x}_m) + b \right) = 1. \tag{7.36}$$

Again, a numerically stable solution is obtained by averaging to give

$$b = \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left(t_n - \sum_{m \in \mathcal{S}} a_m t_m k(\mathbf{x}_n, \mathbf{x}_m) \right) \tag{7.37}$$

where $\mathcal{M}$ denotes the set of indices of data points having $0 < a_n < C$.

An alternative, equivalent formulation of the support vector machine, known as the *ν-SVM*, has been proposed by Schölkopf *et al.* (2000). This involves maximizing

$$\widetilde{L}(\mathbf{a}) = -\frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m t_n t_m k(\mathbf{x}_n, \mathbf{x}_m) \tag{7.38}$$

subject to the constraints

$$0 \leqslant a_n \leqslant 1/N \tag{7.39}$$

$$\sum_{n=1}^{N} a_n t_n = 0 \tag{7.40}$$

$$\sum_{n=1}^{N} a_n \geqslant \nu. \tag{7.41}$$

This approach has the advantage that the parameter ν, which replaces C, can be interpreted as both an upper bound on the fraction of *margin errors* (points for which $\xi_n > 0$ and hence which lie on the wrong side of the margin boundary and which may or may not be misclassified) and a lower bound on the fraction of support vectors. An example of the ν-SVM applied to a synthetic data set is shown in Figure 7.4. Here Gaussian kernels of the form $\exp(-\gamma\|\mathbf{x} - \mathbf{x}'\|^2)$ have been used, with $\gamma = 0.45$.

Although predictions for new inputs are made using only the support vectors, the training phase (i.e., the determination of the parameters $\mathbf{a}$ and b) makes use of the whole data set, and so it is important to have efficient algorithms for solving

Figure 7.4 Illustration of the ν-SVM applied to a nonseparable data set in two dimensions. The support vectors are indicated by circles.

the quadratic programming problem. We first note that the objective function $\widetilde{L}(\mathbf{a})$ given by (7.10) or (7.32) is quadratic and so any local optimum will also be a global optimum provided the constraints define a convex region (which they do as a consequence of being linear). Direct solution of the quadratic programming problem using traditional techniques is often infeasible due to the demanding computation and memory requirements, and so more practical approaches need to be found. The technique of *chunking* (Vapnik, 1982) exploits the fact that the value of the Lagrangian is unchanged if we remove the rows and columns of the kernel matrix corresponding to Lagrange multipliers that have value zero. This allows the full quadratic programming problem to be broken down into a series of smaller ones, whose goal is eventually to identify all of the nonzero Lagrange multipliers and discard the others. Chunking can be implemented using *protected conjugate gradients* (Burges, 1998). Although chunking reduces the size of the matrix in the quadratic function from the number of data points squared to approximately the number of nonzero Lagrange multipliers squared, even this may be too big to fit in memory for large-scale applications. *Decomposition methods* (Osuna *et al.*, 1996) also solve a series of smaller quadratic programming problems but are designed so that each of these is of a fixed size, and so the technique can be applied to arbitrarily large data sets. However, it still involves numerical solution of quadratic programming subproblems and these can be problematic and expensive. One of the most popular approaches to training support vector machines is called *sequential minimal optimization*, or *SMO* (Platt, 1999). It takes the concept of chunking to the extreme limit and considers just two Lagrange multipliers at a time. In this case, the subproblem can be solved analytically, thereby avoiding numerical quadratic programming altogether. Heuristics are given for choosing the pair of Lagrange multipliers to be considered at each step. In practice, SMO is found to have a scaling with the number of data points that is somewhere between linear and quadratic depending on the particular application.

We have seen that kernel functions correspond to inner products in feature spaces that can have high, or even infinite, dimensionality. By working directly in terms of the kernel function, without introducing the feature space explicitly, it might therefore seem that support vector machines somehow manage to avoid the curse of di-

Section 1.4

mensionality. This is not the case, however, because there are constraints amongst the feature values that restrict the effective dimensionality of feature space. To see this consider a simple second-order polynomial kernel that we can expand in terms of its components

$$\begin{aligned} k(\mathbf{x}, \mathbf{z}) &= \left(1 + \mathbf{x}^{\mathrm{T}}\mathbf{z}\right)^2 = (1 + x_1 z_1 + x_2 z_2)^2 \\ &= 1 + 2x_1 z_1 + 2x_2 z_2 + x_1^2 z_1^2 + 2x_1 z_1 x_2 z_2 + x_2^2 z_2^2 \\ &= (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1 x_2, x_2^2)(1, \sqrt{2}z_1, \sqrt{2}z_2, z_1^2, \sqrt{2}z_1 z_2, z_2^2)^{\mathrm{T}} \\ &= \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{z}). \end{aligned} \tag{7.42}$$

This kernel function therefore represents an inner product in a feature space having six dimensions, in which the mapping from input space to feature space is described by the vector function $\boldsymbol{\phi}(\mathbf{x})$. However, the coefficients weighting these different features are constrained to have specific forms. Thus any set of points in the original two-dimensional space $\mathbf{x}$ would be constrained to lie exactly on a two-dimensional nonlinear manifold embedded in the six-dimensional feature space.

We have already highlighted the fact that the support vector machine does not provide probabilistic outputs but instead makes classification decisions for new input vectors. Veropoulos *et al.* (1999) discuss modifications to the SVM to allow the trade-off between false positive and false negative errors to be controlled. However, if we wish to use the SVM as a module in a larger probabilistic system, then probabilistic predictions of the class label t for new inputs $\mathbf{x}$ are required.

To address this issue, Platt (2000) has proposed fitting a logistic sigmoid to the outputs of a previously trained support vector machine. Specifically, the required conditional probability is assumed to be of the form

$$p(t = 1|\mathbf{x}) = \sigma\left(Ay(\mathbf{x}) + B\right) \tag{7.43}$$

where $y(\mathbf{x})$ is defined by (7.1). Values for the parameters A and B are found by minimizing the cross-entropy error function defined by a training set consisting of pairs of values $y(\mathbf{x}_n)$ and t_n. The data used to fit the sigmoid needs to be independent of that used to train the original SVM in order to avoid severe over-fitting. This two-stage approach is equivalent to assuming that the output $y(\mathbf{x})$ of the support vector machine represents the log-odds of $\mathbf{x}$ belonging to class $t = 1$. Because the SVM training procedure is not specifically intended to encourage this, the SVM can give a poor approximation to the posterior probabilities (Tipping, 2001).

7.1.2 Relation to logistic regression

As with the separable case, we can re-cast the SVM for nonseparable distributions in terms of the minimization of a regularized error function. This will also allow us to highlight similarities, and differences, compared to the logistic regression model.

Section 4.3.2

We have seen that for data points that are on the correct side of the margin boundary, and which therefore satisfy $y_n t_n \geqslant 1$, we have $\xi_n = 0$, and for the

Figure 7.5 Plot of the 'hinge' error function used in support vector machines, shown in blue, along with the error function for logistic regression, rescaled by a factor of $1/\ln(2)$ so that it passes through the point $(0, 1)$, shown in red. Also shown are the misclassification error in black and the squared error in green.

remaining points we have $\xi_n = 1 - y_n t_n$. Thus the objective function (7.21) can be written (up to an overall multiplicative constant) in the form

$$\sum_{n=1}^{N} E_{\mathrm{SV}}(y_n t_n) + \lambda \|\mathbf{w}\|^2 \tag{7.44}$$

where $\lambda = (2C)^{-1}$, and $E_{\mathrm{SV}}(\cdot)$ is the *hinge* error function defined by

$$E_{\mathrm{SV}}(y_n t_n) = [1 - y_n t_n]_+ \tag{7.45}$$

where $[\,\cdot\,]_+$ denotes the positive part. The hinge error function, so-called because of its shape, is plotted in Figure 7.5. It can be viewed as an approximation to the misclassification error, i.e., the error function that ideally we would like to minimize, which is also shown in Figure 7.5.

When we considered the logistic regression model in Section 4.3.2, we found it convenient to work with target variable $t \in \{0, 1\}$. For comparison with the support vector machine, we first reformulate maximum likelihood logistic regression using the target variable $t \in \{-1, 1\}$. To do this, we note that $p(t = 1|y) = \sigma(y)$ where $y(\mathbf{x})$ is given by (7.1), and $\sigma(y)$ is the logistic sigmoid function defined by (4.59). It follows that $p(t = -1|y) = 1 - \sigma(y) = \sigma(-y)$, where we have used the properties of the logistic sigmoid function, and so we can write

$$p(t|y) = \sigma(yt). \tag{7.46}$$

From this we can construct an error function by taking the negative logarithm of the likelihood function that, with a quadratic regularizer, takes the form

Exercise 7.6

$$\sum_{n=1}^{N} E_{\mathrm{LR}}(y_n t_n) + \lambda \|\mathbf{w}\|^2. \tag{7.47}$$

where

$$E_{\mathrm{LR}}(yt) = \ln\left(1 + \exp(-yt)\right). \tag{7.48}$$

For comparison with other error functions, we can divide by $\ln(2)$ so that the error function passes through the point $(0, 1)$. This rescaled error function is also plotted in Figure 7.5 and we see that it has a similar form to the support vector error function. The key difference is that the flat region in $E_{\mathrm{SV}}(yt)$ leads to sparse solutions.

Both the logistic error and the hinge loss can be viewed as continuous approximations to the misclassification error. Another continuous error function that has sometimes been used to solve classification problems is the squared error, which is again plotted in Figure 7.5. It has the property, however, of placing increasing emphasis on data points that are correctly classified but that are a long way from the decision boundary on the correct side. Such points will be strongly weighted at the expense of misclassified points, and so if the objective is to minimize the misclassification rate, then a monotonically decreasing error function would be a better choice.

7.1.3 Multiclass SVMs

The support vector machine is fundamentally a two-class classifier. In practice, however, we often have to tackle problems involving $K > 2$ classes. Various methods have therefore been proposed for combining multiple two-class SVMs in order to build a multiclass classifier.

One commonly used approach (Vapnik, 1998) is to construct K separate SVMs, in which the kth model $y_k(\mathbf{x})$ is trained using the data from class $\mathcal{C}_k$ as the positive examples and the data from the remaining $K - 1$ classes as the negative examples. This is known as the *one-versus-the-rest* approach. However, in Figure 4.2 we saw that using the decisions of the individual classifiers can lead to inconsistent results in which an input is assigned to multiple classes simultaneously. This problem is sometimes addressed by making predictions for new inputs $\mathbf{x}$ using

$$y(\mathbf{x}) = \max_k y_k(\mathbf{x}). \tag{7.49}$$

Unfortunately, this heuristic approach suffers from the problem that the different classifiers were trained on different tasks, and there is no guarantee that the real-valued quantities $y_k(\mathbf{x})$ for different classifiers will have appropriate scales.

Another problem with the one-versus-the-rest approach is that the training sets are imbalanced. For instance, if we have ten classes each with equal numbers of training data points, then the individual classifiers are trained on data sets comprising 90% negative examples and only 10% positive examples, and the symmetry of the original problem is lost. A variant of the one-versus-the-rest scheme was proposed by Lee *et al.* (2001) who modify the target values so that the positive class has target $+1$ and the negative class has target $-1/(K-1)$.

Weston and Watkins (1999) define a single objective function for training all K SVMs simultaneously, based on maximizing the margin from each to remaining classes. However, this can result in much slower training because, instead of solving K separate optimization problems each over N data points with an overall cost of $O(KN^2)$, a single optimization problem of size $(K-1)N$ must be solved giving an overall cost of $O(K^2N^2)$.

Another approach is to train $K(K-1)/2$ different 2-class SVMs on all possible pairs of classes, and then to classify test points according to which class has the highest number of 'votes', an approach that is sometimes called *one-versus-one*. Again, we saw in Figure 4.2 that this can lead to ambiguities in the resulting classification. Also, for large K this approach requires significantly more training time than the one-versus-the-rest approach. Similarly, to evaluate test points, significantly more computation is required.

The latter problem can be alleviated by organizing the pairwise classifiers into a directed acyclic graph (not to be confused with a probabilistic graphical model) leading to the *DAGSVM* (Platt *et al.*, 2000). For K classes, the DAGSVM has a total of $K(K-1)/2$ classifiers, and to classify a new test point only $K-1$ pairwise classifiers need to be evaluated, with the particular classifiers used depending on which path through the graph is traversed.

A different approach to multiclass classification, based on error-correcting output codes, was developed by Dietterich and Bakiri (1995) and applied to support vector machines by Allwein *et al.* (2000). This can be viewed as a generalization of the voting scheme of the one-versus-one approach in which more general partitions of the classes are used to train the individual classifiers. The K classes themselves are represented as particular sets of responses from the two-class classifiers chosen, and together with a suitable decoding scheme, this gives robustness to errors and to ambiguity in the outputs of the individual classifiers. Although the application of SVMs to multiclass classification problems remains an open issue, in practice the one-versus-the-rest approach is the most widely used in spite of its ad-hoc formulation and its practical limitations.

There are also *single-class* support vector machines, which solve an unsupervised learning problem related to probability density estimation. Instead of modelling the density of data, however, these methods aim to find a smooth boundary enclosing a region of high density. The boundary is chosen to represent a quantile of the density, that is, the probability that a data point drawn from the distribution will land inside that region is given by a fixed number between 0 and 1 that is specified in advance. This is a more restricted problem than estimating the full density but may be sufficient in specific applications. Two approaches to this problem using support vector machines have been proposed. The algorithm of Schölkopf *et al.* (2001) tries to find a hyperplane that separates all but a fixed fraction ν of the training data from the origin while at the same time maximizing the distance (margin) of the hyperplane from the origin, while Tax and Duin (1999) look for the smallest sphere in feature space that contains all but a fraction ν of the data points. For kernels $k(\mathbf{x}, \mathbf{x}')$ that are functions only of $\mathbf{x} - \mathbf{x}'$, the two algorithms are equivalent.

7.1.4 SVMs for regression

We now extend support vector machines to regression problems while at the same time preserving the property of sparseness. In simple linear regression, we

Section 3.1.4

Figure 7.6 Plot of an ϵ-insensitive error function (in red) in which the error increases linearly with distance beyond the insensitive region. Also shown for comparison is the quadratic error function (in green).

minimize a regularized error function given by

$$\frac{1}{2}\sum_{n=1}^{N}\{y_n - t_n\}^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2. \tag{7.50}$$

To obtain sparse solutions, the quadratic error function is replaced by an ϵ-*insensitive error function* (Vapnik, 1995), which gives zero error if the absolute difference between the prediction $y(\mathbf{x})$ and the target t is less than ϵ where $\epsilon > 0$. A simple example of an ϵ-insensitive error function, having a linear cost associated with errors outside the insensitive region, is given by

$$E_\epsilon(y(\mathbf{x}) - t) = \begin{cases} 0, & \text{if } |y(\mathbf{x}) - t| < \epsilon; \\ |y(\mathbf{x}) - t| - \epsilon, & \text{otherwise} \end{cases} \tag{7.51}$$

and is illustrated in Figure 7.6.

We therefore minimize a regularized error function given by

$$C\sum_{n=1}^{N} E_\epsilon(y(\mathbf{x}_n) - t_n) + \frac{1}{2}\|\mathbf{w}\|^2 \tag{7.52}$$

where $y(\mathbf{x})$ is given by (7.1). By convention the (inverse) regularization parameter, denoted C, appears in front of the error term.

As before, we can re-express the optimization problem by introducing slack variables. For each data point $\mathbf{x}_n$, we now need two slack variables $\xi_n \geqslant 0$ and $\widehat{\xi}_n \geqslant 0$, where $\xi_n > 0$ corresponds to a point for which $t_n > y(\mathbf{x}_n) + \epsilon$, and $\widehat{\xi}_n > 0$ corresponds to a point for which $t_n < y(\mathbf{x}_n) - \epsilon$, as illustrated in Figure 7.7.

The condition for a target point to lie inside the ϵ-tube is that $y_n - \epsilon \leqslant t_n \leqslant y_n + \epsilon$, where $y_n = y(\mathbf{x}_n)$. Introducing the slack variables allows points to lie outside the tube provided the slack variables are nonzero, and the corresponding conditions are

$$t_n \leqslant y(\mathbf{x}_n) + \epsilon + \xi_n \tag{7.53}$$

$$t_n \geqslant y(\mathbf{x}_n) - \epsilon - \widehat{\xi}_n. \tag{7.54}$$

Figure 7.7 Illustration of SVM regression, showing the regression curve together with the ϵ-insensitive 'tube'. Also shown are examples of the slack variables ξ and $\widehat{\xi}$. Points above the ϵ-tube have $\xi > 0$ and $\widehat{\xi} = 0$, points below the ϵ-tube have $\xi = 0$ and $\widehat{\xi} > 0$, and points inside the ϵ-tube have $\xi = \widehat{\xi} = 0$.

The error function for support vector regression can then be written as

$$C\sum_{n=1}^{N}(\xi_n + \widehat{\xi}_n) + \frac{1}{2}\|\mathbf{w}\|^2 \tag{7.55}$$

which must be minimized subject to the constraints $\xi_n \geqslant 0$ and $\widehat{\xi}_n \geqslant 0$ as well as (7.53) and (7.54). This can be achieved by introducing Lagrange multipliers $a_n \geqslant 0$, $\widehat{a}_n \geqslant 0$, $\mu_n \geqslant 0$, and $\widehat{\mu}_n \geqslant 0$ and optimizing the Lagrangian

$$\begin{aligned} L &= C\sum_{n=1}^{N}(\xi_n + \widehat{\xi}_n) + \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{n=1}^{N}(\mu_n\xi_n + \widehat{\mu}_n\widehat{\xi}_n) \\ &\quad - \sum_{n=1}^{N} a_n(\epsilon + \xi_n + y_n - t_n) - \sum_{n=1}^{N}\widehat{a}_n(\epsilon + \widehat{\xi}_n - y_n + t_n). \end{aligned} \tag{7.56}$$

We now substitute for $y(\mathbf{x})$ using (7.1) and then set the derivatives of the Lagrangian with respect to $\mathbf{w}$, b, ξ_n, and $\widehat{\xi}_n$ to zero, giving

$$\frac{\partial L}{\partial \mathbf{w}} = 0 \quad \Rightarrow \quad \mathbf{w} = \sum_{n=1}^{N}(a_n - \widehat{a}_n)\boldsymbol{\phi}(\mathbf{x}_n) \tag{7.57}$$

$$\frac{\partial L}{\partial b} = 0 \quad \Rightarrow \quad \sum_{n=1}^{N}(a_n - \widehat{a}_n) = 0 \tag{7.58}$$

$$\frac{\partial L}{\partial \xi_n} = 0 \quad \Rightarrow \quad a_n + \mu_n = C \tag{7.59}$$

$$\frac{\partial L}{\partial \widehat{\xi}_n} = 0 \quad \Rightarrow \quad \widehat{a}_n + \widehat{\mu}_n = C. \tag{7.60}$$

Exercise 7.7

Using these results to eliminate the corresponding variables from the Lagrangian, we see that the dual problem involves maximizing

$$\widetilde{L}(\mathbf{a}, \widehat{\mathbf{a}}) = -\frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}(a_n - \widehat{a}_n)(a_m - \widehat{a}_m)k(\mathbf{x}_n, \mathbf{x}_m) - \epsilon\sum_{n=1}^{N}(a_n + \widehat{a}_n) + \sum_{n=1}^{N}(a_n - \widehat{a}_n)t_n \tag{7.61}$$

with respect to $\{a_n\}$ and $\{\widehat{a}_n\}$, where we have introduced the kernel $k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}')$. Again, this is a constrained maximization, and to find the constraints we note that $a_n \geqslant 0$ and $\widehat{a}_n \geqslant 0$ are both required because these are Lagrange multipliers. Also $\mu_n \geqslant 0$ and $\widehat{\mu}_n \geqslant 0$ together with (7.59) and (7.60), require $a_n \leqslant C$ and $\widehat{a}_n \leqslant C$, and so again we have the box constraints

$$0 \leqslant a_n \leqslant C \tag{7.62}$$

$$0 \leqslant \widehat{a}_n \leqslant C \tag{7.63}$$

together with the condition (7.58).

Substituting (7.57) into (7.1), we see that predictions for new inputs can be made using

$$y(\mathbf{x}) = \sum_{n=1}^{N}(a_n - \widehat{a}_n)k(\mathbf{x}, \mathbf{x}_n) + b \tag{7.64}$$

which is again expressed in terms of the kernel function.

The corresponding Karush-Kuhn-Tucker (KKT) conditions, which state that at the solution the product of the dual variables and the constraints must vanish, are given by

$$a_n(\epsilon + \xi_n + y_n - t_n) = 0 \tag{7.65}$$

$$\widehat{a}_n(\epsilon + \widehat{\xi}_n - y_n + t_n) = 0 \tag{7.66}$$

$$(C - a_n)\xi_n = 0 \tag{7.67}$$

$$(C - \widehat{a}_n)\widehat{\xi}_n = 0. \tag{7.68}$$

From these we can obtain several useful results. First of all, we note that a coefficient a_n can only be nonzero if $\epsilon + \xi_n + y_n - t_n = 0$, which implies that the data point either lies on the upper boundary of the ϵ-tube ($\xi_n = 0$) or lies above the upper boundary ($\xi_n > 0$). Similarly, a nonzero value for $\widehat{a}_n$ implies $\epsilon + \widehat{\xi}_n - y_n + t_n = 0$, and such points must lie either on or below the lower boundary of the ϵ-tube.

Furthermore, the two constraints $\epsilon + \xi_n + y_n - t_n = 0$ and $\epsilon + \widehat{\xi}_n - y_n + t_n = 0$ are incompatible, as is easily seen by adding them together and noting that ξ_n and $\widehat{\xi}_n$ are nonnegative while ϵ is strictly positive, and so for every data point $\mathbf{x}_n$, either a_n or $\widehat{a}_n$ (or both) must be zero.

The support vectors are those data points that contribute to predictions given by (7.64), in other words those for which either $a_n \neq 0$ or $\widehat{a}_n \neq 0$. These are points that lie on the boundary of the ϵ-tube or outside the tube. All points within the tube have

$a_n = \widehat{a}_n = 0$. We again have a sparse solution, and the only terms that have to be evaluated in the predictive model (7.64) are those that involve the support vectors.

The parameter b can be found by considering a data point for which $0 < a_n < C$, which from (7.67) must have $\xi_n = 0$, and from (7.65) must therefore satisfy $\epsilon + y_n - t_n = 0$. Using (7.1) and solving for b, we obtain

$$\begin{aligned} b &= t_n - \epsilon - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n) \\ &= t_n - \epsilon - \sum_{m=1}^{N}(a_m - \widehat{a}_m)k(\mathbf{x}_n, \mathbf{x}_m) \end{aligned} \tag{7.69}$$

where we have used (7.57). We can obtain an analogous result by considering a point for which $0 < \widehat{a}_n < C$. In practice, it is better to average over all such estimates of b.

As with the classification case, there is an alternative formulation of the SVM for regression in which the parameter governing complexity has a more intuitive interpretation (Schölkopf *et al.*, 2000). In particular, instead of fixing the width ϵ of the insensitive region, we fix instead a parameter ν that bounds the fraction of points lying outside the tube. This involves maximizing

$$\begin{aligned} \widetilde{L}(\mathbf{a}, \widehat{\mathbf{a}}) &= -\frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}(a_n - \widehat{a}_n)(a_m - \widehat{a}_m)k(\mathbf{x}_n, \mathbf{x}_m) \\ &\quad + \sum_{n=1}^{N}(a_n - \widehat{a}_n)t_n \end{aligned} \tag{7.70}$$

subject to the constraints

$$0 \leqslant a_n \leqslant C/N \tag{7.71}$$

$$0 \leqslant \widehat{a}_n \leqslant C/N \tag{7.72}$$

$$\sum_{n=1}^{N}(a_n - \widehat{a}_n) = 0 \tag{7.73}$$

$$\sum_{n=1}^{N}(a_n + \widehat{a}_n) \leqslant \nu C. \tag{7.74}$$

It can be shown that there are at most νN data points falling outside the insensitive tube, while at least νN data points are support vectors and so lie either on the tube or outside it.

The use of a support vector machine to solve a regression problem is illustrated using the sinusoidal data set in Figure 7.8. Here the parameters ν and C have been chosen by hand. In practice, their values would typically be determined by cross-validation.

Appendix A

Figure 7.8 Illustration of the ν-SVM for regression applied to the sinusoidal synthetic data set using Gaussian kernels. The predicted regression curve is shown by the red line, and the ϵ-insensitive tube corresponds to the shaded region. Also, the data points are shown in green, and those with support vectors are indicated by blue circles.

7.1.5 Computational learning theory

Historically, support vector machines have largely been motivated and analysed using a theoretical framework known as *computational learning theory*, also sometimes called *statistical learning theory* (Anthony and Biggs, 1992; Kearns and Vazirani, 1994; Vapnik, 1995; Vapnik, 1998). This has its origins with Valiant (1984) who formulated the *probably approximately correct*, or PAC, learning framework. The goal of the PAC framework is to understand how large a data set needs to be in order to give good generalization. It also gives bounds for the computational cost of learning, although we do not consider these here.

Suppose that a data set $\mathcal{D}$ of size N is drawn from some joint distribution $p(\mathbf{x}, \mathbf{t})$ where $\mathbf{x}$ is the input variable and $\mathbf{t}$ represents the class label, and that we restrict attention to 'noise free' situations in which the class labels are determined by some (unknown) deterministic function $\mathbf{t} = \mathbf{g}(\mathbf{x})$. In PAC learning we say that a function $\mathbf{f}(\mathbf{x}; \mathcal{D})$, drawn from a space $\mathcal{F}$ of such functions on the basis of the training set $\mathcal{D}$, has good generalization if its expected error rate is below some pre-specified threshold ϵ, so that

$$\mathbb{E}_{\mathbf{x},\mathbf{t}} \left[I \left(\mathbf{f}(\mathbf{x}; \mathcal{D}) \neq \mathbf{t} \right) \right] < \epsilon \tag{7.75}$$

where $I(\cdot)$ is the indicator function, and the expectation is with respect to the distribution $p(\mathbf{x}, \mathbf{t})$. The quantity on the left-hand side is a random variable, because it depends on the training set $\mathcal{D}$, and the PAC framework requires that (7.75) holds, with probability greater than $1 - \delta$, for a data set $\mathcal{D}$ drawn randomly from $p(\mathbf{x}, \mathbf{t})$. Here δ is another pre-specified parameter, and the terminology 'probably approximately correct' comes from the requirement that with high probability (greater than $1 - \delta$), the error rate be small (less than ϵ). For a given choice of model space $\mathcal{F}$, and for given parameters ϵ and δ, PAC learning aims to provide bounds on the minimum size N of data set needed to meet this criterion. A key quantity in PAC learning is the *Vapnik-Chervonenkis dimension*, or VC dimension, which provides a measure of the complexity of a space of functions, and which allows the PAC framework to be extended to spaces containing an infinite number of functions.

The bounds derived within the PAC framework are often described as worst-

case, because they apply to *any* choice for the distribution $p(\mathbf{x}, \mathbf{t})$, so long as both the training and the test examples are drawn (independently) from the same distribution, and for *any* choice for the function $\mathbf{f}(\mathbf{x})$ so long as it belongs to $\mathcal{F}$. In real-world applications of machine learning, we deal with distributions that have significant regularity, for example in which large regions of input space carry the same class label. As a consequence of the lack of any assumptions about the form of the distribution, the PAC bounds are very conservative, in other words they strongly over-estimate the size of data sets required to achieve a given generalization performance. For this reason, PAC bounds have found few, if any, practical applications.

One attempt to improve the tightness of the PAC bounds is the *PAC-Bayesian* framework (McAllester, 2003), which considers a distribution over the space $\mathcal{F}$ of functions, somewhat analogous to the prior in a Bayesian treatment. This still considers any possible choice for $p(\mathbf{x}, \mathbf{t})$, and so although the bounds are tighter, they are still very conservative.

7.2. Relevance Vector Machines

Support vector machines have been used in a variety of classification and regression applications. Nevertheless, they suffer from a number of limitations, several of which have been highlighted already in this chapter. In particular, the outputs of an SVM represent decisions rather than posterior probabilities. Also, the SVM was originally formulated for two classes, and the extension to $K > 2$ classes is problematic. There is a complexity parameter C, or ν (as well as a parameter ϵ in the case of regression), that must be found using a hold-out method such as cross-validation. Finally, predictions are expressed as linear combinations of kernel functions that are centred on training data points and that are required to be positive definite.

The *relevance vector machine* or RVM (Tipping, 2001) is a Bayesian sparse kernel technique for regression and classification that shares many of the characteristics of the SVM whilst avoiding its principal limitations. Additionally, it typically leads to much sparser models resulting in correspondingly faster performance on test data whilst maintaining comparable generalization error.

In contrast to the SVM we shall find it more convenient to introduce the regression form of the RVM first and then consider the extension to classification tasks.

7.2.1 RVM for regression

The relevance vector machine for regression is a linear model of the form studied in Chapter 3 but with a modified prior that results in sparse solutions. The model defines a conditional distribution for a real-valued target variable t, given an input vector $\mathbf{x}$, which takes the form

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}), \beta^{-1}) \tag{7.76}$$

where $\beta = \sigma^{-2}$ is the noise precision (inverse noise variance), and the mean is given by a linear model of the form

$$y(\mathbf{x}) = \sum_{i=1}^{M} w_i \phi_i(\mathbf{x}) = \mathbf{w}^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}) \tag{7.77}$$

with fixed nonlinear basis functions $\phi_i(\mathbf{x})$, which will typically include a constant term so that the corresponding weight parameter represents a 'bias'.

The relevance vector machine is a specific instance of this model, which is intended to mirror the structure of the support vector machine. In particular, the basis functions are given by kernels, with one kernel associated with each of the data points from the training set. The general expression (7.77) then takes the SVM-like form

$$y(\mathbf{x}) = \sum_{n=1}^{N} w_n k(\mathbf{x}, \mathbf{x}_n) + b \tag{7.78}$$

where b is a bias parameter. The number of parameters in this case is $M = N + 1$, and $y(\mathbf{x})$ has the same form as the predictive model (7.64) for the SVM, except that the coefficients a_n are here denoted w_n. It should be emphasized that the subsequent analysis is valid for arbitrary choices of basis function, and for generality we shall work with the form (7.77). In contrast to the SVM, there is no restriction to positive-definite kernels, nor are the basis functions tied in either number or location to the training data points.

Suppose we are given a set of N observations of the input vector $\mathbf{x}$, which we denote collectively by a data matrix $\mathbf{X}$ whose n^{th} row is $\mathbf{x}_n^{\mathrm{T}}$ with $n = 1, \ldots, N$. The corresponding target values are given by $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$. Thus, the likelihood function is given by

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} p(t_n|\mathbf{x}_n, \mathbf{w}, \beta^{-1}). \tag{7.79}$$

Next we introduce a prior distribution over the parameter vector $\mathbf{w}$ and as in Chapter 3, we shall consider a zero-mean Gaussian prior. However, the key difference in the RVM is that we introduce a separate hyperparameter α_i for each of the weight parameters w_i instead of a single shared hyperparameter. Thus the weight prior takes the form

$$p(\mathbf{w}|\boldsymbol{\alpha}) = \prod_{i=1}^{M} \mathcal{N}(w_i|0, \alpha_i^{-1}) \tag{7.80}$$

where α_i represents the precision of the corresponding parameter w_i, and $\boldsymbol{\alpha}$ denotes $(\alpha_1, \ldots, \alpha_M)^{\mathrm{T}}$. We shall see that, when we maximize the evidence with respect to these hyperparameters, a significant proportion of them go to infinity, and the corresponding weight parameters have posterior distributions that are concentrated at zero. The basis functions associated with these parameters therefore play no role

in the predictions made by the model and so are effectively pruned out, resulting in a sparse model.

Using the result (3.49) for linear regression models, we see that the posterior distribution for the weights is again Gaussian and takes the form

$$p(\mathbf{w}|\mathbf{t}, \mathbf{X}, \boldsymbol{\alpha}, \beta) = \mathcal{N}(\mathbf{w}|\mathbf{m}, \boldsymbol{\Sigma}) \tag{7.81}$$

where the mean and covariance are given by

$$\mathbf{m} = \beta \boldsymbol{\Sigma} \boldsymbol{\Phi}^{\mathrm{T}} \mathbf{t} \tag{7.82}$$

$$\boldsymbol{\Sigma} = \left(\mathbf{A} + \beta \boldsymbol{\Phi}^{\mathrm{T}} \boldsymbol{\Phi}\right)^{-1} \tag{7.83}$$

where $\boldsymbol{\Phi}$ is the $N \times M$ design matrix with elements $\Phi_{ni} = \phi_i(\mathbf{x}_n)$, and $\mathbf{A} = \mathrm{diag}(\alpha_i)$. Note that in the specific case of the model (7.78), we have $\boldsymbol{\Phi} = \mathbf{K}$, where $\mathbf{K}$ is the symmetric $(N+1) \times (N+1)$ kernel matrix with elements $k(\mathbf{x}_n, \mathbf{x}_m)$.

Section 3.5 The values of $\boldsymbol{\alpha}$ and β are determined using type-2 maximum likelihood, also known as the *evidence approximation*, in which we maximize the marginal likelihood function obtained by integrating out the weight parameters

$$p(\mathbf{t}|\mathbf{X}, \boldsymbol{\alpha}, \beta) = \int p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) p(\mathbf{w}|\boldsymbol{\alpha}) \, \mathrm{d}\mathbf{w}. \tag{7.84}$$

Exercise 7.10 Because this represents the convolution of two Gaussians, it is readily evaluated to give the log marginal likelihood in the form

$$\begin{aligned} \ln p(\mathbf{t}|\mathbf{X}, \boldsymbol{\alpha}, \beta) &= \ln \mathcal{N}(\mathbf{t}|\mathbf{0}, \mathbf{C}) \\ &= -\frac{1}{2} \left\{ N \ln(2\pi) + \ln|\mathbf{C}| + \mathbf{t}^{\mathrm{T}} \mathbf{C}^{-1} \mathbf{t} \right\} \end{aligned} \tag{7.85}$$

where $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$, and we have defined the $N \times N$ matrix $\mathbf{C}$ given by

$$\mathbf{C} = \beta^{-1} \mathbf{I} + \boldsymbol{\Phi} \mathbf{A}^{-1} \boldsymbol{\Phi}^{\mathrm{T}}. \tag{7.86}$$

Our goal is now to maximize (7.85) with respect to the hyperparameters $\boldsymbol{\alpha}$ and β. This requires only a small modification to the results obtained in Section 3.5 for the evidence approximation in the linear regression model. Again, we can identify two approaches. In the first, we simply set the required derivatives of the marginal likelihood to zero and obtain the following re-estimation equations *Exercise 7.12*

$$\alpha_i^{\mathrm{new}} = \frac{\gamma_i}{m_i^2} \tag{7.87}$$

$$(\beta^{\mathrm{new}})^{-1} = \frac{\|\mathbf{t} - \boldsymbol{\Phi}\mathbf{m}\|^2}{N - \sum_i \gamma_i} \tag{7.88}$$

where m_i is the ith component of the posterior mean $\mathbf{m}$ defined by (7.82). The quantity γ_i measures how well the corresponding parameter w_i is determined by the data and is defined by *Section 3.5.3*

$$\gamma_i = 1 - \alpha_i \Sigma_{ii} \tag{7.89}$$

in which Σ_{ii} is the ith diagonal component of the posterior covariance $\boldsymbol{\Sigma}$ given by (7.83). Learning therefore proceeds by choosing initial values for $\boldsymbol{\alpha}$ and β, evaluating the mean and covariance of the posterior using (7.82) and (7.83), respectively, and then alternately re-estimating the hyperparameters, using (7.87) and (7.88), and re-estimating the posterior mean and covariance, using (7.82) and (7.83), until a suitable convergence criterion is satisfied.

The second approach is to use the EM algorithm, and is discussed in Section 9.3.4. These two approaches to finding the values of the hyperparameters that maximize the evidence are formally equivalent. Numerically, however, it is found that the direct optimization approach corresponding to (7.87) and (7.88) gives somewhat faster convergence (Tipping, 2001).

Exercise 9.23

As a result of the optimization, we find that a proportion of the hyperparameters $\{\alpha_i\}$ are driven to large (in principle infinite) values, and so the weight parameters w_i corresponding to these hyperparameters have posterior distributions with mean and variance both zero. Thus those parameters, and the corresponding basis functions $\phi_i(\mathbf{x})$, are removed from the model and play no role in making predictions for new inputs. In the case of models of the form (7.78), the inputs $\mathbf{x}_n$ corresponding to the remaining nonzero weights are called *relevance vectors*, because they are identified through the mechanism of automatic relevance determination, and are analogous to the support vectors of an SVM. It is worth emphasizing, however, that this mechanism for achieving sparsity in probabilistic models through automatic relevance determination is quite general and can be applied to any model expressed as an adaptive linear combination of basis functions.

Section 7.2.2

Having found values $\boldsymbol{\alpha}^\star$ and $\beta^\star$ for the hyperparameters that maximize the marginal likelihood, we can evaluate the predictive distribution over t for a new input $\mathbf{x}$. Using (7.76) and (7.81), this is given by

Exercise 7.14

$$\begin{aligned} p(t|\mathbf{x}, \mathbf{X}, \mathbf{t}, \boldsymbol{\alpha}^\star, \beta^\star) &= \int p(t|\mathbf{x}, \mathbf{w}, \beta^\star) p(\mathbf{w}|\mathbf{X}, \mathbf{t}, \boldsymbol{\alpha}^\star, \beta^\star)\, \mathrm{d}\mathbf{w} \\ &= \mathcal{N}\left(t|\mathbf{m}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}), \sigma^2(\mathbf{x})\right). \end{aligned} \tag{7.90}$$

Thus the predictive mean is given by (7.76) with $\mathbf{w}$ set equal to the posterior mean $\mathbf{m}$, and the variance of the predictive distribution is given by

$$\sigma^2(\mathbf{x}) = (\beta^\star)^{-1} + \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}} \boldsymbol{\Sigma} \boldsymbol{\phi}(\mathbf{x}) \tag{7.91}$$

where $\boldsymbol{\Sigma}$ is given by (7.83) in which $\boldsymbol{\alpha}$ and β are set to their optimized values $\boldsymbol{\alpha}^\star$ and $\beta^\star$. This is just the familiar result (3.59) obtained in the context of linear regression. Recall that for localized basis functions, the predictive variance for linear regression models becomes small in regions of input space where there are no basis functions. In the case of an RVM with the basis functions centred on data points, the model will therefore become increasingly certain of its predictions when extrapolating outside the domain of the data (Rasmussen and Quiñonero-Candela, 2005), which of course is undesirable. The predictive distribution in Gaussian process regression does not

Section 6.4.2

Figure 7.9 Illustration of RVM regression using the same data set, and the same Gaussian kernel functions, as used in Figure 7.8 for the ν-SVM regression model. The mean of the predictive distribution for the RVM is shown by the red line, and the one standard-deviation predictive distribution is shown by the shaded region. Also, the data points are shown in green, and the relevance vectors are indicated by blue circles. Note that there are only 3 relevance vectors compared to 7 support vectors for the ν-SVM in Figure 7.8.

suffer from this problem. However, the computational cost of making predictions with a Gaussian processes is typically much higher than with an RVM.

Figure 7.9 shows an example of the RVM applied to the sinusoidal regression data set. Here the noise precision parameter β is also determined through evidence maximization. We see that the number of relevance vectors in the RVM is significantly smaller than the number of support vectors used by the SVM. For a wide range of regression and classification tasks, the RVM is found to give models that are typically an order of magnitude more compact than the corresponding support vector machine, resulting in a significant improvement in the speed of processing on test data. Remarkably, this greater sparsity is achieved with little or no reduction in generalization error compared with the corresponding SVM.

The principal disadvantage of the RVM compared to the SVM is that training involves optimizing a nonconvex function, and training times can be longer than for a comparable SVM. For a model with M basis functions, the RVM requires inversion of a matrix of size $M \times M$, which in general requires $O(M^3)$ computation. In the specific case of the SVM-like model (7.78), we have $M = N+1$. As we have noted, there are techniques for training SVMs whose cost is roughly quadratic in N. Of course, in the case of the RVM we always have the option of starting with a smaller number of basis functions than $N + 1$. More significantly, in the relevance vector machine the parameters governing complexity and noise variance are determined automatically from a single training run, whereas in the support vector machine the parameters C and ϵ (or ν) are generally found using cross-validation, which involves multiple training runs. Furthermore, in the next section we shall derive an alternative procedure for training the relevance vector machine that improves training speed significantly.

7.2.2 Analysis of sparsity

We have noted earlier that the mechanism of *automatic relevance determination* causes a subset of parameters to be driven to zero. We now examine in more detail

Figure 7.10 Illustration of the mechanism for sparsity in a Bayesian linear regression model, showing a training set vector of target values given by $\mathbf{t} = (t_1, t_2)^{\mathrm{T}}$, indicated by the cross, for a model with one basis vector $\boldsymbol{\varphi} = (\phi(\mathbf{x}_1), \phi(\mathbf{x}_2))^{\mathrm{T}}$, which is poorly aligned with the target data vector $\mathbf{t}$. On the left we see a model having only isotropic noise, so that $\mathbf{C} = \beta^{-1}\mathbf{I}$, corresponding to $\alpha = \infty$, with β set to its most probable value. On the right we see the same model but with a finite value of α. In each case the red ellipse corresponds to unit Mahalanobis distance, with $|\mathbf{C}|$ taking the same value for both plots, while the dashed green circle shows the contrition arising from the noise term β^{-1}. We see that any finite value of α reduces the probability of the observed data, and so for the most probable solution the basis vector is removed.

the mechanism of sparsity in the context of the relevance vector machine. In the process, we will arrive at a significantly faster procedure for optimizing the hyperparameters compared to the direct techniques given above.

Before proceeding with a mathematical analysis, we first give some informal insight into the origin of sparsity in Bayesian linear models. Consider a data set comprising $N = 2$ observations t_1 and t_2, together with a model having a single basis function $\phi(\mathbf{x})$, with hyperparameter α, along with isotropic noise having precision β. From (7.85), the marginal likelihood is given by $p(\mathbf{t}|\alpha, \beta) = \mathcal{N}(\mathbf{t}|\mathbf{0}, \mathbf{C})$ in which the covariance matrix takes the form

$$\mathbf{C} = \frac{1}{\beta}\mathbf{I} + \frac{1}{\alpha}\boldsymbol{\varphi}\boldsymbol{\varphi}^{\mathrm{T}} \tag{7.92}$$

where $\boldsymbol{\varphi}$ denotes the N-dimensional vector $(\phi(\mathbf{x}_1), \phi(\mathbf{x}_2))^{\mathrm{T}}$, and similarly $\mathbf{t} = (t_1, t_2)^{\mathrm{T}}$. Notice that this is just a zero-mean Gaussian process model over $\mathbf{t}$ with covariance $\mathbf{C}$. Given a particular observation for $\mathbf{t}$, our goal is to find $\alpha^\star$ and $\beta^\star$ by maximizing the marginal likelihood. We see from Figure 7.10 that, if there is a poor alignment between the direction of $\boldsymbol{\varphi}$ and that of the training data vector $\mathbf{t}$, then the corresponding hyperparameter α will be driven to ∞, and the basis vector will be pruned from the model. This arises because any finite value for α will always assign a lower probability to the data, thereby decreasing the value of the density at $\mathbf{t}$, provided that β is set to its optimal value. We see that any finite value for α would cause the distribution to be elongated in a direction away from the data, thereby increasing the probability mass in regions away from the observed data and hence reducing the value of the density at the target data vector itself. For the more general case of M

basis vectors $\boldsymbol{\varphi}_1, \ldots, \boldsymbol{\varphi}_M$ a similar intuition holds, namely that if a particular basis vector is poorly aligned with the data vector $\mathbf{t}$, then it is likely to be pruned from the model.

We now investigate the mechanism for sparsity from a more mathematical perspective, for a general case involving M basis functions. To motivate this analysis we first note that, in the result (7.87) for re-estimating the parameter α_i, the terms on the right-hand side are themselves also functions of α_i. These results therefore represent implicit solutions, and iteration would be required even to determine a single α_i with all other α_j for $j \neq i$ fixed.

This suggests a different approach to solving the optimization problem for the RVM, in which we make explicit all of the dependence of the marginal likelihood (7.85) on a particular α_i and then determine its stationary points explicitly (Faul and Tipping, 2002; Tipping and Faul, 2003). To do this, we first pull out the contribution from α_i in the matrix $\mathbf{C}$ defined by (7.86) to give

$$\begin{aligned} \mathbf{C} &= \beta^{-1}\mathbf{I} + \sum_{j \neq i} \alpha_j^{-1} \boldsymbol{\varphi}_j \boldsymbol{\varphi}_j^{\mathrm{T}} + \alpha_i^{-1} \boldsymbol{\varphi}_i \boldsymbol{\varphi}_i^{\mathrm{T}} \\ &= \mathbf{C}_{-i} + \alpha_i^{-1} \boldsymbol{\varphi}_i \boldsymbol{\varphi}_i^{\mathrm{T}} \end{aligned} \tag{7.93}$$

where $\boldsymbol{\varphi}_i$ denotes the ith column of $\boldsymbol{\Phi}$, in other words the N-dimensional vector with elements $(\phi_i(\mathbf{x}_1), \ldots, \phi_i(\mathbf{x}_N))$, in contrast to $\boldsymbol{\phi}_n$, which denotes the nth row of $\boldsymbol{\Phi}$. The matrix $\mathbf{C}_{-i}$ represents the matrix $\mathbf{C}$ with the contribution from basis function i removed. Using the matrix identities (C.7) and (C.15), the determinant and inverse of $\mathbf{C}$ can then be written

$$|\mathbf{C}| = |\mathbf{C}_{-i}||1 + \alpha_i^{-1} \boldsymbol{\varphi}_i^{\mathrm{T}} \mathbf{C}_{-i}^{-1} \boldsymbol{\varphi}_i| \tag{7.94}$$

$$\mathbf{C}^{-1} = \mathbf{C}_{-i}^{-1} - \frac{\mathbf{C}_{-i}^{-1} \boldsymbol{\varphi}_i \boldsymbol{\varphi}_i^{\mathrm{T}} \mathbf{C}_{-i}^{-1}}{\alpha_i + \boldsymbol{\varphi}_i^{\mathrm{T}} \mathbf{C}_{-i}^{-1} \boldsymbol{\varphi}_i}. \tag{7.95}$$

Exercise 7.15

Using these results, we can then write the log marginal likelihood function (7.85) in the form

$$L(\boldsymbol{\alpha}) = L(\boldsymbol{\alpha}_{-i}) + \lambda(\alpha_i) \tag{7.96}$$

where $L(\boldsymbol{\alpha}_{-i})$ is simply the log marginal likelihood with basis function $\boldsymbol{\varphi}_i$ omitted, and the quantity $\lambda(\alpha_i)$ is defined by

$$\lambda(\alpha_i) = \frac{1}{2}\left[\ln \alpha_i - \ln(\alpha_i + s_i) + \frac{q_i^2}{\alpha_i + s_i}\right] \tag{7.97}$$

and contains all of the dependence on α_i. Here we have introduced the two quantities

$$s_i = \boldsymbol{\varphi}_i^{\mathrm{T}} \mathbf{C}_{-i}^{-1} \boldsymbol{\varphi}_i \tag{7.98}$$

$$q_i = \boldsymbol{\varphi}_i^{\mathrm{T}} \mathbf{C}_{-i}^{-1} \mathbf{t}. \tag{7.99}$$

Here s_i is called the *sparsity* and q_i is known as the *quality* of $\boldsymbol{\varphi}_i$, and as we shall see, a large value of s_i relative to the value of q_i means that the basis function $\boldsymbol{\varphi}_i$

Figure 7.11 Plots of the log marginal likelihood $\lambda(\alpha_i)$ versus $\ln \alpha_i$ showing on the left, the single maximum at a finite α_i for $q_i^2 = 4$ and $s_i = 1$ (so that $q_i^2 > s_i$) and on the right, the maximum at $\alpha_i = \infty$ for $q_i^2 = 1$ and $s_i = 2$ (so that $q_i^2 < s_i$).

is more likely to be pruned from the model. The 'sparsity' measures the extent to which basis function $\boldsymbol{\varphi}_i$ overlaps with the other basis vectors in the model, and the 'quality' represents a measure of the alignment of the basis vector $\boldsymbol{\varphi}_n$ with the error between the training set values $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$ and the vector $\mathbf{y}_{-i}$ of predictions that would result from the model with the vector $\boldsymbol{\varphi}_i$ excluded (Tipping and Faul, 2003).

The stationary points of the marginal likelihood with respect to α_i occur when the derivative

$$\frac{\mathrm{d}\lambda(\alpha_i)}{\mathrm{d}\alpha_i} = \frac{\alpha_i^{-1} s_i^2 - (q_i^2 - s_i)}{2(\alpha_i + s_i)^2} \tag{7.100}$$

is equal to zero. There are two possible forms for the solution. Recalling that $\alpha_i \geqslant 0$, we see that if $q_i^2 < s_i$, then $\alpha_i \to \infty$ provides a solution. Conversely, if $q_i^2 > s_i$, we can solve for α_i to obtain

$$\alpha_i = \frac{s_i^2}{q_i^2 - s_i}. \tag{7.101}$$

These two solutions are illustrated in Figure 7.11. We see that the relative size of the quality and sparsity terms determines whether a particular basis vector will be pruned from the model or not. A more complete analysis (Faul and Tipping, 2002), based on the second derivatives of the marginal likelihood, confirms these solutions are indeed the unique maxima of $\lambda(\alpha_i)$.

Exercise 7.16

Note that this approach has yielded a closed-form solution for α_i, for given values of the other hyperparameters. As well as providing insight into the origin of sparsity in the RVM, this analysis also leads to a practical algorithm for optimizing the hyperparameters that has significant speed advantages. This uses a fixed set of candidate basis vectors, and then cycles through them in turn to decide whether each vector should be included in the model or not. The resulting sequential sparse Bayesian learning algorithm is described below.

Sequential Sparse Bayesian Learning Algorithm

1. If solving a regression problem, initialize β.
2. Initialize using one basis function $\boldsymbol{\varphi}_1$, with hyperparameter α_1 set using (7.101), with the remaining hyperparameters α_j for $j \neq i$ initialized to infinity, so that only $\boldsymbol{\varphi}_1$ is included in the model.

3. Evaluate $\boldsymbol{\Sigma}$ and $\mathbf{m}$, along with q_i and s_i for all basis functions.
4. Select a candidate basis function $\boldsymbol{\varphi}_i$.
5. If $q_i^2 > s_i$, and $\alpha_i < \infty$, so that the basis vector $\boldsymbol{\varphi}_i$ is already included in the model, then update α_i using (7.101).
6. If $q_i^2 > s_i$, and $\alpha_i = \infty$, then add $\boldsymbol{\varphi}_i$ to the model, and evaluate hyperparameter α_i using (7.101).
7. If $q_i^2 \leqslant s_i$, and $\alpha_i < \infty$ then remove basis function $\boldsymbol{\varphi}_i$ from the model, and set $\alpha_i = \infty$.
8. If solving a regression problem, update β.
9. If converged terminate, otherwise go to 3.

Note that if $q_i^2 \leqslant s_i$ and $\alpha_i = \infty$, then the basis function $\boldsymbol{\varphi}_i$ is already excluded from the model and no action is required.

In practice, it is convenient to evaluate the quantities

$$Q_i = \boldsymbol{\varphi}_i^{\mathrm{T}} \mathbf{C}^{-1} \mathbf{t} \tag{7.102}$$

$$S_i = \boldsymbol{\varphi}_i^{\mathrm{T}} \mathbf{C}^{-1} \boldsymbol{\varphi}_i. \tag{7.103}$$

The quality and sparseness variables can then be expressed in the form

$$q_i = \frac{\alpha_i Q_i}{\alpha_i - S_i} \tag{7.104}$$

$$s_i = \frac{\alpha_i S_i}{\alpha_i - S_i}. \tag{7.105}$$

Exercise 7.17 Note that when $\alpha_i = \infty$, we have $q_i = Q_i$ and $s_i = S_i$. Using (C.7), we can write

$$Q_i = \beta \boldsymbol{\varphi}_i^{\mathrm{T}} \mathbf{t} - \beta^2 \boldsymbol{\varphi}_i^{\mathrm{T}} \boldsymbol{\Phi} \boldsymbol{\Sigma} \boldsymbol{\Phi}^{\mathrm{T}} \mathbf{t} \tag{7.106}$$

$$S_i = \beta \boldsymbol{\varphi}_i^{\mathrm{T}} \boldsymbol{\varphi}_i - \beta^2 \boldsymbol{\varphi}_i^{\mathrm{T}} \boldsymbol{\Phi} \boldsymbol{\Sigma} \boldsymbol{\Phi}^{\mathrm{T}} \boldsymbol{\varphi}_i \tag{7.107}$$

where $\boldsymbol{\Phi}$ and $\boldsymbol{\Sigma}$ involve only those basis vectors that correspond to finite hyperparameters α_i. At each stage the required computations therefore scale like $O(M^3)$, where M is the number of active basis vectors in the model and is typically much smaller than the number N of training patterns.

7.2.3 RVM for classification

We can extend the relevance vector machine framework to classification problems by applying the ARD prior over weights to a probabilistic linear classification model of the kind studied in Chapter 4. To start with, we consider two-class problems with a binary target variable $t \in \{0, 1\}$. The model now takes the form of a linear combination of basis functions transformed by a logistic sigmoid function

$$y(\mathbf{x}, \mathbf{w}) = \sigma\left(\mathbf{w}^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x})\right) \tag{7.108}$$

where $\sigma(\cdot)$ is the logistic sigmoid function defined by (4.59). If we introduce a Gaussian prior over the weight vector $\mathbf{w}$, then we obtain the model that has been considered already in Chapter 4. The difference here is that in the RVM, this model uses the ARD prior (7.80) in which there is a separate precision hyperparameter associated with each weight parameter.

In contrast to the regression model, we can no longer integrate analytically over the parameter vector $\mathbf{w}$. Here we follow Tipping (2001) and use the Laplace approximation, which was applied to the closely related problem of Bayesian logistic regression in Section 4.5.1.

Section 4.4

We begin by initializing the hyperparameter vector $\boldsymbol{\alpha}$. For this given value of $\boldsymbol{\alpha}$, we then build a Gaussian approximation to the posterior distribution and thereby obtain an approximation to the marginal likelihood. Maximization of this approximate marginal likelihood then leads to a re-estimated value for $\boldsymbol{\alpha}$, and the process is repeated until convergence.

Let us consider the Laplace approximation for this model in more detail. For a fixed value of $\boldsymbol{\alpha}$, the mode of the posterior distribution over $\mathbf{w}$ is obtained by maximizing

$$\begin{aligned}\ln p(\mathbf{w}|\mathbf{t},\boldsymbol{\alpha}) &= \ln\{p(\mathbf{t}|\mathbf{w})p(\mathbf{w}|\boldsymbol{\alpha})\} - \ln p(\mathbf{t}|\boldsymbol{\alpha}) \\ &= \sum_{n=1}^{N}\{t_n \ln y_n + (1-t_n)\ln(1-y_n)\} - \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{A}\mathbf{w} + \text{const} \qquad (7.109)\end{aligned}$$

where $\mathbf{A} = \mathrm{diag}(\alpha_i)$. This can be done using iterative reweighted least squares (IRLS) as discussed in Section 4.3.3. For this, we need the gradient vector and Hessian matrix of the log posterior distribution, which from (7.109) are given by

Exercise 7.18

$$\nabla \ln p(\mathbf{w}|\mathbf{t},\boldsymbol{\alpha}) = \boldsymbol{\Phi}^{\mathrm{T}}(\mathbf{t}-\mathbf{y}) - \mathbf{A}\mathbf{w} \qquad (7.110)$$

$$\nabla\nabla \ln p(\mathbf{w}|\mathbf{t},\boldsymbol{\alpha}) = -\left(\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{B}\boldsymbol{\Phi} + \mathbf{A}\right) \qquad (7.111)$$

where $\mathbf{B}$ is an $N \times N$ diagonal matrix with elements $b_n = y_n(1-y_n)$, the vector $\mathbf{y} = (y_1, \ldots, y_N)^{\mathrm{T}}$, and $\boldsymbol{\Phi}$ is the design matrix with elements $\Phi_{ni} = \phi_i(\mathbf{x}_n)$. Here we have used the property (4.88) for the derivative of the logistic sigmoid function. At convergence of the IRLS algorithm, the negative Hessian represents the inverse covariance matrix for the Gaussian approximation to the posterior distribution.

The mode of the resulting approximation to the posterior distribution, corresponding to the mean of the Gaussian approximation, is obtained setting (7.110) to zero, giving the mean and covariance of the Laplace approximation in the form

$$\mathbf{w}^{\star} = \mathbf{A}^{-1}\boldsymbol{\Phi}^{\mathrm{T}}(\mathbf{t}-\mathbf{y}) \qquad (7.112)$$

$$\boldsymbol{\Sigma} = \left(\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{B}\boldsymbol{\Phi} + \mathbf{A}\right)^{-1}. \qquad (7.113)$$

We can now use this Laplace approximation to evaluate the marginal likelihood. Using the general result (4.135) for an integral evaluated using the Laplace approxi-

mation, we have

$$\begin{aligned} p(\mathbf{t}|\boldsymbol{\alpha}) &= \int p(\mathbf{t}|\mathbf{w})p(\mathbf{w}|\boldsymbol{\alpha})\,\mathrm{d}\mathbf{w} \\ &\simeq p(\mathbf{t}|\mathbf{w}^\star)p(\mathbf{w}^\star|\boldsymbol{\alpha})(2\pi)^{M/2}|\boldsymbol{\Sigma}|^{1/2}. \end{aligned} \tag{7.114}$$

Exercise 7.19

If we substitute for $p(\mathbf{t}|\mathbf{w}^\star)$ and $p(\mathbf{w}^\star|\boldsymbol{\alpha})$ and then set the derivative of the marginal likelihood with respect to α_i equal to zero, we obtain

$$-\frac{1}{2}(w_i^\star)^2 + \frac{1}{2\alpha_i} - \frac{1}{2}\Sigma_{ii} = 0. \tag{7.115}$$

Defining $\gamma_i = 1 - \alpha_i\Sigma_{ii}$ and rearranging then gives

$$\alpha_i^{\text{new}} = \frac{\gamma_i}{(w_i^\star)^2} \tag{7.116}$$

which is identical to the re-estimation formula (7.87) obtained for the regression RVM.

If we define

$$\widehat{\mathbf{t}} = \boldsymbol{\Phi}\mathbf{w}^\star + \mathbf{B}^{-1}(\mathbf{t} - \mathbf{y}) \tag{7.117}$$

we can write the approximate log marginal likelihood in the form

$$\ln p(\mathbf{t}|\boldsymbol{\alpha}, \beta) = -\frac{1}{2}\left\{N\ln(2\pi) + \ln|\mathbf{C}| + (\widehat{\mathbf{t}})^{\mathrm{T}}\mathbf{C}^{-1}\widehat{\mathbf{t}}\right\} \tag{7.118}$$

where

$$\mathbf{C} = \mathbf{B} + \boldsymbol{\Phi}\mathbf{A}\boldsymbol{\Phi}^{\mathrm{T}}. \tag{7.119}$$

This takes the same form as (7.85) in the regression case, and so we can apply the same analysis of sparsity and obtain the same fast learning algorithm in which we fully optimize a single hyperparameter α_i at each step.

Appendix A

Figure 7.12 shows the relevance vector machine applied to a synthetic classification data set. We see that the relevance vectors tend not to lie in the region of the decision boundary, in contrast to the support vector machine. This is consistent with our earlier discussion of sparsity in the RVM, because a basis function $\phi_i(\mathbf{x})$ centred on a data point near the boundary will have a vector $\boldsymbol{\varphi}_i$ that is poorly aligned with the training data vector $\mathbf{t}$.

One of the potential advantages of the relevance vector machine compared with the SVM is that it makes probabilistic predictions. For example, this allows the RVM to be used to help construct an emission density in a nonlinear extension of the linear dynamical system for tracking faces in video sequences (Williams *et al.*, 2005).

Section 13.3

So far, we have considered the RVM for binary classification problems. For $K > 2$ classes, we again make use of the probabilistic approach in Section 4.3.4 in which there are K linear models of the form

$$a_k = \mathbf{w}_k^{\mathrm{T}}\mathbf{x} \tag{7.120}$$

Figure 7.12 Example of the relevance vector machine applied to a synthetic data set, in which the left-hand plot shows the decision boundary and the data points, with the relevance vectors indicated by circles. Comparison with the results shown in Figure 7.4 for the corresponding support vector machine shows that the RVM gives a much sparser model. The right-hand plot shows the posterior probability given by the RVM output in which the proportion of red (blue) ink indicates the probability of that point belonging to the red (blue) class.

which are combined using a softmax function to give outputs

$$y_k(\mathbf{x}) = \frac{\exp(a_k)}{\sum_j \exp(a_j)}. \tag{7.121}$$

The log likelihood function is then given by

$$\ln p(\mathbf{T}|\mathbf{w}_1, \ldots, \mathbf{w}_K) = \prod_{n=1}^{N} \prod_{k=1}^{K} y_{nk}^{t_{nk}} \tag{7.122}$$

where the target values t_{nk} have a 1-of-K coding for each data point n, and $\mathbf{T}$ is a matrix with elements t_{nk}. Again, the Laplace approximation can be used to optimize the hyperparameters (Tipping, 2001), in which the model and its Hessian are found using IRLS. This gives a more principled approach to multiclass classification than the pairwise method used in the support vector machine and also provides probabilistic predictions for new data points. The principal disadvantage is that the Hessian matrix has size $MK \times MK$, where M is the number of active basis functions, which gives an additional factor of K^3 in the computational cost of training compared with the two-class RVM.

The principal disadvantage of the relevance vector machine is the relatively long training times compared with the SVM. This is offset, however, by the avoidance of cross-validation runs to set the model complexity parameters. Furthermore, because it yields sparser models, the computation time on test points, which is usually the more important consideration in practice, is typically much less.

Exercises

7.1 ($\star\star$) **www** Suppose we have a data set of input vectors $\{\mathbf{x}_n\}$ with corresponding target values $t_n \in \{-1, 1\}$, and suppose that we model the density of input vectors within each class separately using a Parzen kernel density estimator (see Section 2.5.1) with a kernel $k(\mathbf{x}, \mathbf{x}')$. Write down the minimum misclassification-rate decision rule assuming the two classes have equal prior probability. Show also that, if the kernel is chosen to be $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^{\mathrm{T}}\mathbf{x}'$, then the classification rule reduces to simply assigning a new input vector to the class having the closest mean. Finally, show that, if the kernel takes the form $k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}')$, that the classification is based on the closest mean in the feature space $\boldsymbol{\phi}(\mathbf{x})$.

7.2 ($\star$) Show that, if the 1 on the right-hand side of the constraint (7.5) is replaced by some arbitrary constant $\gamma > 0$, the solution for the maximum margin hyperplane is unchanged.

7.3 ($\star\star$) Show that, irrespective of the dimensionality of the data space, a data set consisting of just two data points, one from each class, is sufficient to determine the location of the maximum-margin hyperplane.

7.4 ($\star\star$) **www** Show that the value ρ of the margin for the maximum-margin hyperplane is given by

$$\frac{1}{\rho^2} = \sum_{n=1}^{N} a_n \tag{7.123}$$

where $\{a_n\}$ are given by maximizing (7.10) subject to the constraints (7.11) and (7.12).

7.5 ($\star\star$) Show that the values of ρ and $\{a_n\}$ in the previous exercise also satisfy

$$\frac{1}{\rho^2} = 2\widetilde{L}(\mathbf{a}) \tag{7.124}$$

where $\widetilde{L}(\mathbf{a})$ is defined by (7.10). Similarly, show that

$$\frac{1}{\rho^2} = \|\mathbf{w}\|^2. \tag{7.125}$$

7.6 ($\star$) Consider the logistic regression model with a target variable $t \in \{-1, 1\}$. If we define $p(t = 1|y) = \sigma(y)$ where $y(\mathbf{x})$ is given by (7.1), show that the negative log likelihood, with the addition of a quadratic regularization term, takes the form (7.47).

7.7 ($\star$) Consider the Lagrangian (7.56) for the regression support vector machine. By setting the derivatives of the Lagrangian with respect to $\mathbf{w}$, b, ξ_n, and $\widehat{\xi}_n$ to zero and then back substituting to eliminate the corresponding variables, show that the dual Lagrangian is given by (7.61).

7.8 ($\star$) **www** For the regression support vector machine considered in Section 7.1.4, show that all training data points for which $\xi_n > 0$ will have $a_n = C$, and similarly all points for which $\widehat{\xi}_n > 0$ will have $\widehat{a}_n = C$.

7.9 ($\star$) Verify the results (7.82) and (7.83) for the mean and covariance of the posterior distribution over weights in the regression RVM.

7.10 ($\star\star$) **www** Derive the result (7.85) for the marginal likelihood function in the regression RVM, by performing the Gaussian integral over $\mathbf{w}$ in (7.84) using the technique of completing the square in the exponential.

7.11 ($\star\star$) Repeat the above exercise, but this time make use of the general result (2.115).

7.12 ($\star\star$) **www** Show that direct maximization of the log marginal likelihood (7.85) for the regression relevance vector machine leads to the re-estimation equations (7.87) and (7.88) where γ_i is defined by (7.89).

7.13 ($\star\star$) In the evidence framework for RVM regression, we obtained the re-estimation formulae (7.87) and (7.88) by maximizing the marginal likelihood given by (7.85). Extend this approach by inclusion of hyperpriors given by gamma distributions of the form (B.26) and obtain the corresponding re-estimation formulae for $\boldsymbol{\alpha}$ and β by maximizing the corresponding posterior probability $p(\mathbf{t}, \boldsymbol{\alpha}, \beta|\mathbf{X})$ with respect to $\boldsymbol{\alpha}$ and β.

7.14 ($\star\star$) Derive the result (7.90) for the predictive distribution in the relevance vector machine for regression. Show that the predictive variance is given by (7.91).

7.15 ($\star\star$) **www** Using the results (7.94) and (7.95), show that the marginal likelihood (7.85) can be written in the form (7.96), where $\lambda(\alpha_n)$ is defined by (7.97) and the sparsity and quality factors are defined by (7.98) and (7.99), respectively.

7.16 ($\star$) By taking the second derivative of the log marginal likelihood (7.97) for the regression RVM with respect to the hyperparameter α_i, show that the stationary point given by (7.101) is a maximum of the marginal likelihood.

7.17 ($\star\star$) Using (7.83) and (7.86), together with the matrix identity (C.7), show that the quantities S_n and Q_n defined by (7.102) and (7.103) can be written in the form (7.106) and (7.107).

7.18 ($\star$) **www** Show that the gradient vector and Hessian matrix of the log posterior distribution (7.109) for the classification relevance vector machine are given by (7.110) and (7.111).

7.19 ($\star\star$) Verify that maximization of the approximate log marginal likelihood function (7.114) for the classification relevance vector machine leads to the result (7.116) for re-estimation of the hyperparameters.

8 Graphical Models

Probabilities play a central role in modern pattern recognition. We have seen in Chapter 1 that probability theory can be expressed in terms of two simple equations corresponding to the sum rule and the product rule. All of the probabilistic inference and learning manipulations discussed in this book, no matter how complex, amount to repeated application of these two equations. We could therefore proceed to formulate and solve complicated probabilistic models purely by algebraic manipulation. However, we shall find it highly advantageous to augment the analysis using diagrammatic representations of probability distributions, called *probabilistic graphical models*. These offer several useful properties:

1. They provide a simple way to visualize the structure of a probabilistic model and can be used to design and motivate new models.

2. Insights into the properties of the model, including conditional independence properties, can be obtained by inspection of the graph.

3. Complex computations, required to perform inference and learning in sophisticated models, can be expressed in terms of graphical manipulations, in which underlying mathematical expressions are carried along implicitly.

A graph comprises *nodes* (also called *vertices*) connected by *links* (also known as *edges* or *arcs*). In a probabilistic graphical model, each node represents a random variable (or group of random variables), and the links express probabilistic relationships between these variables. The graph then captures the way in which the joint distribution over all of the random variables can be decomposed into a product of factors each depending only on a subset of the variables. We shall begin by discussing *Bayesian networks*, also known as *directed graphical models*, in which the links of the graphs have a particular directionality indicated by arrows. The other major class of graphical models are *Markov random fields*, also known as *undirected graphical models*, in which the links do not carry arrows and have no directional significance. Directed graphs are useful for expressing causal relationships between random variables, whereas undirected graphs are better suited to expressing soft constraints between random variables. For the purposes of solving inference problems, it is often convenient to convert both directed and undirected graphs into a different representation called a *factor graph*.

In this chapter, we shall focus on the key aspects of graphical models as needed for applications in pattern recognition and machine learning. More general treatments of graphical models can be found in the books by Whittaker (1990), Lauritzen (1996), Jensen (1996), Castillo *et al.* (1997), Jordan (1999), Cowell *et al.* (1999), and Jordan (2007).

8.1. Bayesian Networks

In order to motivate the use of directed graphs to describe probability distributions, consider first an arbitrary joint distribution $p(a, b, c)$ over three variables a, b, and c. Note that at this stage, we do not need to specify anything further about these variables, such as whether they are discrete or continuous. Indeed, one of the powerful aspects of graphical models is that a specific graph can make probabilistic statements for a broad class of distributions. By application of the product rule of probability (1.11), we can write the joint distribution in the form

$$p(a, b, c) = p(c|a, b)p(a, b). \tag{8.1}$$

A second application of the product rule, this time to the second term on the right-hand side of (8.1), gives

$$p(a, b, c) = p(c|a, b)p(b|a)p(a). \tag{8.2}$$

Note that this decomposition holds for any choice of the joint distribution. We now represent the right-hand side of (8.2) in terms of a simple graphical model as follows. First we introduce a node for each of the random variables a, b, and c and associate each node with the corresponding conditional distribution on the right-hand side of

Figure 8.1 A directed graphical model representing the joint probability distribution over three variables a, b, and c, corresponding to the decomposition on the right-hand side of (8.2).

(8.2). Then, for each conditional distribution we add directed links (arrows) to the graph from the nodes corresponding to the variables on which the distribution is conditioned. Thus for the factor $p(c|a, b)$, there will be links from nodes a and b to node c, whereas for the factor $p(a)$ there will be no incoming links. The result is the graph shown in Figure 8.1. If there is a link going from a node a to a node b, then we say that node a is the *parent* of node b, and we say that node b is the *child* of node a. Note that we shall not make any formal distinction between a node and the variable to which it corresponds but will simply use the same symbol to refer to both.

An interesting point to note about (8.2) is that the left-hand side is symmetrical with respect to the three variables a, b, and c, whereas the right-hand side is not. Indeed, in making the decomposition in (8.2), we have implicitly chosen a particular ordering, namely a, b, c, and had we chosen a different ordering we would have obtained a different decomposition and hence a different graphical representation. We shall return to this point later.

For the moment let us extend the example of Figure 8.1 by considering the joint distribution over K variables given by $p(x_1, \ldots, x_K)$. By repeated application of the product rule of probability, this joint distribution can be written as a product of conditional distributions, one for each of the variables

$$p(x_1, \ldots, x_K) = p(x_K|x_1, \ldots, x_{K-1}) \ldots p(x_2|x_1)p(x_1). \tag{8.3}$$

For a given choice of K, we can again represent this as a directed graph having K nodes, one for each conditional distribution on the right-hand side of (8.3), with each node having incoming links from all lower numbered nodes. We say that this graph is *fully connected* because there is a link between every pair of nodes.

So far, we have worked with completely general joint distributions, so that the decompositions, and their representations as fully connected graphs, will be applicable to any choice of distribution. As we shall see shortly, it is the *absence* of links in the graph that conveys interesting information about the properties of the class of distributions that the graph represents. Consider the graph shown in Figure 8.2. This is not a fully connected graph because, for instance, there is no link from x_1 to x_2 or from x_3 to x_7.

We shall now go from this graph to the corresponding representation of the joint probability distribution written in terms of the product of a set of conditional distributions, one for each node in the graph. Each such conditional distribution will be conditioned only on the parents of the corresponding node in the graph. For instance, x_5 will be conditioned on x_1 and x_3. The joint distribution of all 7 variables

Figure 8.2 Example of a directed acyclic graph describing the joint distribution over variables $x_1, \ldots, x_7$. The corresponding decomposition of the joint distribution is given by (8.4).

is therefore given by

$$p(x_1)p(x_2)p(x_3)p(x_4|x_1, x_2, x_3)p(x_5|x_1, x_3)p(x_6|x_4)p(x_7|x_4, x_5). \tag{8.4}$$

The reader should take a moment to study carefully the correspondence between (8.4) and Figure 8.2.

We can now state in general terms the relationship between a given directed graph and the corresponding distribution over the variables. The joint distribution defined by a graph is given by the product, over all of the nodes of the graph, of a conditional distribution for each node conditioned on the variables corresponding to the parents of that node in the graph. Thus, for a graph with K nodes, the joint distribution is given by

$$p(\mathbf{x}) = \prod_{k=1}^{K} p(x_k|\mathrm{pa}_k) \tag{8.5}$$

where pa_k denotes the set of parents of x_k, and $\mathbf{x} = \{x_1, \ldots, x_K\}$. This key equation expresses the *factorization* properties of the joint distribution for a directed graphical model. Although we have considered each node to correspond to a single variable, we can equally well associate sets of variables and vector-valued variables with the nodes of a graph. It is easy to show that the representation on the right-hand side of (8.5) is always correctly normalized provided the individual conditional distributions are normalized.

Exercise 8.1

The directed graphs that we are considering are subject to an important restriction namely that there must be no *directed cycles*, in other words there are no closed paths within the graph such that we can move from node to node along links following the direction of the arrows and end up back at the starting node. Such graphs are also called *directed acyclic graphs*, or *DAGs*. This is equivalent to the statement that there exists an ordering of the nodes such that there are no links that go from any node to any lower numbered node.

Exercise 8.2

8.1.1 Example: Polynomial regression

As an illustration of the use of directed graphs to describe probability distributions, we consider the Bayesian polynomial regression model introduced in Sec-

Figure 8.3 Directed graphical model representing the joint distribution (8.6) corresponding to the Bayesian polynomial regression model introduced in Section 1.2.6.

tion 1.2.6. The random variables in this model are the vector of polynomial coefficients $\mathbf{w}$ and the observed data $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$. In addition, this model contains the input data $\mathbf{x} = (x_1, \ldots, x_N)^{\mathrm{T}}$, the noise variance σ^2, and the hyperparameter α representing the precision of the Gaussian prior over $\mathbf{w}$, all of which are parameters of the model rather than random variables. Focussing just on the random variables for the moment, we see that the joint distribution is given by the product of the prior $p(\mathbf{w})$ and N conditional distributions $p(t_n|\mathbf{w})$ for $n = 1, \ldots, N$ so that

$$p(\mathbf{t}, \mathbf{w}) = p(\mathbf{w}) \prod_{n=1}^{N} p(t_n|\mathbf{w}). \tag{8.6}$$

This joint distribution can be represented by a graphical model shown in Figure 8.3.

When we start to deal with more complex models later in the book, we shall find it inconvenient to have to write out multiple nodes of the form $t_1, \ldots, t_N$ explicitly as in Figure 8.3. We therefore introduce a graphical notation that allows such multiple nodes to be expressed more compactly, in which we draw a single representative node t_n and then surround this with a box, called a *plate*, labelled with N indicating that there are N nodes of this kind. Re-writing the graph of Figure 8.3 in this way, we obtain the graph shown in Figure 8.4.

We shall sometimes find it helpful to make the parameters of a model, as well as its stochastic variables, explicit. In this case, (8.6) becomes

$$p(\mathbf{t}, \mathbf{w}|\mathbf{x}, \alpha, \sigma^2) = p(\mathbf{w}|\alpha) \prod_{n=1}^{N} p(t_n|\mathbf{w}, x_n, \sigma^2).$$

Correspondingly, we can make $\mathbf{x}$ and α explicit in the graphical representation. To do this, we shall adopt the convention that random variables will be denoted by open circles, and deterministic parameters will be denoted by smaller solid circles. If we take the graph of Figure 8.4 and include the deterministic parameters, we obtain the graph shown in Figure 8.5.

When we apply a graphical model to a problem in machine learning or pattern recognition, we will typically set some of the random variables to specific observed

Figure 8.4 An alternative, more compact, representation of the graph shown in Figure 8.3 in which we have introduced a *plate* (the box labelled N) that represents N nodes of which only a single example t_n is shown explicitly.

Figure 8.5 This shows the same model as in Figure 8.4 but with the deterministic parameters shown explicitly by the smaller solid nodes.

values, for example the variables $\{t_n\}$ from the training set in the case of polynomial curve fitting. In a graphical model, we will denote such *observed variables* by shading the corresponding nodes. Thus the graph corresponding to Figure 8.5 in which the variables $\{t_n\}$ are observed is shown in Figure 8.6. Note that the value of $\mathbf{w}$ is not observed, and so $\mathbf{w}$ is an example of a *latent* variable, also known as a *hidden* variable. Such variables play a crucial role in many probabilistic models and will form the focus of Chapters 9 and 12.

Having observed the values $\{t_n\}$ we can, if desired, evaluate the posterior distribution of the polynomial coefficients $\mathbf{w}$ as discussed in Section 1.2.5. For the moment, we note that this involves a straightforward application of Bayes' theorem

$$p(\mathbf{w}|\mathbf{T}) \propto p(\mathbf{w}) \prod_{n=1}^{N} p(t_n|\mathbf{w}) \tag{8.7}$$

where again we have omitted the deterministic parameters in order to keep the notation uncluttered.

In general, model parameters such as $\mathbf{w}$ are of little direct interest in themselves, because our ultimate goal is to make predictions for new input values. Suppose we are given a new input value $\widehat{x}$ and we wish to find the corresponding probability distribution for $\widehat{t}$ conditioned on the observed data. The graphical model that describes this problem is shown in Figure 8.7, and the corresponding joint distribution of all of the random variables in this model, conditioned on the deterministic parameters, is then given by

$$p(\widehat{t}, \mathbf{t}, \mathbf{w}|\widehat{x}, \mathbf{x}, \alpha, \sigma^2) = \left[\prod_{n=1}^{N} p(t_n|x_n, \mathbf{w}, \sigma^2)\right] p(\mathbf{w}|\alpha) p(\widehat{t}|\widehat{x}, \mathbf{w}, \sigma^2). \tag{8.8}$$

Figure 8.6 As in Figure 8.5 but with the nodes $\{t_n\}$ shaded to indicate that the corresponding random variables have been set to their observed (training set) values.

Figure 8.7 The polynomial regression model, corresponding to Figure 8.6, showing also a new input value $\widehat{x}$ together with the corresponding model prediction $\widehat{t}$.

The required predictive distribution for $\widehat{t}$ is then obtained, from the sum rule of probability, by integrating out the model parameters $\mathbf{w}$ so that

$$p(\widehat{t}|\widehat{x}, \mathbf{x}, \mathbf{t}, \alpha, \sigma^2) \propto \int p(\widehat{t}, \mathbf{t}, \mathbf{w}|\widehat{x}, \mathbf{x}, \alpha, \sigma^2)\, \mathrm{d}\mathbf{w}$$

where we are implicitly setting the random variables in $\mathbf{t}$ to the specific values observed in the data set. The details of this calculation were discussed in Chapter 3.

8.1.2 Generative models

There are many situations in which we wish to draw samples from a given probability distribution. Although we shall devote the whole of Chapter 11 to a detailed discussion of sampling methods, it is instructive to outline here one technique, called *ancestral sampling*, which is particularly relevant to graphical models. Consider a joint distribution $p(x_1, \ldots, x_K)$ over K variables that factorizes according to (8.5) corresponding to a directed acyclic graph. We shall suppose that the variables have been ordered such that there are no links from any node to any lower numbered node, in other words each node has a higher number than any of its parents. Our goal is to draw a sample $\widehat{x}_1, \ldots, \widehat{x}_K$ from the joint distribution.

To do this, we start with the lowest-numbered node and draw a sample from the distribution $p(x_1)$, which we call $\widehat{x}_1$. We then work through each of the nodes in order, so that for node n we draw a sample from the conditional distribution $p(x_n|\mathrm{pa}_n)$ in which the parent variables have been set to their sampled values. Note that at each stage, these parent values will always be available because they correspond to lower-numbered nodes that have already been sampled. Techniques for sampling from specific distributions will be discussed in detail in Chapter 11. Once we have sampled from the final variable x_K, we will have achieved our objective of obtaining a sample from the joint distribution. To obtain a sample from some marginal distribution corresponding to a subset of the variables, we simply take the sampled values for the required nodes and ignore the sampled values for the remaining nodes. For example, to draw a sample from the distribution $p(x_2, x_4)$, we simply sample from the full joint distribution and then retain the values $\widehat{x}_2, \widehat{x}_4$ and discard the remaining values $\{\widehat{x}_{j\neq 2,4}\}$.

Figure 8.8 A graphical model representing the process by which images of objects are created, in which the identity of an object (a discrete variable) and the position and orientation of that object (continuous variables) have independent prior probabilities. The image (a vector of pixel intensities) has a probability distribution that is dependent on the identity of the object as well as on its position and orientation.

For practical applications of probabilistic models, it will typically be the higher-numbered variables corresponding to terminal nodes of the graph that represent the observations, with lower-numbered nodes corresponding to latent variables. The primary role of the latent variables is to allow a complicated distribution over the observed variables to be represented in terms of a model constructed from simpler (typically exponential family) conditional distributions.

We can interpret such models as expressing the processes by which the observed data arose. For instance, consider an object recognition task in which each observed data point corresponds to an image (comprising a vector of pixel intensities) of one of the objects. In this case, the latent variables might have an interpretation as the position and orientation of the object. Given a particular observed image, our goal is to find the posterior distribution over objects, in which we integrate over all possible positions and orientations. We can represent this problem using a graphical model of the form show in Figure 8.8.

The graphical model captures the *causal* process (Pearl, 1988) by which the observed data was generated. For this reason, such models are often called *generative* models. By contrast, the polynomial regression model described by Figure 8.5 is not generative because there is no probability distribution associated with the input variable x, and so it is not possible to generate synthetic data points from this model. We could make it generative by introducing a suitable prior distribution $p(x)$, at the expense of a more complex model.

The hidden variables in a probabilistic model need not, however, have any explicit physical interpretation but may be introduced simply to allow a more complex joint distribution to be constructed from simpler components. In either case, the technique of ancestral sampling applied to a generative model mimics the creation of the observed data and would therefore give rise to 'fantasy' data whose probability distribution (if the model were a perfect representation of reality) would be the same as that of the observed data. In practice, producing synthetic observations from a generative model can prove informative in understanding the form of the probability distribution represented by that model.

8.1.3 Discrete variables

Section 2.4

We have discussed the importance of probability distributions that are members of the exponential family, and we have seen that this family includes many well-known distributions as particular cases. Although such distributions are relatively simple, they form useful building blocks for constructing more complex probability

Figure 8.9 (a) This fully-connected graph describes a general distribution over two K-state discrete variables having a total of $K^2 - 1$ parameters. (b) By dropping the link between the nodes, the number of parameters is reduced to $2(K-1)$.

distributions, and the framework of graphical models is very useful in expressing the way in which these building blocks are linked together.

Such models have particularly nice properties if we choose the relationship between each parent-child pair in a directed graph to be conjugate, and we shall explore several examples of this shortly. Two cases are particularly worthy of note, namely when the parent and child node each correspond to discrete variables and when they each correspond to Gaussian variables, because in these two cases the relationship can be extended hierarchically to construct arbitrarily complex directed acyclic graphs. We begin by examining the discrete case.

The probability distribution $p(\mathbf{x}|\boldsymbol{\mu})$ for a single discrete variable $\mathbf{x}$ having K possible states (using the 1-of-K representation) is given by

$$p(\mathbf{x}|\boldsymbol{\mu}) = \prod_{k=1}^{K} \mu_k^{x_k} \tag{8.9}$$

and is governed by the parameters $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_K)^{\mathrm{T}}$. Due to the constraint $\sum_k \mu_k = 1$, only $K-1$ values for μ_k need to be specified in order to define the distribution.

Now suppose that we have two discrete variables, $\mathbf{x}_1$ and $\mathbf{x}_2$, each of which has K states, and we wish to model their joint distribution. We denote the probability of observing both $x_{1k} = 1$ and $x_{2l} = 1$ by the parameter μ_{kl}, where x_{1k} denotes the kth component of $\mathbf{x}_1$, and similarly for x_{2l}. The joint distribution can be written

$$p(\mathbf{x}_1, \mathbf{x}_2|\boldsymbol{\mu}) = \prod_{k=1}^{K} \prod_{l=1}^{K} \mu_{kl}^{x_{1k} x_{2l}}.$$

Because the parameters μ_{kl} are subject to the constraint $\sum_k \sum_l \mu_{kl} = 1$, this distribution is governed by $K^2 - 1$ parameters. It is easily seen that the total number of parameters that must be specified for an arbitrary joint distribution over M variables is $K^M - 1$ and therefore grows exponentially with the number M of variables.

Using the product rule, we can factor the joint distribution $p(\mathbf{x}_1, \mathbf{x}_2)$ in the form $p(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_1)$, which corresponds to a two-node graph with a link going from the $\mathbf{x}_1$ node to the $\mathbf{x}_2$ node as shown in Figure 8.9(a). The marginal distribution $p(\mathbf{x}_1)$ is governed by $K-1$ parameters, as before, Similarly, the conditional distribution $p(\mathbf{x}_2|\mathbf{x}_1)$ requires the specification of $K-1$ parameters for each of the K possible values of $\mathbf{x}_1$. The total number of parameters that must be specified in the joint distribution is therefore $(K-1) + K(K-1) = K^2 - 1$ as before.

Now suppose that the variables $\mathbf{x}_1$ and $\mathbf{x}_2$ were independent, corresponding to the graphical model shown in Figure 8.9(b). Each variable is then described by

Figure 8.10 This chain of M discrete nodes, each having K states, requires the specification of $K-1+(M-1)K(K-1)$ parameters, which grows linearly with the length M of the chain. In contrast, a fully connected graph of M nodes would have K^M-1 parameters, which grows exponentially with M.

a separate multinomial distribution, and the total number of parameters would be $2(K-1)$. For a distribution over M independent discrete variables, each having K states, the total number of parameters would be $M(K-1)$, which therefore grows linearly with the number of variables. From a graphical perspective, we have reduced the number of parameters by dropping links in the graph, at the expense of having a restricted class of distributions.

More generally, if we have M discrete variables $\mathbf{x}_1, \ldots, \mathbf{x}_M$, we can model the joint distribution using a directed graph with one variable corresponding to each node. The conditional distribution at each node is given by a set of nonnegative parameters subject to the usual normalization constraint. If the graph is fully connected then we have a completely general distribution having K^M-1 parameters, whereas if there are no links in the graph the joint distribution factorizes into the product of the marginals, and the total number of parameters is $M(K-1)$. Graphs having intermediate levels of connectivity allow for more general distributions than the fully factorized one while requiring fewer parameters than the general joint distribution. As an illustration, consider the chain of nodes shown in Figure 8.10. The marginal distribution $p(\mathbf{x}_1)$ requires $K-1$ parameters, whereas each of the $M-1$ conditional distributions $p(\mathbf{x}_i|\mathbf{x}_{i-1})$, for $i=2,\ldots,M$, requires $K(K-1)$ parameters. This gives a total parameter count of $K-1+(M-1)K(K-1)$, which is quadratic in K and which grows linearly (rather than exponentially) with the length M of the chain.

An alternative way to reduce the number of independent parameters in a model is by *sharing* parameters (also known as *tying* of parameters). For instance, in the chain example of Figure 8.10, we can arrange that all of the conditional distributions $p(\mathbf{x}_i|\mathbf{x}_{i-1})$, for $i=2,\ldots,M$, are governed by the same set of $K(K-1)$ parameters. Together with the $K-1$ parameters governing the distribution of $\mathbf{x}_1$, this gives a total of K^2-1 parameters that must be specified in order to define the joint distribution.

We can turn a graph over discrete variables into a Bayesian model by introducing Dirichlet priors for the parameters. From a graphical point of view, each node then acquires an additional parent representing the Dirichlet distribution over the parameters associated with the corresponding discrete node. This is illustrated for the chain model in Figure 8.11. The corresponding model in which we tie the parameters governing the conditional distributions $p(\mathbf{x}_i|\mathbf{x}_{i-1})$, for $i=2,\ldots,M$, is shown in Figure 8.12.

Another way of controlling the exponential growth in the number of parameters in models of discrete variables is to use parameterized models for the conditional distributions instead of complete tables of conditional probability values. To illustrate this idea, consider the graph in Figure 8.13 in which all of the nodes represent binary variables. Each of the parent variables x_i is governed by a single parame-

Figure 8.11 An extension of the model of Figure 8.10 to include Dirichlet priors over the parameters governing the discrete distributions.

Figure 8.12 As in Figure 8.11 but with a single set of parameters $\boldsymbol{\mu}$ shared amongst all of the conditional distributions $p(\mathbf{x}_i|\mathbf{x}_{i-1})$.

ter μ_i representing the probability $p(x_i = 1)$, giving M parameters in total for the parent nodes. The conditional distribution $p(y|x_1, \ldots, x_M)$, however, would require 2^M parameters representing the probability $p(y = 1)$ for each of the 2^M possible settings of the parent variables. Thus in general the number of parameters required to specify this conditional distribution will grow exponentially with M. We can obtain a more parsimonious form for the conditional distribution by using a logistic sigmoid function acting on a linear combination of the parent variables, giving

Section 2.4

$$p(y = 1|x_1, \ldots, x_M) = \sigma\left(w_0 + \sum_{i=1}^{M} w_i x_i\right) = \sigma(\mathbf{w}^{\mathrm{T}}\mathbf{x}) \tag{8.10}$$

where $\sigma(a) = (1+\exp(-a))^{-1}$ is the logistic sigmoid, $\mathbf{x} = (x_0, x_1, \ldots, x_M)^{\mathrm{T}}$ is an $(M+1)$-dimensional vector of parent states augmented with an additional variable x_0 whose value is clamped to 1, and $\mathbf{w} = (w_0, w_1, \ldots, w_M)^{\mathrm{T}}$ is a vector of $M+1$ parameters. This is a more restricted form of conditional distribution than the general case but is now governed by a number of parameters that grows linearly with M. In this sense, it is analogous to the choice of a restrictive form of covariance matrix (for example, a diagonal matrix) in a multivariate Gaussian distribution. The motivation for the logistic sigmoid representation was discussed in Section 4.2.

Figure 8.13 A graph comprising M parents $x_1, \ldots, x_M$ and a single child y, used to illustrate the idea of parameterized conditional distributions for discrete variables.

8.1.4 Linear-Gaussian models

In the previous section, we saw how to construct joint probability distributions over a set of discrete variables by expressing the variables as nodes in a directed acyclic graph. Here we show how a multivariate Gaussian can be expressed as a directed graph corresponding to a linear-Gaussian model over the component variables. This allows us to impose interesting structure on the distribution, with the general Gaussian and the diagonal covariance Gaussian representing opposite extremes. Several widely used techniques are examples of linear-Gaussian models, such as probabilistic principal component analysis, factor analysis, and linear dynamical systems (Roweis and Ghahramani, 1999). We shall make extensive use of the results of this section in later chapters when we consider some of these techniques in detail.

Consider an arbitrary directed acyclic graph over D variables in which node i represents a single continuous random variable x_i having a Gaussian distribution. The mean of this distribution is taken to be a linear combination of the states of its parent nodes pa_i of node i

$$p(x_i|\mathrm{pa}_i) = \mathcal{N}\left(x_i \middle| \sum_{j\in\mathrm{pa}_i} w_{ij}x_j + b_i, v_i\right) \tag{8.11}$$

where w_{ij} and b_i are parameters governing the mean, and v_i is the variance of the conditional distribution for x_i. The log of the joint distribution is then the log of the product of these conditionals over all nodes in the graph and hence takes the form

$$\ln p(\mathbf{x}) = \sum_{i=1}^{D} \ln p(x_i|\mathrm{pa}_i) \tag{8.12}$$

$$= -\sum_{i=1}^{D} \frac{1}{2v_i}\left(x_i - \sum_{j\in\mathrm{pa}_i} w_{ij}x_j - b_i\right)^2 + \text{const} \tag{8.13}$$

where $\mathbf{x} = (x_1, \ldots, x_D)^{\mathrm{T}}$ and 'const' denotes terms independent of $\mathbf{x}$. We see that this is a quadratic function of the components of $\mathbf{x}$, and hence the joint distribution $p(\mathbf{x})$ is a multivariate Gaussian.

We can determine the mean and covariance of the joint distribution recursively as follows. Each variable x_i has (conditional on the states of its parents) a Gaussian distribution of the form (8.11) and so

$$x_i = \sum_{j\in\mathrm{pa}_i} w_{ij}x_j + b_i + \sqrt{v_i}\epsilon_i \tag{8.14}$$

where ϵ_i is a zero mean, unit variance Gaussian random variable satisfying $\mathbb{E}[\epsilon_i] = 0$ and $\mathbb{E}[\epsilon_i\epsilon_j] = I_{ij}$, where I_{ij} is the i, j element of the identity matrix. Taking the expectation of (8.14), we have

$$\mathbb{E}[x_i] = \sum_{j\in\mathrm{pa}_i} w_{ij}\mathbb{E}[x_j] + b_i. \tag{8.15}$$

Figure 8.14 A directed graph over three Gaussian variables, with one missing link.

Thus we can find the components of $\mathbb{E}[\mathbf{x}] = (\mathbb{E}[x_1], \ldots, \mathbb{E}[x_D])^{\mathrm{T}}$ by starting at the lowest numbered node and working recursively through the graph (here we again assume that the nodes are numbered such that each node has a higher number than its parents). Similarly, we can use (8.14) and (8.15) to obtain the i, j element of the covariance matrix for $p(\mathbf{x})$ in the form of a recursion relation

$$\begin{aligned} \mathrm{cov}[x_i, x_j] &= \mathbb{E}\left[(x_i - \mathbb{E}[x_i])(x_j - \mathbb{E}[x_j])\right] \\ &= \mathbb{E}\left[(x_i - \mathbb{E}[x_i])\left\{\sum_{k \in \mathrm{pa}_j} w_{jk}(x_k - \mathbb{E}[x_k]) + \sqrt{v_j}\epsilon_j\right\}\right] \\ &= \sum_{k \in \mathrm{pa}_j} w_{jk}\mathrm{cov}[x_i, x_k] + I_{ij}v_j \end{aligned} \tag{8.16}$$

and so the covariance can similarly be evaluated recursively starting from the lowest numbered node.

Let us consider two extreme cases. First of all, suppose that there are no links in the graph, which therefore comprises D isolated nodes. In this case, there are no parameters w_{ij} and so there are just D parameters b_i and D parameters v_i. From the recursion relations (8.15) and (8.16), we see that the mean of $p(\mathbf{x})$ is given by $(b_1, \ldots, b_D)^{\mathrm{T}}$ and the covariance matrix is diagonal of the form $\mathrm{diag}(v_1, \ldots, v_D)$. The joint distribution has a total of $2D$ parameters and represents a set of D independent univariate Gaussian distributions.

Now consider a fully connected graph in which each node has all lower numbered nodes as parents. The matrix w_{ij} then has $i - 1$ entries on the ith row and hence is a lower triangular matrix (with no entries on the leading diagonal). Then the total number of parameters w_{ij} is obtained by taking the number D^2 of elements in a $D \times D$ matrix, subtracting D to account for the absence of elements on the leading diagonal, and then dividing by 2 because the matrix has elements only below the diagonal, giving a total of $D(D-1)/2$. The total number of independent parameters $\{w_{ij}\}$ and $\{v_i\}$ in the covariance matrix is therefore $D(D+1)/2$ corresponding to a general symmetric covariance matrix.

Section 2.3

Graphs having some intermediate level of complexity correspond to joint Gaussian distributions with partially constrained covariance matrices. Consider for example the graph shown in Figure 8.14, which has a link missing between variables x_1 and x_3. Using the recursion relations (8.15) and (8.16), we see that the mean and covariance of the joint distribution are given by

Exercise 8.7

$$\boldsymbol{\mu} = (b_1, b_2 + w_{21}b_1, b_3 + w_{32}b_2 + w_{32}w_{21}b_1)^{\mathrm{T}} \tag{8.17}$$

$$\boldsymbol{\Sigma} = \begin{pmatrix} v_1 & w_{21}v_1 & w_{32}w_{21}v_1 \\ w_{21}v_1 & v_2 + w_{21}^2 v_1 & w_{32}(v_2 + w_{21}^2 v_1) \\ w_{32}w_{21}v_1 & w_{32}(v_2 + w_{21}^2 v_1) & v_3 + w_{32}^2(v_2 + w_{21}^2 v_1) \end{pmatrix}. \tag{8.18}$$

We can readily extend the linear-Gaussian graphical model to the case in which the nodes of the graph represent multivariate Gaussian variables. In this case, we can write the conditional distribution for node i in the form

$$p(\mathbf{x}_i|\mathrm{pa}_i) = \mathcal{N}\left(\mathbf{x}_i \middle| \sum_{j\in\mathrm{pa}_i} \mathbf{W}_{ij}\mathbf{x}_j + \mathbf{b}_i, \mathbf{\Sigma}_i\right) \tag{8.19}$$

where now $\mathbf{W}_{ij}$ is a matrix (which is nonsquare if $\mathbf{x}_i$ and $\mathbf{x}_j$ have different dimensionalities). Again it is easy to verify that the joint distribution over all variables is Gaussian.

Section 2.3.6

Note that we have already encountered a specific example of the linear-Gaussian relationship when we saw that the conjugate prior for the mean $\boldsymbol{\mu}$ of a Gaussian variable $\mathbf{x}$ is itself a Gaussian distribution over $\boldsymbol{\mu}$. The joint distribution over $\mathbf{x}$ and $\boldsymbol{\mu}$ is therefore Gaussian. This corresponds to a simple two-node graph in which the node representing $\boldsymbol{\mu}$ is the parent of the node representing $\mathbf{x}$. The mean of the distribution over $\boldsymbol{\mu}$ is a parameter controlling a prior, and so it can be viewed as a hyperparameter. Because the value of this hyperparameter may itself be unknown, we can again treat it from a Bayesian perspective by introducing a prior over the hyperparameter, sometimes called a *hyperprior*, which is again given by a Gaussian distribution. This type of construction can be extended in principle to any level and is an illustration of a *hierarchical Bayesian model*, of which we shall encounter further examples in later chapters.

8.2. Conditional Independence

An important concept for probability distributions over multiple variables is that of *conditional independence* (Dawid, 1980). Consider three variables a, b, and c, and suppose that the conditional distribution of a, given b and c, is such that it does not depend on the value of b, so that

$$p(a|b,c) = p(a|c). \tag{8.20}$$

We say that a is conditionally independent of b given c. This can be expressed in a slightly different way if we consider the joint distribution of a and b conditioned on c, which we can write in the form

$$\begin{aligned} p(a,b|c) &= p(a|b,c)p(b|c) \\ &= p(a|c)p(b|c). \end{aligned} \tag{8.21}$$

where we have used the product rule of probability together with (8.20). Thus we see that, conditioned on c, the joint distribution of a and b factorizes into the product of the marginal distribution of a and the marginal distribution of b (again both conditioned on c). This says that the variables a and b are statistically independent, given c. Note that our definition of conditional independence will require that (8.20),

Figure 8.15 The first of three examples of graphs over three variables a, b, and c used to discuss conditional independence properties of directed graphical models.

or equivalently (8.21), must hold for every possible value of c, and not just for some values. We shall sometimes use a shorthand notation for conditional independence (Dawid, 1979) in which

$$a \perp\!\!\!\perp b \mid c \tag{8.22}$$

denotes that a is conditionally independent of b given c and is equivalent to (8.20).

Conditional independence properties play an important role in using probabilistic models for pattern recognition by simplifying both the structure of a model and the computations needed to perform inference and learning under that model. We shall see examples of this shortly.

If we are given an expression for the joint distribution over a set of variables in terms of a product of conditional distributions (i.e., the mathematical representation underlying a directed graph), then we could in principle test whether any potential conditional independence property holds by repeated application of the sum and product rules of probability. In practice, such an approach would be very time consuming. An important and elegant feature of graphical models is that conditional independence properties of the joint distribution can be read directly from the graph without having to perform any analytical manipulations. The general framework for achieving this is called *d-separation*, where the 'd' stands for 'directed' (Pearl, 1988). Here we shall motivate the concept of d-separation and give a general statement of the d-separation criterion. A formal proof can be found in Lauritzen (1996).

8.2.1 Three example graphs

We begin our discussion of the conditional independence properties of directed graphs by considering three simple examples each involving graphs having just three nodes. Together, these will motivate and illustrate the key concepts of d-separation. The first of the three examples is shown in Figure 8.15, and the joint distribution corresponding to this graph is easily written down using the general result (8.5) to give

$$p(a, b, c) = p(a|c)p(b|c)p(c). \tag{8.23}$$

If none of the variables are observed, then we can investigate whether a and b are independent by marginalizing both sides of (8.23) with respect to c to give

$$p(a, b) = \sum_c p(a|c)p(b|c)p(c). \tag{8.24}$$

In general, this does not factorize into the product $p(a)p(b)$, and so

$$a \not\perp\!\!\!\perp b \mid \emptyset \tag{8.25}$$

Figure 8.16 As in Figure 8.15 but where we have conditioned on the value of variable c.

where $\emptyset$ denotes the empty set, and the symbol $\not\perp\!\!\!\perp$ means that the conditional independence property does not hold in general. Of course, it may hold for a particular distribution by virtue of the specific numerical values associated with the various conditional probabilities, but it does not follow in general from the structure of the graph.

Now suppose we condition on the variable c, as represented by the graph of Figure 8.16. From (8.23), we can easily write down the conditional distribution of a and b, given c, in the form

$$\begin{aligned} p(a,b|c) &= \frac{p(a,b,c)}{p(c)} \\ &= p(a|c)p(b|c) \end{aligned}$$

and so we obtain the conditional independence property

$$a \perp\!\!\!\perp b \mid c.$$

We can provide a simple graphical interpretation of this result by considering the path from node a to node b via c. The node c is said to be *tail-to-tail* with respect to this path because the node is connected to the tails of the two arrows, and the presence of such a path connecting nodes a and b causes these nodes to be dependent. However, when we condition on node c, as in Figure 8.16, the conditioned node 'blocks' the path from a to b and causes a and b to become (conditionally) independent.

We can similarly consider the graph shown in Figure 8.17. The joint distribution corresponding to this graph is again obtained from our general formula (8.5) to give

$$p(a,b,c) = p(a)p(c|a)p(b|c). \tag{8.26}$$

First of all, suppose that none of the variables are observed. Again, we can test to see if a and b are independent by marginalizing over c to give

$$p(a,b) = p(a)\sum_c p(c|a)p(b|c) = p(a)p(b|a).$$

Figure 8.17 The second of our three examples of 3-node graphs used to motivate the conditional independence framework for directed graphical models.

Figure 8.18 As in Figure 8.17 but now conditioning on node c.

which in general does not factorize into $p(a)p(b)$, and so

$$a \not\perp\!\!\!\perp b \mid \emptyset \tag{8.27}$$

as before.

Now suppose we condition on node c, as shown in Figure 8.18. Using Bayes' theorem, together with (8.26), we obtain

$$\begin{aligned} p(a,b|c) &= \frac{p(a,b,c)}{p(c)} \\ &= \frac{p(a)p(c|a)p(b|c)}{p(c)} \\ &= p(a|c)p(b|c) \end{aligned}$$

and so again we obtain the conditional independence property

$$a \perp\!\!\!\perp b \mid c.$$

As before, we can interpret these results graphically. The node c is said to be *head-to-tail* with respect to the path from node a to node b. Such a path connects nodes a and b and renders them dependent. If we now observe c, as in Figure 8.18, then this observation 'blocks' the path from a to b and so we obtain the conditional independence property $a \perp\!\!\!\perp b \mid c$.

Finally, we consider the third of our 3-node examples, shown by the graph in Figure 8.19. As we shall see, this has a more subtle behaviour than the two previous graphs.

The joint distribution can again be written down using our general result (8.5) to give

$$p(a,b,c) = p(a)p(b)p(c|a,b). \tag{8.28}$$

Consider first the case where none of the variables are observed. Marginalizing both sides of (8.28) over c we obtain

$$p(a,b) = p(a)p(b)$$

Figure 8.19 The last of our three examples of 3-node graphs used to explore conditional independence properties in graphical models. This graph has rather different properties from the two previous examples.

Figure 8.20 As in Figure 8.19 but conditioning on the value of node c. In this graph, the act of conditioning induces a dependence between a and b.

and so a and b are independent with no variables observed, in contrast to the two previous examples. We can write this result as

$$a \perp\!\!\!\perp b \mid \emptyset. \tag{8.29}$$

Now suppose we condition on c, as indicated in Figure 8.20. The conditional distribution of a and b is then given by

$$\begin{aligned} p(a, b|c) &= \frac{p(a, b, c)}{p(c)} \\ &= \frac{p(a)p(b)p(c|a, b)}{p(c)} \end{aligned}$$

which in general does not factorize into the product $p(a)p(b)$, and so

$$a \not\perp\!\!\!\perp b \mid c.$$

Thus our third example has the opposite behaviour from the first two. Graphically, we say that node c is *head-to-head* with respect to the path from a to b because it connects to the heads of the two arrows. When node c is unobserved, it 'blocks' the path, and the variables a and b are independent. However, conditioning on c 'unblocks' the path and renders a and b dependent.

There is one more subtlety associated with this third example that we need to consider. First we introduce some more terminology. We say that node y is a *descendant* of node x if there is a path from x to y in which each step of the path follows the directions of the arrows. Then it can be shown that a head-to-head path will become unblocked if either the node, *or any of its descendants*, is observed.

Exercise 8.10

In summary, a tail-to-tail node or a head-to-tail node leaves a path unblocked unless it is observed in which case it blocks the path. By contrast, a head-to-head node blocks a path if it is unobserved, but once the node, and/or at least one of its descendants, is observed the path becomes unblocked.

It is worth spending a moment to understand further the unusual behaviour of the graph of Figure 8.20. Consider a particular instance of such a graph corresponding to a problem with three binary random variables relating to the fuel system on a car, as shown in Figure 8.21. The variables are called B, representing the state of a battery that is either charged ($B = 1$) or flat ($B = 0$), F representing the state of the fuel tank that is either full of fuel ($F = 1$) or empty ($F = 0$), and G, which is the state of an electric fuel gauge and which indicates either full ($G = 1$) or empty

Figure 8.21 An example of a 3-node graph used to illustrate the phenomenon of 'explaining away'. The three nodes represent the state of the battery (B), the state of the fuel tank (F) and the reading on the electric fuel gauge (G). See the text for details.

($G = 0$). The battery is either charged or flat, and independently the fuel tank is either full or empty, with prior probabilities

$$\begin{aligned} p(B = 1) &= 0.9 \\ p(F = 1) &= 0.9. \end{aligned}$$

Given the state of the fuel tank and the battery, the fuel gauge reads full with probabilities given by

$$\begin{aligned} p(G = 1|B = 1, F = 1) &= 0.8 \\ p(G = 1|B = 1, F = 0) &= 0.2 \\ p(G = 1|B = 0, F = 1) &= 0.2 \\ p(G = 1|B = 0, F = 0) &= 0.1 \end{aligned}$$

so this is a rather unreliable fuel gauge! All remaining probabilities are determined by the requirement that probabilities sum to one, and so we have a complete specification of the probabilistic model.

Before we observe any data, the prior probability of the fuel tank being empty is $p(F = 0) = 0.1$. Now suppose that we observe the fuel gauge and discover that it reads empty, i.e., $G = 0$, corresponding to the middle graph in Figure 8.21. We can use Bayes' theorem to evaluate the posterior probability of the fuel tank being empty. First we evaluate the denominator for Bayes' theorem given by

$$p(G = 0) = \sum_{B \in \{0,1\}} \sum_{F \in \{0,1\}} p(G = 0|B, F)p(B)p(F) = 0.315 \tag{8.30}$$

and similarly we evaluate

$$p(G = 0|F = 0) = \sum_{B \in \{0,1\}} p(G = 0|B, F = 0)p(B) = 0.81 \tag{8.31}$$

and using these results we have

$$p(F = 0|G = 0) = \frac{p(G = 0|F = 0)p(F = 0)}{p(G = 0)} \simeq 0.257 \tag{8.32}$$

and so $p(F = 0|G = 0) > p(F = 0)$. Thus observing that the gauge reads empty makes it more likely that the tank is indeed empty, as we would intuitively expect. Next suppose that we also check the state of the battery and find that it is flat, i.e., $B = 0$. We have now observed the states of both the fuel gauge and the battery, as shown by the right-hand graph in Figure 8.21. The posterior probability that the fuel tank is empty given the observations of both the fuel gauge and the battery state is then given by

$$p(F = 0|G = 0, B = 0) = \frac{p(G = 0|B = 0, F = 0)p(F = 0)}{\sum_{F\in\{0,1\}} p(G = 0|B = 0, F)p(F)} \simeq 0.111 \quad (8.33)$$

where the prior probability $p(B = 0)$ has cancelled between numerator and denominator. Thus the probability that the tank is empty has *decreased* (from 0.257 to 0.111) as a result of the observation of the state of the battery. This accords with our intuition that finding out that the battery is flat *explains away* the observation that the fuel gauge reads empty. We see that the state of the fuel tank and that of the battery have indeed become dependent on each other as a result of observing the reading on the fuel gauge. In fact, this would also be the case if, instead of observing the fuel gauge directly, we observed the state of some descendant of G. Note that the probability $p(F = 0|G = 0, B = 0) \simeq 0.111$ is greater than the prior probability $p(F = 0) = 0.1$ because the observation that the fuel gauge reads zero still provides some evidence in favour of an empty fuel tank.

8.2.2 D-separation

We now give a general statement of the d-separation property (Pearl, 1988) for directed graphs. Consider a general directed graph in which A, B, and C are arbitrary nonintersecting sets of nodes (whose union may be smaller than the complete set of nodes in the graph). We wish to ascertain whether a particular conditional independence statement $A \perp\!\!\!\perp B \mid C$ is implied by a given directed acyclic graph. To do so, we consider all possible paths from any node in A to any node in B. Any such path is said to be *blocked* if it includes a node such that either

(a) the arrows on the path meet either head-to-tail or tail-to-tail at the node, and the node is in the set C, or

(b) the arrows meet head-to-head at the node, and neither the node, nor any of its descendants, is in the set C.

If all paths are blocked, then A is said to be d-separated from B by C, and the joint distribution over all of the variables in the graph will satisfy $A \perp\!\!\!\perp B \mid C$.

The concept of d-separation is illustrated in Figure 8.22. In graph (a), the path from a to b is not blocked by node f because it is a tail-to-tail node for this path and is not observed, nor is it blocked by node e because, although the latter is a head-to-head node, it has a descendant c because is in the conditioning set. Thus the conditional independence statement $a \perp\!\!\!\perp b \mid c$ does *not* follow from this graph. In graph (b), the path from a to b is blocked by node f because this is a tail-to-tail node that is observed, and so the conditional independence property $a \perp\!\!\!\perp b \mid f$ will

Figure 8.22 Illustration of the concept of d-separation. See the text for details.

be satisfied by any distribution that factorizes according to this graph. Note that this path is also blocked by node e because e is a head-to-head node and neither it nor its descendant are in the conditioning set.

For the purposes of d-separation, parameters such as α and σ^2 in Figure 8.5, indicated by small filled circles, behave in the same was as observed nodes. However, there are no marginal distributions associated with such nodes. Consequently parameter nodes never themselves have parents and so all paths through these nodes will always be tail-to-tail and hence blocked. Consequently they play no role in d-separation.

Another example of conditional independence and d-separation is provided by the concept of i.i.d. (independent identically distributed) data introduced in Section 1.2.4. Consider the problem of finding the posterior distribution for the mean of a univariate Gaussian distribution. This can be represented by the directed graph shown in Figure 8.23 in which the joint distribution is defined by a prior $p(\mu)$ together with a set of conditional distributions $p(x_n|\mu)$ for $n = 1, \ldots, N$. In practice, we observe $\mathcal{D} = \{x_1, \ldots, x_N\}$ and our goal is to infer μ. Suppose, for a moment, that we condition on μ and consider the joint distribution of the observations. Using d-separation, we note that there is a unique path from any x_i to any other $x_{j \neq i}$ and that this path is tail-to-tail with respect to the observed node μ. Every such path is blocked and so the observations $D = \{x_1, \ldots, x_N\}$ are independent given μ, so that

Section 2.3

$$p(\mathcal{D}|\mu) = \prod_{n=1}^{N} p(x_n|\mu). \tag{8.34}$$

Figure 8.23 (a) Directed graph corresponding to the problem of inferring the mean μ of a univariate Gaussian distribution from observations $x_1, \ldots, x_N$. (b) The same graph drawn using the plate notation.

Figure 8.24 A graphical representation of the 'naive Bayes' model for classification. Conditioned on the class label $\mathbf{z}$, the components of the observed vector $\mathbf{x} = (x_1, \ldots, x_D)^{\mathrm{T}}$ are assumed to be independent.

However, if we integrate over μ, the observations are in general no longer independent

$$p(\mathcal{D}) = \int_0^{\infty} p(\mathcal{D}|\mu)p(\mu)\,\mathrm{d}\mu \neq \prod_{n=1}^{N} p(x_n). \tag{8.35}$$

Here μ is a latent variable, because its value is not observed.

Another example of a model representing i.i.d. data is the graph in Figure 8.7 corresponding to Bayesian polynomial regression. Here the stochastic nodes correspond to $\{t_n\}$, $\mathbf{w}$ and $\widehat{t}$. We see that the node for $\mathbf{w}$ is tail-to-tail with respect to the path from $\widehat{t}$ to any one of the nodes t_n and so we have the following conditional independence property

$$\widehat{t} \perp\!\!\!\perp t_n \mid \mathbf{w}. \tag{8.36}$$

Thus, conditioned on the polynomial coefficients $\mathbf{w}$, the predictive distribution for $\widehat{t}$ is independent of the training data $\{t_1, \ldots, t_N\}$. We can therefore first use the training data to determine the posterior distribution over the coefficients $\mathbf{w}$ and then we can discard the training data and use the posterior distribution for $\mathbf{w}$ to make predictions of $\widehat{t}$ for new input observations $\widehat{x}$.

Section 3.3

A related graphical structure arises in an approach to classification called the *naive Bayes* model, in which we use conditional independence assumptions to simplify the model structure. Suppose our observed variable consists of a D-dimensional vector $\mathbf{x} = (x_1, \ldots, x_D)^{\mathrm{T}}$, and we wish to assign observed values of $\mathbf{x}$ to one of K classes. Using the 1-of-K encoding scheme, we can represent these classes by a K-dimensional binary vector $\mathbf{z}$. We can then define a generative model by introducing a multinomial prior $p(\mathbf{z}|\boldsymbol{\mu})$ over the class labels, where the kth component μ_k of $\boldsymbol{\mu}$ is the prior probability of class $\mathcal{C}_k$, together with a conditional distribution $p(\mathbf{x}|\mathbf{z})$ for the observed vector $\mathbf{x}$. The key assumption of the naive Bayes model is that, conditioned on the class $\mathbf{z}$, the distributions of the input variables $x_1, \ldots, x_D$ are independent. The graphical representation of this model is shown in Figure 8.24. We see that observation of $\mathbf{z}$ blocks the path between x_i and x_j for $j \neq i$ (because such paths are tail-to-tail at the node $\mathbf{z}$) and so x_i and x_j are conditionally independent given $\mathbf{z}$. If, however, we marginalize out $\mathbf{z}$ (so that $\mathbf{z}$ is unobserved) the tail-to-tail path from x_i to x_j is no longer blocked. This tells us that in general the marginal density $p(\mathbf{x})$ will not factorize with respect to the components of $\mathbf{x}$. We encountered a simple application of the naive Bayes model in the context of fusing data from different sources for medical diagnosis in Section 1.5.

If we are given a labelled training set, comprising inputs $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ together with their class labels, then we can fit the naive Bayes model to the training data

using maximum likelihood assuming that the data are drawn independently from the model. The solution is obtained by fitting the model for each class separately using the correspondingly labelled data. As an example, suppose that the probability density within each class is chosen to be Gaussian. In this case, the naive Bayes assumption then implies that the covariance matrix for each Gaussian is diagonal, and the contours of constant density within each class will be axis-aligned ellipsoids. The marginal density, however, is given by a superposition of diagonal Gaussians (with weighting coefficients given by the class priors) and so will no longer factorize with respect to its components.

The naive Bayes assumption is helpful when the dimensionality D of the input space is high, making density estimation in the full D-dimensional space more challenging. It is also useful if the input vector contains both discrete and continuous variables, since each can be represented separately using appropriate models (e.g., Bernoulli distributions for binary observations or Gaussians for real-valued variables). The conditional independence assumption of this model is clearly a strong one that may lead to rather poor representations of the class-conditional densities. Nevertheless, even if this assumption is not precisely satisfied, the model may still give good classification performance in practice because the decision boundaries can be insensitive to some of the details in the class-conditional densities, as illustrated in Figure 1.27.

We have seen that a particular directed graph represents a specific decomposition of a joint probability distribution into a product of conditional probabilities. The graph also expresses a set of conditional independence statements obtained through the d-separation criterion, and the d-separation theorem is really an expression of the equivalence of these two properties. In order to make this clear, it is helpful to think of a directed graph as a filter. Suppose we consider a particular joint probability distribution $p(\mathbf{x})$ over the variables $\mathbf{x}$ corresponding to the (nonobserved) nodes of the graph. The filter will allow this distribution to pass through if, and only if, it can be expressed in terms of the factorization (8.5) implied by the graph. If we present to the filter the set of all possible distributions $p(\mathbf{x})$ over the set of variables $\mathbf{x}$, then the subset of distributions that are passed by the filter will be denoted $\mathcal{DF}$, for *directed factorization*. This is illustrated in Figure 8.25. Alternatively, we can use the graph as a different kind of filter by first listing all of the conditional independence properties obtained by applying the d-separation criterion to the graph, and then allowing a distribution to pass only if it satisfies all of these properties. If we present all possible distributions $p(\mathbf{x})$ to this second kind of filter, then the d-separation theorem tells us that the set of distributions that will be allowed through is precisely the set $\mathcal{DF}$.

It should be emphasized that the conditional independence properties obtained from d-separation apply to any probabilistic model described by that particular directed graph. This will be true, for instance, whether the variables are discrete or continuous or a combination of these. Again, we see that a particular graph is describing a whole family of probability distributions.

At one extreme we have a fully connected graph that exhibits no conditional independence properties at all, and which can represent any possible joint probability distribution over the given variables. The set $\mathcal{DF}$ will contain all possible distribu-

Figure 8.25 We can view a graphical model (in this case a directed graph) as a filter in which a probability distribution $p(\mathbf{x})$ is allowed through the filter if, and only if, it satisfies the directed factorization property (8.5). The set of all possible probability distributions $p(\mathbf{x})$ that pass through the filter is denoted $\mathcal{DF}$. We can alternatively use the graph to filter distributions according to whether they respect all of the conditional independencies implied by the d-separation properties of the graph. The d-separation theorem says that it is the same set of distributions $\mathcal{DF}$ that will be allowed through this second kind of filter.

tions $p(\mathbf{x})$. At the other extreme, we have the fully disconnected graph, i.e., one having no links at all. This corresponds to joint distributions which factorize into the product of the marginal distributions over the variables comprising the nodes of the graph.

Note that for any given graph, the set of distributions $\mathcal{DF}$ will include any distributions that have additional independence properties beyond those described by the graph. For instance, a fully factorized distribution will always be passed through the filter implied by any graph over the corresponding set of variables.

We end our discussion of conditional independence properties by exploring the concept of a *Markov blanket* or *Markov boundary*. Consider a joint distribution $p(\mathbf{x}_1, \ldots, \mathbf{x}_D)$ represented by a directed graph having D nodes, and consider the conditional distribution of a particular node with variables $\mathbf{x}_i$ conditioned on all of the remaining variables $\mathbf{x}_{j \neq i}$. Using the factorization property (8.5), we can express this conditional distribution in the form

$$
\begin{aligned}
p(\mathbf{x}_i | \mathbf{x}_{\{j \neq i\}}) &= \frac{p(\mathbf{x}_1, \ldots, \mathbf{x}_D)}{\displaystyle\int p(\mathbf{x}_1, \ldots, \mathbf{x}_D)\, \mathrm{d}\mathbf{x}_i} \\
&= \frac{\displaystyle\prod_k p(\mathbf{x}_k | \mathrm{pa}_k)}{\displaystyle\int \prod_k p(\mathbf{x}_k | \mathrm{pa}_k)\, \mathrm{d}\mathbf{x}_i}
\end{aligned}
$$

in which the integral is replaced by a summation in the case of discrete variables. We now observe that any factor $p(\mathbf{x}_k | \mathrm{pa}_k)$ that does not have any functional dependence on $\mathbf{x}_i$ can be taken outside the integral over $\mathbf{x}_i$, and will therefore cancel between numerator and denominator. The only factors that remain will be the conditional distribution $p(\mathbf{x}_i | \mathrm{pa}_i)$ for node $\mathbf{x}_i$ itself, together with the conditional distributions for any nodes $\mathbf{x}_k$ such that node $\mathbf{x}_i$ is in the conditioning set of $p(\mathbf{x}_k | \mathrm{pa}_k)$, in other words for which $\mathbf{x}_i$ is a parent of $\mathbf{x}_k$. The conditional $p(\mathbf{x}_i | \mathrm{pa}_i)$ will depend on the parents of node $\mathbf{x}_i$, whereas the conditionals $p(\mathbf{x}_k | \mathrm{pa}_k)$ will depend on the children

Figure 8.26 The Markov blanket of a node $\mathbf{x}_i$ comprises the set of parents, children and co-parents of the node. It has the property that the conditional distribution of $\mathbf{x}_i$, conditioned on all the remaining variables in the graph, is dependent only on the variables in the Markov blanket.

of $\mathbf{x}_i$ as well as on the *co-parents*, in other words variables corresponding to parents of node $\mathbf{x}_k$ other than node $\mathbf{x}_i$. The set of nodes comprising the parents, the children and the co-parents is called the Markov blanket and is illustrated in Figure 8.26. We can think of the Markov blanket of a node $\mathbf{x}_i$ as being the minimal set of nodes that isolates $\mathbf{x}_i$ from the rest of the graph. Note that it is not sufficient to include only the parents and children of node $\mathbf{x}_i$ because the phenomenon of explaining away means that observations of the child nodes will not block paths to the co-parents. We must therefore observe the co-parent nodes also.

8.3. Markov Random Fields

We have seen that directed graphical models specify a factorization of the joint distribution over a set of variables into a product of local conditional distributions. They also define a set of conditional independence properties that must be satisfied by any distribution that factorizes according to the graph. We turn now to the second major class of graphical models that are described by undirected graphs and that again specify both a factorization and a set of conditional independence relations.

A *Markov random field*, also known as a *Markov network* or an *undirected graphical model* (Kindermann and Snell, 1980), has a set of nodes each of which corresponds to a variable or group of variables, as well as a set of links each of which connects a pair of nodes. The links are undirected, that is they do not carry arrows. In the case of undirected graphs, it is convenient to begin with a discussion of conditional independence properties.

8.3.1 Conditional independence properties

Section 8.2

In the case of directed graphs, we saw that it was possible to test whether a particular conditional independence property holds by applying a graphical test called d-separation. This involved testing whether or not the paths connecting two sets of nodes were 'blocked'. The definition of blocked, however, was somewhat subtle due to the presence of paths having head-to-head nodes. We might ask whether it is possible to define an alternative graphical semantics for probability distributions such that conditional independence is determined by simple graph separation. This is indeed the case and corresponds to undirected graphical models. By removing the

Figure 8.27 An example of an undirected graph in which every path from any node in set A to any node in set B passes through at least one node in set C. Consequently the conditional independence property $A \perp\!\!\!\perp B \mid C$ holds for any probability distribution described by this graph.

directionality from the links of the graph, the asymmetry between parent and child nodes is removed, and so the subtleties associated with head-to-head nodes no longer arise.

Suppose that in an undirected graph we identify three sets of nodes, denoted A, B, and C, and that we consider the conditional independence property

$$A \perp\!\!\!\perp B \mid C. \tag{8.37}$$

To test whether this property is satisfied by a probability distribution defined by a graph we consider all possible paths that connect nodes in set A to nodes in set B. If all such paths pass through one or more nodes in set C, then all such paths are 'blocked' and so the conditional independence property holds. However, if there is at least one such path that is not blocked, then the property does not necessarily hold, or more precisely there will exist at least some distributions corresponding to the graph that do not satisfy this conditional independence relation. This is illustrated with an example in Figure 8.27. Note that this is exactly the same as the d-separation criterion except that there is no 'explaining away' phenomenon. Testing for conditional independence in undirected graphs is therefore simpler than in directed graphs.

An alternative way to view the conditional independence test is to imagine removing all nodes in set C from the graph together with any links that connect to those nodes. We then ask if there exists a path that connects any node in A to any node in B. If there are no such paths, then the conditional independence property must hold.

The Markov blanket for an undirected graph takes a particularly simple form, because a node will be conditionally independent of all other nodes conditioned only on the neighbouring nodes, as illustrated in Figure 8.28.

8.3.2 Factorization properties

We now seek a factorization rule for undirected graphs that will correspond to the above conditional independence test. Again, this will involve expressing the joint distribution $p(\mathbf{x})$ as a product of functions defined over sets of variables that are local to the graph. We therefore need to decide what is the appropriate notion of locality in this case.

Figure 8.28 For an undirected graph, the Markov blanket of a node x_i consists of the set of neighbouring nodes. It has the property that the conditional distribution of x_i, conditioned on all the remaining variables in the graph, is dependent only on the variables in the Markov blanket.

If we consider two nodes x_i and x_j that are not connected by a link, then these variables must be conditionally independent given all other nodes in the graph. This follows from the fact that there is no direct path between the two nodes, and all other paths pass through nodes that are observed, and hence those paths are blocked. This conditional independence property can be expressed as

$$p(x_i, x_j|\mathbf{x}_{\setminus\{i,j\}}) = p(x_i|\mathbf{x}_{\setminus\{i,j\}})p(x_j|\mathbf{x}_{\setminus\{i,j\}}) \tag{8.38}$$

where $\mathbf{x}_{\setminus\{i,j\}}$ denotes the set $\mathbf{x}$ of all variables with x_i and x_j removed. The factorization of the joint distribution must therefore be such that x_i and x_j do not appear in the same factor in order for the conditional independence property to hold for all possible distributions belonging to the graph.

This leads us to consider a graphical concept called a *clique*, which is defined as a subset of the nodes in a graph such that there exists a link between all pairs of nodes in the subset. In other words, the set of nodes in a clique is fully connected. Furthermore, a *maximal clique* is a clique such that it is not possible to include any other nodes from the graph in the set without it ceasing to be a clique. These concepts are illustrated by the undirected graph over four variables shown in Figure 8.29. This graph has five cliques of two nodes given by $\{x_1, x_2\}$, $\{x_2, x_3\}$, $\{x_3, x_4\}$, $\{x_4, x_2\}$, and $\{x_1, x_3\}$, as well as two maximal cliques given by $\{x_1, x_2, x_3\}$ and $\{x_2, x_3, x_4\}$. The set $\{x_1, x_2, x_3, x_4\}$ is not a clique because of the missing link from x_1 to x_4.

We can therefore define the factors in the decomposition of the joint distribution to be functions of the variables in the cliques. In fact, we can consider functions of the maximal cliques, without loss of generality, because other cliques must be subsets of maximal cliques. Thus, if $\{x_1, x_2, x_3\}$ is a maximal clique and we define an arbitrary function over this clique, then including another factor defined over a subset of these variables would be redundant.

Let us denote a clique by C and the set of variables in that clique by $\mathbf{x}_C$. Then

Figure 8.29 A four-node undirected graph showing a clique (outlined in green) and a maximal clique (outlined in blue).

the joint distribution is written as a product of *potential functions* $\psi_C(\mathbf{x}_C)$ over the maximal cliques of the graph

$$p(\mathbf{x}) = \frac{1}{Z} \prod_C \psi_C(\mathbf{x}_C). \tag{8.39}$$

Here the quantity Z, sometimes called the *partition function*, is a normalization constant and is given by

$$Z = \sum_{\mathbf{x}} \prod_C \psi_C(\mathbf{x}_C) \tag{8.40}$$

which ensures that the distribution $p(\mathbf{x})$ given by (8.39) is correctly normalized. By considering only potential functions which satisfy $\psi_C(\mathbf{x}_C) \geqslant 0$ we ensure that $p(\mathbf{x}) \geqslant 0$. In (8.40) we have assumed that $\mathbf{x}$ comprises discrete variables, but the framework is equally applicable to continuous variables, or a combination of the two, in which the summation is replaced by the appropriate combination of summation and integration.

Note that we do not restrict the choice of potential functions to those that have a specific probabilistic interpretation as marginal or conditional distributions. This is in contrast to directed graphs in which each factor represents the conditional distribution of the corresponding variable, conditioned on the state of its parents. However, in special cases, for instance where the undirected graph is constructed by starting with a directed graph, the potential functions may indeed have such an interpretation, as we shall see shortly.

One consequence of the generality of the potential functions $\psi_C(\mathbf{x}_C)$ is that their product will in general not be correctly normalized. We therefore have to introduce an explicit normalization factor given by (8.40). Recall that for directed graphs, the joint distribution was automatically normalized as a consequence of the normalization of each of the conditional distributions in the factorization.

The presence of this normalization constant is one of the major limitations of undirected graphs. If we have a model with M discrete nodes each having K states, then the evaluation of the normalization term involves summing over K^M states and so (in the worst case) is exponential in the size of the model. The partition function is needed for parameter learning because it will be a function of any parameters that govern the potential functions $\psi_C(\mathbf{x}_C)$. However, for evaluation of local conditional distributions, the partition function is not needed because a conditional is the ratio of two marginals, and the partition function cancels between numerator and denominator when evaluating this ratio. Similarly, for evaluating local marginal probabilities we can work with the unnormalized joint distribution and then normalize the marginals explicitly at the end. Provided the marginals only involves a small number of variables, the evaluation of their normalization coefficient will be feasible.

So far, we have discussed the notion of conditional independence based on simple graph separation and we have proposed a factorization of the joint distribution that is intended to correspond to this conditional independence structure. However, we have not made any formal connection between conditional independence and factorization for undirected graphs. To do so we need to restrict attention to potential functions $\psi_C(\mathbf{x}_C)$ that are strictly positive (i.e., never zero or negative for any

choice of $\mathbf{x}_C$). Given this restriction, we can make a precise relationship between factorization and conditional independence.

To do this we again return to the concept of a graphical model as a filter, corresponding to Figure 8.25. Consider the set of all possible distributions defined over a fixed set of variables corresponding to the nodes of a particular undirected graph. We can define $\mathcal{UI}$ to be the set of such distributions that are consistent with the set of conditional independence statements that can be read from the graph using graph separation. Similarly, we can define $\mathcal{UF}$ to be the set of such distributions that can be expressed as a factorization of the form (8.39) with respect to the maximal cliques of the graph. The *Hammersley-Clifford* theorem (Clifford, 1990) states that the sets $\mathcal{UI}$ and $\mathcal{UF}$ are identical.

Because we are restricted to potential functions which are strictly positive it is convenient to express them as exponentials, so that

$$\psi_C(\mathbf{x}_C) = \exp\{-E(\mathbf{x}_C)\} \tag{8.41}$$

where $E(\mathbf{x}_C)$ is called an *energy function*, and the exponential representation is called the *Boltzmann distribution*. The joint distribution is defined as the product of potentials, and so the total energy is obtained by adding the energies of each of the maximal cliques.

In contrast to the factors in the joint distribution for a directed graph, the potentials in an undirected graph do not have a specific probabilistic interpretation. Although this gives greater flexibility in choosing the potential functions, because there is no normalization constraint, it does raise the question of how to motivate a choice of potential function for a particular application. This can be done by viewing the potential function as expressing which configurations of the local variables are preferred to others. Global configurations that have a relatively high probability are those that find a good balance in satisfying the (possibly conflicting) influences of the clique potentials. We turn now to a specific example to illustrate the use of undirected graphs.

8.3.3 Illustration: Image de-noising

We can illustrate the application of undirected graphs using an example of noise removal from a binary image (Besag, 1974; Geman and Geman, 1984; Besag, 1986). Although a very simple example, this is typical of more sophisticated applications. Let the observed noisy image be described by an array of binary pixel values $y_i \in \{-1, +1\}$, where the index $i = 1, \ldots, D$ runs over all pixels. We shall suppose that the image is obtained by taking an unknown noise-free image, described by binary pixel values $x_i \in \{-1, +1\}$ and randomly flipping the sign of pixels with some small probability. An example binary image, together with a noise corrupted image obtained by flipping the sign of the pixels with probability 10%, is shown in Figure 8.30. Given the noisy image, our goal is to recover the original noise-free image.

Because the noise level is small, we know that there will be a strong correlation between x_i and y_i. We also know that neighbouring pixels x_i and x_j in an image are strongly correlated. This prior knowledge can be captured using the Markov

Figure 8.30 Illustration of image de-noising using a Markov random field. The top row shows the original binary image on the left and the corrupted image after randomly changing 10% of the pixels on the right. The bottom row shows the restored images obtained using iterated conditional models (ICM) on the left and using the graph-cut algorithm on the right. ICM produces an image where 96% of the pixels agree with the original image, whereas the corresponding number for graph-cut is 99%.

random field model whose undirected graph is shown in Figure 8.31. This graph has two types of cliques, each of which contains two variables. The cliques of the form $\{x_i, y_i\}$ have an associated energy function that expresses the correlation between these variables. We choose a very simple energy function for these cliques of the form $-\eta x_i y_i$ where η is a positive constant. This has the desired effect of giving a lower energy (thus encouraging a higher probability) when x_i and y_i have the same sign and a higher energy when they have the opposite sign.

The remaining cliques comprise pairs of variables $\{x_i, x_j\}$ where i and j are indices of neighbouring pixels. Again, we want the energy to be lower when the pixels have the same sign than when they have the opposite sign, and so we choose an energy given by $-\beta x_i x_j$ where β is a positive constant.

Because a potential function is an arbitrary, nonnegative function over a maximal clique, we can multiply it by any nonnegative functions of subsets of the clique, or

Figure 8.31 An undirected graphical model representing a Markov random field for image de-noising, in which x_i is a binary variable denoting the state of pixel i in the unknown noise-free image, and y_i denotes the corresponding value of pixel i in the observed noisy image.

equivalently we can add the corresponding energies. In this example, this allows us to add an extra term hx_i for each pixel i in the noise-free image. Such a term has the effect of biasing the model towards pixel values that have one particular sign in preference to the other.

The complete energy function for the model then takes the form

$$E(\mathbf{x}, \mathbf{y}) = h \sum_i x_i - \beta \sum_{\{i,j\}} x_i x_j - \eta \sum_i x_i y_i \tag{8.42}$$

which defines a joint distribution over $\mathbf{x}$ and $\mathbf{y}$ given by

$$p(\mathbf{x}, \mathbf{y}) = \frac{1}{Z} \exp\{-E(\mathbf{x}, \mathbf{y})\}. \tag{8.43}$$

We now fix the elements of $\mathbf{y}$ to the observed values given by the pixels of the noisy image, which implicitly defines a conditional distribution $p(\mathbf{x}|\mathbf{y})$ over noise-free images. This is an example of the *Ising model*, which has been widely studied in statistical physics. For the purposes of image restoration, we wish to find an image $\mathbf{x}$ having a high probability (ideally the maximum probability). To do this we shall use a simple iterative technique called *iterated conditional modes*, or *ICM* (Kittler and Föglein, 1984), which is simply an application of coordinate-wise gradient ascent. The idea is first to initialize the variables $\{x_i\}$, which we do by simply setting $x_i = y_i$ for all i. Then we take one node x_j at a time and we evaluate the total energy for the two possible states $x_j = +1$ and $x_j = -1$, keeping all other node variables fixed, and set x_j to whichever state has the lower energy. This will either leave the probability unchanged, if x_j is unchanged, or will increase it. Because only one variable is changed, this is a simple local computation that can be performed efficiently. We then repeat the update for another site, and so on, until some suitable stopping criterion is satisfied. The nodes may be updated in a systematic way, for instance by repeatedly raster scanning through the image, or by choosing nodes at random.

Exercise 8.13

If we have a sequence of updates in which every site is visited at least once, and in which no changes to the variables are made, then by definition the algorithm

Figure 8.32 (a) Example of a directed graph. (b) The equivalent undirected graph.

will have converged to a local maximum of the probability. This need not, however, correspond to the global maximum.

For the purposes of this simple illustration, we have fixed the parameters to be $\beta = 1.0$, $\eta = 2.1$ and $h = 0$. Note that leaving $h = 0$ simply means that the prior probabilities of the two states of x_i are equal. Starting with the observed noisy image as the initial configuration, we run ICM until convergence, leading to the de-noised image shown in the lower left panel of Figure 8.30. Note that if we set $\beta = 0$, which effectively removes the links between neighbouring pixels, then the global most probable solution is given by $x_i = y_i$ for all i, corresponding to the observed noisy image.

Exercise 8.14

Later we shall discuss a more effective algorithm for finding high probability solutions called the max-product algorithm, which typically leads to better solutions, although this is still not guaranteed to find the global maximum of the posterior distribution. However, for certain classes of model, including the one given by (8.42), there exist efficient algorithms based on *graph cuts* that are guaranteed to find the global maximum (Greig *et al.*, 1989; Boykov *et al.*, 2001; Kolmogorov and Zabih, 2004). The lower right panel of Figure 8.30 shows the result of applying a graph-cut algorithm to the de-noising problem.

Section 8.4

8.3.4 Relation to directed graphs

We have introduced two graphical frameworks for representing probability distributions, corresponding to directed and undirected graphs, and it is instructive to discuss the relation between these. Consider first the problem of taking a model that is specified using a directed graph and trying to convert it to an undirected graph. In some cases this is straightforward, as in the simple example in Figure 8.32. Here the joint distribution for the directed graph is given as a product of conditionals in the form

$$p(\mathbf{x}) = p(x_1)p(x_2|x_1)p(x_3|x_2)\cdots p(x_N|x_{N-1}). \tag{8.44}$$

Now let us convert this to an undirected graph representation, as shown in Figure 8.32. In the undirected graph, the maximal cliques are simply the pairs of neighbouring nodes, and so from (8.39) we wish to write the joint distribution in the form

$$p(\mathbf{x}) = \frac{1}{Z}\psi_{1,2}(x_1, x_2)\psi_{2,3}(x_2, x_3)\cdots\psi_{N-1,N}(x_{N-1}, x_N). \tag{8.45}$$

Figure 8.33 Example of a simple directed graph (a) and the corresponding moral graph (b).

This is easily done by identifying

$$\begin{aligned}
\psi_{1,2}(x_1, x_2) &= p(x_1)p(x_2|x_1) \\
\psi_{2,3}(x_2, x_3) &= p(x_3|x_2) \\
&\vdots \\
\psi_{N-1,N}(x_{N-1}, x_N) &= p(x_N|x_{N-1})
\end{aligned}$$

where we have absorbed the marginal $p(x_1)$ for the first node into the first potential function. Note that in this case, the partition function $Z = 1$.

Let us consider how to generalize this construction, so that we can convert any distribution specified by a factorization over a directed graph into one specified by a factorization over an undirected graph. This can be achieved if the clique potentials of the undirected graph are given by the conditional distributions of the directed graph. In order for this to be valid, we must ensure that the set of variables that appears in each of the conditional distributions is a member of at least one clique of the undirected graph. For nodes on the directed graph having just one parent, this is achieved simply by replacing the directed link with an undirected link. However, for nodes in the directed graph having more than one parent, this is not sufficient. These are nodes that have 'head-to-head' paths encountered in our discussion of conditional independence. Consider a simple directed graph over 4 nodes shown in Figure 8.33. The joint distribution for the directed graph takes the form

$$p(\mathbf{x}) = p(x_1)p(x_2)p(x_3)p(x_4|x_1, x_2, x_3). \tag{8.46}$$

We see that the factor $p(x_4|x_1, x_2, x_3)$ involves the four variables x_1, x_2, x_3, and x_4, and so these must all belong to a single clique if this conditional distribution is to be absorbed into a clique potential. To ensure this, we add extra links between all pairs of parents of the node x_4. Anachronistically, this process of 'marrying the parents' has become known as *moralization*, and the resulting undirected graph, after dropping the arrows, is called the *moral graph*. It is important to observe that the moral graph in this example is fully connected and so exhibits no conditional independence properties, in contrast to the original directed graph.

Thus in general to convert a directed graph into an undirected graph, we first add additional undirected links between all pairs of parents for each node in the graph and

then drop the arrows on the original links to give the moral graph. Then we initialize all of the clique potentials of the moral graph to 1. We then take each conditional distribution factor in the original directed graph and multiply it into one of the clique potentials. There will always exist at least one maximal clique that contains all of the variables in the factor as a result of the moralization step. Note that in all cases the partition function is given by $Z = 1$.

The process of converting a directed graph into an undirected graph plays an important role in exact inference techniques such as the *junction tree algorithm*. Converting from an undirected to a directed representation is much less common and in general presents problems due to the normalization constraints.

Section 8.4

We saw that in going from a directed to an undirected representation we had to discard some conditional independence properties from the graph. Of course, we could always trivially convert any distribution over a directed graph into one over an undirected graph by simply using a fully connected undirected graph. This would, however, discard all conditional independence properties and so would be vacuous. The process of moralization adds the fewest extra links and so retains the maximum number of independence properties.

We have seen that the procedure for determining the conditional independence properties is different between directed and undirected graphs. It turns out that the two types of graph can express different conditional independence properties, and it is worth exploring this issue in more detail. To do so, we return to the view of a specific (directed or undirected) graph as a filter, so that the set of all possible distributions over the given variables could be reduced to a subset that respects the conditional independencies implied by the graph. A graph is said to be a *D map* (for 'dependency map') of a distribution if every conditional independence statement satisfied by the distribution is reflected in the graph. Thus a completely disconnected graph (no links) will be a trivial D map for any distribution.

Section 8.2

Alternatively, we can consider a specific distribution and ask which graphs have the appropriate conditional independence properties. If every conditional independence statement implied by a graph is satisfied by a specific distribution, then the graph is said to be an *I map* (for 'independence map') of that distribution. Clearly a fully connected graph will be a trivial I map for any distribution.

If it is the case that every conditional independence property of the distribution is reflected in the graph, and vice versa, then the graph is said to be a *perfect map* for

Figure 8.34 Venn diagram illustrating the set of all distributions P over a given set of variables, together with the set of distributions D that can be represented as a perfect map using a directed graph, and the set U that can be represented as a perfect map using an undirected graph.

Figure 8.35 A directed graph whose conditional independence properties cannot be expressed using an undirected graph over the same three variables.

that distribution. A perfect map is therefore both an I map and a D map.

Consider the set of distributions such that for each distribution there exists a directed graph that is a perfect map. This set is distinct from the set of distributions such that for each distribution there exists an undirected graph that is a perfect map. In addition there are distributions for which neither directed nor undirected graphs offer a perfect map. This is illustrated as a Venn diagram in Figure 8.34.

Figure 8.35 shows an example of a directed graph that is a perfect map for a distribution satisfying the conditional independence properties $A \perp\!\!\!\perp B \mid \emptyset$ and $A \not\perp\!\!\!\perp B \mid C$. There is no corresponding undirected graph over the same three variables that is a perfect map.

Conversely, consider the undirected graph over four variables shown in Figure 8.36. This graph exhibits the properties $A \not\perp\!\!\!\perp B \mid \emptyset$, $C \perp\!\!\!\perp D \mid A \cup B$ and $A \perp\!\!\!\perp B \mid C \cup D$. There is no directed graph over four variables that implies the same set of conditional independence properties.

The graphical framework can be extended in a consistent way to graphs that include both directed and undirected links. These are called *chain graphs* (Lauritzen and Wermuth, 1989; Frydenberg, 1990), and contain the directed and undirected graphs considered so far as special cases. Although such graphs can represent a broader class of distributions than either directed or undirected alone, there remain distributions for which even a chain graph cannot provide a perfect map. Chain graphs are not discussed further in this book.

Figure 8.36 An undirected graph whose conditional independence properties cannot be expressed in terms of a directed graph over the same variables.

8.4. Inference in Graphical Models

We turn now to the problem of inference in graphical models, in which some of the nodes in a graph are clamped to observed values, and we wish to compute the posterior distributions of one or more subsets of other nodes. As we shall see, we can exploit the graphical structure both to find efficient algorithms for inference, and

Figure 8.37 A graphical representation of Bayes' theorem. See the text for details.

to make the structure of those algorithms transparent. Specifically, we shall see that many algorithms can be expressed in terms of the propagation of local *messages* around the graph. In this section, we shall focus primarily on techniques for exact inference, and in Chapter 10 we shall consider a number of approximate inference algorithms.

To start with, let us consider the graphical interpretation of Bayes' theorem. Suppose we decompose the joint distribution $p(x, y)$ over two variables x and y into a product of factors in the form $p(x, y) = p(x)p(y|x)$. This can be represented by the directed graph shown in Figure 8.37(a). Now suppose we observe the value of y, as indicated by the shaded node in Figure 8.37(b). We can view the marginal distribution $p(x)$ as a prior over the latent variable x, and our goal is to infer the corresponding posterior distribution over x. Using the sum and product rules of probability we can evaluate

$$p(y) = \sum_{x'} p(y|x')p(x') \tag{8.47}$$

which can then be used in Bayes' theorem to calculate

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)}. \tag{8.48}$$

Thus the joint distribution is now expressed in terms of $p(y)$ and $p(x|y)$. From a graphical perspective, the joint distribution $p(x, y)$ is now represented by the graph shown in Figure 8.37(c), in which the direction of the arrow is reversed. This is the simplest example of an inference problem for a graphical model.

8.4.1 Inference on a chain

Now consider a more complex problem involving the chain of nodes of the form shown in Figure 8.32. This example will lay the foundation for a discussion of exact inference in more general graphs later in this section.

Specifically, we shall consider the undirected graph in Figure 8.32(b). We have already seen that the directed chain can be transformed into an equivalent undirected chain. Because the directed graph does not have any nodes with more than one parent, this does not require the addition of any extra links, and the directed and undirected versions of this graph express exactly the same set of conditional independence statements.

The joint distribution for this graph takes the form

$$p(\mathbf{x}) = \frac{1}{Z}\psi_{1,2}(x_1, x_2)\psi_{2,3}(x_2, x_3)\cdots\psi_{N-1,N}(x_{N-1}, x_N). \tag{8.49}$$

We shall consider the specific case in which the N nodes represent discrete variables each having K states, in which case each potential function $\psi_{n-1,n}(x_{n-1}, x_n)$ comprises an $K \times K$ table, and so the joint distribution has $(N-1)K^2$ parameters.

Let us consider the inference problem of finding the marginal distribution $p(x_n)$ for a specific node x_n that is part way along the chain. Note that, for the moment, there are no observed nodes. By definition, the required marginal is obtained by summing the joint distribution over all variables except x_n, so that

$$p(x_n) = \sum_{x_1}\cdots\sum_{x_{n-1}}\sum_{x_{n+1}}\cdots\sum_{x_N} p(\mathbf{x}). \tag{8.50}$$

In a naive implementation, we would first evaluate the joint distribution and then perform the summations explicitly. The joint distribution can be represented as a set of numbers, one for each possible value for $\mathbf{x}$. Because there are N variables each with K states, there are K^N values for $\mathbf{x}$ and so evaluation and storage of the joint distribution, as well as marginalization to obtain $p(x_n)$, all involve storage and computation that scale exponentially with the length N of the chain.

We can, however, obtain a much more efficient algorithm by exploiting the conditional independence properties of the graphical model. If we substitute the factorized expression (8.49) for the joint distribution into (8.50), then we can rearrange the order of the summations and the multiplications to allow the required marginal to be evaluated much more efficiently. Consider for instance the summation over x_N. The potential $\psi_{N-1,N}(x_{N-1}, x_N)$ is the only one that depends on x_N, and so we can perform the summation

$$\sum_{x_N}\psi_{N-1,N}(x_{N-1}, x_N) \tag{8.51}$$

first to give a function of x_{N-1}. We can then use this to perform the summation over x_{N-1}, which will involve only this new function together with the potential $\psi_{N-2,N-1}(x_{N-2}, x_{N-1})$, because this is the only other place that x_{N-1} appears. Similarly, the summation over x_1 involves only the potential $\psi_{1,2}(x_1, x_2)$ and so can be performed separately to give a function of x_2, and so on. Because each summation effectively removes a variable from the distribution, this can be viewed as the removal of a node from the graph.

If we group the potentials and summations together in this way, we can express

the desired marginal in the form

$$p(x_n) = \frac{1}{Z} \underbrace{\left[\sum_{x_{n-1}} \psi_{n-1,n}(x_{n-1}, x_n) \cdots \left[\sum_{x_2} \psi_{2,3}(x_2, x_3) \left[\sum_{x_1} \psi_{1,2}(x_1, x_2) \right] \right] \cdots \right]}_{\mu_\alpha(x_n)} \underbrace{\left[\sum_{x_{n+1}} \psi_{n,n+1}(x_n, x_{n+1}) \cdots \left[\sum_{x_N} \psi_{N-1,N}(x_{N-1}, x_N) \right] \cdots \right]}_{\mu_\beta(x_n)}. \tag{8.52}$$

The reader is encouraged to study this re-ordering carefully as the underlying idea forms the basis for the later discussion of the general sum-product algorithm. Here the key concept that we are exploiting is that multiplication is distributive over addition, so that

$$ab + ac = a(b + c) \tag{8.53}$$

in which the left-hand side involves three arithmetic operations whereas the right-hand side reduces this to two operations.

Let us work out the computational cost of evaluating the required marginal using this re-ordered expression. We have to perform $N - 1$ summations each of which is over K states and each of which involves a function of two variables. For instance, the summation over x_1 involves only the function $\psi_{1,2}(x_1, x_2)$, which is a table of $K \times K$ numbers. We have to sum this table over x_1 for each value of x_2 and so this has $O(K^2)$ cost. The resulting vector of K numbers is multiplied by the matrix of numbers $\psi_{2,3}(x_2, x_3)$ and so is again $O(K^2)$. Because there are $N - 1$ summations and multiplications of this kind, the total cost of evaluating the marginal $p(x_n)$ is $O(NK^2)$. This is linear in the length of the chain, in contrast to the exponential cost of a naive approach. We have therefore been able to exploit the many conditional independence properties of this simple graph in order to obtain an efficient calculation. If the graph had been fully connected, there would have been no conditional independence properties, and we would have been forced to work directly with the full joint distribution.

We now give a powerful interpretation of this calculation in terms of the passing of local *messages* around on the graph. From (8.52) we see that the expression for the marginal $p(x_n)$ decomposes into the product of two factors times the normalization constant

$$p(x_n) = \frac{1}{Z} \mu_\alpha(x_n) \mu_\beta(x_n). \tag{8.54}$$

We shall interpret $\mu_\alpha(x_n)$ as a message passed forwards along the chain from node x_{n-1} to node x_n. Similarly, $\mu_\beta(x_n)$ can be viewed as a message passed backwards

Figure 8.38 The marginal distribution $p(x_n)$ for a node x_n along the chain is obtained by multiplying the two messages $\mu_\alpha(x_n)$ and $\mu_\beta(x_n)$, and then normalizing. These messages can themselves be evaluated recursively by passing messages from both ends of the chain towards node x_n.

along the chain to node x_n from node x_{n+1}. Note that each of the messages comprises a set of K values, one for each choice of x_n, and so the product of two messages should be interpreted as the point-wise multiplication of the elements of the two messages to give another set of K values.

The message $\mu_\alpha(x_n)$ can be evaluated recursively because

$$\begin{aligned}\mu_\alpha(x_n) &= \sum_{x_{n-1}} \psi_{n-1,n}(x_{n-1}, x_n) \left[\sum_{x_{n-2}} \cdots \right] \\ &= \sum_{x_{n-1}} \psi_{n-1,n}(x_{n-1}, x_n) \mu_\alpha(x_{n-1}). \end{aligned} \tag{8.55}$$

We therefore first evaluate

$$\mu_\alpha(x_2) = \sum_{x_1} \psi_{1,2}(x_1, x_2) \tag{8.56}$$

and then apply (8.55) repeatedly until we reach the desired node. Note carefully the structure of the message passing equation. The outgoing message $\mu_\alpha(x_n)$ in (8.55) is obtained by multiplying the incoming message $\mu_\alpha(x_{n-1})$ by the local potential involving the node variable and the outgoing variable and then summing over the node variable.

Similarly, the message $\mu_\beta(x_n)$ can be evaluated recursively by starting with node x_N and using

$$\begin{aligned}\mu_\beta(x_n) &= \sum_{x_{n+1}} \psi_{n+1,n}(x_{n+1}, x_n) \left[\sum_{x_{n+2}} \cdots \right] \\ &= \sum_{x_{n+1}} \psi_{n+1,n}(x_{n+1}, x_n) \mu_\beta(x_{n+1}). \end{aligned} \tag{8.57}$$

This recursive message passing is illustrated in Figure 8.38. The normalization constant Z is easily evaluated by summing the right-hand side of (8.54) over all states of x_n, an operation that requires only $O(K)$ computation.

Graphs of the form shown in Figure 8.38 are called *Markov chains*, and the corresponding message passing equations represent an example of the *Chapman-Kolmogorov* equations for Markov processes (Papoulis, 1984).

Now suppose we wish to evaluate the marginals $p(x_n)$ for every node $n \in \{1, \ldots, N\}$ in the chain. Simply applying the above procedure separately for each node will have computational cost that is $O(N^2M^2)$. However, such an approach would be very wasteful of computation. For instance, to find $p(x_1)$ we need to propagate a message $\mu_\beta(\cdot)$ from node x_N back to node x_2. Similarly, to evaluate $p(x_2)$ we need to propagate a messages $\mu_\beta(\cdot)$ from node x_N back to node x_3. This will involve much duplicated computation because most of the messages will be identical in the two cases.

Suppose instead we first launch a message $\mu_\beta(x_{N-1})$ starting from node x_N and propagate corresponding messages all the way back to node x_1, and suppose we similarly launch a message $\mu_\alpha(x_2)$ starting from node x_1 and propagate the corresponding messages all the way forward to node x_N. Provided we store all of the intermediate messages along the way, then any node can evaluate its marginal simply by applying (8.54). The computational cost is only twice that for finding the marginal of a single node, rather than N times as much. Observe that a message has passed once in each direction across each link in the graph. Note also that the normalization constant Z need be evaluated only once, using any convenient node.

If some of the nodes in the graph are observed, then the corresponding variables are simply clamped to their observed values and there is no summation. To see this, note that the effect of clamping a variable x_n to an observed value $\widehat{x}_n$ can be expressed by multiplying the joint distribution by (one or more copies of) an additional function $I(x_n, \widehat{x}_n)$, which takes the value 1 when $x_n = \widehat{x}_n$ and the value 0 otherwise. One such function can then be absorbed into each of the potentials that contain x_n. Summations over x_n then contain only one term in which $x_n = \widehat{x}_n$.

Now suppose we wish to calculate the joint distribution $p(x_{n-1}, x_n)$ for two neighbouring nodes on the chain. This is similar to the evaluation of the marginal for a single node, except that there are now two variables that are not summed out. A few moments thought will show that the required joint distribution can be written in the form

Exercise 8.15

$$p(x_{n-1}, x_n) = \frac{1}{Z}\mu_\alpha(x_{n-1})\psi_{n-1,n}(x_{n-1}, x_n)\mu_\beta(x_n). \tag{8.58}$$

Thus we can obtain the joint distributions over all of the sets of variables in each of the potentials directly once we have completed the message passing required to obtain the marginals.

This is a useful result because in practice we may wish to use parametric forms for the clique potentials, or equivalently for the conditional distributions if we started from a directed graph. In order to learn the parameters of these potentials in situations where not all of the variables are observed, we can employ the *EM algorithm*, and it turns out that the local joint distributions of the cliques, conditioned on any observed data, is precisely what is needed in the E step. We shall consider some examples of this in detail in Chapter 13.

Chapter 9

8.4.2 Trees

We have seen that exact inference on a graph comprising a chain of nodes can be performed efficiently in time that is linear in the number of nodes, using an algorithm

Figure 8.39 Examples of tree-structured graphs, showing (a) an undirected tree, (b) a directed tree, and (c) a directed polytree.

that can be interpreted in terms of messages passed along the chain. More generally, inference can be performed efficiently using local message passing on a broader class of graphs called *trees*. In particular, we shall shortly generalize the message passing formalism derived above for chains to give the *sum-product* algorithm, which provides an efficient framework for exact inference in tree-structured graphs.

In the case of an undirected graph, a tree is defined as a graph in which there is one, and only one, path between any pair of nodes. Such graphs therefore do not have loops. In the case of directed graphs, a tree is defined such that there is a single node, called the *root*, which has no parents, and all other nodes have one parent. If we convert a directed tree into an undirected graph, we see that the moralization step will not add any links as all nodes have at most one parent, and as a consequence the corresponding moralized graph will be an undirected tree. Examples of undirected and directed trees are shown in Figure 8.39(a) and 8.39(b). Note that a distribution represented as a directed tree can easily be converted into one represented by an undirected tree, and vice versa.

Exercise 8.18

If there are nodes in a directed graph that have more than one parent, but there is still only one path (ignoring the direction of the arrows) between any two nodes, then the graph is a called a *polytree*, as illustrated in Figure 8.39(c). Such a graph will have more than one node with the property of having no parents, and furthermore, the corresponding moralized undirected graph will have loops.

8.4.3 Factor graphs

The sum-product algorithm that we derive in the next section is applicable to undirected and directed trees and to polytrees. It can be cast in a particularly simple and general form if we first introduce a new graphical construction called a *factor graph* (Frey, 1998; Kschischnang *et al.*, 2001).

Both directed and undirected graphs allow a global function of several variables to be expressed as a product of factors over subsets of those variables. Factor graphs make this decomposition explicit by introducing additional nodes for the factors themselves in addition to the nodes representing the variables. They also allow us to be more explicit about the details of the factorization, as we shall see.

Let us write the joint distribution over a set of variables in the form of a product of factors

$$p(\mathbf{x}) = \prod_s f_s(\mathbf{x}_s) \tag{8.59}$$

where $\mathbf{x}_s$ denotes a subset of the variables. For convenience, we shall denote the

Figure 8.40 Example of a factor graph, which corresponds to the factorization (8.60).

individual variables by x_i, however, as in earlier discussions, these can comprise groups of variables (such as vectors or matrices). Each factor f_s is a function of a corresponding set of variables $\mathbf{x}_s$.

Directed graphs, whose factorization is defined by (8.5), represent special cases of (8.59) in which the factors $f_s(\mathbf{x}_s)$ are local conditional distributions. Similarly, undirected graphs, given by (8.39), are a special case in which the factors are potential functions over the maximal cliques (the normalizing coefficient $1/Z$ can be viewed as a factor defined over the empty set of variables).

In a factor graph, there is a node (depicted as usual by a circle) for every variable in the distribution, as was the case for directed and undirected graphs. There are also additional nodes (depicted by small squares) for each factor $f_s(\mathbf{x}_s)$ in the joint distribution. Finally, there are undirected links connecting each factor node to all of the variables nodes on which that factor depends. Consider, for example, a distribution that is expressed in terms of the factorization

$$p(\mathbf{x}) = f_a(x_1, x_2) f_b(x_1, x_2) f_c(x_2, x_3) f_d(x_3). \tag{8.60}$$

This can be expressed by the factor graph shown in Figure 8.40. Note that there are two factors $f_a(x_1, x_2)$ and $f_b(x_1, x_2)$ that are defined over the same set of variables. In an undirected graph, the product of two such factors would simply be lumped together into the same clique potential. Similarly, $f_c(x_2, x_3)$ and $f_d(x_3)$ could be combined into a single potential over x_2 and x_3. The factor graph, however, keeps such factors explicit and so is able to convey more detailed information about the underlying factorization.

Figure 8.41 (a) An undirected graph with a single clique potential $\psi(x_1, x_2, x_3)$. (b) A factor graph with factor $f(x_1, x_2, x_3) = \psi(x_1, x_2, x_3)$ representing the same distribution as the undirected graph. (c) A different factor graph representing the same distribution, whose factors satisfy $f_a(x_1, x_2, x_3) f_b(x_1, x_2) = \psi(x_1, x_2, x_3)$.

Figure 8.42 (a) A directed graph with the factorization $p(x_1)p(x_2)p(x_3|x_1, x_2)$. (b) A factor graph representing the same distribution as the directed graph, whose factor satisfies $f(x_1, x_2, x_3) = p(x_1)p(x_2)p(x_3|x_1, x_2)$. (c) A different factor graph representing the same distribution with factors $f_a(x_1) = p(x_1)$, $f_b(x_2) = p(x_2)$ and $f_c(x_1, x_2, x_3) = p(x_3|x_1, x_2)$.

Factor graphs are said to be *bipartite* because they consist of two distinct kinds of nodes, and all links go between nodes of opposite type. In general, factor graphs can therefore always be drawn as two rows of nodes (variable nodes at the top and factor nodes at the bottom) with links between the rows, as shown in the example in Figure 8.40. In some situations, however, other ways of laying out the graph may be more intuitive, for example when the factor graph is derived from a directed or undirected graph, as we shall see.

If we are given a distribution that is expressed in terms of an undirected graph, then we can readily convert it to a factor graph. To do this, we create variable nodes corresponding to the nodes in the original undirected graph, and then create additional factor nodes corresponding to the maximal cliques $\mathbf{x}_s$. The factors $f_s(\mathbf{x}_s)$ are then set equal to the clique potentials. Note that there may be several different factor graphs that correspond to the same undirected graph. These concepts are illustrated in Figure 8.41.

Similarly, to convert a directed graph to a factor graph, we simply create variable nodes in the factor graph corresponding to the nodes of the directed graph, and then create factor nodes corresponding to the conditional distributions, and then finally add the appropriate links. Again, there can be multiple factor graphs all of which correspond to the same directed graph. The conversion of a directed graph to a factor graph is illustrated in Figure 8.42.

We have already noted the importance of tree-structured graphs for performing efficient inference. If we take a directed or undirected tree and convert it into a factor graph, then the result will again be a tree (in other words, the factor graph will have no loops, and there will be one and only one path connecting any two nodes). In the case of a directed polytree, conversion to an undirected graph results in loops due to the moralization step, whereas conversion to a factor graph again results in a tree, as illustrated in Figure 8.43. In fact, local cycles in a directed graph due to links connecting parents of a node can be removed on conversion to a factor graph by defining the appropriate factor function, as shown in Figure 8.44.

We have seen that multiple different factor graphs can represent the same directed or undirected graph. This allows factor graphs to be more specific about the

Figure 8.43 (a) A directed polytree. (b) The result of converting the polytree into an undirected graph showing the creation of loops. (c) The result of converting the polytree into a factor graph, which retains the tree structure.

precise form of the factorization. Figure 8.45 shows an example of a fully connected undirected graph along with two different factor graphs. In (b), the joint distribution is given by a general form $p(\mathbf{x}) = f(x_1, x_2, x_3)$, whereas in (c), it is given by the more specific factorization $p(\mathbf{x}) = f_a(x_1, x_2)f_b(x_1, x_3)f_c(x_2, x_3)$. It should be emphasized that the factorization in (c) does not correspond to any conditional independence properties.

8.4.4 The sum-product algorithm

We shall now make use of the factor graph framework to derive a powerful class of efficient, exact inference algorithms that are applicable to tree-structured graphs. Here we shall focus on the problem of evaluating local marginals over nodes or subsets of nodes, which will lead us to the *sum-product* algorithm. Later we shall modify the technique to allow the most probable state to be found, giving rise to the *max-sum* algorithm.

Also we shall suppose that all of the variables in the model are discrete, and so marginalization corresponds to performing sums. The framework, however, is equally applicable to linear-Gaussian models in which case marginalization involves integration, and we shall consider an example of this in detail when we discuss linear dynamical systems.

Section 13.3

Figure 8.44 (a) A fragment of a directed graph having a local cycle. (b) Conversion to a fragment of a factor graph having a tree structure, in which $f(x_1, x_2, x_3) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2)$.

Figure 8.45 (a) A fully connected undirected graph. (b) and (c) Two factor graphs each of which corresponds to the undirected graph in (a).

There is an algorithm for exact inference on directed graphs without loops known as *belief propagation* (Pearl, 1988; Lauritzen and Spiegelhalter, 1988), and is equivalent to a special case of the sum-product algorithm. Here we shall consider only the sum-product algorithm because it is simpler to derive and to apply, as well as being more general.

We shall assume that the original graph is an undirected tree or a directed tree or polytree, so that the corresponding factor graph has a tree structure. We first convert the original graph into a factor graph so that we can deal with both directed and undirected models using the same framework. Our goal is to exploit the structure of the graph to achieve two things: (i) to obtain an efficient, exact inference algorithm for finding marginals; (ii) in situations where several marginals are required to allow computations to be shared efficiently.

We begin by considering the problem of finding the marginal $p(x)$ for particular variable node x. For the moment, we shall suppose that all of the variables are hidden. Later we shall see how to modify the algorithm to incorporate evidence corresponding to observed variables. By definition, the marginal is obtained by summing the joint distribution over all variables except x so that

$$p(x) = \sum_{\mathbf{x} \setminus x} p(\mathbf{x}) \tag{8.61}$$

where $\mathbf{x} \setminus x$ denotes the set of variables in $\mathbf{x}$ with variable x omitted. The idea is to substitute for $p(\mathbf{x})$ using the factor graph expression (8.59) and then interchange summations and products in order to obtain an efficient algorithm. Consider the fragment of graph shown in Figure 8.46 in which we see that the tree structure of the graph allows us to partition the factors in the joint distribution into groups, with one group associated with each of the factor nodes that is a neighbour of the variable node x. We see that the joint distribution can be written as a product of the form

$$p(\mathbf{x}) = \prod_{s \in \mathrm{ne}(x)} F_s(x, X_s) \tag{8.62}$$

$\mathrm{ne}(x)$ denotes the set of factor nodes that are neighbours of x, and X_s denotes the set of all variables in the subtree connected to the variable node x via the factor node

Figure 8.46 A fragment of a factor graph illustrating the evaluation of the marginal $p(x)$.

f_s, and $F_s(x, X_s)$ represents the product of all the factors in the group associated with factor f_s.

Substituting (8.62) into (8.61) and interchanging the sums and products, we obtain

$$\begin{aligned} p(x) &= \prod_{s \in \mathrm{ne}(x)} \left[\sum_{X_s} F_s(x, X_s) \right] \\ &= \prod_{s \in \mathrm{ne}(x)} \mu_{f_s \to x}(x). \end{aligned} \tag{8.63}$$

Here we have introduced a set of functions $\mu_{f_s \to x}(x)$, defined by

$$\mu_{f_s \to x}(x) \equiv \sum_{X_s} F_s(x, X_s) \tag{8.64}$$

which can be viewed as *messages* from the factor nodes f_s to the variable node x. We see that the required marginal $p(x)$ is given by the product of all the incoming messages arriving at node x.

In order to evaluate these messages, we again turn to Figure 8.46 and note that each factor $F_s(x, X_s)$ is described by a factor (sub-)graph and so can itself be factorized. In particular, we can write

$$F_s(x, X_s) = f_s(x, x_1, \ldots, x_M) G_1(x_1, X_{s1}) \ldots G_M(x_M, X_{sM}) \tag{8.65}$$

where, for convenience, we have denoted the variables associated with factor f_x, in addition to x, by $x_1, \ldots, x_M$. This factorization is illustrated in Figure 8.47. Note that the set of variables $\{x, x_1, \ldots, x_M\}$ is the set of variables on which the factor f_s depends, and so it can also be denoted $\mathbf{x}_s$, using the notation of (8.59).

Substituting (8.65) into (8.64) we obtain

$$\begin{aligned} \mu_{f_s \to x}(x) &= \sum_{x_1} \ldots \sum_{x_M} f_s(x, x_1, \ldots, x_M) \prod_{m \in \mathrm{ne}(f_s) \setminus x} \left[\sum_{X_{xm}} G_m(x_m, X_{sm}) \right] \\ &= \sum_{x_1} \ldots \sum_{x_M} f_s(x, x_1, \ldots, x_M) \prod_{m \in \mathrm{ne}(f_s) \setminus x} \mu_{x_m \to f_s}(x_m) \end{aligned} \tag{8.66}$$

Figure 8.47 Illustration of the factorization of the subgraph associated with factor node f_s.

where $\mathrm{ne}(f_s)$ denotes the set of variable nodes that are neighbours of the factor node f_s, and $\mathrm{ne}(f_s) \setminus x$ denotes the same set but with node x removed. Here we have defined the following messages from variable nodes to factor nodes

$$\mu_{x_m \to f_s}(x_m) \equiv \sum_{X_{sm}} G_m(x_m, X_{sm}). \tag{8.67}$$

We have therefore introduced two distinct kinds of message, those that go from factor nodes to variable nodes denoted $\mu_{f \to x}(x)$, and those that go from variable nodes to factor nodes denoted $\mu_{x \to f}(x)$. In each case, we see that messages passed along a link are always a function of the variable associated with the variable node that link connects to.

The result (8.66) says that to evaluate the message sent by a factor node to a variable node along the link connecting them, take the product of the incoming messages along all other links coming into the factor node, multiply by the factor associated with that node, and then marginalize over all of the variables associated with the incoming messages. This is illustrated in Figure 8.47. It is important to note that a factor node can send a message to a variable node once it has received incoming messages from all other neighbouring variable nodes.

Finally, we derive an expression for evaluating the messages from variable nodes to factor nodes, again by making use of the (sub-)graph factorization. From Figure 8.48, we see that term $G_m(x_m, X_{sm})$ associated with node x_m is given by a product of terms $F_l(x_m, X_{ml})$ each associated with one of the factor nodes f_l that is linked to node x_m (excluding node f_s), so that

$$G_m(x_m, X_{sm}) = \prod_{l \in \mathrm{ne}(x_m) \setminus f_s} F_l(x_m, X_{ml}) \tag{8.68}$$

where the product is taken over all neighbours of node x_m except for node f_s. Note that each of the factors $F_l(x_m, X_{ml})$ represents a subtree of the original graph of precisely the same kind as introduced in (8.62). Substituting (8.68) into (8.67), we

Figure 8.48 Illustration of the evaluation of the message sent by a variable node to an adjacent factor node.

then obtain

$$\begin{aligned}\mu_{x_m \to f_s}(x_m) &= \prod_{l \in \mathrm{ne}(x_m) \setminus f_s} \left[\sum_{X_{ml}} F_l(x_m, X_{ml}) \right] \\ &= \prod_{l \in \mathrm{ne}(x_m) \setminus f_s} \mu_{f_l \to x_m}(x_m) \end{aligned} \tag{8.69}$$

where we have used the definition (8.64) of the messages passed from factor nodes to variable nodes. Thus to evaluate the message sent by a variable node to an adjacent factor node along the connecting link, we simply take the product of the incoming messages along all of the other links. Note that any variable node that has only two neighbours performs no computation but simply passes messages through unchanged. Also, we note that a variable node can send a message to a factor node once it has received incoming messages from all other neighbouring factor nodes.

Recall that our goal is to calculate the marginal for variable node x, and that this marginal is given by the product of incoming messages along all of the links arriving at that node. Each of these messages can be computed recursively in terms of other messages. In order to start this recursion, we can view the node x as the root of the tree and begin at the leaf nodes. From the definition (8.69), we see that if a leaf node is a variable node, then the message that it sends along its one and only link is given by

$$\mu_{x \to f}(x) = 1 \tag{8.70}$$

as illustrated in Figure 8.49(a). Similarly, if the leaf node is a factor node, we see from (8.66) that the message sent should take the form

$$\mu_{f \to x}(x) = f(x) \tag{8.71}$$

Figure 8.49 The sum-product algorithm begins with messages sent by the leaf nodes, which depend on whether the leaf node is (a) a variable node, or (b) a factor node.

as illustrated in Figure 8.49(b).

At this point, it is worth pausing to summarize the particular version of the sum-product algorithm obtained so far for evaluating the marginal $p(x)$. We start by viewing the variable node x as the root of the factor graph and initiating messages at the leaves of the graph using (8.70) and (8.71). The message passing steps (8.66) and (8.69) are then applied recursively until messages have been propagated along every link, and the root node has received messages from all of its neighbours. Each node can send a message towards the root once it has received messages from all of its other neighbours. Once the root node has received messages from all of its neighbours, the required marginal can be evaluated using (8.63). We shall illustrate this process shortly.

To see that each node will always receive enough messages to be able to send out a message, we can use a simple inductive argument as follows. Clearly, for a graph comprising a variable root node connected directly to several factor leaf nodes, the algorithm trivially involves sending messages of the form (8.71) directly from the leaves to the root. Now imagine building up a general graph by adding nodes one at a time, and suppose that for some particular graph we have a valid algorithm. When one more (variable or factor) node is added, it can be connected only by a single link because the overall graph must remain a tree, and so the new node will be a leaf node. It therefore sends a message to the node to which it is linked, which in turn will therefore receive all the messages it requires in order to send its own message towards the root, and so again we have a valid algorithm, thereby completing the proof.

Now suppose we wish to find the marginals for every variable node in the graph. This could be done by simply running the above algorithm afresh for each such node. However, this would be very wasteful as many of the required computations would be repeated. We can obtain a much more efficient procedure by 'overlaying' these multiple message passing algorithms to obtain the general sum-product algorithm as follows. Arbitrarily pick any (variable or factor) node and designate it as the root. Propagate messages from the leaves to the root as before. At this point, the root node will have received messages from all of its neighbours. It can therefore send out messages to all of its neighbours. These in turn will then have received messages from all of their neighbours and so can send out messages along the links going away from the root, and so on. In this way, messages are passed outwards from the root all the way to the leaves. By now, a message will have passed in both directions across every link in the graph, and every node will have received a message from all of its neighbours. Again a simple inductive argument can be used to verify the validity of this message passing protocol. Because every variable node will have received messages from all of its neighbours, we can readily calculate the marginal distribution for every variable in the graph. The number of messages that have to be computed is given by twice the number of links in the graph and so involves only twice the computation involved in finding a single marginal. By comparison, if we had run the sum-product algorithm separately for each node, the amount of computation would grow quadratically with the size of the graph. Note that this algorithm is in fact independent of which node was designated as the root,

Exercise 8.20

Figure 8.50 The sum-product algorithm can be viewed purely in terms of messages sent out by factor nodes to other factor nodes. In this example, the outgoing message shown by the blue arrow is obtained by taking the product of all the incoming messages shown by green arrows, multiplying by the factor f_s, and marginalizing over the variables x_1 and x_2.

and indeed the notion of one node having a special status was introduced only as a convenient way to explain the message passing protocol.

Next suppose we wish to find the marginal distributions $p(\mathbf{x}_s)$ associated with the sets of variables belonging to each of the factors. By a similar argument to that used above, it is easy to see that the marginal associated with a factor is given by the product of messages arriving at the factor node and the local factor at that node

Exercise 8.21

$$p(\mathbf{x}_s) = f_s(\mathbf{x}_s) \prod_{i \in \mathrm{ne}(f_s)} \mu_{x_i \to f_s}(x_i) \tag{8.72}$$

in complete analogy with the marginals at the variable nodes. If the factors are parameterized functions and we wish to learn the values of the parameters using the EM algorithm, then these marginals are precisely the quantities we will need to calculate in the E step, as we shall see in detail when we discuss the hidden Markov model in Chapter 13.

The message sent by a variable node to a factor node, as we have seen, is simply the product of the incoming messages on other links. We can if we wish view the sum-product algorithm in a slightly different form by eliminating messages from variable nodes to factor nodes and simply considering messages that are sent out by factor nodes. This is most easily seen by considering the example in Figure 8.50.

So far, we have rather neglected the issue of normalization. If the factor graph was derived from a directed graph, then the joint distribution is already correctly normalized, and so the marginals obtained by the sum-product algorithm will similarly be normalized correctly. However, if we started from an undirected graph, then in general there will be an unknown normalization coefficient $1/Z$. As with the simple chain example of Figure 8.38, this is easily handled by working with an unnormalized version $\widetilde{p}(\mathbf{x})$ of the joint distribution, where $p(\mathbf{x}) = \widetilde{p}(\mathbf{x})/Z$. We first run the sum-product algorithm to find the corresponding unnormalized marginals $\widetilde{p}(x_i)$. The coefficient $1/Z$ is then easily obtained by normalizing any one of these marginals, and this is computationally efficient because the normalization is done over a single variable rather than over the entire set of variables as would be required to normalize $\widetilde{p}(\mathbf{x})$ directly.

At this point, it may be helpful to consider a simple example to illustrate the operation of the sum-product algorithm. Figure 8.51 shows a simple 4-node factor

Figure 8.51 A simple factor graph used to illustrate the sum-product algorithm.

graph whose unnormalized joint distribution is given by

$$\widetilde{p}(\mathbf{x}) = f_a(x_1, x_2) f_b(x_2, x_3) f_c(x_2, x_4). \tag{8.73}$$

In order to apply the sum-product algorithm to this graph, let us designate node x_3 as the root, in which case there are two leaf nodes x_1 and x_4. Starting with the leaf nodes, we then have the following sequence of six messages

$$\mu_{x_1 \to f_a}(x_1) = 1 \tag{8.74}$$

$$\mu_{f_a \to x_2}(x_2) = \sum_{x_1} f_a(x_1, x_2) \tag{8.75}$$

$$\mu_{x_4 \to f_c}(x_4) = 1 \tag{8.76}$$

$$\mu_{f_c \to x_2}(x_2) = \sum_{x_4} f_c(x_2, x_4) \tag{8.77}$$

$$\mu_{x_2 \to f_b}(x_2) = \mu_{f_a \to x_2}(x_2) \mu_{f_c \to x_2}(x_2) \tag{8.78}$$

$$\mu_{f_b \to x_3}(x_3) = \sum_{x_2} f_b(x_2, x_3) \mu_{x_2 \to f_b}. \tag{8.79}$$

The direction of flow of these messages is illustrated in Figure 8.52. Once this message propagation is complete, we can then propagate messages from the root node out to the leaf nodes, and these are given by

$$\mu_{x_3 \to f_b}(x_3) = 1 \tag{8.80}$$

$$\mu_{f_b \to x_2}(x_2) = \sum_{x_3} f_b(x_2, x_3) \tag{8.81}$$

$$\mu_{x_2 \to f_a}(x_2) = \mu_{f_b \to x_2}(x_2) \mu_{f_c \to x_2}(x_2) \tag{8.82}$$

$$\mu_{f_a \to x_1}(x_1) = \sum_{x_2} f_a(x_1, x_2) \mu_{x_2 \to f_a}(x_2) \tag{8.83}$$

$$\mu_{x_2 \to f_c}(x_2) = \mu_{f_a \to x_2}(x_2) \mu_{f_b \to x_2}(x_2) \tag{8.84}$$

$$\mu_{f_c \to x_4}(x_4) = \sum_{x_2} f_c(x_2, x_4) \mu_{x_2 \to f_c}(x_2). \tag{8.85}$$

Figure 8.52 Flow of messages for the sum-product algorithm applied to the example graph in Figure 8.51. (a) From the leaf nodes x_1 and x_4 towards the root node x_3. (b) From the root node towards the leaf nodes.

One message has now passed in each direction across each link, and we can now evaluate the marginals. As a simple check, let us verify that the marginal $p(x_2)$ is given by the correct expression. Using (8.63) and substituting for the messages using the above results, we have

$$\begin{aligned}
\widetilde{p}(x_2) &= \mu_{f_a \to x_2}(x_2)\mu_{f_b \to x_2}(x_2)\mu_{f_c \to x_2}(x_2) \\
&= \left[\sum_{x_1} f_a(x_1, x_2)\right]\left[\sum_{x_3} f_b(x_2, x_3)\right]\left[\sum_{x_4} f_c(x_2, x_4)\right] \\
&= \sum_{x_1}\sum_{x_2}\sum_{x_4} f_a(x_1, x_2) f_b(x_2, x_3) f_c(x_2, x_4) \\
&= \sum_{x_1}\sum_{x_3}\sum_{x_4} \widetilde{p}(\mathbf{x}) \qquad (8.86)
\end{aligned}$$

as required.

So far, we have assumed that all of the variables in the graph are hidden. In most practical applications, a subset of the variables will be observed, and we wish to calculate posterior distributions conditioned on these observations. Observed nodes are easily handled within the sum-product algorithm as follows. Suppose we partition $\mathbf{x}$ into hidden variables $\mathbf{h}$ and observed variables $\mathbf{v}$, and that the observed value of $\mathbf{v}$ is denoted $\widehat{\mathbf{v}}$. Then we simply multiply the joint distribution $p(\mathbf{x})$ by $\prod_i I(v_i, \widehat{v}_i)$, where $I(v, \widehat{v}) = 1$ if $v = \widehat{v}$ and $I(v, \widehat{v}) = 0$ otherwise. This product corresponds to $p(\mathbf{h}, \mathbf{v} = \widehat{\mathbf{v}})$ and hence is an unnormalized version of $p(\mathbf{h}|\mathbf{v} = \widehat{\mathbf{v}})$. By running the sum-product algorithm, we can efficiently calculate the posterior marginals $p(h_i|\mathbf{v} = \widehat{\mathbf{v}})$ up to a normalization coefficient whose value can be found efficiently using a local computation. Any summations over variables in $\mathbf{v}$ then collapse into a single term.

We have assumed throughout this section that we are dealing with discrete variables. However, there is nothing specific to discrete variables either in the graphical framework or in the probabilistic construction of the sum-product algorithm. For

Table 8.1 Example of a joint distribution over two binary variables for which the maximum of the joint distribution occurs for different variable values compared to the maxima of the two marginals.

	$x = 0$	$x = 1$
$y = 0$	0.3	0.4
$y = 1$	0.3	0.0

continuous variables the summations are simply replaced by integrations. We shall give an example of the sum-product algorithm applied to a graph of linear-Gaussian variables when we consider linear dynamical systems.

Section 13.3

8.4.5 The max-sum algorithm

The sum-product algorithm allows us to take a joint distribution $p(\mathbf{x})$ expressed as a factor graph and efficiently find marginals over the component variables. Two other common tasks are to find a setting of the variables that has the largest probability and to find the value of that probability. These can be addressed through a closely related algorithm called *max-sum*, which can be viewed as an application of *dynamic programming* in the context of graphical models (Cormen *et al.*, 2001).

A simple approach to finding latent variable values having high probability would be to run the sum-product algorithm to obtain the marginals $p(x_i)$ for every variable, and then, for each marginal in turn, to find the value $x_i^\star$ that maximizes that marginal. However, this would give the set of values that are *individually* the most probable. In practice, we typically wish to find the set of values that *jointly* have the largest probability, in other words the vector $\mathbf{x}^{\max}$ that maximizes the joint distribution, so that

$$\mathbf{x}^{\max} = \arg\max_{\mathbf{x}} p(\mathbf{x}) \tag{8.87}$$

for which the corresponding value of the joint probability will be given by

$$p(\mathbf{x}^{\max}) = \max_{\mathbf{x}} p(\mathbf{x}). \tag{8.88}$$

In general, $\mathbf{x}^{\max}$ is not the same as the set of $x_i^\star$ values, as we can easily show using a simple example. Consider the joint distribution $p(x, y)$ over two binary variables $x, y \in \{0, 1\}$ given in Table 8.1. The joint distribution is maximized by setting $x = 1$ and $y = 0$, corresponding the value 0.4. However, the marginal for $p(x)$, obtained by summing over both values of y, is given by $p(x = 0) = 0.6$ and $p(x = 1) = 0.4$, and similarly the marginal for y is given by $p(y = 0) = 0.7$ and $p(y = 1) = 0.3$, and so the marginals are maximized by $x = 0$ and $y = 0$, which corresponds to a value of 0.3 for the joint distribution. In fact, it is not difficult to construct examples for which the set of individually most probable values has probability zero under the joint distribution.

Exercise 8.27

We therefore seek an efficient algorithm for finding the value of $\mathbf{x}$ that maximizes the joint distribution $p(\mathbf{x})$ and that will allow us to obtain the value of the joint distribution at its maximum. To address the second of these problems, we shall simply write out the max operator in terms of its components

$$\max_{\mathbf{x}} p(\mathbf{x}) = \max_{x_1} \ldots \max_{x_M} p(\mathbf{x}) \tag{8.89}$$

where M is the total number of variables, and then substitute for $p(\mathbf{x})$ using its expansion in terms of a product of factors. In deriving the sum-product algorithm, we made use of the distributive law (8.53) for multiplication. Here we make use of the analogous law for the max operator

$$\max(ab, ac) = a \max(b, c) \tag{8.90}$$

which holds if $a \geqslant 0$ (as will always be the case for the factors in a graphical model). This allows us to exchange products with maximizations.

Consider first the simple example of a chain of nodes described by (8.49). The evaluation of the probability maximum can be written as

$$\begin{aligned} \max_{\mathbf{x}} p(\mathbf{x}) &= \frac{1}{Z} \max_{x_1} \cdots \max_{x_N} \left[\psi_{1,2}(x_1, x_2) \cdots \psi_{N-1,N}(x_{N-1}, x_N)\right] \\ &= \frac{1}{Z} \max_{x_1} \left[\psi_{1,2}(x_1, x_2) \left[\cdots \max_{x_N} \psi_{N-1,N}(x_{N-1}, x_N)\right]\right]. \end{aligned}$$

As with the calculation of marginals, we see that exchanging the max and product operators results in a much more efficient computation, and one that is easily interpreted in terms of messages passed from node x_N backwards along the chain to node x_1.

We can readily generalize this result to arbitrary tree-structured factor graphs by substituting the expression (8.59) for the factor graph expansion into (8.89) and again exchanging maximizations with products. The structure of this calculation is identical to that of the sum-product algorithm, and so we can simply translate those results into the present context. In particular, suppose that we designate a particular variable node as the 'root' of the graph. Then we start a set of messages propagating inwards from the leaves of the tree towards the root, with each node sending its message towards the root once it has received all incoming messages from its other neighbours. The final maximization is performed over the product of all messages arriving at the root node, and gives the maximum value for $p(\mathbf{x})$. This could be called the *max-product* algorithm and is identical to the sum-product algorithm except that summations are replaced by maximizations. Note that at this stage, messages have been sent from leaves to the root, but not in the other direction.

In practice, products of many small probabilities can lead to numerical underflow problems, and so it is convenient to work with the logarithm of the joint distribution. The logarithm is a monotonic function, so that if $a > b$ then $\ln a > \ln b$, and hence the max operator and the logarithm function can be interchanged, so that

$$\ln\left(\max_{\mathbf{x}} p(\mathbf{x})\right) = \max_{\mathbf{x}} \ln p(\mathbf{x}). \tag{8.91}$$

The distributive property is preserved because

$$\max(a + b, a + c) = a + \max(b, c). \tag{8.92}$$

Thus taking the logarithm simply has the effect of replacing the products in the max-product algorithm with sums, and so we obtain the *max-sum* algorithm. From

the results (8.66) and (8.69) derived earlier for the sum-product algorithm, we can readily write down the max-sum algorithm in terms of message passing simply by replacing 'sum' with 'max' and replacing products with sums of logarithms to give

$$\mu_{f\to x}(x) = \max_{x_1,\ldots,x_M}\left[\ln f(x,x_1,\ldots,x_M) + \sum_{m\in \mathrm{ne}(f_s)\setminus x}\mu_{x_m\to f}(x_m)\right] \tag{8.93}$$

$$\mu_{x\to f}(x) = \sum_{l\in \mathrm{ne}(x)\setminus f}\mu_{f_l\to x}(x). \tag{8.94}$$

The initial messages sent by the leaf nodes are obtained by analogy with (8.70) and (8.71) and are given by

$$\mu_{x\to f}(x) = 0 \tag{8.95}$$

$$\mu_{f\to x}(x) = \ln f(x) \tag{8.96}$$

while at the root node the maximum probability can then be computed, by analogy with (8.63), using

$$p^{\max} = \max_x\left[\sum_{s\in \mathrm{ne}(x)}\mu_{f_s\to x}(x)\right]. \tag{8.97}$$

So far, we have seen how to find the maximum of the joint distribution by propagating messages from the leaves to an arbitrarily chosen root node. The result will be the same irrespective of which node is chosen as the root. Now we turn to the second problem of finding the configuration of the variables for which the joint distribution attains this maximum value. So far, we have sent messages from the leaves to the root. The process of evaluating (8.97) will also give the value $x^{\max}$ for the most probable value of the root node variable, defined by

$$x^{\max} = \arg\max_x\left[\sum_{s\in \mathrm{ne}(x)}\mu_{f_s\to x}(x)\right]. \tag{8.98}$$

At this point, we might be tempted simply to continue with the message passing algorithm and send messages from the root back out to the leaves, using (8.93) and (8.94), then apply (8.98) to all of the remaining variable nodes. However, because we are now maximizing rather than summing, it is possible that there may be multiple configurations of $\mathbf{x}$ all of which give rise to the maximum value for $p(\mathbf{x})$. In such cases, this strategy can fail because it is possible for the individual variable values obtained by maximizing the product of messages at each node to belong to different maximizing configurations, giving an overall configuration that no longer corresponds to a maximum.

The problem can be resolved by adopting a rather different kind of message passing from the root node to the leaves. To see how this works, let us return once again to the simple chain example of N variables $x_1,\ldots,x_N$ each having K states,

Figure 8.53 A lattice, or trellis, diagram showing explicitly the K possible states (one per row of the diagram) for each of the variables x_n in the chain model. In this illustration $K = 3$. The arrow shows the direction of message passing in the max-product algorithm. For every state k of each variable x_n (corresponding to column n of the diagram) the function $\phi(x_n)$ defines a unique state at the previous variable, indicated by the black lines. The two paths through the lattice correspond to configurations that give the global maximum of the joint probability distribution, and either of these can be found by tracing back along the black lines in the opposite direction to the arrow.

corresponding to the graph shown in Figure 8.38. Suppose we take node x_N to be the root node. Then in the first phase, we propagate messages from the leaf node x_1 to the root node using

$$\begin{aligned}
\mu_{x_n \to f_{n,n+1}}(x_n) &= \mu_{f_{n-1,n} \to x_n}(x_n) \\
\mu_{f_{n-1,n} \to x_n}(x_n) &= \max_{x_{n-1}} \left[\ln f_{n-1,n}(x_{n-1}, x_n) + \mu_{x_{n-1} \to fn-1,n}(x_n) \right]
\end{aligned}$$

which follow from applying (8.94) and (8.93) to this particular graph. The initial message sent from the leaf node is simply

$$\mu_{x_1 \to f_{1,2}}(x_1) = 0. \tag{8.99}$$

The most probable value for x_N is then given by

$$x_N^{\max} = \arg\max_{x_N} \left[\mu_{f_{N-1,N} \to x_N}(x_N) \right]. \tag{8.100}$$

Now we need to determine the states of the previous variables that correspond to the same maximizing configuration. This can be done by keeping track of which values of the variables gave rise to the maximum state of each variable, in other words by storing quantities given by

$$\phi(x_n) = \arg\max_{x_{n-1}} \left[\ln f_{n-1,n}(x_{n-1}, x_n) + \mu_{x_{n-1} \to fn-1,n}(x_n) \right]. \tag{8.101}$$

To understand better what is happening, it is helpful to represent the chain of variables in terms of a *lattice* or *trellis* diagram as shown in Figure 8.53. Note that this is not a probabilistic graphical model because the nodes represent individual states of variables, while each variable corresponds to a column of such states in the diagram. For each state of a given variable, there is a unique state of the previous variable that maximizes the probability (ties are broken either systematically or at random), corresponding to the function $\phi(x_n)$ given by (8.101), and this is indicated

by the lines connecting the nodes. Once we know the most probable value of the final node x_N, we can then simply follow the link back to find the most probable state of node x_{N-1} and so on back to the initial node x_1. This corresponds to propagating a message back down the chain using

$$x_{n-1}^{\max} = \phi(x_n^{\max}) \tag{8.102}$$

and is known as *back-tracking*. Note that there could be several values of x_{n-1} all of which give the maximum value in (8.101). Provided we chose one of these values when we do the back-tracking, we are assured of a globally consistent maximizing configuration.

In Figure 8.53, we have indicated two paths, each of which we shall suppose corresponds to a global maximum of the joint probability distribution. If $k = 2$ and $k = 3$ each represent possible values of $x_N^{\max}$, then starting from either state and tracing back along the black lines, which corresponds to iterating (8.102), we obtain a valid global maximum configuration. Note that if we had run a forward pass of max-sum message passing followed by a backward pass and then applied (8.98) at each node separately, we could end up selecting some states from one path and some from the other path, giving an overall configuration that is not a global maximizer. We see that it is necessary instead to keep track of the maximizing states during the forward pass using the functions $\phi(x_n)$ and then use back-tracking to find a consistent solution.

The extension to a general tree-structured factor graph should now be clear. If a message is sent from a factor node f to a variable node x, a maximization is performed over all other variable nodes $x_1, \ldots, x_M$ that are neighbours of that factor node, using (8.93). When we perform this maximization, we keep a record of which values of the variables $x_1, \ldots, x_M$ gave rise to the maximum. Then in the back-tracking step, having found $x^{\max}$, we can then use these stored values to assign consistent maximizing states $x_1^{\max}, \ldots, x_M^{\max}$. The max-sum algorithm, with back-tracking, gives an exact maximizing configuration for the variables provided the factor graph is a tree. An important application of this technique is for finding the most probable sequence of hidden states in a hidden Markov model, in which case it is known as the *Viterbi* algorithm.

Section 13.2

As with the sum-product algorithm, the inclusion of evidence in the form of observed variables is straightforward. The observed variables are clamped to their observed values, and the maximization is performed over the remaining hidden variables. This can be shown formally by including identity functions for the observed variables into the factor functions, as we did for the sum-product algorithm.

It is interesting to compare max-sum with the iterated conditional modes (ICM) algorithm described on page 389. Each step in ICM is computationally simpler because the 'messages' that are passed from one node to the next comprise a single value consisting of the new state of the node for which the conditional distribution is maximized. The max-sum algorithm is more complex because the messages are functions of node variables x and hence comprise a set of K values for each possible state of x. Unlike max-sum, however, ICM is not guaranteed to find a global maximum even for tree-structured graphs.

8.4.6 Exact inference in general graphs

The sum-product and max-sum algorithms provide efficient and exact solutions to inference problems in tree-structured graphs. For many practical applications, however, we have to deal with graphs having loops.

The message passing framework can be generalized to arbitrary graph topologies, giving an exact inference procedure known as the *junction tree algorithm* (Lauritzen and Spiegelhalter, 1988; Jordan, 2007). Here we give a brief outline of the key steps involved. This is not intended to convey a detailed understanding of the algorithm, but rather to give a flavour of the various stages involved. If the starting point is a directed graph, it is first converted to an undirected graph by moralization, whereas if starting from an undirected graph this step is not required. Next the graph is *triangulated*, which involves finding chord-less cycles containing four or more nodes and adding extra links to eliminate such chord-less cycles. For instance, in the graph in Figure 8.36, the cycle A–C–B–D–A is chord-less a link could be added between A and B or alternatively between C and D. Note that the joint distribution for the resulting triangulated graph is still defined by a product of the same potential functions, but these are now considered to be functions over expanded sets of variables. Next the triangulated graph is used to construct a new tree-structured undirected graph called a *join tree*, whose nodes correspond to the maximal cliques of the triangulated graph, and whose links connect pairs of cliques that have variables in common. The selection of which pairs of cliques to connect in this way is important and is done so as to give a *maximal spanning tree* defined as follows. Of all possible trees that link up the cliques, the one that is chosen is one for which the *weight* of the tree is largest, where the weight for a link is the number of nodes shared by the two cliques it connects, and the weight for the tree is the sum of the weights for the links. If the tree is condensed, so that any clique that is a subset of another clique is absorbed into the larger clique, this gives a *junction tree*. As a consequence of the triangulation step, the resulting tree satisfies the *running intersection property*, which means that if a variable is contained in two cliques, then it must also be contained in every clique on the path that connects them. This ensures that inference about variables will be consistent across the graph. Finally, a two-stage message passing algorithm, essentially equivalent to the sum-product algorithm, can now be applied to this junction tree in order to find marginals and conditionals. Although the junction tree algorithm sounds complicated, at its heart is the simple idea that we have used already of exploiting the factorization properties of the distribution to allow sums and products to be interchanged so that partial summations can be performed, thereby avoiding having to work directly with the joint distribution. The role of the junction tree is to provide a precise and efficient way to organize these computations. It is worth emphasizing that this is achieved using purely graphical operations!

The junction tree is exact for arbitrary graphs and is efficient in the sense that for a given graph there does not in general exist a computationally cheaper approach. Unfortunately, the algorithm must work with the joint distributions within each node (each of which corresponds to a clique of the triangulated graph) and so the computational cost of the algorithm is determined by the number of variables in the largest

clique and will grow exponentially with this number in the case of discrete variables. An important concept is the *treewidth* of a graph (Bodlaender, 1993), which is defined in terms of the number of variables in the largest clique. In fact, it is defined to be as one less than the size of the largest clique, to ensure that a tree has a treewidth of 1. Because there in general there can be multiple different junction trees that can be constructed from a given starting graph, the treewidth is defined by the junction tree for which the largest clique has the fewest variables. If the treewidth of the original graph is high, the junction tree algorithm becomes impractical.

8.4.7 Loopy belief propagation

For many problems of practical interest, it will not be feasible to use exact inference, and so we need to exploit effective approximation methods. An important class of such approximations, that can broadly be called *variational* methods, will be discussed in detail in Chapter 10. Complementing these deterministic approaches is a wide range of *sampling* methods, also called *Monte Carlo* methods, that are based on stochastic numerical sampling from distributions and that will be discussed at length in Chapter 11.

Here we consider one simple approach to approximate inference in graphs with loops, which builds directly on the previous discussion of exact inference in trees. The idea is simply to apply the sum-product algorithm even though there is no guarantee that it will yield good results. This approach is known as *loopy belief propagation* (Frey and MacKay, 1998) and is possible because the message passing rules (8.66) and (8.69) for the sum-product algorithm are purely local. However, because the graph now has cycles, information can flow many times around the graph. For some models, the algorithm will converge, whereas for others it will not.

In order to apply this approach, we need to define a *message passing schedule*. Let us assume that one message is passed at a time on any given link and in any given direction. Each message sent from a node replaces any previous message sent in the same direction across the same link and will itself be a function only of the most recent messages received by that node at previous steps of the algorithm.

We have seen that a message can only be sent across a link from a node when all other messages have been received by that node across its other links. Because there are loops in the graph, this raises the problem of how to initiate the message passing algorithm. To resolve this, we suppose that an initial message given by the unit function has been passed across every link in each direction. Every node is then in a position to send a message.

There are now many possible ways to organize the message passing schedule. For example, the *flooding schedule* simultaneously passes a message across every link in both directions at each time step, whereas schedules that pass one message at a time are called *serial schedules*.

Following Kschischnang *et al.* (2001), we will say that a (variable or factor) node a has a message *pending* on its link to a node b if node a has received any message on any of its other links since the last time it send a message to b. Thus, when a node receives a message on one of its links, this creates pending messages on all of its other links. Only pending messages need to be transmitted because

other messages would simply duplicate the previous message on the same link. For graphs that have a tree structure, any schedule that sends only pending messages will eventually terminate once a message has passed in each direction across every link. At this point, there are no pending messages, and the product of the received messages at every variable give the exact marginal. In graphs having loops, however, the algorithm may never terminate because there might always be pending messages, although in practice it is generally found to converge within a reasonable time for most applications. Once the algorithm has converged, or once it has been stopped if convergence is not observed, the (approximate) local marginals can be computed using the product of the most recently received incoming messages to each variable node or factor node on every link.

Exercise 8.29

In some applications, the loopy belief propagation algorithm can give poor results, whereas in other applications it has proven to be very effective. In particular, state-of-the-art algorithms for decoding certain kinds of error-correcting codes are equivalent to loopy belief propagation (Gallager, 1963; Berrou *et al.*, 1993; McEliece *et al.*, 1998; MacKay and Neal, 1999; Frey, 1998).

8.4.8 Learning the graph structure

In our discussion of inference in graphical models, we have assumed that the structure of the graph is known and fixed. However, there is also interest in going beyond the inference problem and learning the graph structure itself from data (Friedman and Koller, 2003). This requires that we define a space of possible structures as well as a measure that can be used to score each structure.

From a Bayesian viewpoint, we would ideally like to compute a posterior distribution over graph structures and to make predictions by averaging with respect to this distribution. If we have a prior $p(m)$ over graphs indexed by m, then the posterior distribution is given by

$$p(m|\mathcal{D}) \propto p(m)p(\mathcal{D}|m) \tag{8.103}$$

where $\mathcal{D}$ is the observed data set. The model evidence $p(\mathcal{D}|m)$ then provides the score for each model. However, evaluation of the evidence involves marginalization over the latent variables and presents a challenging computational problem for many models.

Exploring the space of structures can also be problematic. Because the number of different graph structures grows exponentially with the number of nodes, it is often necessary to resort to heuristics to find good candidates.

Exercises

8.1 ($\star$) **WWW** By marginalizing out the variables in order, show that the representation (8.5) for the joint distribution of a directed graph is correctly normalized, provided each of the conditional distributions is normalized.

8.2 ($\star$) **WWW** Show that the property of there being no directed cycles in a directed graph follows from the statement that there exists an ordered numbering of the nodes such that for each node there are no links going to a lower-numbered node.

Table 8.2 The joint distribution over three binary variables.

a	b	c	$p(a,b,c)$
0	0	0	0.192
0	0	1	0.144
0	1	0	0.048
0	1	1	0.216
1	0	0	0.192
1	0	1	0.064
1	1	0	0.048
1	1	1	0.096

8.3 ($\star\star$) Consider three binary variables $a, b, c \in \{0, 1\}$ having the joint distribution given in Table 8.2. Show by direct evaluation that this distribution has the property that a and b are marginally dependent, so that $p(a, b) \neq p(a)p(b)$, but that they become independent when conditioned on c, so that $p(a, b|c) = p(a|c)p(b|c)$ for both $c = 0$ and $c = 1$.

8.4 ($\star\star$) Evaluate the distributions $p(a)$, $p(b|c)$, and $p(c|a)$ corresponding to the joint distribution given in Table 8.2. Hence show by direct evaluation that $p(a, b, c) = p(a)p(c|a)p(b|c)$. Draw the corresponding directed graph.

8.5 ($\star$) **WWW** Draw a directed probabilistic graphical model corresponding to the relevance vector machine described by (7.79) and (7.80).

8.6 ($\star$) For the model shown in Figure 8.13, we have seen that the number of parameters required to specify the conditional distribution $p(y|x_1, \ldots, x_M)$, where $x_i \in \{0, 1\}$, could be reduced from 2^M to $M + 1$ by making use of the logistic sigmoid representation (8.10). An alternative representation (Pearl, 1988) is given by

$$p(y = 1|x_1, \ldots, x_M) = 1 - (1 - \mu_0) \prod_{i=1}^{M} (1 - \mu_i)^{x_i} \tag{8.104}$$

where the parameters μ_i represent the probabilities $p(x_i = 1)$, and μ_0 is an additional parameters satisfying $0 \leqslant \mu_0 \leqslant 1$. The conditional distribution (8.104) is known as the *noisy-OR*. Show that this can be interpreted as a 'soft' (probabilistic) form of the logical OR function (i.e., the function that gives $y = 1$ whenever at least one of the $x_i = 1$). Discuss the interpretation of μ_0.

8.7 ($\star\star$) Using the recursion relations (8.15) and (8.16), show that the mean and covariance of the joint distribution for the graph shown in Figure 8.14 are given by (8.17) and (8.18), respectively.

8.8 ($\star$) **WWW** Show that $a \perp\!\!\!\perp b, c \mid d$ implies $a \perp\!\!\!\perp b \mid d$.

8.9 ($\star$) **WWW** Using the d-separation criterion, show that the conditional distribution for a node $\mathbf{x}$ in a directed graph, conditioned on all of the nodes in the Markov blanket, is independent of the remaining variables in the graph.

Figure 8.54 Example of a graphical model used to explore the conditional independence properties of the head-to-head path a–c–b when a descendant of c, namely the node d, is observed.

8.10 ($\star$) Consider the directed graph shown in Figure 8.54 in which none of the variables is observed. Show that $a \perp\!\!\!\perp b \mid \emptyset$. Suppose we now observe the variable d. Show that in general $a \not\perp\!\!\!\perp b \mid d$.

8.11 ($\star\star$) Consider the example of the car fuel system shown in Figure 8.21, and suppose that instead of observing the state of the fuel gauge G directly, the gauge is seen by the driver D who reports to us the reading on the gauge. This report is either that the gauge shows full $D = 1$ or that it shows empty $D = 0$. Our driver is a bit unreliable, as expressed through the following probabilities

$$p(D = 1|G = 1) = 0.9 \tag{8.105}$$
$$p(D = 0|G = 0) = 0.9. \tag{8.106}$$

Suppose that the driver tells us that the fuel gauge shows empty, in other words that we observe $D = 0$. Evaluate the probability that the tank is empty given only this observation. Similarly, evaluate the corresponding probability given also the observation that the battery is flat, and note that this second probability is lower. Discuss the intuition behind this result, and relate the result to Figure 8.54.

8.12 ($\star$) **WWW** Show that there are $2^{M(M-1)/2}$ distinct undirected graphs over a set of M distinct random variables. Draw the 8 possibilities for the case of $M = 3$.

8.13 ($\star$) Consider the use of iterated conditional modes (ICM) to minimize the energy function given by (8.42). Write down an expression for the difference in the values of the energy associated with the two states of a particular variable x_j, with all other variables held fixed, and show that it depends only on quantities that are local to x_j in the graph.

8.14 ($\star$) Consider a particular case of the energy function given by (8.42) in which the coefficients $\beta = h = 0$. Show that the most probable configuration of the latent variables is given by $x_i = y_i$ for all i.

8.15 ($\star\star$) **WWW** Show that the joint distribution $p(x_{n-1}, x_n)$ for two neighbouring nodes in the graph shown in Figure 8.38 is given by an expression of the form (8.58).

8.16 ($\star\star$) Consider the inference problem of evaluating $p(\mathbf{x}_n|\mathbf{x}_N)$ for the graph shown in Figure 8.38, for all nodes $n \in \{1, \ldots, N-1\}$. Show that the message passing algorithm discussed in Section 8.4.1 can be used to solve this efficiently, and discuss which messages are modified and in what way.

8.17 ($\star\star$) Consider a graph of the form shown in Figure 8.38 having $N = 5$ nodes, in which nodes x_3 and x_5 are observed. Use d-separation to show that $x_2 \perp\!\!\!\perp x_5 \mid x_3$. Show that if the message passing algorithm of Section 8.4.1 is applied to the evaluation of $p(x_2|x_3, x_5)$, the result will be independent of the value of x_5.

8.18 ($\star\star$) **WWW** Show that a distribution represented by a directed tree can trivially be written as an equivalent distribution over the corresponding undirected tree. Also show that a distribution expressed as an undirected tree can, by suitable normalization of the clique potentials, be written as a directed tree. Calculate the number of distinct directed trees that can be constructed from a given undirected tree.

8.19 ($\star\star$) Apply the sum-product algorithm derived in Section 8.4.4 to the chain-of-nodes model discussed in Section 8.4.1 and show that the results (8.54), (8.55), and (8.57) are recovered as a special case.

8.20 ($\star$) **WWW** Consider the message passing protocol for the sum-product algorithm on a tree-structured factor graph in which messages are first propagated from the leaves to an arbitrarily chosen root node and then from the root node out to the leaves. Use proof by induction to show that the messages can be passed in such an order that at every step, each node that must send a message has received all of the incoming messages necessary to construct its outgoing messages.

8.21 ($\star\star$) **WWW** Show that the marginal distributions $p(\mathbf{x}_s)$ over the sets of variables $\mathbf{x}_s$ associated with each of the factors $f_x(\mathbf{x}_s)$ in a factor graph can be found by first running the sum-product message passing algorithm and then evaluating the required marginals using (8.72).

8.22 ($\star$) Consider a tree-structured factor graph, in which a given subset of the variable nodes form a connected subgraph (i.e., any variable node of the subset is connected to at least one of the other variable nodes via a single factor node). Show how the sum-product algorithm can be used to compute the marginal distribution over that subset.

8.23 ($\star\star$) **WWW** In Section 8.4.4, we showed that the marginal distribution $p(x_i)$ for a variable node x_i in a factor graph is given by the product of the messages arriving at this node from neighbouring factor nodes in the form (8.63). Show that the marginal $p(x_i)$ can also be written as the product of the incoming message along any one of the links with the outgoing message along the same link.

8.24 ($\star\star$) Show that the marginal distribution for the variables $\mathbf{x}_s$ in a factor $f_s(\mathbf{x}_s)$ in a tree-structured factor graph, after running the sum-product message passing algorithm, can be written as the product of the message arriving at the factor node along all its links, times the local factor $f(\mathbf{x}_s)$, in the form (8.72).

8.25 ($\star\star$) In (8.86), we verified that the sum-product algorithm run on the graph in Figure 8.51 with node x_3 designated as the root node gives the correct marginal for x_2. Show that the correct marginals are obtained also for x_1 and x_3. Similarly, show that the use of the result (8.72) after running the sum-product algorithm on this graph gives the correct joint distribution for x_1, x_2.

8.26 ($\star$) Consider a tree-structured factor graph over discrete variables, and suppose we wish to evaluate the joint distribution $p(x_a, x_b)$ associated with two variables x_a and x_b that do not belong to a common factor. Define a procedure for using the sum-product algorithm to evaluate this joint distribution in which one of the variables is successively clamped to each of its allowed values.

8.27 ($\star\star$) Consider two discrete variables x and y each having three possible states, for example $x, y \in \{0, 1, 2\}$. Construct a joint distribution $p(x, y)$ over these variables having the property that the value $\widehat{x}$ that maximizes the marginal $p(x)$, along with the value $\widehat{y}$ that maximizes the marginal $p(y)$, together have probability zero under the joint distribution, so that $p(\widehat{x}, \widehat{y}) = 0$.

8.28 ($\star\star$) **www** The concept of a *pending* message in the sum-product algorithm for a factor graph was defined in Section 8.4.7. Show that if the graph has one or more cycles, there will always be at least one pending message irrespective of how long the algorithm runs.

8.29 ($\star\star$) **www** Show that if the sum-product algorithm is run on a factor graph with a tree structure (no loops), then after a finite number of messages have been sent, there will be no pending messages.

9

Mixture Models and EM

If we define a joint distribution over observed and latent variables, the corresponding distribution of the observed variables alone is obtained by marginalization. This allows relatively complex marginal distributions over observed variables to be expressed in terms of more tractable joint distributions over the expanded space of observed and latent variables. The introduction of latent variables thereby allows complicated distributions to be formed from simpler components. In this chapter, we shall see that mixture distributions, such as the Gaussian mixture discussed in Section 2.3.9, can be interpreted in terms of discrete latent variables. Continuous latent variables will form the subject of Chapter 12.

As well as providing a framework for building more complex probability distributions, mixture models can also be used to cluster data. We therefore begin our discussion of mixture distributions by considering the problem of finding clusters in a set of data points, which we approach first using a nonprobabilistic technique called the K-means algorithm (Lloyd, 1982). Then we introduce the latent variable

Section 9.1

view of mixture distributions in which the discrete latent variables can be interpreted as defining assignments of data points to specific components of the mixture. A general technique for finding maximum likelihood estimators in latent variable models is the expectation-maximization (EM) algorithm. We first of all use the Gaussian mixture distribution to motivate the EM algorithm in a fairly informal way, and then we give a more careful treatment based on the latent variable viewpoint. We shall see that the K-means algorithm corresponds to a particular nonprobabilistic limit of EM applied to mixtures of Gaussians. Finally, we discuss EM in some generality.

Section 9.2

Section 9.3

Section 9.4

Gaussian mixture models are widely used in data mining, pattern recognition, machine learning, and statistical analysis. In many applications, their parameters are determined by maximum likelihood, typically using the EM algorithm. However, as we shall see there are some significant limitations to the maximum likelihood approach, and in Chapter 10 we shall show that an elegant Bayesian treatment can be given using the framework of variational inference. This requires little additional computation compared with EM, and it resolves the principal difficulties of maximum likelihood while also allowing the number of components in the mixture to be inferred automatically from the data.

9.1. K-means Clustering

We begin by considering the problem of identifying groups, or clusters, of data points in a multidimensional space. Suppose we have a data set $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ consisting of N observations of a random D-dimensional Euclidean variable $\mathbf{x}$. Our goal is to partition the data set into some number K of clusters, where we shall suppose for the moment that the value of K is given. Intuitively, we might think of a cluster as comprising a group of data points whose inter-point distances are small compared with the distances to points outside of the cluster. We can formalize this notion by first introducing a set of D-dimensional vectors $\boldsymbol{\mu}_k$, where $k = 1, \ldots, K$, in which $\boldsymbol{\mu}_k$ is a prototype associated with the kth cluster. As we shall see shortly, we can think of the $\boldsymbol{\mu}_k$ as representing the centres of the clusters. Our goal is then to find an assignment of data points to clusters, as well as a set of vectors $\{\boldsymbol{\mu}_k\}$, such that the sum of the squares of the distances of each data point to its closest vector $\boldsymbol{\mu}_k$, is a minimum.

It is convenient at this point to define some notation to describe the assignment of data points to clusters. For each data point $\mathbf{x}_n$, we introduce a corresponding set of binary indicator variables $r_{nk} \in \{0, 1\}$, where $k = 1, \ldots, K$ describing which of the K clusters the data point $\mathbf{x}_n$ is assigned to, so that if data point $\mathbf{x}_n$ is assigned to cluster k then $r_{nk} = 1$, and $r_{nj} = 0$ for $j \neq k$. This is known as the 1-of-K coding scheme. We can then define an objective function, sometimes called a *distortion measure*, given by

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2 \tag{9.1}$$

which represents the sum of the squares of the distances of each data point to its

assigned vector $\boldsymbol{\mu}_k$. Our goal is to find values for the $\{r_{nk}\}$ and the $\{\boldsymbol{\mu}_k\}$ so as to minimize J. We can do this through an iterative procedure in which each iteration involves two successive steps corresponding to successive optimizations with respect to the r_{nk} and the $\boldsymbol{\mu}_k$. First we choose some initial values for the $\boldsymbol{\mu}_k$. Then in the first phase we minimize J with respect to the r_{nk}, keeping the $\boldsymbol{\mu}_k$ fixed. In the second phase we minimize J with respect to the $\boldsymbol{\mu}_k$, keeping r_{nk} fixed. This two-stage optimization is then repeated until convergence. We shall see that these two stages of updating r_{nk} and updating $\boldsymbol{\mu}_k$ correspond respectively to the E (expectation) and M (maximization) steps of the EM algorithm, and to emphasize this we shall use the terms E step and M step in the context of the K-means algorithm.

Section 9.4

Consider first the determination of the r_{nk}. Because J in (9.1) is a linear function of r_{nk}, this optimization can be performed easily to give a closed form solution. The terms involving different n are independent and so we can optimize for each n separately by choosing r_{nk} to be 1 for whichever value of k gives the minimum value of $\|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2$. In other words, we simply assign the nth data point to the closest cluster centre. More formally, this can be expressed as

$$r_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_j \|\mathbf{x}_n - \boldsymbol{\mu}_j\|^2 \\ 0 & \text{otherwise.} \end{cases} \tag{9.2}$$

Now consider the optimization of the $\boldsymbol{\mu}_k$ with the r_{nk} held fixed. The objective function J is a quadratic function of $\boldsymbol{\mu}_k$, and it can be minimized by setting its derivative with respect to $\boldsymbol{\mu}_k$ to zero giving

$$2\sum_{n=1}^{N} r_{nk}(\mathbf{x}_n - \boldsymbol{\mu}_k) = 0 \tag{9.3}$$

which we can easily solve for $\boldsymbol{\mu}_k$ to give

$$\boldsymbol{\mu}_k = \frac{\sum_n r_{nk}\mathbf{x}_n}{\sum_n r_{nk}}. \tag{9.4}$$

The denominator in this expression is equal to the number of points assigned to cluster k, and so this result has a simple interpretation, namely set $\boldsymbol{\mu}_k$ equal to the mean of all of the data points $\mathbf{x}_n$ assigned to cluster k. For this reason, the procedure is known as the K-*means* algorithm.

The two phases of re-assigning data points to clusters and re-computing the cluster means are repeated in turn until there is no further change in the assignments (or until some maximum number of iterations is exceeded). Because each phase reduces the value of the objective function J, convergence of the algorithm is assured. However, it may converge to a local rather than global minimum of J. The convergence properties of the K-means algorithm were studied by MacQueen (1967).

Exercise 9.1

Appendix A

The K-means algorithm is illustrated using the Old Faithful data set in Figure 9.1. For the purposes of this example, we have made a linear re-scaling of the data, known as *standardizing*, such that each of the variables has zero mean and unit standard deviation. For this example, we have chosen $K = 2$, and so in this

Figure 9.1 Illustration of the K-means algorithm using the re-scaled Old Faithful data set. (a) Green points denote the data set in a two-dimensional Euclidean space. The initial choices for centres $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ are shown by the red and blue crosses, respectively. (b) In the initial E step, each data point is assigned either to the red cluster or to the blue cluster, according to which cluster centre is nearer. This is equivalent to classifying the points according to which side of the perpendicular bisector of the two cluster centres, shown by the magenta line, they lie on. (c) In the subsequent M step, each cluster centre is re-computed to be the mean of the points assigned to the corresponding cluster. (d)–(i) show successive E and M steps through to final convergence of the algorithm.

Figure 9.2 Plot of the cost function J given by (9.1) after each E step (blue points) and M step (red points) of the K-means algorithm for the example shown in Figure 9.1. The algorithm has converged after the third M step, and the final EM cycle produces no changes in either the assignments or the prototype vectors.

case, the assignment of each data point to the nearest cluster centre is equivalent to a classification of the data points according to which side they lie of the perpendicular bisector of the two cluster centres. A plot of the cost function J given by (9.1) for the Old Faithful example is shown in Figure 9.2.

Note that we have deliberately chosen poor initial values for the cluster centres so that the algorithm takes several steps before convergence. In practice, a better initialization procedure would be to choose the cluster centres $\boldsymbol{\mu}_k$ to be equal to a random subset of K data points. It is also worth noting that the K-means algorithm itself is often used to initialize the parameters in a Gaussian mixture model before applying the EM algorithm.

Section 9.2.2

A direct implementation of the K-means algorithm as discussed here can be relatively slow, because in each E step it is necessary to compute the Euclidean distance between every prototype vector and every data point. Various schemes have been proposed for speeding up the K-means algorithm, some of which are based on precomputing a data structure such as a tree such that nearby points are in the same subtree (Ramasubramanian and Paliwal, 1990; Moore, 2000). Other approaches make use of the triangle inequality for distances, thereby avoiding unnecessary distance calculations (Hodgson, 1998; Elkan, 2003).

So far, we have considered a batch version of K-means in which the whole data set is used together to update the prototype vectors. We can also derive an on-line stochastic algorithm (MacQueen, 1967) by applying the Robbins-Monro procedure to the problem of finding the roots of the regression function given by the derivatives of J in (9.1) with respect to $\boldsymbol{\mu}_k$. This leads to a sequential update in which, for each data point $\mathbf{x}_n$ in turn, we update the nearest prototype $\boldsymbol{\mu}_k$ using

Section 2.3.5

Exercise 9.2

$$\boldsymbol{\mu}_k^{\text{new}} = \boldsymbol{\mu}_k^{\text{old}} + \eta_n(\mathbf{x}_n - \boldsymbol{\mu}_k^{\text{old}}) \tag{9.5}$$

where η_n is the learning rate parameter, which is typically made to decrease monotonically as more data points are considered.

The K-means algorithm is based on the use of squared Euclidean distance as the measure of dissimilarity between a data point and a prototype vector. Not only does this limit the type of data variables that can be considered (it would be inappropriate for cases where some or all of the variables represent categorical labels for instance),

Section 2.3.7

but it can also make the determination of the cluster means nonrobust to outliers. We can generalize the K-means algorithm by introducing a more general dissimilarity measure $\mathcal{V}(\mathbf{x}, \mathbf{x}')$ between two vectors $\mathbf{x}$ and $\mathbf{x}'$ and then minimizing the following distortion measure

$$\widetilde{J} = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \mathcal{V}(\mathbf{x}_n, \boldsymbol{\mu}_k) \tag{9.6}$$

which gives the K-*medoids* algorithm. The E step again involves, for given cluster prototypes $\boldsymbol{\mu}_k$, assigning each data point to the cluster for which the dissimilarity to the corresponding prototype is smallest. The computational cost of this is $O(KN)$, as is the case for the standard K-means algorithm. For a general choice of dissimilarity measure, the M step is potentially more complex than for K-means, and so it is common to restrict each cluster prototype to be equal to one of the data vectors assigned to that cluster, as this allows the algorithm to be implemented for any choice of dissimilarity measure $\mathcal{V}(\cdot, \cdot)$ so long as it can be readily evaluated. Thus the M step involves, for each cluster k, a discrete search over the N_k points assigned to that cluster, which requires $O(N_k^2)$ evaluations of $\mathcal{V}(\cdot, \cdot)$.

One notable feature of the K-means algorithm is that at each iteration, every data point is assigned uniquely to one, and only one, of the clusters. Whereas some data points will be much closer to a particular centre $\boldsymbol{\mu}_k$ than to any other centre, there may be other data points that lie roughly midway between cluster centres. In the latter case, it is not clear that the hard assignment to the nearest cluster is the most appropriate. We shall see in the next section that by adopting a probabilistic approach, we obtain 'soft' assignments of data points to clusters in a way that reflects the level of uncertainty over the most appropriate assignment. This probabilistic formulation brings with it numerous benefits.

9.1.1 Image segmentation and compression

As an illustration of the application of the K-means algorithm, we consider the related problems of image segmentation and image compression. The goal of segmentation is to partition an image into regions each of which has a reasonably homogeneous visual appearance or which corresponds to objects or parts of objects (Forsyth and Ponce, 2003). Each pixel in an image is a point in a 3-dimensional space comprising the intensities of the red, blue, and green channels, and our segmentation algorithm simply treats each pixel in the image as a separate data point. Note that strictly this space is not Euclidean because the channel intensities are bounded by the interval $[0, 1]$. Nevertheless, we can apply the K-means algorithm without difficulty. We illustrate the result of running K-means to convergence, for any particular value of K, by re-drawing the image replacing each pixel vector with the $\{R, G, B\}$ intensity triplet given by the centre $\boldsymbol{\mu}_k$ to which that pixel has been assigned. Results for various values of K are shown in Figure 9.3. We see that for a given value of K, the algorithm is representing the image using a palette of only K colours. It should be emphasized that this use of K-means is not a particularly sophisticated approach to image segmentation, not least because it takes no account of the spatial proximity of different pixels. The image segmentation problem is in general extremely difficult

Figure 9.3 Two examples of the application of the K-means clustering algorithm to image segmentation showing the initial images together with their K-means segmentations obtained using various values of K. This also illustrates of the use of vector quantization for data compression, in which smaller values of K give higher compression at the expense of poorer image quality.

and remains the subject of active research and is introduced here simply to illustrate the behaviour of the K-means algorithm.

We can also use the result of a clustering algorithm to perform data compression. It is important to distinguish between *lossless data compression*, in which the goal is to be able to reconstruct the original data exactly from the compressed representation, and *lossy data compression*, in which we accept some errors in the reconstruction in return for higher levels of compression than can be achieved in the lossless case. We can apply the K-means algorithm to the problem of lossy data compression as follows. For each of the N data points, we store only the identity k of the cluster to which it is assigned. We also store the values of the K cluster centres $\boldsymbol{\mu}_k$, which typically requires significantly less data, provided we choose $K \ll N$. Each data point is then approximated by its nearest centre $\boldsymbol{\mu}_k$. New data points can similarly be compressed by first finding the nearest $\boldsymbol{\mu}_k$ and then storing the label k instead of the original data vector. This framework is often called *vector quantization*, and the vectors $\boldsymbol{\mu}_k$ are called *code-book vectors*.

The image segmentation problem discussed above also provides an illustration of the use of clustering for data compression. Suppose the original image has N pixels comprising $\{R, G, B\}$ values each of which is stored with 8 bits of precision. Then to transmit the whole image directly would cost $24N$ bits. Now suppose we first run K-means on the image data, and then instead of transmitting the original pixel intensity vectors we transmit the identity of the nearest vector $\boldsymbol{\mu}_k$. Because there are K such vectors, this requires $\log_2 K$ bits per pixel. We must also transmit the K code book vectors $\boldsymbol{\mu}_k$, which requires $24K$ bits, and so the total number of bits required to transmit the image is $24K + N \log_2 K$ (rounding up to the nearest integer). The original image shown in Figure 9.3 has $240 \times 180 = 43,200$ pixels and so requires $24 \times 43,200 = 1,036,800$ bits to transmit directly. By comparison, the compressed images require $43,248$ bits ($K = 2$), $86,472$ bits ($K = 3$), and $173,040$ bits ($K = 10$), respectively, to transmit. These represent compression ratios compared to the original image of 4.2%, 8.3%, and 16.7%, respectively. We see that there is a trade-off between degree of compression and image quality. Note that our aim in this example is to illustrate the K-means algorithm. If we had been aiming to produce a good image compressor, then it would be more fruitful to consider small blocks of adjacent pixels, for instance 5×5, and thereby exploit the correlations that exist in natural images between nearby pixels.

9.2. Mixtures of Gaussians

In Section 2.3.9 we motivated the Gaussian mixture model as a simple linear superposition of Gaussian components, aimed at providing a richer class of density models than the single Gaussian. We now turn to a formulation of Gaussian mixtures in terms of discrete *latent* variables. This will provide us with a deeper insight into this important distribution, and will also serve to motivate the expectation-maximization algorithm.

Recall from (2.188) that the Gaussian mixture distribution can be written as a linear superposition of Gaussians in the form

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k). \tag{9.7}$$

Let us introduce a K-dimensional binary random variable $\mathbf{z}$ having a 1-of-K representation in which a particular element z_k is equal to 1 and all other elements are equal to 0. The values of z_k therefore satisfy $z_k \in \{0, 1\}$ and $\sum_k z_k = 1$, and we see that there are K possible states for the vector $\mathbf{z}$ according to which element is nonzero. We shall define the joint distribution $p(\mathbf{x}, \mathbf{z})$ in terms of a marginal distribution $p(\mathbf{z})$ and a conditional distribution $p(\mathbf{x}|\mathbf{z})$, corresponding to the graphical model in Figure 9.4. The marginal distribution over $\mathbf{z}$ is specified in terms of the mixing coefficients π_k, such that

$$p(z_k = 1) = \pi_k$$

Figure 9.4 Graphical representation of a mixture model, in which the joint distribution is expressed in the form $p(\mathbf{x}, \mathbf{z}) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$.

where the parameters $\{\pi_k\}$ must satisfy

$$0 \leqslant \pi_k \leqslant 1 \tag{9.8}$$

together with

$$\sum_{k=1}^{K} \pi_k = 1 \tag{9.9}$$

in order to be valid probabilities. Because $\mathbf{z}$ uses a 1-of-K representation, we can also write this distribution in the form

$$p(\mathbf{z}) = \prod_{k=1}^{K} \pi_k^{z_k}. \tag{9.10}$$

Similarly, the conditional distribution of $\mathbf{x}$ given a particular value for $\mathbf{z}$ is a Gaussian

$$p(\mathbf{x}|z_k = 1) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

which can also be written in the form

$$p(\mathbf{x}|\mathbf{z}) = \prod_{k=1}^{K} \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_k}. \tag{9.11}$$

Exercise 9.3

The joint distribution is given by $p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$, and the marginal distribution of $\mathbf{x}$ is then obtained by summing the joint distribution over all possible states of $\mathbf{z}$ to give

$$p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{z})p(\mathbf{x}|\mathbf{z}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{9.12}$$

where we have made use of (9.10) and (9.11). Thus the marginal distribution of $\mathbf{x}$ is a Gaussian mixture of the form (9.7). If we have several observations $\mathbf{x}_1, \ldots, \mathbf{x}_N$, then, because we have represented the marginal distribution in the form $p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{x}, \mathbf{z})$, it follows that for every observed data point $\mathbf{x}_n$ there is a corresponding latent variable $\mathbf{z}_n$.

We have therefore found an equivalent formulation of the Gaussian mixture involving an explicit latent variable. It might seem that we have not gained much by doing so. However, we are now able to work with the joint distribution $p(\mathbf{x}, \mathbf{z})$

instead of the marginal distribution $p(\mathbf{x})$, and this will lead to significant simplifications, most notably through the introduction of the expectation-maximization (EM) algorithm.

Another quantity that will play an important role is the conditional probability of $\mathbf{z}$ given $\mathbf{x}$. We shall use $\gamma(z_k)$ to denote $p(z_k = 1|\mathbf{x})$, whose value can be found using Bayes' theorem

$$\begin{aligned}\gamma(z_k) \equiv p(z_k = 1|\mathbf{x}) &= \frac{p(z_k = 1)p(\mathbf{x}|z_k = 1)}{\sum_{j=1}^{K} p(z_j = 1)p(\mathbf{x}|z_j = 1)} \\ &= \frac{\pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}. \qquad (9.13)\end{aligned}$$

We shall view π_k as the prior probability of $z_k = 1$, and the quantity $\gamma(z_k)$ as the corresponding posterior probability once we have observed $\mathbf{x}$. As we shall see later, $\gamma(z_k)$ can also be viewed as the *responsibility* that component k takes for 'explaining' the observation $\mathbf{x}$.

Section 8.1.2

We can use the technique of ancestral sampling to generate random samples distributed according to the Gaussian mixture model. To do this, we first generate a value for $\mathbf{z}$, which we denote $\widehat{\mathbf{z}}$, from the marginal distribution $p(\mathbf{z})$ and then generate a value for $\mathbf{x}$ from the conditional distribution $p(\mathbf{x}|\widehat{\mathbf{z}})$. Techniques for sampling from standard distributions are discussed in Chapter 11. We can depict samples from the joint distribution $p(\mathbf{x}, \mathbf{z})$ by plotting points at the corresponding values of $\mathbf{x}$ and then colouring them according to the value of $\mathbf{z}$, in other words according to which Gaussian component was responsible for generating them, as shown in Figure 9.5(a). Similarly samples from the marginal distribution $p(\mathbf{x})$ are obtained by taking the samples from the joint distribution and ignoring the values of $\mathbf{z}$. These are illustrated in Figure 9.5(b) by plotting the $\mathbf{x}$ values without any coloured labels.

We can also use this synthetic data set to illustrate the 'responsibilities' by evaluating, for every data point, the posterior probability for each component in the mixture distribution from which this data set was generated. In particular, we can represent the value of the responsibilities $\gamma(z_{nk})$ associated with data point $\mathbf{x}_n$ by plotting the corresponding point using proportions of red, blue, and green ink given by $\gamma(z_{nk})$ for $k = 1, 2, 3$, respectively, as shown in Figure 9.5(c). So, for instance, a data point for which $\gamma(z_{n1}) = 1$ will be coloured red, whereas one for which $\gamma(z_{n2}) = \gamma(z_{n3}) = 0.5$ will be coloured with equal proportions of blue and green ink and so will appear cyan. This should be compared with Figure 9.5(a) in which the data points were labelled using the true identity of the component from which they were generated.

9.2.1 Maximum likelihood

Suppose we have a data set of observations $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, and we wish to model this data using a mixture of Gaussians. We can represent this data set as an $N \times D$

Figure 9.5 Example of 500 points drawn from the mixture of 3 Gaussians shown in Figure 2.23. (a) Samples from the joint distribution $p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$ in which the three states of $\mathbf{z}$, corresponding to the three components of the mixture, are depicted in red, green, and blue, and (b) the corresponding samples from the marginal distribution $p(\mathbf{x})$, which is obtained by simply ignoring the values of $\mathbf{z}$ and just plotting the $\mathbf{x}$ values. The data set in (a) is said to be *complete*, whereas that in (b) is *incomplete*. (c) The same samples in which the colours represent the value of the responsibilities $\gamma(z_{nk})$ associated with data point $\mathbf{x}_n$, obtained by plotting the corresponding point using proportions of red, blue, and green ink given by $\gamma(z_{nk})$ for $k = 1, 2, 3$, respectively

matrix $\mathbf{X}$ in which the nth row is given by $\mathbf{x}_n^{\mathrm{T}}$. Similarly, the corresponding latent variables will be denoted by an $N \times K$ matrix $\mathbf{Z}$ with rows $\mathbf{z}_n^{\mathrm{T}}$. If we assume that the data points are drawn independently from the distribution, then we can express the Gaussian mixture model for this i.i.d. data set using the graphical representation shown in Figure 9.6. From (9.7) the log of the likelihood function is given by

$$\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}. \tag{9.14}$$

Before discussing how to maximize this function, it is worth emphasizing that there is a significant problem associated with the maximum likelihood framework applied to Gaussian mixture models, due to the presence of singularities. For simplicity, consider a Gaussian mixture whose components have covariance matrices given by $\boldsymbol{\Sigma}_k = \sigma_k^2 \mathbf{I}$, where $\mathbf{I}$ is the unit matrix, although the conclusions will hold for general covariance matrices. Suppose that one of the components of the mixture model, let us say the jth component, has its mean $\boldsymbol{\mu}_j$ exactly equal to one of the data

Figure 9.6 Graphical representation of a Gaussian mixture model for a set of N i.i.d. data points $\{\mathbf{x}_n\}$, with corresponding latent points $\{\mathbf{z}_n\}$, where $n = 1, \ldots, N$.

Figure 9.7 Illustration of how singularities in the likelihood function arise with mixtures of Gaussians. This should be compared with the case of a single Gaussian shown in Figure 1.14 for which no singularities arise.

points so that $\boldsymbol{\mu}_j = \mathbf{x}_n$ for some value of n. This data point will then contribute a term in the likelihood function of the form

$$\mathcal{N}(\mathbf{x}_n|\mathbf{x}_n, \sigma_j^2\mathbf{I}) = \frac{1}{(2\pi)^{1/2}}\frac{1}{\sigma_j}. \tag{9.15}$$

If we consider the limit $\sigma_j \to 0$, then we see that this term goes to infinity and so the log likelihood function will also go to infinity. Thus the maximization of the log likelihood function is not a well posed problem because such singularities will always be present and will occur whenever one of the Gaussian components 'collapses' onto a specific data point. Recall that this problem did not arise in the case of a single Gaussian distribution. To understand the difference, note that if a single Gaussian collapses onto a data point it will contribute multiplicative factors to the likelihood function arising from the other data points and these factors will go to zero exponentially fast, giving an overall likelihood that goes to zero rather than infinity. However, once we have (at least) two components in the mixture, one of the components can have a finite variance and therefore assign finite probability to all of the data points while the other component can shrink onto one specific data point and thereby contribute an ever increasing additive value to the log likelihood. This is illustrated in Figure 9.7. These singularities provide another example of the severe over-fitting that can occur in a maximum likelihood approach. We shall see that this difficulty does not occur if we adopt a Bayesian approach. For the moment, however, we simply note that in applying maximum likelihood to Gaussian mixture models we must take steps to avoid finding such pathological solutions and instead seek local maxima of the likelihood function that are well behaved. We can hope to avoid the singularities by using suitable heuristics, for instance by detecting when a Gaussian component is collapsing and resetting its mean to a randomly chosen value while also resetting its covariance to some large value, and then continuing with the optimization.

Section 10.1

A further issue in finding maximum likelihood solutions arises from the fact that for any given maximum likelihood solution, a K-component mixture will have a total of $K!$ equivalent solutions corresponding to the $K!$ ways of assigning K sets of parameters to K components. In other words, for any given (nondegenerate) point in the space of parameter values there will be a further $K! - 1$ additional points all of which give rise to exactly the same distribution. This problem is known as

identifiability (Casella and Berger, 2002) and is an important issue when we wish to interpret the parameter values discovered by a model. Identifiability will also arise when we discuss models having continuous latent variables in Chapter 12. However, for the purposes of finding a good density model, it is irrelevant because any of the equivalent solutions is as good as any other.

Maximizing the log likelihood function (9.14) for a Gaussian mixture model turns out to be a more complex problem than for the case of a single Gaussian. The difficulty arises from the presence of the summation over k that appears inside the logarithm in (9.14), so that the logarithm function no longer acts directly on the Gaussian. If we set the derivatives of the log likelihood to zero, we will no longer obtain a closed form solution, as we shall see shortly.

One approach is to apply gradient-based optimization techniques (Fletcher, 1987; Nocedal and Wright, 1999; Bishop and Nabney, 2008). Although gradient-based techniques are feasible, and indeed will play an important role when we discuss mixture density networks in Chapter 5, we now consider an alternative approach known as the EM algorithm which has broad applicability and which will lay the foundations for a discussion of variational inference techniques in Chapter 10.

9.2.2 EM for Gaussian mixtures

An elegant and powerful method for finding maximum likelihood solutions for models with latent variables is called the *expectation-maximization* algorithm, or *EM* algorithm (Dempster *et al.*, 1977; McLachlan and Krishnan, 1997). Later we shall give a general treatment of EM, and we shall also show how EM can be generalized to obtain the variational inference framework. Initially, we shall motivate the EM algorithm by giving a relatively informal treatment in the context of the Gaussian mixture model. We emphasize, however, that EM has broad applicability, and indeed it will be encountered in the context of a variety of different models in this book.

Section 10.1

Let us begin by writing down the conditions that must be satisfied at a maximum of the likelihood function. Setting the derivatives of $\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ in (9.14) with respect to the means $\boldsymbol{\mu}_k$ of the Gaussian components to zero, we obtain

$$0 = -\sum_{n=1}^{N} \underbrace{\frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_j \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}}_{\gamma(z_{nk})} \boldsymbol{\Sigma}_k(\mathbf{x}_n - \boldsymbol{\mu}_k) \tag{9.16}$$

where we have made use of the form (2.43) for the Gaussian distribution. Note that the posterior probabilities, or responsibilities, given by (9.13) appear naturally on the right-hand side. Multiplying by $\boldsymbol{\Sigma}_k^{-1}$ (which we assume to be nonsingular) and rearranging we obtain

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n \tag{9.17}$$

where we have defined

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}). \tag{9.18}$$

We can interpret N_k as the effective number of points assigned to cluster k. Note carefully the form of this solution. We see that the mean $\boldsymbol{\mu}_k$ for the kth Gaussian component is obtained by taking a weighted mean of all of the points in the data set, in which the weighting factor for data point $\mathbf{x}_n$ is given by the posterior probability $\gamma(z_{nk})$ that component k was responsible for generating $\mathbf{x}_n$.

If we set the derivative of $\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ with respect to $\boldsymbol{\Sigma}_k$ to zero, and follow a similar line of reasoning, making use of the result for the maximum likelihood solution for the covariance matrix of a single Gaussian, we obtain

Section 2.3.4

$$\boldsymbol{\Sigma}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}} \tag{9.19}$$

which has the same form as the corresponding result for a single Gaussian fitted to the data set, but again with each data point weighted by the corresponding posterior probability and with the denominator given by the effective number of points associated with the corresponding component.

Finally, we maximize $\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ with respect to the mixing coefficients π_k. Here we must take account of the constraint (9.9), which requires the mixing coefficients to sum to one. This can be achieved using a Lagrange multiplier and maximizing the following quantity

Appendix E

$$\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) + \lambda \left(\sum_{k=1}^{K} \pi_k - 1 \right) \tag{9.20}$$

which gives

$$0 = \sum_{n=1}^{N} \frac{\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_j \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} + \lambda \tag{9.21}$$

where again we see the appearance of the responsibilities. If we now multiply both sides by π_k and sum over k making use of the constraint (9.9), we find $\lambda = -N$. Using this to eliminate λ and rearranging we obtain

$$\pi_k = \frac{N_k}{N} \tag{9.22}$$

so that the mixing coefficient for the kth component is given by the average responsibility which that component takes for explaining the data points.

It is worth emphasizing that the results (9.17), (9.19), and (9.22) do not constitute a closed-form solution for the parameters of the mixture model because the responsibilities $\gamma(z_{nk})$ depend on those parameters in a complex way through (9.13). However, these results do suggest a simple iterative scheme for finding a solution to the maximum likelihood problem, which as we shall see turns out to be an instance of the EM algorithm for the particular case of the Gaussian mixture model. We first choose some initial values for the means, covariances, and mixing coefficients. Then we alternate between the following two updates that we shall call the E step

Figure 9.8 Illustration of the EM algorithm using the Old Faithful set as used for the illustration of the K-means algorithm in Figure 9.1. See the text for details.

and the M step, for reasons that will become apparent shortly. In the *expectation* step, or E step, we use the current values for the parameters to evaluate the posterior probabilities, or responsibilities, given by (9.13). We then use these probabilities in the *maximization* step, or M step, to re-estimate the means, covariances, and mixing coefficients using the results (9.17), (9.19), and (9.22). Note that in so doing we first evaluate the new means using (9.17) and then use these new values to find the covariances using (9.19), in keeping with the corresponding result for a single Gaussian distribution. We shall show that each update to the parameters resulting from an E step followed by an M step is guaranteed to increase the log likelihood function. In practice, the algorithm is deemed to have converged when the change in the log likelihood function, or alternatively in the parameters, falls below some threshold. We illustrate the EM algorithm for a mixture of two Gaussians applied to the rescaled Old Faithful data set in Figure 9.8. Here a mixture of two Gaussians is used, with centres initialized using the same values as for the K-means algorithm in Figure 9.1, and with precision matrices initialized to be proportional to the unit matrix. Plot (a) shows the data points in green, together with the initial configuration of the mixture model in which the one standard-deviation contours for the two

Section 9.4

Gaussian components are shown as blue and red circles. Plot (b) shows the result of the initial E step, in which each data point is depicted using a proportion of blue ink equal to the posterior probability of having been generated from the blue component, and a corresponding proportion of red ink given by the posterior probability of having been generated by the red component. Thus, points that have a significant probability for belonging to either cluster appear purple. The situation after the first M step is shown in plot (c), in which the mean of the blue Gaussian has moved to the mean of the data set, weighted by the probabilities of each data point belonging to the blue cluster, in other words it has moved to the centre of mass of the blue ink. Similarly, the covariance of the blue Gaussian is set equal to the covariance of the blue ink. Analogous results hold for the red component. Plots (d), (e), and (f) show the results after 2, 5, and 20 complete cycles of EM, respectively. In plot (f) the algorithm is close to convergence.

Note that the EM algorithm takes many more iterations to reach (approximate) convergence compared with the K-means algorithm, and that each cycle requires significantly more computation. It is therefore common to run the K-means algorithm in order to find a suitable initialization for a Gaussian mixture model that is subsequently adapted using EM. The covariance matrices can conveniently be initialized to the sample covariances of the clusters found by the K-means algorithm, and the mixing coefficients can be set to the fractions of data points assigned to the respective clusters. As with gradient-based approaches for maximizing the log likelihood, techniques must be employed to avoid singularities of the likelihood function in which a Gaussian component collapses onto a particular data point. It should be emphasized that there will generally be multiple local maxima of the log likelihood function, and that EM is not guaranteed to find the largest of these maxima. Because the EM algorithm for Gaussian mixtures plays such an important role, we summarize it below.

EM for Gaussian Mixtures

Given a Gaussian mixture model, the goal is to maximize the likelihood function with respect to the parameters (comprising the means and covariances of the components and the mixing coefficients).

1. Initialize the means $\boldsymbol{\mu}_k$, covariances $\boldsymbol{\Sigma}_k$ and mixing coefficients π_k, and evaluate the initial value of the log likelihood.
2. **E step**. Evaluate the responsibilities using the current parameter values

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}. \tag{9.23}$$

3. **M step**. Re-estimate the parameters using the current responsibilities

$$\boldsymbol{\mu}_k^{\text{new}} = \frac{1}{N_k}\sum_{n=1}^{N}\gamma(z_{nk})\mathbf{x}_n \tag{9.24}$$

$$\boldsymbol{\Sigma}_k^{\text{new}} = \frac{1}{N_k}\sum_{n=1}^{N}\gamma(z_{nk})\left(\mathbf{x}_n - \boldsymbol{\mu}_k^{\text{new}}\right)\left(\mathbf{x}_n - \boldsymbol{\mu}_k^{\text{new}}\right)^{\mathrm{T}} \tag{9.25}$$

$$\pi_k^{\text{new}} = \frac{N_k}{N} \tag{9.26}$$

where

$$N_k = \sum_{n=1}^{N}\gamma(z_{nk}). \tag{9.27}$$

4. Evaluate the log likelihood

$$\ln p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \sum_{n=1}^{N}\ln\left\{\sum_{k=1}^{K}\pi_k\mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right\} \tag{9.28}$$

and check for convergence of either the parameters or the log likelihood. If the convergence criterion is not satisfied return to step 2.

9.3. An Alternative View of EM

In this section, we present a complementary view of the EM algorithm that recognizes the key role played by latent variables. We discuss this approach first of all in an abstract setting, and then for illustration we consider once again the case of Gaussian mixtures.

The goal of the EM algorithm is to find maximum likelihood solutions for models having latent variables. We denote the set of all observed data by $\mathbf{X}$, in which the nth row represents $\mathbf{x}_n^{\mathrm{T}}$, and similarly we denote the set of all latent variables by $\mathbf{Z}$, with a corresponding row $\mathbf{z}_n^{\mathrm{T}}$. The set of all model parameters is denoted by $\boldsymbol{\theta}$, and so the log likelihood function is given by

$$\ln p(\mathbf{X}|\boldsymbol{\theta}) = \ln\left\{\sum_{\mathbf{Z}} p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})\right\}. \tag{9.29}$$

Note that our discussion will apply equally well to continuous latent variables simply by replacing the sum over $\mathbf{Z}$ with an integral.

A key observation is that the summation over the latent variables appears inside the logarithm. Even if the joint distribution $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ belongs to the exponential

family, the marginal distribution $p(\mathbf{X}|\boldsymbol{\theta})$ typically does not as a result of this summation. The presence of the sum prevents the logarithm from acting directly on the joint distribution, resulting in complicated expressions for the maximum likelihood solution.

Now suppose that, for each observation in $\mathbf{X}$, we were told the corresponding value of the latent variable $\mathbf{Z}$. We shall call $\{\mathbf{X}, \mathbf{Z}\}$ the *complete* data set, and we shall refer to the actual observed data $\mathbf{X}$ as *incomplete*, as illustrated in Figure 9.5. The likelihood function for the complete data set simply takes the form $\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$, and we shall suppose that maximization of this complete-data log likelihood function is straightforward.

In practice, however, we are not given the complete data set $\{\mathbf{X}, \mathbf{Z}\}$, but only the incomplete data $\mathbf{X}$. Our state of knowledge of the values of the latent variables in $\mathbf{Z}$ is given only by the posterior distribution $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})$. Because we cannot use the complete-data log likelihood, we consider instead its expected value under the posterior distribution of the latent variable, which corresponds (as we shall see) to the E step of the EM algorithm. In the subsequent M step, we maximize this expectation. If the current estimate for the parameters is denoted $\boldsymbol{\theta}^{\text{old}}$, then a pair of successive E and M steps gives rise to a revised estimate $\boldsymbol{\theta}^{\text{new}}$. The algorithm is initialized by choosing some starting value for the parameters $\boldsymbol{\theta}_0$. The use of the expectation may seem somewhat arbitrary. However, we shall see the motivation for this choice when we give a deeper treatment of EM in Section 9.4.

In the E step, we use the current parameter values $\boldsymbol{\theta}^{\text{old}}$ to find the posterior distribution of the latent variables given by $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}})$. We then use this posterior distribution to find the expectation of the complete-data log likelihood evaluated for some general parameter value $\boldsymbol{\theta}$. This expectation, denoted $\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$, is given by

$$\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) = \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}) \ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}). \tag{9.30}$$

In the M step, we determine the revised parameter estimate $\boldsymbol{\theta}^{\text{new}}$ by maximizing this function

$$\boldsymbol{\theta}^{\text{new}} = \arg\max_{\boldsymbol{\theta}} \mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}). \tag{9.31}$$

Note that in the definition of $\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$, the logarithm acts directly on the joint distribution $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$, and so the corresponding M-step maximization will, by supposition, be tractable.

The general EM algorithm is summarized below. It has the property, as we shall show later, that each cycle of EM will increase the incomplete-data log likelihood (unless it is already at a local maximum).

Section 9.4

The General EM Algorithm

Given a joint distribution $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ over observed variables $\mathbf{X}$ and latent variables $\mathbf{Z}$, governed by parameters $\boldsymbol{\theta}$, the goal is to maximize the likelihood function $p(\mathbf{X}|\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$.

1. Choose an initial setting for the parameters $\boldsymbol{\theta}^{\text{old}}$.

2. **E step** Evaluate $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}})$.

3. **M step** Evaluate $\boldsymbol{\theta}^{\text{new}}$ given by

$$\boldsymbol{\theta}^{\text{new}} = \arg\max_{\boldsymbol{\theta}} \mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) \tag{9.32}$$

where

$$\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) = \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}) \ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}). \tag{9.33}$$

4. Check for convergence of either the log likelihood or the parameter values. If the convergence criterion is not satisfied, then let

$$\boldsymbol{\theta}^{\text{old}} \leftarrow \boldsymbol{\theta}^{\text{new}} \tag{9.34}$$

and return to step 2.

Exercise 9.4

The EM algorithm can also be used to find MAP (maximum posterior) solutions for models in which a prior $p(\boldsymbol{\theta})$ is defined over the parameters. In this case the E step remains the same as in the maximum likelihood case, whereas in the M step the quantity to be maximized is given by $\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) + \ln p(\boldsymbol{\theta})$. Suitable choices for the prior will remove the singularities of the kind illustrated in Figure 9.7.

Here we have considered the use of the EM algorithm to maximize a likelihood function when there are discrete latent variables. However, it can also be applied when the unobserved variables correspond to missing values in the data set. The distribution of the observed values is obtained by taking the joint distribution of all the variables and then marginalizing over the missing ones. EM can then be used to maximize the corresponding likelihood function. We shall show an example of the application of this technique in the context of principal component analysis in Figure 12.11. This will be a valid procedure if the data values are *missing at random*, meaning that the mechanism causing values to be missing does not depend on the unobserved values. In many situations this will not be the case, for instance if a sensor fails to return a value whenever the quantity it is measuring exceeds some threshold.

9.3.1 Gaussian mixtures revisited

We now consider the application of this latent variable view of EM to the specific case of a Gaussian mixture model. Recall that our goal is to maximize the log likelihood function (9.14), which is computed using the observed data set $\mathbf{X}$, and we saw that this was more difficult than for the case of a single Gaussian distribution due to the presence of the summation over k that occurs inside the logarithm. Suppose then that in addition to the observed data set $\mathbf{X}$, we were also given the values of the corresponding discrete variables $\mathbf{Z}$. Recall that Figure 9.5(a) shows a 'complete' data set (i.e., one that includes labels showing which component generated each data point) while Figure 9.5(b) shows the corresponding 'incomplete' data set. The graphical model for the complete data is shown in Figure 9.9.

Figure 9.9 This shows the same graph as in Figure 9.6 except that we now suppose that the discrete variables $\mathbf{z}_n$ are observed, as well as the data variables $\mathbf{x}_n$.

Now consider the problem of maximizing the likelihood for the complete data set $\{\mathbf{X}, \mathbf{Z}\}$. From (9.10) and (9.11), this likelihood function takes the form

$$p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \prod_{n=1}^{N} \prod_{k=1}^{K} \pi_k^{z_{nk}} \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_{nk}} \tag{9.35}$$

where z_{nk} denotes the kth component of $\mathbf{z}_n$. Taking the logarithm, we obtain

$$\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \left\{ \ln \pi_k + \ln \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}. \tag{9.36}$$

Comparison with the log likelihood function (9.14) for the incomplete data shows that the summation over k and the logarithm have been interchanged. The logarithm now acts directly on the Gaussian distribution, which itself is a member of the exponential family. Not surprisingly, this leads to a much simpler solution to the maximum likelihood problem, as we now show. Consider first the maximization with respect to the means and covariances. Because $\mathbf{z}_n$ is a K-dimensional vector with all elements equal to 0 except for a single element having the value 1, the complete-data log likelihood function is simply a sum of K independent contributions, one for each mixture component. Thus the maximization with respect to a mean or a covariance is exactly as for a single Gaussian, except that it involves only the subset of data points that are 'assigned' to that component. For the maximization with respect to the mixing coefficients, we note that these are coupled for different values of k by virtue of the summation constraint (9.9). Again, this can be enforced using a Lagrange multiplier as before, and leads to the result

$$\pi_k = \frac{1}{N} \sum_{n=1}^{N} z_{nk} \tag{9.37}$$

so that the mixing coefficients are equal to the fractions of data points assigned to the corresponding components.

Thus we see that the complete-data log likelihood function can be maximized trivially in closed form. In practice, however, we do not have values for the latent variables so, as discussed earlier, we consider the expectation, with respect to the posterior distribution of the latent variables, of the complete-data log likelihood.

Using (9.10) and (9.11) together with Bayes' theorem, we see that this posterior distribution takes the form

$$p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) \propto \prod_{n=1}^{N} \prod_{k=1}^{K} [\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)]^{z_{nk}} . \tag{9.38}$$

Exercise 9.5
Section 8.2

and hence factorizes over n so that under the posterior distribution the $\{\mathbf{z}_n\}$ are independent. This is easily verified by inspection of the directed graph in Figure 9.6 and making use of the d-separation criterion. The expected value of the indicator variable z_{nk} under this posterior distribution is then given by

$$\begin{aligned} \mathbb{E}[z_{nk}] &= \frac{\sum_{z_{nk}} z_{nk} \left[\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right]^{z_{nk}}}{\sum_{z_{nj}} \left[\pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)\right]^{z_{nj}}} \\ &= \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} = \gamma(z_{nk}) \end{aligned} \tag{9.39}$$

which is just the responsibility of component k for data point $\mathbf{x}_n$. The expected value of the complete-data log likelihood function is therefore given by

$$\mathbb{E}_{\mathbf{Z}}[\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi})] = \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma(z_{nk}) \left\{\ln \pi_k + \ln \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right\} . \tag{9.40}$$

We can now proceed as follows. First we choose some initial values for the parameters $\boldsymbol{\mu}^{\text{old}}$, $\boldsymbol{\Sigma}^{\text{old}}$ and $\boldsymbol{\pi}^{\text{old}}$, and use these to evaluate the responsibilities (the E step). We then keep the responsibilities fixed and maximize (9.40) with respect to $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$ and π_k (the M step). This leads to closed form solutions for $\boldsymbol{\mu}^{\text{new}}$, $\boldsymbol{\Sigma}^{\text{new}}$ and $\boldsymbol{\pi}^{\text{new}}$ given by (9.17), (9.19), and (9.22) as before. This is precisely the EM algorithm for Gaussian mixtures as derived earlier. We shall gain more insight into the role of the expected complete-data log likelihood function when we give a proof of convergence of the EM algorithm in Section 9.4.

Exercise 9.8

9.3.2 Relation to K-means

Comparison of the K-means algorithm with the EM algorithm for Gaussian mixtures shows that there is a close similarity. Whereas the K-means algorithm performs a *hard* assignment of data points to clusters, in which each data point is associated uniquely with one cluster, the EM algorithm makes a *soft* assignment based on the posterior probabilities. In fact, we can derive the K-means algorithm as a particular limit of EM for Gaussian mixtures as follows.

Consider a Gaussian mixture model in which the covariance matrices of the mixture components are given by $\epsilon\mathbf{I}$, where ϵ is a variance parameter that is shared

by all of the components, and $\mathbf{I}$ is the identity matrix, so that

$$p(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{(2\pi\epsilon)^{1/2}} \exp\left\{-\frac{1}{2\epsilon}\|\mathbf{x} - \boldsymbol{\mu}_k\|^2\right\}. \tag{9.41}$$

We now consider the EM algorithm for a mixture of K Gaussians of this form in which we treat ϵ as a fixed constant, instead of a parameter to be re-estimated. From (9.13) the posterior probabilities, or responsibilities, for a particular data point $\mathbf{x}_n$, are given by

$$\gamma(z_{nk}) = \frac{\pi_k \exp\left\{-\|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2/2\epsilon\right\}}{\sum_j \pi_j \exp\left\{-\|\mathbf{x}_n - \boldsymbol{\mu}_j\|^2/2\epsilon\right\}}. \tag{9.42}$$

If we consider the limit $\epsilon \to 0$, we see that in the denominator the term for which $\|\mathbf{x}_n - \boldsymbol{\mu}_j\|^2$ is smallest will go to zero most slowly, and hence the responsibilities $\gamma(z_{nk})$ for the data point $\mathbf{x}_n$ all go to zero except for term j, for which the responsibility $\gamma(z_{nj})$ will go to unity. Note that this holds independently of the values of the π_k so long as none of the π_k is zero. Thus, in this limit, we obtain a hard assignment of data points to clusters, just as in the K-means algorithm, so that $\gamma(z_{nk}) \to r_{nk}$ where r_{nk} is defined by (9.2). Each data point is thereby assigned to the cluster having the closest mean.

The EM re-estimation equation for the $\boldsymbol{\mu}_k$, given by (9.17), then reduces to the K-means result (9.4). Note that the re-estimation formula for the mixing coefficients (9.22) simply re-sets the value of π_k to be equal to the fraction of data points assigned to cluster k, although these parameters no longer play an active role in the algorithm.

Finally, in the limit $\epsilon \to 0$ the expected complete-data log likelihood, given by (9.40), becomes

Exercise 9.11

$$\mathbb{E}_{\mathbf{Z}}[\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi})] \to -\frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2 + \text{const}. \tag{9.43}$$

Thus we see that in this limit, maximizing the expected complete-data log likelihood is equivalent to minimizing the distortion measure J for the K-means algorithm given by (9.1).

Note that the K-means algorithm does not estimate the covariances of the clusters but only the cluster means. A hard-assignment version of the Gaussian mixture model with general covariance matrices, known as the *elliptical K-means* algorithm, has been considered by Sung and Poggio (1994).

9.3.3 Mixtures of Bernoulli distributions

So far in this chapter, we have focussed on distributions over continuous variables described by mixtures of Gaussians. As a further example of mixture modelling, and to illustrate the EM algorithm in a different context, we now discuss mixtures of discrete binary variables described by Bernoulli distributions. This model is also known as *latent class analysis* (Lazarsfeld and Henry, 1968; McLachlan and Peel, 2000). As well as being of practical importance in its own right, our discussion of Bernoulli mixtures will also lay the foundation for a consideration of hidden Markov models over discrete variables.

Section 13.2

Consider a set of D binary variables x_i, where $i = 1, \ldots, D$, each of which is governed by a Bernoulli distribution with parameter μ_i, so that

$$p(\mathbf{x}|\boldsymbol{\mu}) = \prod_{i=1}^{D} \mu_i^{x_i}(1-\mu_i)^{(1-x_i)} \tag{9.44}$$

where $\mathbf{x} = (x_1, \ldots, x_D)^{\mathrm{T}}$ and $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_D)^{\mathrm{T}}$. We see that the individual variables x_i are independent, given $\boldsymbol{\mu}$. The mean and covariance of this distribution are easily seen to be

$$\mathbb{E}[\mathbf{x}] = \boldsymbol{\mu} \tag{9.45}$$

$$\mathrm{cov}[\mathbf{x}] = \mathrm{diag}\{\mu_i(1-\mu_i)\}. \tag{9.46}$$

Now let us consider a finite mixture of these distributions given by

$$p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\pi}) = \sum_{k=1}^{K} \pi_k p(\mathbf{x}|\boldsymbol{\mu}_k) \tag{9.47}$$

where $\boldsymbol{\mu} = \{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K\}$, $\boldsymbol{\pi} = \{\pi_1, \ldots, \pi_K\}$, and

$$p(\mathbf{x}|\boldsymbol{\mu}_k) = \prod_{i=1}^{D} \mu_{ki}^{x_i}(1-\mu_{ki})^{(1-x_i)}. \tag{9.48}$$

Exercise 9.12

The mean and covariance of this mixture distribution are given by

$$\mathbb{E}[\mathbf{x}] = \sum_{k=1}^{K} \pi_k \boldsymbol{\mu}_k \tag{9.49}$$

$$\mathrm{cov}[\mathbf{x}] = \sum_{k=1}^{K} \pi_k \left\{\boldsymbol{\Sigma}_k + \boldsymbol{\mu}_k \boldsymbol{\mu}_k^{\mathrm{T}}\right\} - \mathbb{E}[\mathbf{x}]\mathbb{E}[\mathbf{x}]^{\mathrm{T}} \tag{9.50}$$

where $\boldsymbol{\Sigma}_k = \mathrm{diag}\{\mu_{ki}(1-\mu_{ki})\}$. Because the covariance matrix $\mathrm{cov}[\mathbf{x}]$ is no longer diagonal, the mixture distribution can capture correlations between the variables, unlike a single Bernoulli distribution.

If we are given a data set $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ then the log likelihood function for this model is given by

$$\ln p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\pi}) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k p(\mathbf{x}_n|\boldsymbol{\mu}_k) \right\}. \tag{9.51}$$

Again we see the appearance of the summation inside the logarithm, so that the maximum likelihood solution no longer has closed form.

We now derive the EM algorithm for maximizing the likelihood function for the mixture of Bernoulli distributions. To do this, we first introduce an explicit latent

variable $\mathbf{z}$ associated with each instance of $\mathbf{x}$. As in the case of the Gaussian mixture, $\mathbf{z} = (z_1, \ldots, z_K)^{\mathrm{T}}$ is a binary K-dimensional variable having a single component equal to 1, with all other components equal to 0. We can then write the conditional distribution of $\mathbf{x}$, given the latent variable, as

$$p(\mathbf{x}|\mathbf{z}, \boldsymbol{\mu}) = \prod_{k=1}^{K} p(\mathbf{x}|\boldsymbol{\mu}_k)^{z_k} \tag{9.52}$$

while the prior distribution for the latent variables is the same as for the mixture of Gaussians model, so that

$$p(\mathbf{z}|\boldsymbol{\pi}) = \prod_{k=1}^{K} \pi_k^{z_k}. \tag{9.53}$$

If we form the product of $p(\mathbf{x}|\mathbf{z}, \boldsymbol{\mu})$ and $p(\mathbf{z}|\boldsymbol{\pi})$ and then marginalize over $\mathbf{z}$, then we recover (9.47).

Exercise 9.14

In order to derive the EM algorithm, we first write down the complete-data log likelihood function, which is given by

$$\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\pi}) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \left\{ \ln \pi_k + \sum_{i=1}^{D} [x_{ni} \ln \mu_{ki} + (1 - x_{ni}) \ln(1 - \mu_{ki})] \right\} \tag{9.54}$$

where $\mathbf{X} = \{\mathbf{x}_n\}$ and $\mathbf{Z} = \{\mathbf{z}_n\}$. Next we take the expectation of the complete-data log likelihood with respect to the posterior distribution of the latent variables to give

$$\mathbb{E}_{\mathbf{Z}}[\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \boldsymbol{\pi})] = \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma(z_{nk}) \left\{ \ln \pi_k + \sum_{i=1}^{D} [x_{ni} \ln \mu_{ki} + (1 - x_{ni}) \ln(1 - \mu_{ki})] \right\} \tag{9.55}$$

where $\gamma(z_{nk}) = \mathbb{E}[z_{nk}]$ is the posterior probability, or responsibility, of component k given data point $\mathbf{x}_n$. In the E step, these responsibilities are evaluated using Bayes' theorem, which takes the form

$$\begin{aligned} \gamma(z_{nk}) = \mathbb{E}[z_{nk}] &= \frac{\sum_{z_{nk}} z_{nk} \left[\pi_k p(\mathbf{x}_n|\boldsymbol{\mu}_k)\right]^{z_{nk}}}{\sum_{z_{nj}} \left[\pi_j p(\mathbf{x}_n|\boldsymbol{\mu}_j)\right]^{z_{nj}}} \\ &= \frac{\pi_k p(\mathbf{x}_n|\boldsymbol{\mu}_k)}{\sum_{j=1}^{K} \pi_j p(\mathbf{x}_n|\boldsymbol{\mu}_j)}. \end{aligned} \tag{9.56}$$

If we consider the sum over n in (9.55), we see that the responsibilities enter only through two terms, which can be written as

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}) \tag{9.57}$$

$$\overline{\mathbf{x}}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \mathbf{x}_n \tag{9.58}$$

where N_k is the effective number of data points associated with component k. In the M step, we maximize the expected complete-data log likelihood with respect to the parameters $\boldsymbol{\mu}_k$ and $\boldsymbol{\pi}$. If we set the derivative of (9.55) with respect to $\boldsymbol{\mu}_k$ equal to zero and rearrange the terms, we obtain

Exercise 9.15

$$\boldsymbol{\mu}_k = \overline{\mathbf{x}}_k. \tag{9.59}$$

We see that this sets the mean of component k equal to a weighted mean of the data, with weighting coefficients given by the responsibilities that component k takes for data points. For the maximization with respect to π_k, we need to introduce a Lagrange multiplier to enforce the constraint $\sum_k \pi_k = 1$. Following analogous steps to those used for the mixture of Gaussians, we then obtain

Exercise 9.16

$$\pi_k = \frac{N_k}{N} \tag{9.60}$$

which represents the intuitively reasonable result that the mixing coefficient for component k is given by the effective fraction of points in the data set explained by that component.

Note that in contrast to the mixture of Gaussians, there are no singularities in which the likelihood function goes to infinity. This can be seen by noting that the likelihood function is bounded above because $0 \leqslant p(\mathbf{x}_n|\boldsymbol{\mu}_k) \leqslant 1$. There exist singularities at which the likelihood function goes to zero, but these will not be found by EM provided it is not initialized to a pathological starting point, because the EM algorithm always increases the value of the likelihood function, until a local maximum is found. We illustrate the Bernoulli mixture model in Figure 9.10 by using it to model handwritten digits. Here the digit images have been turned into binary vectors by setting all elements whose values exceed 0.5 to 1 and setting the remaining elements to 0. We now fit a data set of $N = 600$ such digits, comprising the digits '2', '3', and '4', with a mixture of $K = 3$ Bernoulli distributions by running 10 iterations of the EM algorithm. The mixing coefficients were initialized to $\pi_k = 1/K$, and the parameters μ_{kj} were set to random values chosen uniformly in the range $(0.25, 0.75)$ and then normalized to satisfy the constraint that $\sum_j \mu_{kj} = 1$. We see that a mixture of 3 Bernoulli distributions is able to find the three clusters in the data set corresponding to the different digits.

Exercise 9.17

Section 9.4

The conjugate prior for the parameters of a Bernoulli distribution is given by the beta distribution, and we have seen that a beta prior is equivalent to introducing

Figure 9.10 Illustration of the Bernoulli mixture model in which the top row shows examples from the digits data set after converting the pixel values from grey scale to binary using a threshold of 0.5. On the bottom row the first three images show the parameters μ_{ki} for each of the three components in the mixture model. As a comparison, we also fit the same data set using a single multivariate Bernoulli distribution, again using maximum likelihood. This amounts to simply averaging the counts in each pixel and is shown by the right-most image on the bottom row.

Section 2.1.1 additional effective observations of $\mathbf{x}$. We can similarly introduce priors into the Bernoulli mixture model, and use EM to maximize the posterior probability distributions.

Exercise 9.18

It is straightforward to extend the analysis of Bernoulli mixtures to the case of multinomial binary variables having $M > 2$ states by making use of the discrete distribution (2.26). Again, we can introduce Dirichlet priors over the model parameters if desired.

Exercise 9.19

9.3.4 EM for Bayesian linear regression

As a third example of the application of EM, we return to the evidence approximation for Bayesian linear regression. In Section 3.5.2, we obtained the re-estimation equations for the hyperparameters α and β by evaluation of the evidence and then setting the derivatives of the resulting expression to zero. We now turn to an alternative approach for finding α and β based on the EM algorithm. Recall that our goal is to maximize the evidence function $p(\mathbf{t}|\alpha, \beta)$ given by (3.77) with respect to α and β. Because the parameter vector $\mathbf{w}$ is marginalized out, we can regard it as a latent variable, and hence we can optimize this marginal likelihood function using EM. In the E step, we compute the posterior distribution of $\mathbf{w}$ given the current setting of the parameters α and β and then use this to find the expected complete-data log likelihood. In the M step, we maximize this quantity with respect to α and β. We have already derived the posterior distribution of $\mathbf{w}$ because this is given by (3.49). The complete-data log likelihood function is then given by

$$\ln p(\mathbf{t}, \mathbf{w}|\alpha, \beta) = \ln p(\mathbf{t}|\mathbf{w}, \beta) + \ln p(\mathbf{w}|\alpha) \tag{9.61}$$

where the likelihood $p(\mathbf{t}|\mathbf{w}, \beta)$ and the prior $p(\mathbf{w}|\alpha)$ are given by (3.10) and (3.52), respectively, and $y(\mathbf{x}, \mathbf{w})$ is given by (3.3). Taking the expectation with respect to the posterior distribution of $\mathbf{w}$ then gives

$$\begin{aligned}\mathbb{E}\left[\ln p(\mathbf{t}, \mathbf{w}|\alpha, \beta)\right] &= \frac{M}{2}\ln\left(\frac{\alpha}{2\pi}\right) - \frac{\alpha}{2}\mathbb{E}\left[\mathbf{w}^{\mathrm{T}}\mathbf{w}\right] + \frac{N}{2}\ln\left(\frac{\beta}{2\pi}\right) \\ &\quad -\frac{\beta}{2}\sum_{n=1}^{N}\mathbb{E}\left[(t_n - \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n)^2\right]. \end{aligned} \tag{9.62}$$

Setting the derivatives with respect to α to zero, we obtain the M step re-estimation equation

Exercise 9.20

$$\alpha = \frac{M}{\mathbb{E}\left[\mathbf{w}^{\mathrm{T}}\mathbf{w}\right]} = \frac{M}{\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N + \mathrm{Tr}(\mathbf{S}_N)}. \tag{9.63}$$

Exercise 9.21

An analogous result holds for β.

Note that this re-estimation equation takes a slightly different form from the corresponding result (3.92) derived by direct evaluation of the evidence function. However, they each involve computation and inversion (or eigen decomposition) of an $M \times M$ matrix and hence will have comparable computational cost per iteration.

These two approaches to determining α should of course converge to the same result (assuming they find the same local maximum of the evidence function). This can be verified by first noting that the quantity γ is defined by

$$\gamma = M - \alpha\sum_{i=1}^{M}\frac{1}{\lambda_i + \alpha} = M - \alpha\mathrm{Tr}(\mathbf{S}_N). \tag{9.64}$$

At a stationary point of the evidence function, the re-estimation equation (3.92) will be self-consistently satisfied, and hence we can substitute for γ to give

$$\alpha\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N = \gamma = M - \alpha\mathrm{Tr}(\mathbf{S}_N) \tag{9.65}$$

and solving for α we obtain (9.63), which is precisely the EM re-estimation equation.

As a final example, we consider a closely related model, namely the relevance vector machine for regression discussed in Section 7.2.1. There we used direct maximization of the marginal likelihood to derive re-estimation equations for the hyperparameters α and β. Here we consider an alternative approach in which we view the weight vector $\mathbf{w}$ as a latent variable and apply the EM algorithm. The E step involves finding the posterior distribution over the weights, and this is given by (7.81). In the M step we maximize the expected complete-data log likelihood, which is defined by

$$\mathbb{E}_{\mathbf{w}}\left[\ln p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta)p(\mathbf{w}|\boldsymbol{\alpha})\right] \tag{9.66}$$

where the expectation is taken with respect to the posterior distribution computed using the 'old' parameter values. To compute the new parameter values we maximize with respect to $\boldsymbol{\alpha}$ and β to give

Exercise 9.22

$$\alpha_i^{\text{new}} = \frac{1}{m_i^2 + \Sigma_{ii}} \tag{9.67}$$

$$(\beta^{\text{new}})^{-1} = \frac{\|\mathbf{t} - \mathbf{\Phi}\mathbf{m}_N\|^2 + \beta^{-1}\sum_i \gamma_i}{N} \tag{9.68}$$

These re-estimation equations are formally equivalent to those obtained by direct maxmization.

Exercise 9.23

9.4. The EM Algorithm in General

The *expectation maximization* algorithm, or EM algorithm, is a general technique for finding maximum likelihood solutions for probabilistic models having latent variables (Dempster *et al.*, 1977; McLachlan and Krishnan, 1997). Here we give a very general treatment of the EM algorithm and in the process provide a proof that the EM algorithm derived heuristically in Sections 9.2 and 9.3 for Gaussian mixtures does indeed maximize the likelihood function (Csiszàr and Tusnàdy, 1984; Hathaway, 1986; Neal and Hinton, 1999). Our discussion will also form the basis for the derivation of the variational inference framework.

Section 10.1

Consider a probabilistic model in which we collectively denote all of the observed variables by $\mathbf{X}$ and all of the hidden variables by $\mathbf{Z}$. The joint distribution $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ is governed by a set of parameters denoted $\boldsymbol{\theta}$. Our goal is to maximize the likelihood function that is given by

$$p(\mathbf{X}|\boldsymbol{\theta}) = \sum_{\mathbf{Z}} p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}). \tag{9.69}$$

Here we are assuming $\mathbf{Z}$ is discrete, although the discussion is identical if $\mathbf{Z}$ comprises continuous variables or a combination of discrete and continuous variables, with summation replaced by integration as appropriate.

We shall suppose that direct optimization of $p(\mathbf{X}|\boldsymbol{\theta})$ is difficult, but that optimization of the complete-data likelihood function $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ is significantly easier. Next we introduce a distribution $q(\mathbf{Z})$ defined over the latent variables, and we observe that, for any choice of $q(\mathbf{Z})$, the following decomposition holds

$$\ln p(\mathbf{X}|\boldsymbol{\theta}) = \mathcal{L}(q, \boldsymbol{\theta}) + \mathrm{KL}(q\|p) \tag{9.70}$$

where we have defined

$$\mathcal{L}(q, \boldsymbol{\theta}) = \sum_{\mathbf{Z}} q(\mathbf{Z}) \ln \left\{ \frac{p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})}{q(\mathbf{Z})} \right\} \tag{9.71}$$

$$\mathrm{KL}(q\|p) = -\sum_{\mathbf{Z}} q(\mathbf{Z}) \ln \left\{ \frac{p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})}{q(\mathbf{Z})} \right\}. \tag{9.72}$$

Note that $\mathcal{L}(q, \boldsymbol{\theta})$ is a functional (see Appendix D for a discussion of functionals) of the distribution $q(\mathbf{Z})$, and a function of the parameters $\boldsymbol{\theta}$. It is worth studying

Figure 9.11 Illustration of the decomposition given by (9.70), which holds for any choice of distribution $q(\mathbf{Z})$. Because the Kullback-Leibler divergence satisfies $\mathrm{KL}(q\|p) \geqslant 0$, we see that the quantity $\mathcal{L}(q,\boldsymbol{\theta})$ is a lower bound on the log likelihood function $\ln p(\mathbf{X}|\boldsymbol{\theta})$.

carefully the forms of the expressions (9.71) and (9.72), and in particular noting that they differ in sign and also that $\mathcal{L}(q,\boldsymbol{\theta})$ contains the joint distribution of $\mathbf{X}$ and $\mathbf{Z}$ while $\mathrm{KL}(q\|p)$ contains the conditional distribution of $\mathbf{Z}$ given $\mathbf{X}$. To verify the decomposition (9.70), we first make use of the product rule of probability to give

Exercise 9.24

$$\ln p(\mathbf{X},\mathbf{Z}|\boldsymbol{\theta}) = \ln p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}) + \ln p(\mathbf{X}|\boldsymbol{\theta}) \tag{9.73}$$

which we then substitute into the expression for $\mathcal{L}(q,\boldsymbol{\theta})$. This gives rise to two terms, one of which cancels $\mathrm{KL}(q\|p)$ while the other gives the required log likelihood $\ln p(\mathbf{X}|\boldsymbol{\theta})$ after noting that $q(\mathbf{Z})$ is a normalized distribution that sums to 1.

From (9.72), we see that $\mathrm{KL}(q\|p)$ is the Kullback-Leibler divergence between $q(\mathbf{Z})$ and the posterior distribution $p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta})$. Recall that the Kullback-Leibler divergence satisfies $\mathrm{KL}(q\|p) \geqslant 0$, with equality if, and only if, $q(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta})$. It therefore follows from (9.70) that $\mathcal{L}(q,\boldsymbol{\theta}) \leqslant \ln p(\mathbf{X}|\boldsymbol{\theta})$, in other words that $\mathcal{L}(q,\boldsymbol{\theta})$ is a lower bound on $\ln p(\mathbf{X}|\boldsymbol{\theta})$. The decomposition (9.70) is illustrated in Figure 9.11.

Section 1.6.1

The EM algorithm is a two-stage iterative optimization technique for finding maximum likelihood solutions. We can use the decomposition (9.70) to define the EM algorithm and to demonstrate that it does indeed maximize the log likelihood. Suppose that the current value of the parameter vector is $\boldsymbol{\theta}^{\mathrm{old}}$. In the E step, the lower bound $\mathcal{L}(q,\boldsymbol{\theta}^{\mathrm{old}})$ is maximized with respect to $q(\mathbf{Z})$ while holding $\boldsymbol{\theta}^{\mathrm{old}}$ fixed. The solution to this maximization problem is easily seen by noting that the value of $\ln p(\mathbf{X}|\boldsymbol{\theta}^{\mathrm{old}})$ does not depend on $q(\mathbf{Z})$ and so the largest value of $\mathcal{L}(q,\boldsymbol{\theta}^{\mathrm{old}})$ will occur when the Kullback-Leibler divergence vanishes, in other words when $q(\mathbf{Z})$ is equal to the posterior distribution $p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}^{\mathrm{old}})$. In this case, the lower bound will equal the log likelihood, as illustrated in Figure 9.12.

In the subsequent M step, the distribution $q(\mathbf{Z})$ is held fixed and the lower bound $\mathcal{L}(q,\boldsymbol{\theta})$ is maximized with respect to $\boldsymbol{\theta}$ to give some new value $\boldsymbol{\theta}^{\mathrm{new}}$. This will cause the lower bound $\mathcal{L}$ to increase (unless it is already at a maximum), which will necessarily cause the corresponding log likelihood function to increase. Because the distribution q is determined using the old parameter values rather than the new values and is held fixed during the M step, it will not equal the new posterior distribution $p(\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}^{\mathrm{new}})$, and hence there will be a nonzero KL divergence. The increase in the log likelihood function is therefore greater than the increase in the lower bound, as

Figure 9.12 Illustration of the E step of the EM algorithm. The q distribution is set equal to the posterior distribution for the current parameter values $\boldsymbol{\theta}^{\mathrm{old}}$, causing the lower bound to move up to the same value as the log likelihood function, with the KL divergence vanishing.

shown in Figure 9.13. If we substitute $q(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\mathrm{old}})$ into (9.71), we see that, after the E step, the lower bound takes the form

$$\begin{aligned} \mathcal{L}(q, \boldsymbol{\theta}) &= \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\mathrm{old}}) \ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) - \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\mathrm{old}}) \ln p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\mathrm{old}}) \\ &= \mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\mathrm{old}}) + \mathrm{const} \end{aligned} \tag{9.74}$$

where the constant is simply the negative entropy of the q distribution and is therefore independent of $\boldsymbol{\theta}$. Thus in the M step, the quantity that is being maximized is the expectation of the complete-data log likelihood, as we saw earlier in the case of mixtures of Gaussians. Note that the variable $\boldsymbol{\theta}$ over which we are optimizing appears only inside the logarithm. If the joint distribution $p(\mathbf{Z}, \mathbf{X}|\boldsymbol{\theta})$ comprises a member of the exponential family, or a product of such members, then we see that the logarithm will cancel the exponential and lead to an M step that will be typically much simpler than the maximization of the corresponding incomplete-data log likelihood function $p(\mathbf{X}|\boldsymbol{\theta})$.

The operation of the EM algorithm can also be viewed in the space of parameters, as illustrated schematically in Figure 9.14. Here the red curve depicts the (in-

Figure 9.13 Illustration of the M step of the EM algorithm. The distribution $q(\mathbf{Z})$ is held fixed and the lower bound $\mathcal{L}(q, \boldsymbol{\theta})$ is maximized with respect to the parameter vector $\boldsymbol{\theta}$ to give a revised value $\boldsymbol{\theta}^{\mathrm{new}}$. Because the KL divergence is nonnegative, this causes the log likelihood $\ln p(\mathbf{X}|\boldsymbol{\theta})$ to increase by at least as much as the lower bound does.

Figure 9.14 The EM algorithm involves alternately computing a lower bound on the log likelihood for the current parameter values and then maximizing this bound to obtain the new parameter values. See the text for a full discussion.

complete data) log likelihood function whose value we wish to maximize. We start with some initial parameter value $\boldsymbol{\theta}^{\text{old}}$, and in the first E step we evaluate the posterior distribution over latent variables, which gives rise to a lower bound $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\theta}^{(\text{old})})$ whose value equals the log likelihood at $\boldsymbol{\theta}^{(\text{old})}$, as shown by the blue curve. Note that the bound makes a tangential contact with the log likelihood at $\boldsymbol{\theta}^{(\text{old})}$, so that both curves have the same gradient. This bound is a convex function having a unique maximum (for mixture components from the exponential family). In the M step, the bound is maximized giving the value $\boldsymbol{\theta}^{(\text{new})}$, which gives a larger value of log likelihood than $\boldsymbol{\theta}^{(\text{old})}$. The subsequent E step then constructs a bound that is tangential at $\boldsymbol{\theta}^{(\text{new})}$ as shown by the green curve.

Exercise 9.25

For the particular case of an independent, identically distributed data set, $\mathbf{X}$ will comprise N data points $\{\mathbf{x}_n\}$ while $\mathbf{Z}$ will comprise N corresponding latent variables $\{\mathbf{z}_n\}$, where $n = 1, \ldots, N$. From the independence assumption, we have $p(\mathbf{X}, \mathbf{Z}) = \prod_n p(\mathbf{x}_n, \mathbf{z}_n)$ and, by marginalizing over the $\{\mathbf{z}_n\}$ we have $p(\mathbf{X}) = \prod_n p(\mathbf{x}_n)$. Using the sum and product rules, we see that the posterior probability that is evaluated in the E step takes the form

$$p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}) = \frac{p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})}{\sum_{\mathbf{Z}} p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})} = \frac{\prod_{n=1}^{N} p(\mathbf{x}_n, \mathbf{z}_n|\boldsymbol{\theta})}{\sum_{\mathbf{Z}} \prod_{n=1}^{N} p(\mathbf{x}_n, \mathbf{z}_n|\boldsymbol{\theta})} = \prod_{n=1}^{N} p(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\theta}) \tag{9.75}$$

and so the posterior distribution also factorizes with respect to n. In the case of the Gaussian mixture model this simply says that the responsibility that each of the mixture components takes for a particular data point $\mathbf{x}_n$ depends only on the value of $\mathbf{x}_n$ and on the parameters $\boldsymbol{\theta}$ of the mixture components, not on the values of the other data points.

We have seen that both the E and the M steps of the EM algorithm are increasing the value of a well-defined bound on the log likelihood function and that the

complete EM cycle will change the model parameters in such a way as to cause the log likelihood to increase (unless it is already at a maximum, in which case the parameters remain unchanged).

We can also use the EM algorithm to maximize the posterior distribution $p(\boldsymbol{\theta}|\mathbf{X})$ for models in which we have introduced a prior $p(\boldsymbol{\theta})$ over the parameters. To see this, we note that as a function of $\boldsymbol{\theta}$, we have $p(\boldsymbol{\theta}|\mathbf{X}) = p(\boldsymbol{\theta}, \mathbf{X})/p(\mathbf{X})$ and so

$$\ln p(\boldsymbol{\theta}|\mathbf{X}) = \ln p(\boldsymbol{\theta}, \mathbf{X}) - \ln p(\mathbf{X}). \tag{9.76}$$

Making use of the decomposition (9.70), we have

$$\begin{aligned} \ln p(\boldsymbol{\theta}|\mathbf{X}) &= \mathcal{L}(q, \boldsymbol{\theta}) + \mathrm{KL}(q\|p) + \ln p(\boldsymbol{\theta}) - \ln p(\mathbf{X}) \\ &\geqslant \mathcal{L}(q, \boldsymbol{\theta}) + \ln p(\boldsymbol{\theta}) - \ln p(\mathbf{X}). \end{aligned} \tag{9.77}$$

where $\ln p(\mathbf{X})$ is a constant. We can again optimize the right-hand side alternately with respect to q and $\boldsymbol{\theta}$. The optimization with respect to q gives rise to the same E-step equations as for the standard EM algorithm, because q only appears in $\mathcal{L}(q, \boldsymbol{\theta})$. The M-step equations are modified through the introduction of the prior term $\ln p(\boldsymbol{\theta})$, which typically requires only a small modification to the standard maximum likelihood M-step equations.

The EM algorithm breaks down the potentially difficult problem of maximizing the likelihood function into two stages, the E step and the M step, each of which will often prove simpler to implement. Nevertheless, for complex models it may be the case that either the E step or the M step, or indeed both, remain intractable. This leads to two possible extensions of the EM algorithm, as follows.

The *generalized EM*, or *GEM*, algorithm addresses the problem of an intractable M step. Instead of aiming to maximize $\mathcal{L}(q, \boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$, it seeks instead to change the parameters in such a way as to increase its value. Again, because $\mathcal{L}(q, \boldsymbol{\theta})$ is a lower bound on the log likelihood function, each complete EM cycle of the GEM algorithm is guaranteed to increase the value of the log likelihood (unless the parameters already correspond to a local maximum). One way to exploit the GEM approach would be to use one of the nonlinear optimization strategies, such as the conjugate gradients algorithm, during the M step. Another form of GEM algorithm, known as the *expectation conditional maximization*, or ECM, algorithm, involves making several constrained optimizations within each M step (Meng and Rubin, 1993). For instance, the parameters might be partitioned into groups, and the M step is broken down into multiple steps each of which involves optimizing one of the subset with the remainder held fixed.

We can similarly generalize the E step of the EM algorithm by performing a partial, rather than complete, optimization of $\mathcal{L}(q, \boldsymbol{\theta})$ with respect to $q(\mathbf{Z})$ (Neal and Hinton, 1999). As we have seen, for any given value of $\boldsymbol{\theta}$ there is a unique maximum of $\mathcal{L}(q, \boldsymbol{\theta})$ with respect to $q(\mathbf{Z})$ that corresponds to the posterior distribution $q_{\boldsymbol{\theta}}(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})$ and that for this choice of $q(\mathbf{Z})$ the bound $\mathcal{L}(q, \boldsymbol{\theta})$ is equal to the log likelihood function $\ln p(\mathbf{X}|\boldsymbol{\theta})$. It follows that any algorithm that converges to the global maximum of $\mathcal{L}(q, \boldsymbol{\theta})$ will find a value of $\boldsymbol{\theta}$ that is also a global maximum of the log likelihood $\ln p(\mathbf{X}|\boldsymbol{\theta})$. Provided $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ is a continuous function of $\boldsymbol{\theta}$

then, by continuity, any local maximum of $\mathcal{L}(q, \boldsymbol{\theta})$ will also be a local maximum of $\ln p(\mathbf{X}|\boldsymbol{\theta})$.

Consider the case of N independent data points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ with corresponding latent variables $\mathbf{z}_1, \ldots, \mathbf{z}_N$. The joint distribution $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ factorizes over the data points, and this structure can be exploited in an incremental form of EM in which at each EM cycle only one data point is processed at a time. In the E step, instead of recomputing the responsibilities for all of the data points, we just re-evaluate the responsibilities for one data point. It might appear that the subsequent M step would require computation involving the responsibilities for all of the data points. However, if the mixture components are members of the exponential family, then the responsibilities enter only through simple sufficient statistics, and these can be updated efficiently. Consider, for instance, the case of a Gaussian mixture, and suppose we perform an update for data point m in which the corresponding old and new values of the responsibilities are denoted $\gamma^{\text{old}}(z_{mk})$ and $\gamma^{\text{new}}(z_{mk})$. In the M step, the required sufficient statistics can be updated incrementally. For instance, for the means the sufficient statistics are defined by (9.17) and (9.18) from which we obtain

Exercise 9.26

$$\boldsymbol{\mu}_k^{\text{new}} = \boldsymbol{\mu}_k^{\text{old}} + \left(\frac{\gamma^{\text{new}}(z_{mk}) - \gamma^{\text{old}}(z_{mk})}{N_k^{\text{new}}} \right) \left(\mathbf{x}_m - \boldsymbol{\mu}_k^{\text{old}} \right) \tag{9.78}$$

together with

$$N_k^{\text{new}} = N_k^{\text{old}} + \gamma^{\text{new}}(z_{mk}) - \gamma^{\text{old}}(z_{mk}). \tag{9.79}$$

The corresponding results for the covariances and the mixing coefficients are analogous.

Thus both the E step and the M step take fixed time that is independent of the total number of data points. Because the parameters are revised after each data point, rather than waiting until after the whole data set is processed, this incremental version can converge faster than the batch version. Each E or M step in this incremental algorithm is increasing the value of $\mathcal{L}(q, \boldsymbol{\theta})$ and, as we have shown above, if the algorithm converges to a local (or global) maximum of $\mathcal{L}(q, \boldsymbol{\theta})$, this will correspond to a local (or global) maximum of the log likelihood function $\ln p(\mathbf{X}|\boldsymbol{\theta})$.

Exercises

9.1 ($\star$) **WWW** Consider the K-means algorithm discussed in Section 9.1. Show that as a consequence of there being a finite number of possible assignments for the set of discrete indicator variables r_{nk}, and that for each such assignment there is a unique optimum for the $\{\boldsymbol{\mu}_k\}$, the K-means algorithm must converge after a finite number of iterations.

9.2 ($\star$) Apply the Robbins-Monro sequential estimation procedure described in Section 2.3.5 to the problem of finding the roots of the regression function given by the derivatives of J in (9.1) with respect to $\boldsymbol{\mu}_k$. Show that this leads to a stochastic K-means algorithm in which, for each data point $\mathbf{x}_n$, the nearest prototype $\boldsymbol{\mu}_k$ is updated using (9.5).

9.3 ($\star$) **WWW** Consider a Gaussian mixture model in which the marginal distribution $p(\mathbf{z})$ for the latent variable is given by (9.10), and the conditional distribution $p(\mathbf{x}|\mathbf{z})$ for the observed variable is given by (9.11). Show that the marginal distribution $p(\mathbf{x})$, obtained by summing $p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$ over all possible values of $\mathbf{z}$, is a Gaussian mixture of the form (9.7).

9.4 ($\star$) Suppose we wish to use the EM algorithm to maximize the posterior distribution over parameters $p(\boldsymbol{\theta}|\mathbf{X})$ for a model containing latent variables, where $\mathbf{X}$ is the observed data set. Show that the E step remains the same as in the maximum likelihood case, whereas in the M step the quantity to be maximized is given by $\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) + \ln p(\boldsymbol{\theta})$ where $\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$ is defined by (9.30).

9.5 ($\star$) Consider the directed graph for a Gaussian mixture model shown in Figure 9.6. By making use of the d-separation criterion discussed in Section 8.2, show that the posterior distribution of the latent variables factorizes with respect to the different data points so that

$$p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \prod_{n=1}^{N} p(\mathbf{z}_n|\mathbf{x}_n, \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}). \tag{9.80}$$

9.6 ($\star\star$) Consider a special case of a Gaussian mixture model in which the covariance matrices $\boldsymbol{\Sigma}_k$ of the components are all constrained to have a common value $\boldsymbol{\Sigma}$. Derive the EM equations for maximizing the likelihood function under such a model.

9.7 ($\star$) **WWW** Verify that maximization of the complete-data log likelihood (9.36) for a Gaussian mixture model leads to the result that the means and covariances of each component are fitted independently to the corresponding group of data points, and the mixing coefficients are given by the fractions of points in each group.

9.8 ($\star$) **WWW** Show that if we maximize (9.40) with respect to $\boldsymbol{\mu}_k$ while keeping the responsibilities $\gamma(z_{nk})$ fixed, we obtain the closed form solution given by (9.17).

9.9 ($\star$) Show that if we maximize (9.40) with respect to $\boldsymbol{\Sigma}_k$ and π_k while keeping the responsibilities $\gamma(z_{nk})$ fixed, we obtain the closed form solutions given by (9.19) and (9.22).

9.10 ($\star\star$) Consider a density model given by a mixture distribution

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k p(\mathbf{x}|k) \tag{9.81}$$

and suppose that we partition the vector $\mathbf{x}$ into two parts so that $\mathbf{x} = (\mathbf{x}_a, \mathbf{x}_b)$. Show that the conditional density $p(\mathbf{x}_b|\mathbf{x}_a)$ is itself a mixture distribution and find expressions for the mixing coefficients and for the component densities.

9.11 ($\star$) In Section 9.3.2, we obtained a relationship between K means and EM for Gaussian mixtures by considering a mixture model in which all components have covariance $\epsilon\mathbf{I}$. Show that in the limit $\epsilon \to 0$, maximizing the expected complete-data log likelihood for this model, given by (9.40), is equivalent to minimizing the distortion measure J for the K-means algorithm given by (9.1).

9.12 ($\star$) **www** Consider a mixture distribution of the form

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k p(\mathbf{x}|k) \tag{9.82}$$

where the elements of $\mathbf{x}$ could be discrete or continuous or a combination of these. Denote the mean and covariance of $p(\mathbf{x}|k)$ by $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$, respectively. Show that the mean and covariance of the mixture distribution are given by (9.49) and (9.50).

9.13 ($\star\star$) Using the re-estimation equations for the EM algorithm, show that a mixture of Bernoulli distributions, with its parameters set to values corresponding to a maximum of the likelihood function, has the property that

$$\mathbb{E}[\mathbf{x}] = \frac{1}{N}\sum_{n=1}^{N} \mathbf{x}_n \equiv \overline{\mathbf{x}}. \tag{9.83}$$

Hence show that if the parameters of this model are initialized such that all components have the same mean $\boldsymbol{\mu}_k = \widehat{\boldsymbol{\mu}}$ for $k = 1, \ldots, K$, then the EM algorithm will converge after one iteration, for any choice of the initial mixing coefficients, and that this solution has the property $\boldsymbol{\mu}_k = \overline{\mathbf{x}}$. Note that this represents a degenerate case of the mixture model in which all of the components are identical, and in practice we try to avoid such solutions by using an appropriate initialization.

9.14 ($\star$) Consider the joint distribution of latent and observed variables for the Bernoulli distribution obtained by forming the product of $p(\mathbf{x}|\mathbf{z}, \boldsymbol{\mu})$ given by (9.52) and $p(\mathbf{z}|\boldsymbol{\pi})$ given by (9.53). Show that if we marginalize this joint distribution with respect to $\mathbf{z}$, then we obtain (9.47).

9.15 ($\star$) **www** Show that if we maximize the expected complete-data log likelihood function (9.55) for a mixture of Bernoulli distributions with respect to $\boldsymbol{\mu}_k$, we obtain the M step equation (9.59).

9.16 ($\star$) Show that if we maximize the expected complete-data log likelihood function (9.55) for a mixture of Bernoulli distributions with respect to the mixing coefficients π_k, using a Lagrange multiplier to enforce the summation constraint, we obtain the M step equation (9.60).

9.17 ($\star$) **www** Show that as a consequence of the constraint $0 \leqslant p(\mathbf{x}_n|\boldsymbol{\mu}_k) \leqslant 1$ for the discrete variable $\mathbf{x}_n$, the incomplete-data log likelihood function for a mixture of Bernoulli distributions is bounded above, and hence that there are no singularities for which the likelihood goes to infinity.

9.18 ($\star\star$) Consider a Bernoulli mixture model as discussed in Section 9.3.3, together with a prior distribution $p(\boldsymbol{\mu}_k|a_k, b_k)$ over each of the parameter vectors $\boldsymbol{\mu}_k$ given by the beta distribution (2.13), and a Dirichlet prior $p(\boldsymbol{\pi}|\boldsymbol{\alpha})$ given by (2.38). Derive the EM algorithm for maximizing the posterior probability $p(\boldsymbol{\mu}, \boldsymbol{\pi}|\mathbf{X})$.

9.19 ($\star\star$) Consider a D-dimensional variable $\mathbf{x}$ each of whose components i is itself a multinomial variable of degree M so that $\mathbf{x}$ is a binary vector with components x_{ij} where $i = 1, \ldots, D$ and $j = 1, \ldots, M$, subject to the constraint that $\sum_j x_{ij} = 1$ for all i. Suppose that the distribution of these variables is described by a mixture of the discrete multinomial distributions considered in Section 2.2 so that

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k p(\mathbf{x}|\boldsymbol{\mu}_k) \tag{9.84}$$

where

$$p(\mathbf{x}|\boldsymbol{\mu}_k) = \prod_{i=1}^{D} \prod_{j=1}^{M} \mu_{kij}^{x_{ij}}. \tag{9.85}$$

The parameters μ_{kij} represent the probabilities $p(x_{ij} = 1|\boldsymbol{\mu}_k)$ and must satisfy $0 \leqslant \mu_{kij} \leqslant 1$ together with the constraint $\sum_j \mu_{kij} = 1$ for all values of k and i. Given an observed data set $\{\mathbf{x}_n\}$, where $n = 1, \ldots, N$, derive the E and M step equations of the EM algorithm for optimizing the mixing coefficients π_k and the component parameters μ_{kij} of this distribution by maximum likelihood.

9.20 ($\star$) **WWW** Show that maximization of the expected complete-data log likelihood function (9.62) for the Bayesian linear regression model leads to the M step re-estimation result (9.63) for α.

9.21 ($\star\star$) Using the evidence framework of Section 3.5, derive the M-step re-estimation equations for the parameter β in the Bayesian linear regression model, analogous to the result (9.63) for α.

9.22 ($\star\star$) By maximization of the expected complete-data log likelihood defined by (9.66), derive the M step equations (9.67) and (9.68) for re-estimating the hyperparameters of the relevance vector machine for regression.

9.23 ($\star\star$) **WWW** In Section 7.2.1 we used direct maximization of the marginal likelihood to derive the re-estimation equations (7.87) and (7.88) for finding values of the hyperparameters $\boldsymbol{\alpha}$ and β for the regression RVM. Similarly, in Section 9.3.4 we used the EM algorithm to maximize the same marginal likelihood, giving the re-estimation equations (9.67) and (9.68). Show that these two sets of re-estimation equations are formally equivalent.

9.24 ($\star$) Verify the relation (9.70) in which $\mathcal{L}(q, \boldsymbol{\theta})$ and $\mathrm{KL}(q\|p)$ are defined by (9.71) and (9.72), respectively.

9.25 ($\star$) **www** Show that the lower bound $\mathcal{L}(q, \boldsymbol{\theta})$ given by (9.71), with $q(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{(\text{old})})$, has the same gradient with respect to $\boldsymbol{\theta}$ as the log likelihood function $\ln p(\mathbf{X}|\boldsymbol{\theta})$ at the point $\boldsymbol{\theta} = \boldsymbol{\theta}^{(\text{old})}$.

9.26 ($\star$) **www** Consider the incremental form of the EM algorithm for a mixture of Gaussians, in which the responsibilities are recomputed only for a specific data point $\mathbf{x}_m$. Starting from the M-step formulae (9.17) and (9.18), derive the results (9.78) and (9.79) for updating the component means.

9.27 ($\star\star$) Derive M-step formulae for updating the covariance matrices and mixing coefficients in a Gaussian mixture model when the responsibilities are updated incrementally, analogous to the result (9.78) for updating the means.

10

Approximate Inference

A central task in the application of probabilistic models is the evaluation of the posterior distribution $p(\mathbf{Z}|\mathbf{X})$ of the latent variables $\mathbf{Z}$ given the observed (visible) data variables $\mathbf{X}$, and the evaluation of expectations computed with respect to this distribution. The model might also contain some deterministic parameters, which we will leave implicit for the moment, or it may be a fully Bayesian model in which any unknown parameters are given prior distributions and are absorbed into the set of latent variables denoted by the vector $\mathbf{Z}$. For instance, in the EM algorithm we need to evaluate the expectation of the complete-data log likelihood with respect to the posterior distribution of the latent variables. For many models of practical interest, it will be infeasible to evaluate the posterior distribution or indeed to compute expectations with respect to this distribution. This could be because the dimensionality of the latent space is too high to work with directly or because the posterior distribution has a highly complex form for which expectations are not analytically tractable. In the case of continuous variables, the required integrations may not have closed-form

analytical solutions, while the dimensionality of the space and the complexity of the integrand may prohibit numerical integration. For discrete variables, the marginalizations involve summing over all possible configurations of the hidden variables, and though this is always possible in principle, we often find in practice that there may be exponentially many hidden states so that exact calculation is prohibitively expensive.

In such situations, we need to resort to approximation schemes, and these fall broadly into two classes, according to whether they rely on stochastic or deterministic approximations. Stochastic techniques such as Markov chain Monte Carlo, described in Chapter 11, have enabled the widespread use of Bayesian methods across many domains. They generally have the property that given infinite computational resource, they can generate exact results, and the approximation arises from the use of a finite amount of processor time. In practice, sampling methods can be computationally demanding, often limiting their use to small-scale problems. Also, it can be difficult to know whether a sampling scheme is generating independent samples from the required distribution.

In this chapter, we introduce a range of deterministic approximation schemes, some of which scale well to large applications. These are based on analytical approximations to the posterior distribution, for example by assuming that it factorizes in a particular way or that it has a specific parametric form such as a Gaussian. As such, they can never generate exact results, and so their strengths and weaknesses are complementary to those of sampling methods.

In Section 4.4, we discussed the Laplace approximation, which is based on a local Gaussian approximation to a mode (i.e., a maximum) of the distribution. Here we turn to a family of approximation techniques called *variational inference* or *variational Bayes*, which use more global criteria and which have been widely applied. We conclude with a brief introduction to an alternative variational framework known as *expectation propagation*.

10.1. Variational Inference

Variational methods have their origins in the 18th century with the work of Euler, Lagrange, and others on the *calculus of variations*. Standard calculus is concerned with finding derivatives of functions. We can think of a function as a mapping that takes the value of a variable as the input and returns the value of the function as the output. The derivative of the function then describes how the output value varies as we make infinitesimal changes to the input value. Similarly, we can define a *functional* as a mapping that takes a function as the input and that returns the value of the functional as the output. An example would be the entropy $\mathrm{H}[p]$, which takes a probability distribution $p(x)$ as the input and returns the quantity

$$\mathrm{H}[p] = \int p(x) \ln p(x) \, \mathrm{d}x \tag{10.1}$$

as the output. We can the introduce the concept of a *functional derivative*, which expresses how the value of the functional changes in response to infinitesimal changes to the input function (Feynman *et al.*, 1964). The rules for the calculus of variations mirror those of standard calculus and are discussed in Appendix D. Many problems can be expressed in terms of an optimization problem in which the quantity being optimized is a functional. The solution is obtained by exploring all possible input functions to find the one that maximizes, or minimizes, the functional. Variational methods have broad applicability and include such areas as finite element methods (Kapur, 1989) and maximum entropy (Schwarz, 1988).

Although there is nothing intrinsically approximate about variational methods, they do naturally lend themselves to finding approximate solutions. This is done by restricting the range of functions over which the optimization is performed, for instance by considering only quadratic functions or by considering functions composed of a linear combination of fixed basis functions in which only the coefficients of the linear combination can vary. In the case of applications to probabilistic inference, the restriction may for example take the form of factorization assumptions (Jordan *et al.*, 1999; Jaakkola, 2001).

Now let us consider in more detail how the concept of variational optimization can be applied to the inference problem. Suppose we have a fully Bayesian model in which all parameters are given prior distributions. The model may also have latent variables as well as parameters, and we shall denote the set of all latent variables and parameters by $\mathbf{Z}$. Similarly, we denote the set of all observed variables by $\mathbf{X}$. For example, we might have a set of N independent, identically distributed data, for which $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ and $\mathbf{Z} = \{\mathbf{z}_1, \ldots, \mathbf{z}_N\}$. Our probabilistic model specifies the joint distribution $p(\mathbf{X}, \mathbf{Z})$, and our goal is to find an approximation for the posterior distribution $p(\mathbf{Z}|\mathbf{X})$ as well as for the model evidence $p(\mathbf{X})$. As in our discussion of EM, we can decompose the log marginal probability using

$$\ln p(\mathbf{X}) = \mathcal{L}(q) + \mathrm{KL}(q\|p) \tag{10.2}$$

where we have defined

$$\mathcal{L}(q) = \int q(\mathbf{Z}) \ln \left\{ \frac{p(\mathbf{X}, \mathbf{Z})}{q(\mathbf{Z})} \right\} \mathrm{d}\mathbf{Z} \tag{10.3}$$

$$\mathrm{KL}(q\|p) = -\int q(\mathbf{Z}) \ln \left\{ \frac{p(\mathbf{Z}|\mathbf{X})}{q(\mathbf{Z})} \right\} \mathrm{d}\mathbf{Z}. \tag{10.4}$$

This differs from our discussion of EM only in that the parameter vector $\boldsymbol{\theta}$ no longer appears, because the parameters are now stochastic variables and are absorbed into $\mathbf{Z}$. Since in this chapter we will mainly be interested in continuous variables we have used integrations rather than summations in formulating this decomposition. However, the analysis goes through unchanged if some or all of the variables are discrete simply by replacing the integrations with summations as required. As before, we can maximize the lower bound $\mathcal{L}(q)$ by optimization with respect to the distribution $q(\mathbf{Z})$, which is equivalent to minimizing the KL divergence. If we allow any possible choice for $q(\mathbf{Z})$, then the maximum of the lower bound occurs when the KL divergence vanishes, which occurs when $q(\mathbf{Z})$ equals the posterior distribution $p(\mathbf{Z}|\mathbf{X})$.

Figure 10.1 Illustration of the variational approximation for the example considered earlier in Figure 4.14. The left-hand plot shows the original distribution (yellow) along with the Laplace (red) and variational (green) approximations, and the right-hand plot shows the negative logarithms of the corresponding curves.

However, we shall suppose the model is such that working with the true posterior distribution is intractable.

We therefore consider instead a restricted family of distributions $q(\mathbf{Z})$ and then seek the member of this family for which the KL divergence is minimized. Our goal is to restrict the family sufficiently that they comprise only tractable distributions, while at the same time allowing the family to be sufficiently rich and flexible that it can provide a good approximation to the true posterior distribution. It is important to emphasize that the restriction is imposed purely to achieve tractability, and that subject to this requirement we should use as rich a family of approximating distributions as possible. In particular, there is no 'over-fitting' associated with highly flexible distributions. Using more flexible approximations simply allows us to approach the true posterior distribution more closely.

One way to restrict the family of approximating distributions is to use a parametric distribution $q(\mathbf{Z}|\boldsymbol{\omega})$ governed by a set of parameters $\boldsymbol{\omega}$. The lower bound $\mathcal{L}(q)$ then becomes a function of $\boldsymbol{\omega}$, and we can exploit standard nonlinear optimization techniques to determine the optimal values for the parameters. An example of this approach, in which the variational distribution is a Gaussian and we have optimized with respect to its mean and variance, is shown in Figure 10.1.

10.1.1 Factorized distributions

Here we consider an alternative way in which to restrict the family of distributions $q(\mathbf{Z})$. Suppose we partition the elements of $\mathbf{Z}$ into disjoint groups that we denote by $\mathbf{Z}_i$ where $i = 1, \ldots, M$. We then assume that the q distribution factorizes with respect to these groups, so that

$$q(\mathbf{Z}) = \prod_{i=1}^{M} q_i(\mathbf{Z}_i). \tag{10.5}$$

It should be emphasized that we are making no further assumptions about the distribution. In particular, we place no restriction on the functional forms of the individual factors $q_i(\mathbf{Z}_i)$. This factorized form of variational inference corresponds to an approximation framework developed in physics called *mean field theory* (Parisi, 1988).

Amongst all distributions $q(\mathbf{Z})$ having the form (10.5), we now seek that distribution for which the lower bound $\mathcal{L}(q)$ is largest. We therefore wish to make a free form (variational) optimization of $\mathcal{L}(q)$ with respect to all of the distributions $q_i(\mathbf{Z}_i)$, which we do by optimizing with respect to each of the factors in turn. To achieve this, we first substitute (10.5) into (10.3) and then dissect out the dependence on one of the factors $q_j(\mathbf{Z}_j)$. Denoting $q_j(\mathbf{Z}_j)$ by simply q_j to keep the notation uncluttered, we then obtain

$$\begin{aligned}
\mathcal{L}(q) &= \int \prod_i q_i \left\{ \ln p(\mathbf{X}, \mathbf{Z}) - \sum_i \ln q_i \right\} \mathrm{d}\mathbf{Z} \\
&= \int q_j \left\{ \int \ln p(\mathbf{X}, \mathbf{Z}) \prod_{i \neq j} q_i \, \mathrm{d}\mathbf{Z}_i \right\} \mathrm{d}\mathbf{Z}_j - \int q_j \ln q_j \, \mathrm{d}\mathbf{Z}_j + \text{const} \\
&= \int q_j \ln \widetilde{p}(\mathbf{X}, \mathbf{Z}_j) \, \mathrm{d}\mathbf{Z}_j - \int q_j \ln q_j \, \mathrm{d}\mathbf{Z}_j + \text{const} \qquad (10.6)
\end{aligned}$$

where we have defined a new distribution $\widetilde{p}(\mathbf{X}, \mathbf{Z}_j)$ by the relation

$$\ln \widetilde{p}(\mathbf{X}, \mathbf{Z}_j) = \mathbb{E}_{i \neq j}[\ln p(\mathbf{X}, \mathbf{Z})] + \text{const}. \qquad (10.7)$$

Here the notation $\mathbb{E}_{i \neq j}[\cdots]$ denotes an expectation with respect to the q distributions over all variables $\mathbf{z}_i$ for $i \neq j$, so that

$$\mathbb{E}_{i \neq j}[\ln p(\mathbf{X}, \mathbf{Z})] = \int \ln p(\mathbf{X}, \mathbf{Z}) \prod_{i \neq j} q_i \, \mathrm{d}\mathbf{Z}_i. \qquad (10.8)$$

Now suppose we keep the $\{q_{i \neq j}\}$ fixed and maximize $\mathcal{L}(q)$ in (10.6) with respect to all possible forms for the distribution $q_j(\mathbf{Z}_j)$. This is easily done by recognizing that (10.6) is a negative Kullback-Leibler divergence between $q_j(\mathbf{Z}_j)$ and $\widetilde{p}(\mathbf{X}, \mathbf{Z}_j)$. Thus maximizing (10.6) is equivalent to minimizing the Kullback-Leibler

Leonhard Euler
1707–1783

Euler was a Swiss mathematician and physicist who worked in St. Petersburg and Berlin and who is widely considered to be one of the greatest mathematicians of all time. He is certainly the most prolific, and his collected works fill 75 volumes. Amongst his many contributions, he formulated the modern theory of the function, he developed (together with Lagrange) the calculus of variations, and he discovered the formula $e^{i\pi} = -1$, which relates four of the most important numbers in mathematics. During the last 17 years of his life, he was almost totally blind, and yet he produced nearly half of his results during this period.

divergence, and the minimum occurs when $q_j(\mathbf{Z}_j) = \widetilde{p}(\mathbf{X}, \mathbf{Z}_j)$. Thus we obtain a general expression for the optimal solution $q_j^\star(\mathbf{Z}_j)$ given by

$$\ln q_j^\star(\mathbf{Z}_j) = \mathbb{E}_{i \neq j}[\ln p(\mathbf{X}, \mathbf{Z})] + \text{const}. \tag{10.9}$$

It is worth taking a few moments to study the form of this solution as it provides the basis for applications of variational methods. It says that the log of the optimal solution for factor q_j is obtained simply by considering the log of the joint distribution over all hidden and visible variables and then taking the expectation with respect to all of the other factors $\{q_i\}$ for $i \neq j$.

The additive constant in (10.9) is set by normalizing the distribution $q_j^\star(\mathbf{Z}_j)$. Thus if we take the exponential of both sides and normalize, we have

$$q_j^\star(\mathbf{Z}_j) = \frac{\exp\left(\mathbb{E}_{i \neq j}[\ln p(\mathbf{X}, \mathbf{Z})]\right)}{\displaystyle\int \exp\left(\mathbb{E}_{i \neq j}[\ln p(\mathbf{X}, \mathbf{Z})]\right) \, \mathrm{d}\mathbf{Z}_j}.$$

In practice, we shall find it more convenient to work with the form (10.9) and then reinstate the normalization constant (where required) by inspection. This will become clear from subsequent examples.

The set of equations given by (10.9) for $j = 1, \ldots, M$ represent a set of consistency conditions for the maximum of the lower bound subject to the factorization constraint. However, they do not represent an explicit solution because the expression on the right-hand side of (10.9) for the optimum $q_j^\star(\mathbf{Z}_j)$ depends on expectations computed with respect to the other factors $q_i(\mathbf{Z}_i)$ for $i \neq j$. We will therefore seek a consistent solution by first initializing all of the factors $q_i(\mathbf{Z}_i)$ appropriately and then cycling through the factors and replacing each in turn with a revised estimate given by the right-hand side of (10.9) evaluated using the current estimates for all of the other factors. Convergence is guaranteed because bound is convex with respect to each of the factors $q_i(\mathbf{Z}_i)$ (Boyd and Vandenberghe, 2004).

10.1.2 Properties of factorized approximations

Our approach to variational inference is based on a factorized approximation to the true posterior distribution. Let us consider for a moment the problem of approximating a general distribution by a factorized distribution. To begin with, we discuss the problem of approximating a Gaussian distribution using a factorized Gaussian, which will provide useful insight into the types of inaccuracy introduced in using factorized approximations. Consider a Gaussian distribution $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Lambda}^{-1})$ over two correlated variables $\mathbf{z} = (z_1, z_2)$ in which the mean and precision have elements

$$\boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \qquad \boldsymbol{\Lambda} = \begin{pmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{21} & \Lambda_{22} \end{pmatrix} \tag{10.10}$$

and $\Lambda_{21} = \Lambda_{12}$ due to the symmetry of the precision matrix. Now suppose we wish to approximate this distribution using a factorized Gaussian of the form $q(\mathbf{z}) = q_1(z_1)q_2(z_2)$. We first apply the general result (10.9) to find an expression for the

optimal factor $q_1^\star(z_1)$. In doing so it is useful to note that on the right-hand side we only need to retain those terms that have some functional dependence on z_1 because all other terms can be absorbed into the normalization constant. Thus we have

$$\begin{aligned}\ln q_1^\star(z_1) &= \mathbb{E}_{z_2}[\ln p(\mathbf{z})] + \text{const} \\ &= \mathbb{E}_{z_2}\left[-\frac{1}{2}(z_1-\mu_1)^2\Lambda_{11} - (z_1-\mu_1)\Lambda_{12}(z_2-\mu_2)\right] + \text{const} \\ &= -\frac{1}{2}z_1^2\Lambda_{11} + z_1\mu_1\Lambda_{11} - z_1\Lambda_{12}\left(\mathbb{E}[z_2]-\mu_2\right) + \text{const}. \qquad (10.11)\end{aligned}$$

Next we observe that the right-hand side of this expression is a quadratic function of z_1, and so we can identify $q^\star(z_1)$ as a Gaussian distribution. It is worth emphasizing that we did not assume that $q(z_i)$ is Gaussian, but rather we derived this result by variational optimization of the KL divergence over all possible distributions $q(z_i)$. Note also that we do not need to consider the additive constant in (10.9) explicitly because it represents the normalization constant that can be found at the end by inspection if required. Using the technique of completing the square, we can identify the mean and precision of this Gaussian, giving

Section 2.3.1

$$q^\star(z_1) = \mathcal{N}(z_1|m_1, \Lambda_{11}^{-1}) \qquad (10.12)$$

where

$$m_1 = \mu_1 - \Lambda_{11}^{-1}\Lambda_{12}\left(\mathbb{E}[z_2]-\mu_2\right). \qquad (10.13)$$

By symmetry, $q_2^\star(z_2)$ is also Gaussian and can be written as

$$q_2^\star(z_2) = \mathcal{N}(z_2|m_2, \Lambda_{22}^{-1}) \qquad (10.14)$$

in which

$$m_2 = \mu_2 - \Lambda_{22}^{-1}\Lambda_{21}\left(\mathbb{E}[z_1]-\mu_1\right). \qquad (10.15)$$

Note that these solutions are coupled, so that $q^\star(z_1)$ depends on expectations computed with respect to $q^\star(z_2)$ and vice versa. In general, we address this by treating the variational solutions as re-estimation equations and cycling through the variables in turn updating them until some convergence criterion is satisfied. We shall see an example of this shortly. Here, however, we note that the problem is sufficiently simple that a closed form solution can be found. In particular, because $\mathbb{E}[z_1] = m_1$ and $\mathbb{E}[z_2] = m_2$, we see that the two equations are satisfied if we take $\mathbb{E}[z_1] = \mu_1$ and $\mathbb{E}[z_2] = \mu_2$, and it is easily shown that this is the only solution provided the distribution is nonsingular. This result is illustrated in Figure 10.2(a). We see that the mean is correctly captured but that the variance of $q(\mathbf{z})$ is controlled by the direction of smallest variance of $p(\mathbf{z})$, and that the variance along the orthogonal direction is significantly under-estimated. It is a general result that a factorized variational approximation tends to give approximations to the posterior distribution that are too compact.

Exercise 10.2

By way of comparison, suppose instead that we had been minimizing the reverse Kullback-Leibler divergence $\mathrm{KL}(p\|q)$. As we shall see, this form of KL divergence

Figure 10.2 Comparison of the two alternative forms for the Kullback-Leibler divergence. The green contours corresponding to 1, 2, and 3 standard deviations for a correlated Gaussian distribution $p(\mathbf{z})$ over two variables z_1 and z_2, and the red contours represent the corresponding levels for an approximating distribution $q(\mathbf{z})$ over the same variables given by the product of two independent univariate Gaussian distributions whose parameters are obtained by minimization of (a) the Kullback-Leibler divergence $\mathrm{KL}(q\|p)$, and (b) the reverse Kullback-Leibler divergence $\mathrm{KL}(p\|q)$.

Section 10.7

is used in an alternative approximate inference framework called *expectation propagation*. We therefore consider the general problem of minimizing $\mathrm{KL}(p\|q)$ when $q(\mathbf{Z})$ is a factorized approximation of the form (10.5). The KL divergence can then be written in the form

$$\mathrm{KL}(p\|q) = -\int p(\mathbf{Z}) \left[\sum_{i=1}^{M} \ln q_i(\mathbf{Z}_i)\right] \mathrm{d}\mathbf{Z} + \mathrm{const} \tag{10.16}$$

Exercise 10.3

where the constant term is simply the entropy of $p(\mathbf{Z})$ and so does not depend on $q(\mathbf{Z})$. We can now optimize with respect to each of the factors $q_j(\mathbf{Z}_j)$, which is easily done using a Lagrange multiplier to give

$$q_j^\star(\mathbf{Z}_j) = \int p(\mathbf{Z}) \prod_{i\neq j} \mathrm{d}\mathbf{Z}_i = p(\mathbf{Z}_j). \tag{10.17}$$

In this case, we find that the optimal solution for $q_j(\mathbf{Z}_j)$ is just given by the corresponding marginal distribution of $p(\mathbf{Z})$. Note that this is a closed-form solution and so does not require iteration.

To apply this result to the illustrative example of a Gaussian distribution $p(\mathbf{z})$ over a vector $\mathbf{z}$ we can use (2.98), which gives the result shown in Figure 10.2(b). We see that once again the mean of the approximation is correct, but that it places significant probability mass in regions of variable space that have very low probability.

The difference between these two results can be understood by noting that there is a large positive contribution to the Kullback-Leibler divergence

$$\mathrm{KL}(q\|p) = -\int q(\mathbf{Z}) \ln \left\{\frac{p(\mathbf{Z})}{q(\mathbf{Z})}\right\} \mathrm{d}\mathbf{Z} \tag{10.18}$$

Figure 10.3 Another comparison of the two alternative forms for the Kullback-Leibler divergence. (a) The blue contours show a bimodal distribution $p(\mathbf{Z})$ given by a mixture of two Gaussians, and the red contours correspond to the single Gaussian distribution $q(\mathbf{Z})$ that best approximates $p(\mathbf{Z})$ in the sense of minimizing the Kullback-Leibler divergence $\mathrm{KL}(p\|q)$. (b) As in (a) but now the red contours correspond to a Gaussian distribution $q(\mathbf{Z})$ found by numerical minimization of the Kullback-Leibler divergence $\mathrm{KL}(q\|p)$. (c) As in (b) but showing a different local minimum of the Kullback-Leibler divergence.

from regions of $\mathbf{Z}$ space in which $p(\mathbf{Z})$ is near zero unless $q(\mathbf{Z})$ is also close to zero. Thus minimizing this form of KL divergence leads to distributions $q(\mathbf{Z})$ that avoid regions in which $p(\mathbf{Z})$ is small. Conversely, the Kullback-Leibler divergence $\mathrm{KL}(p\|q)$ is minimized by distributions $q(\mathbf{Z})$ that are nonzero in regions where $p(\mathbf{Z})$ is nonzero.

We can gain further insight into the different behaviour of the two KL divergences if we consider approximating a multimodal distribution by a unimodal one, as illustrated in Figure 10.3. In practical applications, the true posterior distribution will often be multimodal, with most of the posterior mass concentrated in some number of relatively small regions of parameter space. These multiple modes may arise through nonidentifiability in the latent space or through complex nonlinear dependence on the parameters. Both types of multimodality were encountered in Chapter 9 in the context of Gaussian mixtures, where they manifested themselves as multiple maxima in the likelihood function, and a variational treatment based on the minimization of $\mathrm{KL}(q\|p)$ will tend to find one of these modes. By contrast, if we were to minimize $\mathrm{KL}(p\|q)$, the resulting approximations would average across all of the modes and, in the context of the mixture model, would lead to poor predictive distributions (because the average of two good parameter values is typically itself not a good parameter value). It is possible to make use of $\mathrm{KL}(p\|q)$ to define a useful inference procedure, but this requires a rather different approach to the one discussed here, and will be considered in detail when we discuss expectation propagation.

Section 10.7

The two forms of Kullback-Leibler divergence are members of the *alpha family*

of divergences (Ali and Silvey, 1966; Amari, 1985; Minka, 2005) defined by

$$\mathrm{D}_\alpha(p\|q) = \frac{4}{1-\alpha^2}\left(1 - \int p(x)^{(1+\alpha)/2} q(x)^{(1-\alpha)/2}\,\mathrm{d}x\right) \tag{10.19}$$

where $-\infty < \alpha < \infty$ is a continuous parameter. The Kullback-Leibler divergence $\mathrm{KL}(p\|q)$ corresponds to the limit $\alpha \to 1$, whereas $\mathrm{KL}(q\|p)$ corresponds to the limit $\alpha \to -1$. For all values of α we have $\mathrm{D}_\alpha(p\|q) \geqslant 0$, with equality if, and only if, $p(x) = q(x)$. Suppose $p(x)$ is a fixed distribution, and we minimize $\mathrm{D}_\alpha(p\|q)$ with respect to some set of distributions $q(x)$. Then for $\alpha \leqslant -1$ the divergence is *zero forcing*, so that any values of x for which $p(x) = 0$ will have $q(x) = 0$, and typically $q(x)$ will under-estimate the support of $p(x)$ and will tend to seek the mode with the largest mass. Conversely for $\alpha \geqslant 1$ the divergence is *zero-avoiding*, so that values of x for which $p(x) > 0$ will have $q(x) > 0$, and typically $q(x)$ will stretch to cover all of $p(x)$, and will over-estimate the support of $p(x)$. When $\alpha = 0$ we obtain a symmetric divergence that is linearly related to the *Hellinger distance* given by

Exercise 10.6

$$\mathrm{D}_\mathrm{H}(p\|q) = \int \left(p(x)^{1/2} - q(x)^{1/2}\right)\,\mathrm{d}x. \tag{10.20}$$

The square root of the Hellinger distance is a valid distance metric.

10.1.3 Example: The univariate Gaussian

We now illustrate the factorized variational approximation using a Gaussian distribution over a single variable x (MacKay, 2003). Our goal is to infer the posterior distribution for the mean μ and precision τ, given a data set $\mathcal{D} = \{x_1, \ldots, x_N\}$ of observed values of x which are assumed to be drawn independently from the Gaussian. The likelihood function is given by

$$p(\mathcal{D}|\mu,\tau) = \left(\frac{\tau}{2\pi}\right)^{N/2} \exp\left\{-\frac{\tau}{2}\sum_{n=1}^{N}(x_n-\mu)^2\right\}. \tag{10.21}$$

We now introduce conjugate prior distributions for μ and τ given by

$$p(\mu|\tau) = \mathcal{N}\left(\mu|\mu_0, (\lambda_0\tau)^{-1}\right) \tag{10.22}$$

$$p(\tau) = \mathrm{Gam}(\tau|a_0, b_0) \tag{10.23}$$

where $\mathrm{Gam}(\tau|a_0, b_0)$ is the gamma distribution defined by (2.146). Together these distributions constitute a Gaussian-Gamma conjugate prior distribution.

Section 2.3.6

For this simple problem the posterior distribution can be found exactly, and again takes the form of a Gaussian-gamma distribution. However, for tutorial purposes we will consider a factorized variational approximation to the posterior distribution given by

Exercise 2.44

$$q(\mu,\tau) = q_\mu(\mu)q_\tau(\tau). \tag{10.24}$$

Note that the true posterior distribution does not factorize in this way. The optimum factors $q_\mu(\mu)$ and $q_\tau(\tau)$ can be obtained from the general result (10.9) as follows. For $q_\mu(\mu)$ we have

$$\begin{aligned}\ln q_\mu^\star(\mu) &= \mathbb{E}_\tau\left[\ln p(\mathcal{D}|\mu,\tau) + \ln p(\mu|\tau)\right] + \text{const} \\ &= -\frac{\mathbb{E}[\tau]}{2}\left\{\lambda_0(\mu-\mu_0)^2 + \sum_{n=1}^{N}(x_n-\mu)^2\right\} + \text{const}. \quad (10.25)\end{aligned}$$

Completing the square over μ we see that $q_\mu(\mu)$ is a Gaussian $\mathcal{N}\left(\mu|\mu_N, \lambda_N^{-1}\right)$ with mean and precision given by

Exercise 10.7

$$\mu_N = \frac{\lambda_0\mu_0 + N\overline{x}}{\lambda_0 + N} \quad (10.26)$$

$$\lambda_N = (\lambda_0 + N)\mathbb{E}[\tau]. \quad (10.27)$$

Note that for $N \to \infty$ this gives the maximum likelihood result in which $\mu_N = \overline{x}$ and the precision is infinite.

Similarly, the optimal solution for the factor $q_\tau(\tau)$ is given by

$$\begin{aligned}\ln q_\tau^\star(\tau) &= \mathbb{E}_\mu\left[\ln p(\mathcal{D}|\mu,\tau) + \ln p(\mu|\tau)\right] + \ln p(\tau) + \text{const} \\ &= (a_0-1)\ln\tau - b_0\tau + \frac{N}{2}\ln\tau \\ &\quad -\frac{\tau}{2}\mathbb{E}_\mu\left[\sum_{n=1}^{N}(x_n-\mu)^2 + \lambda_0(\mu-\mu_0)^2\right] + \text{const} \quad (10.28)\end{aligned}$$

and hence $q_\tau(\tau)$ is a gamma distribution $\mathrm{Gam}(\tau|a_N, b_N)$ with parameters

$$a_N = a_0 + \frac{N}{2} \quad (10.29)$$

$$b_N = b_0 + \frac{1}{2}\mathbb{E}_\mu\left[\sum_{n=1}^{N}(x_n-\mu)^2 + \lambda_0(\mu-\mu_0)^2\right]. \quad (10.30)$$

Exercise 10.8

Again this exhibits the expected behaviour when $N \to \infty$.

It should be emphasized that we did not assume these specific functional forms for the optimal distributions $q_\mu(\mu)$ and $q_\tau(\tau)$. They arose naturally from the structure of the likelihood function and the corresponding conjugate priors.

Section 10.4.1

Thus we have expressions for the optimal distributions $q_\mu(\mu)$ and $q_\tau(\tau)$ each of which depends on moments evaluated with respect to the other distribution. One approach to finding a solution is therefore to make an initial guess for, say, the moment $\mathbb{E}[\tau]$ and use this to re-compute the distribution $q_\mu(\mu)$. Given this revised distribution we can then extract the required moments $\mathbb{E}[\mu]$ and $\mathbb{E}[\mu^2]$, and use these to recompute the distribution $q_\tau(\tau)$, and so on. Since the space of hidden variables for this example is only two dimensional, we can illustrate the variational approximation to the posterior distribution by plotting contours of both the true posterior and the factorized approximation, as illustrated in Figure 10.4.

Figure 10.4 Illustration of variational inference for the mean μ and precision τ of a univariate Gaussian distribution. Contours of the true posterior distribution $p(\mu, \tau|D)$ are shown in green. (a) Contours of the initial factorized approximation $q_\mu(\mu)q_\tau(\tau)$ are shown in blue. (b) After re-estimating the factor $q_\mu(\mu)$. (c) After re-estimating the factor $q_\tau(\tau)$. (d) Contours of the optimal factorized approximation, to which the iterative scheme converges, are shown in red.

In general, we will need to use an iterative approach such as this in order to solve for the optimal factorized posterior distribution. For the very simple example we are considering here, however, we can find an explicit solution by solving the simultaneous equations for the optimal factors $q_\mu(\mu)$ and $q_\tau(\tau)$. Before doing this, we can simplify these expressions by considering broad, noninformative priors in which $\mu_0 = a_0 = b_0 = \lambda_0 = 0$. Although these parameter settings correspond to improper priors, we see that the posterior distribution is still well defined. Using the standard result $\mathbb{E}[\tau] = a_N/b_N$ for the mean of a gamma distribution, together with (10.29) and (10.30), we have

Appendix B

$$\frac{1}{\mathbb{E}[\tau]} = \mathbb{E}\left[\frac{1}{N}\sum_{n=1}^{N}(x_n - \mu)^2\right] = \overline{x^2} - 2\overline{x}\mathbb{E}[\mu] + \mathbb{E}[\mu^2]. \tag{10.31}$$

Then, using (10.26) and (10.27), we obtain the first and second order moments of

$q_\mu(\mu)$ in the form

$$\mathbb{E}[\mu] = \overline{x}, \qquad \mathbb{E}[\mu^2] = \overline{x}^2 + \frac{1}{N\mathbb{E}[\tau]}. \tag{10.32}$$

Exercise 10.9 We can now substitute these moments into (10.31) and then solve for $\mathbb{E}[\tau]$ to give

$$\begin{aligned} \frac{1}{\mathbb{E}[\tau]} &= \frac{1}{N-1}(\overline{x^2} - \overline{x}^2) \\ &= \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \overline{x})^2. \end{aligned} \tag{10.33}$$

We recognize the right-hand side as the familiar unbiased estimator for the variance of a univariate Gaussian distribution, and so we see that the use of a Bayesian approach has avoided the bias of the maximum likelihood solution.

Section 1.2.4

10.1.4 Model comparison

As well as performing inference over the hidden variables $\mathbf{Z}$, we may also wish to compare a set of candidate models, labelled by the index m, and having prior probabilities $p(m)$. Our goal is then to approximate the posterior probabilities $p(m|\mathbf{X})$, where $\mathbf{X}$ is the observed data. This is a slightly more complex situation than that considered so far because different models may have different structure and indeed different dimensionality for the hidden variables $\mathbf{Z}$. We cannot therefore simply consider a factorized approximation $q(\mathbf{Z})q(m)$, but must instead recognize that the posterior over $\mathbf{Z}$ must be conditioned on m, and so we must consider $q(\mathbf{Z}, m) = q(\mathbf{Z}|m)q(m)$. We can readily verify the following decomposition based on this variational distribution

Exercise 10.10

$$\ln p(\mathbf{X}) = \mathcal{L}_m - \sum_m \sum_{\mathbf{Z}} q(\mathbf{Z}|m)q(m) \ln \left\{ \frac{p(\mathbf{Z}, m|\mathbf{X})}{q(\mathbf{Z}|m)q(m)} \right\} \tag{10.34}$$

where the $\mathcal{L}_m$ is a lower bound on $\ln p(\mathbf{X})$ and is given by

$$\mathcal{L}_m = \sum_m \sum_{\mathbf{Z}} q(\mathbf{Z}|m)q(m) \ln \left\{ \frac{p(\mathbf{Z}, \mathbf{X}, m)}{q(\mathbf{Z}|m)q(m)} \right\}. \tag{10.35}$$

Here we are assuming discrete $\mathbf{Z}$, but the same analysis applies to continuous latent variables provided the summations are replaced with integrations. We can maximize $\mathcal{L}_m$ with respect to the distribution $q(m)$ using a Lagrange multiplier, with the result

Exercise 10.11

$$q(m) \propto p(m) \exp\{\mathcal{L}_m\}. \tag{10.36}$$

However, if we maximize $\mathcal{L}_m$ with respect to the $q(\mathbf{Z}|m)$, we find that the solutions for different m are coupled, as we expect because they are conditioned on m. We proceed instead by first optimizing each of the $q(\mathbf{Z}|m)$ individually by optimization

of (10.35), and then subsequently determining the $q(m)$ using (10.36). After normalization the resulting values for $q(m)$ can be used for model selection or model averaging in the usual way.

10.2. Illustration: Variational Mixture of Gaussians

We now return to our discussion of the Gaussian mixture model and apply the variational inference machinery developed in the previous section. This will provide a good illustration of the application of variational methods and will also demonstrate how a Bayesian treatment elegantly resolves many of the difficulties associated with the maximum likelihood approach (Attias, 1999b). The reader is encouraged to work through this example in detail as it provides many insights into the practical application of variational methods. Many Bayesian models, corresponding to much more sophisticated distributions, can be solved by straightforward extensions and generalizations of this analysis.

Our starting point is the likelihood function for the Gaussian mixture model, illustrated by the graphical model in Figure 9.6. For each observation $\mathbf{x}_n$ we have a corresponding latent variable $\mathbf{z}_n$ comprising a 1-of-K binary vector with elements z_{nk} for $k = 1, \ldots, K$. As before we denote the observed data set by $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, and similarly we denote the latent variables by $\mathbf{Z} = \{\mathbf{z}_1, \ldots, \mathbf{z}_N\}$. From (9.10) we can write down the conditional distribution of $\mathbf{Z}$, given the mixing coefficients $\boldsymbol{\pi}$, in the form

$$p(\mathbf{Z}|\boldsymbol{\pi}) = \prod_{n=1}^{N} \prod_{k=1}^{K} \pi_k^{z_{nk}}. \tag{10.37}$$

Similarly, from (9.11), we can write down the conditional distribution of the observed data vectors, given the latent variables and the component parameters

$$p(\mathbf{X}|\mathbf{Z}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) = \prod_{n=1}^{N} \prod_{k=1}^{K} \mathcal{N}\left(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k^{-1}\right)^{z_{nk}} \tag{10.38}$$

where $\boldsymbol{\mu} = \{\boldsymbol{\mu}_k\}$ and $\boldsymbol{\Lambda} = \{\boldsymbol{\Lambda}_k\}$. Note that we are working in terms of precision matrices rather than covariance matrices as this somewhat simplifies the mathematics.

Next we introduce priors over the parameters $\boldsymbol{\mu}$, $\boldsymbol{\Lambda}$ and $\boldsymbol{\pi}$. The analysis is considerably simplified if we use conjugate prior distributions. We therefore choose a Dirichlet distribution over the mixing coefficients $\boldsymbol{\pi}$

Section 10.4.1

$$p(\boldsymbol{\pi}) = \mathrm{Dir}(\boldsymbol{\pi}|\boldsymbol{\alpha}_0) = C(\boldsymbol{\alpha}_0) \prod_{k=1}^{K} \pi_k^{\alpha_0 - 1} \tag{10.39}$$

where by symmetry we have chosen the same parameter α_0 for each of the components, and $C(\boldsymbol{\alpha}_0)$ is the normalization constant for the Dirichlet distribution defined

Figure 10.5 Directed acyclic graph representing the Bayesian mixture of Gaussians model, in which the box (plate) denotes a set of N i.i.d. observations. Here $\boldsymbol{\mu}$ denotes $\{\boldsymbol{\mu}_k\}$ and $\boldsymbol{\Lambda}$ denotes $\{\boldsymbol{\Lambda}_k\}$.

Section 2.2.1

by (B.23). As we have seen, the parameter α_0 can be interpreted as the effective prior number of observations associated with each component of the mixture. If the value of α_0 is small, then the posterior distribution will be influenced primarily by the data rather than by the prior.

Similarly, we introduce an independent Gaussian-Wishart prior governing the mean and precision of each Gaussian component, given by

$$\begin{aligned} p(\boldsymbol{\mu}, \boldsymbol{\Lambda}) &= p(\boldsymbol{\mu}|\boldsymbol{\Lambda})p(\boldsymbol{\Lambda}) \\ &= \prod_{k=1}^{K} \mathcal{N}\left(\boldsymbol{\mu}_k|\mathbf{m}_0, (\beta_0\boldsymbol{\Lambda}_k)^{-1}\right) \mathcal{W}(\boldsymbol{\Lambda}_k|\mathbf{W}_0, \nu_0) \end{aligned} \tag{10.40}$$

because this represents the conjugate prior distribution when both the mean and precision are unknown. Typically we would choose $\mathbf{m}_0 = \mathbf{0}$ by symmetry.

Section 2.3.6

The resulting model can be represented as a directed graph as shown in Figure 10.5. Note that there is a link from $\boldsymbol{\Lambda}$ to $\boldsymbol{\mu}$ since the variance of the distribution over $\boldsymbol{\mu}$ in (10.40) is a function of $\boldsymbol{\Lambda}$.

This example provides a nice illustration of the distinction between latent variables and parameters. Variables such as $\mathbf{z}_n$ that appear inside the plate are regarded as latent variables because the number of such variables grows with the size of the data set. By contrast, variables such as $\boldsymbol{\mu}$ that are outside the plate are fixed in number independently of the size of the data set, and so are regarded as parameters. From the perspective of graphical models, however, there is really no fundamental difference between them.

10.2.1 Variational distribution

In order to formulate a variational treatment of this model, we next write down the joint distribution of all of the random variables, which is given by

$$p(\mathbf{X}, \mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) = p(\mathbf{X}|\mathbf{Z}, \boldsymbol{\mu}, \boldsymbol{\Lambda})p(\mathbf{Z}|\boldsymbol{\pi})p(\boldsymbol{\pi})p(\boldsymbol{\mu}|\boldsymbol{\Lambda})p(\boldsymbol{\Lambda}) \tag{10.41}$$

in which the various factors are defined above. The reader should take a moment to verify that this decomposition does indeed correspond to the probabilistic graphical model shown in Figure 10.5. Note that only the variables $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ are observed.

We now consider a variational distribution which factorizes between the latent variables and the parameters so that

$$q(\mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) = q(\mathbf{Z})q(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}). \tag{10.42}$$

It is remarkable that this is the *only* assumption that we need to make in order to obtain a tractable practical solution to our Bayesian mixture model. In particular, the functional form of the factors $q(\mathbf{Z})$ and $q(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda})$ will be determined automatically by optimization of the variational distribution. Note that we are omitting the subscripts on the q distributions, much as we do with the p distributions in (10.41), and are relying on the arguments to distinguish the different distributions.

The corresponding sequential update equations for these factors can be easily derived by making use of the general result (10.9). Let us consider the derivation of the update equation for the factor $q(\mathbf{Z})$. The log of the optimized factor is given by

$$\ln q^\star(\mathbf{Z}) = \mathbb{E}_{\boldsymbol{\pi},\boldsymbol{\mu},\boldsymbol{\Lambda}}[\ln p(\mathbf{X}, \mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda})] + \text{const}. \tag{10.43}$$

We now make use of the decomposition (10.41). Note that we are only interested in the functional dependence of the right-hand side on the variable $\mathbf{Z}$. Thus any terms that do not depend on $\mathbf{Z}$ can be absorbed into the additive normalization constant, giving

$$\ln q^\star(\mathbf{Z}) = \mathbb{E}_{\boldsymbol{\pi}}[\ln p(\mathbf{Z}|\boldsymbol{\pi})] + \mathbb{E}_{\boldsymbol{\mu},\boldsymbol{\Lambda}}[\ln p(\mathbf{X}|\mathbf{Z}, \boldsymbol{\mu}, \boldsymbol{\Lambda})] + \text{const}. \tag{10.44}$$

Substituting for the two conditional distributions on the right-hand side, and again absorbing any terms that are independent of $\mathbf{Z}$ into the additive constant, we have

$$\ln q^\star(\mathbf{Z}) = \sum_{n=1}^{N}\sum_{k=1}^{K} z_{nk} \ln \rho_{nk} + \text{const} \tag{10.45}$$

where we have defined

$$\begin{aligned} \ln \rho_{nk} &= \mathbb{E}[\ln \pi_k] + \frac{1}{2}\mathbb{E}\left[\ln |\boldsymbol{\Lambda}_k|\right] - \frac{D}{2}\ln(2\pi) \\ &\quad -\frac{1}{2}\mathbb{E}_{\boldsymbol{\mu}_k,\boldsymbol{\Lambda}_k}\left[(\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}}\boldsymbol{\Lambda}_k(\mathbf{x}_n - \boldsymbol{\mu}_k)\right] \end{aligned} \tag{10.46}$$

where D is the dimensionality of the data variable $\mathbf{x}$. Taking the exponential of both sides of (10.45) we obtain

$$q^\star(\mathbf{Z}) \propto \prod_{n=1}^{N}\prod_{k=1}^{K} \rho_{nk}^{z_{nk}}. \tag{10.47}$$

Requiring that this distribution be normalized, and noting that for each value of n the quantities z_{nk} are binary and sum to 1 over all values of k, we obtain

Exercise 10.12

$$q^\star(\mathbf{Z}) = \prod_{n=1}^{N}\prod_{k=1}^{K} r_{nk}^{z_{nk}} \tag{10.48}$$

where

$$r_{nk} = \frac{\rho_{nk}}{\sum_{j=1}^{K} \rho_{nj}}. \tag{10.49}$$

We see that the optimal solution for the factor $q(\mathbf{Z})$ takes the same functional form as the prior $p(\mathbf{Z}|\boldsymbol{\pi})$. Note that because ρ_{nk} is given by the exponential of a real quantity, the quantities r_{nk} will be nonnegative and will sum to one, as required.

For the discrete distribution $q^\star(\mathbf{Z})$ we have the standard result

$$\mathbb{E}[z_{nk}] = r_{nk} \tag{10.50}$$

from which we see that the quantities r_{nk} are playing the role of responsibilities. Note that the optimal solution for $q^\star(\mathbf{Z})$ depends on moments evaluated with respect to the distributions of other variables, and so again the variational update equations are coupled and must be solved iteratively.

At this point, we shall find it convenient to define three statistics of the observed data set evaluated with respect to the responsibilities, given by

$$N_k = \sum_{n=1}^{N} r_{nk} \tag{10.51}$$

$$\overline{\mathbf{x}}_k = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk} \mathbf{x}_n \tag{10.52}$$

$$\mathbf{S}_k = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk} (\mathbf{x}_n - \overline{\mathbf{x}}_k)(\mathbf{x}_n - \overline{\mathbf{x}}_k)^{\mathrm{T}}. \tag{10.53}$$

Note that these are analogous to quantities evaluated in the maximum likelihood EM algorithm for the Gaussian mixture model.

Now let us consider the factor $q(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda})$ in the variational posterior distribution. Again using the general result (10.9) we have

$$\begin{aligned} \ln q^\star(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) = \ln p(\boldsymbol{\pi}) + \sum_{k=1}^{K} \ln p(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k) + \mathbb{E}_{\mathbf{Z}} \left[\ln p(\mathbf{Z}|\boldsymbol{\pi})\right] \\ + \sum_{k=1}^{K} \sum_{n=1}^{N} \mathbb{E}[z_{nk}] \ln \mathcal{N}\left(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k^{-1}\right) + \text{const}. \end{aligned} \tag{10.54}$$

We observe that the right-hand side of this expression decomposes into a sum of terms involving only $\boldsymbol{\pi}$ together with terms only involving $\boldsymbol{\mu}$ and $\boldsymbol{\Lambda}$, which implies that the variational posterior $q(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda})$ factorizes to give $q(\boldsymbol{\pi})q(\boldsymbol{\mu}, \boldsymbol{\Lambda})$. Furthermore, the terms involving $\boldsymbol{\mu}$ and $\boldsymbol{\Lambda}$ themselves comprise a sum over k of terms involving $\boldsymbol{\mu}_k$ and $\boldsymbol{\Lambda}_k$ leading to the further factorization

$$q(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) = q(\boldsymbol{\pi}) \prod_{k=1}^{K} q(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k). \tag{10.55}$$

Identifying the terms on the right-hand side of (10.54) that depend on $\boldsymbol{\pi}$, we have

$$\ln q^\star(\boldsymbol{\pi}) = (\alpha_0 - 1)\sum_{k=1}^{K} \ln \pi_k + \sum_{k=1}^{K}\sum_{n=1}^{N} r_{nk} \ln \pi_k + \text{const} \tag{10.56}$$

where we have used (10.50). Taking the exponential of both sides, we recognize $q^\star(\boldsymbol{\pi})$ as a Dirichlet distribution

$$q^\star(\boldsymbol{\pi}) = \text{Dir}(\boldsymbol{\pi}|\boldsymbol{\alpha}) \tag{10.57}$$

where $\boldsymbol{\alpha}$ has components α_k given by

$$\alpha_k = \alpha_0 + N_k. \tag{10.58}$$

Finally, the variational posterior distribution $q^\star(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k)$ does not factorize into the product of the marginals, but we can always use the product rule to write it in the form $q^\star(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k) = q^\star(\boldsymbol{\mu}_k|\boldsymbol{\Lambda}_k)q^\star(\boldsymbol{\Lambda}_k)$. The two factors can be found by inspecting (10.54) and reading off those terms that involve $\boldsymbol{\mu}_k$ and $\boldsymbol{\Lambda}_k$. The result, as expected, is a Gaussian-Wishart distribution and is given by

Exercise 10.13

$$q^\star(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k) = \mathcal{N}\left(\boldsymbol{\mu}_k|\mathbf{m}_k, (\beta_k\boldsymbol{\Lambda}_k)^{-1}\right)\, \mathcal{W}(\boldsymbol{\Lambda}_k|\mathbf{W}_k, \nu_k) \tag{10.59}$$

where we have defined

$$\beta_k = \beta_0 + N_k \tag{10.60}$$

$$\mathbf{m}_k = \frac{1}{\beta_k}\left(\beta_0\mathbf{m}_0 + N_k\overline{\mathbf{x}}_k\right) \tag{10.61}$$

$$\mathbf{W}_k^{-1} = \mathbf{W}_0^{-1} + N_k\mathbf{S}_k + \frac{\beta_0 N_k}{\beta_0 + N_k}(\overline{\mathbf{x}}_k - \mathbf{m}_0)(\overline{\mathbf{x}}_k - \mathbf{m}_0)^{\mathrm{T}} \tag{10.62}$$

$$\nu_k = \nu_0 + N_k. \tag{10.63}$$

These update equations are analogous to the M-step equations of the EM algorithm for the maximum likelihood solution of the mixture of Gaussians. We see that the computations that must be performed in order to update the variational posterior distribution over the model parameters involve evaluation of the same sums over the data set, as arose in the maximum likelihood treatment.

In order to perform this variational M step, we need the expectations $\mathbb{E}[z_{nk}] = r_{nk}$ representing the responsibilities. These are obtained by normalizing the ρ_{nk} that are given by (10.46). We see that this expression involves expectations with respect to the variational distributions of the parameters, and these are easily evaluated to give

Exercise 10.14

$$\mathbb{E}_{\boldsymbol{\mu}_k,\boldsymbol{\Lambda}_k}\left[(\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}}\boldsymbol{\Lambda}_k(\mathbf{x}_n - \boldsymbol{\mu}_k)\right] = D\beta_k^{-1} + \nu_k(\mathbf{x}_n - \mathbf{m}_k)^{\mathrm{T}}\mathbf{W}_k(\mathbf{x}_n - \mathbf{m}_k) \tag{10.64}$$

$$\ln \widetilde{\Lambda}_k \equiv \mathbb{E}\left[\ln|\boldsymbol{\Lambda}_k|\right] = \sum_{i=1}^{D}\psi\left(\frac{\nu_k + 1 - i}{2}\right) + D\ln 2 + \ln|\mathbf{W}_k| \tag{10.65}$$

$$\ln \widetilde{\pi}_k \equiv \mathbb{E}\left[\ln \pi_k\right] = \psi(\alpha_k) - \psi(\widehat{\alpha}) \tag{10.66}$$

where we have introduced definitions of $\widetilde{\Lambda}_k$ and $\widetilde{\pi}_k$, and $\psi(\cdot)$ is the digamma function defined by (B.25), with $\widehat{\alpha} = \sum_k \alpha_k$. The results (10.65) and (10.66) follow from the standard properties of the Wishart and Dirichlet distributions.

Appendix B

If we substitute (10.64), (10.65), and (10.66) into (10.46) and make use of (10.49), we obtain the following result for the responsibilities

$$r_{nk} \propto \widetilde{\pi}_k \widetilde{\Lambda}_k^{1/2} \exp\left\{-\frac{D}{2\beta_k} - \frac{\nu_k}{2}(\mathbf{x}_n - \mathbf{m}_k)^{\mathrm{T}}\mathbf{W}_k(\mathbf{x}_n - \mathbf{m}_k)\right\}. \tag{10.67}$$

Notice the similarity to the corresponding result for the responsibilities in maximum likelihood EM, which from (9.13) can be written in the form

$$r_{nk} \propto \pi_k |\boldsymbol{\Lambda}_k|^{1/2} \exp\left\{-\frac{1}{2}(\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}}\boldsymbol{\Lambda}_k(\mathbf{x}_n - \boldsymbol{\mu}_k)\right\} \tag{10.68}$$

where we have used the precision in place of the covariance to highlight the similarity to (10.67).

Thus the optimization of the variational posterior distribution involves cycling between two stages analogous to the E and M steps of the maximum likelihood EM algorithm. In the variational equivalent of the E step, we use the current distributions over the model parameters to evaluate the moments in (10.64), (10.65), and (10.66) and hence evaluate $\mathbb{E}[z_{nk}] = r_{nk}$. Then in the subsequent variational equivalent of the M step, we keep these responsibilities fixed and use them to re-compute the variational distribution over the parameters using (10.57) and (10.59). In each case, we see that the variational posterior distribution has the same functional form as the corresponding factor in the joint distribution (10.41). This is a general result and is a consequence of the choice of conjugate distributions.

Section 10.4.1

Figure 10.6 shows the results of applying this approach to the rescaled Old Faithful data set for a Gaussian mixture model having $K = 6$ components. We see that after convergence, there are only two components for which the expected values of the mixing coefficients are numerically distinguishable from their prior values. This effect can be understood qualitatively in terms of the automatic trade-off in a Bayesian model between fitting the data and the complexity of the model, in which the complexity penalty arises from components whose parameters are pushed away from their prior values. Components that take essentially no responsibility for explaining the data points have $r_{nk} \simeq 0$ and hence $N_k \simeq 0$. From (10.58), we see that $\alpha_k \simeq \alpha_0$ and from (10.60)–(10.63) we see that the other parameters revert to their prior values. In principle such components are fitted slightly to the data points, but for broad priors this effect is too small to be seen numerically. For the variational Gaussian mixture model the expected values of the mixing coefficients in the posterior distribution are given by

Section 3.4

Exercise 10.15

$$\mathbb{E}[\pi_k] = \frac{\alpha_k + N_k}{K\alpha_0 + N}. \tag{10.69}$$

Consider a component for which $N_k \simeq 0$ and $\alpha_k \simeq \alpha_0$. If the prior is broad so that $\alpha_0 \to 0$, then $\mathbb{E}[\pi_k] \to 0$ and the component plays no role in the model, whereas if

Figure 10.6 Variational Bayesian mixture of $K = 6$ Gaussians applied to the Old Faithful data set, in which the ellipses denote the one standard-deviation density contours for each of the components, and the density of red ink inside each ellipse corresponds to the mean value of the mixing coefficient for each component. The number in the top left of each diagram shows the number of iterations of variational inference. Components whose expected mixing coefficient are numerically indistinguishable from zero are not plotted.

the prior tightly constrains the mixing coefficients so that $\alpha_0 \to \infty$, then $\mathbb{E}[\pi_k] \to 1/K$.

In Figure 10.6, the prior over the mixing coefficients is a Dirichlet of the form (10.39). Recall from Figure 2.5 that for $\alpha_0 < 1$ the prior favours solutions in which some of the mixing coefficients are zero. Figure 10.6 was obtained using $\alpha_0 = 10^{-3}$, and resulted in two components having nonzero mixing coefficients. If instead we choose $\alpha_0 = 1$ we obtain three components with nonzero mixing coefficients, and for $\alpha = 10$ all six components have nonzero mixing coefficients.

As we have seen there is a close similarity between the variational solution for the Bayesian mixture of Gaussians and the EM algorithm for maximum likelihood. In fact if we consider the limit $N \to \infty$ then the Bayesian treatment converges to the maximum likelihood EM algorithm. For anything other than very small data sets, the dominant computational cost of the variational algorithm for Gaussian mixtures arises from the evaluation of the responsibilities, together with the evaluation and inversion of the weighted data covariance matrices. These computations mirror precisely those that arise in the maximum likelihood EM algorithm, and so there is little computational overhead in using this Bayesian approach as compared to the traditional maximum likelihood one. There are, however, some substantial advantages. First of all, the singularities that arise in maximum likelihood when a Gaussian component 'collapses' onto a specific data point are absent in the Bayesian treatment.

Indeed, these singularities are removed if we simply introduce a prior and then use a MAP estimate instead of maximum likelihood. Furthermore, there is no over-fitting if we choose a large number K of components in the mixture, as we saw in Figure 10.6. Finally, the variational treatment opens up the possibility of determining the optimal number of components in the mixture without resorting to techniques such as cross validation.

Section 10.2.4

10.2.2 Variational lower bound

We can also straightforwardly evaluate the lower bound (10.3) for this model. In practice, it is useful to be able to monitor the bound during the re-estimation in order to test for convergence. It can also provide a valuable check on both the mathematical expressions for the solutions and their software implementation, because at each step of the iterative re-estimation procedure the value of this bound should not decrease. We can take this a stage further to provide a deeper test of the correctness of both the mathematical derivation of the update equations and of their software implementation by using finite differences to check that each update does indeed give a (constrained) maximum of the bound (Svensén and Bishop, 2004).

For the variational mixture of Gaussians, the lower bound (10.3) is given by

$$\begin{aligned}
\mathcal{L} &= \sum_{\mathbf{Z}} \iiint q(\mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}) \ln \left\{ \frac{p(\mathbf{X}, \mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda})}{q(\mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda})} \right\} \mathrm{d}\boldsymbol{\pi}\, \mathrm{d}\boldsymbol{\mu}\, \mathrm{d}\boldsymbol{\Lambda} \\
&= \mathbb{E}[\ln p(\mathbf{X}, \mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda})] - \mathbb{E}[\ln q(\mathbf{Z}, \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda})] \\
&= \mathbb{E}[\ln p(\mathbf{X}|\mathbf{Z}, \boldsymbol{\mu}, \boldsymbol{\Lambda})] + \mathbb{E}[\ln p(\mathbf{Z}|\boldsymbol{\pi})] + \mathbb{E}[\ln p(\boldsymbol{\pi})] + \mathbb{E}[\ln p(\boldsymbol{\mu}, \boldsymbol{\Lambda})] \\
&\quad - \mathbb{E}[\ln q(\mathbf{Z})] - \mathbb{E}[\ln q(\boldsymbol{\pi})] - \mathbb{E}[\ln q(\boldsymbol{\mu}, \boldsymbol{\Lambda})] \qquad (10.70)
\end{aligned}$$

where, to keep the notation uncluttered, we have omitted the $\star$ superscript on the q distributions, along with the subscripts on the expectation operators because each expectation is taken with respect to all of the random variables in its argument. The various terms in the bound are easily evaluated to give the following results

Exercise 10.16

$$\begin{aligned}
\mathbb{E}[\ln p(\mathbf{X}|\mathbf{Z}, \boldsymbol{\mu}, \boldsymbol{\Lambda})] = \frac{1}{2} \sum_{k=1}^{K} N_k \Big\{ & \ln \widetilde{\Lambda}_k - D\beta_k^{-1} - \nu_k \mathrm{Tr}(\mathbf{S}_k \mathbf{W}_k) \\
& - \nu_k (\overline{\mathbf{x}}_k - \mathbf{m}_k)^{\mathrm{T}} \mathbf{W}_k (\overline{\mathbf{x}}_k - \mathbf{m}_k) - D \ln(2\pi) \Big\} \qquad (10.71)
\end{aligned}$$

$$\mathbb{E}[\ln p(\mathbf{Z}|\boldsymbol{\pi})] = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \ln \widetilde{\pi}_k \qquad (10.72)$$

$$\mathbb{E}[\ln p(\boldsymbol{\pi})] = \ln C(\boldsymbol{\alpha}_0) + (\alpha_0 - 1) \sum_{k=1}^{K} \ln \widetilde{\pi}_k \qquad (10.73)$$

$$\mathbb{E}[\ln p(\boldsymbol{\mu}, \boldsymbol{\Lambda})] = \frac{1}{2}\sum_{k=1}^{K}\left\{D\ln(\beta_0/2\pi) + \ln\widetilde{\Lambda}_k - \frac{D\beta_0}{\beta_k}\right.$$
$$\left. -\beta_0\nu_k(\mathbf{m}_k - \mathbf{m}_0)^{\mathrm{T}}\mathbf{W}_k(\mathbf{m}_k - \mathbf{m}_0)\right\} + K\ln B(\mathbf{W}_0, \nu_0)$$
$$+\frac{(\nu_0 - D - 1)}{2}\sum_{k=1}^{K}\ln\widetilde{\Lambda}_k - \frac{1}{2}\sum_{k=1}^{K}\nu_k\mathrm{Tr}(\mathbf{W}_0^{-1}\mathbf{W}_k) \tag{10.74}$$

$$\mathbb{E}[\ln q(\mathbf{Z})] = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}\ln r_{nk} \tag{10.75}$$

$$\mathbb{E}[\ln q(\boldsymbol{\pi})] = \sum_{k=1}^{K}(\alpha_k - 1)\ln\widetilde{\pi}_k + \ln C(\boldsymbol{\alpha}) \tag{10.76}$$

$$\mathbb{E}[\ln q(\boldsymbol{\mu}, \boldsymbol{\Lambda})] = \sum_{k=1}^{K}\left\{\frac{1}{2}\ln\widetilde{\Lambda}_k + \frac{D}{2}\ln\left(\frac{\beta_k}{2\pi}\right) - \frac{D}{2} - \mathrm{H}\left[q(\boldsymbol{\Lambda}_k)\right]\right\} \tag{10.77}$$

where D is the dimensionality of $\mathbf{x}$, $\mathrm{H}[q(\boldsymbol{\Lambda}_k)]$ is the entropy of the Wishart distribution given by (B.82), and the coefficients $C(\boldsymbol{\alpha})$ and $B(\mathbf{W}, \nu)$ are defined by (B.23) and (B.79), respectively. Note that the terms involving expectations of the logs of the q distributions simply represent the negative entropies of those distributions. Some simplifications and combination of terms can be performed when these expressions are summed to give the lower bound. However, we have kept the expressions separate for ease of understanding.

Finally, it is worth noting that the lower bound provides an alternative approach for deriving the variational re-estimation equations obtained in Section 10.2.1. To do this we use the fact that, since the model has conjugate priors, the functional form of the factors in the variational posterior distribution is known, namely discrete for $\mathbf{Z}$, Dirichlet for π, and Gaussian-Wishart for $(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k)$. By taking general parametric forms for these distributions we can derive the form of the lower bound as a function of the parameters of the distributions. Maximizing the bound with respect to these parameters then gives the required re-estimation equations.

Exercise 10.18

10.2.3 Predictive density

In applications of the Bayesian mixture of Gaussians model we will often be interested in the predictive density for a new value $\widehat{\mathbf{x}}$ of the observed variable. Associated with this observation will be a corresponding latent variable $\widehat{\mathbf{z}}$, and the predictive density is then given by

$$p(\widehat{\mathbf{x}}|\mathbf{X}) = \sum_{\widehat{\mathbf{z}}}\iiint p(\widehat{\mathbf{x}}|\widehat{\mathbf{z}}, \boldsymbol{\mu}, \boldsymbol{\Lambda})p(\widehat{\mathbf{z}}|\boldsymbol{\pi})p(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}|\mathbf{X})\,\mathrm{d}\boldsymbol{\pi}\,\mathrm{d}\boldsymbol{\mu}\,\mathrm{d}\boldsymbol{\Lambda} \tag{10.78}$$

where $p(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}|\mathbf{X})$ is the (unknown) true posterior distribution of the parameters. Using (10.37) and (10.38) we can first perform the summation over $\widehat{\mathbf{z}}$ to give

$$p(\widehat{\mathbf{x}}|\mathbf{X}) = \sum_{k=1}^{K} \iiint \pi_k \mathcal{N}\left(\widehat{\mathbf{x}}|\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k^{-1}\right) p(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}|\mathbf{X})\, \mathrm{d}\boldsymbol{\pi}\, \mathrm{d}\boldsymbol{\mu}\, \mathrm{d}\boldsymbol{\Lambda}. \tag{10.79}$$

Because the remaining integrations are intractable, we approximate the predictive density by replacing the true posterior distribution $p(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Lambda}|\mathbf{X})$ with its variational approximation $q(\boldsymbol{\pi})q(\boldsymbol{\mu}, \boldsymbol{\Lambda})$ to give

$$p(\widehat{\mathbf{x}}|\mathbf{X}) = \sum_{k=1}^{K} \iiint \pi_k \mathcal{N}\left(\widehat{\mathbf{x}}|\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k^{-1}\right) q(\boldsymbol{\pi})q(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k)\, \mathrm{d}\boldsymbol{\pi}\, \mathrm{d}\boldsymbol{\mu}_k\, \mathrm{d}\boldsymbol{\Lambda}_k \tag{10.80}$$

where we have made use of the factorization (10.55) and in each term we have implicitly integrated out all variables $\{\boldsymbol{\mu}_j, \boldsymbol{\Lambda}_j\}$ for $j \neq k$ The remaining integrations can now be evaluated analytically giving a mixture of Student's t-distributions

Exercise 10.19

$$p(\widehat{\mathbf{x}}|\mathbf{X}) = \frac{1}{\widehat{\alpha}} \sum_{k=1}^{K} \alpha_k \mathrm{St}(\widehat{\mathbf{x}}|\mathbf{m}_k, \mathbf{L}_k, \nu_k + 1 - D) \tag{10.81}$$

in which the kth component has mean $\mathbf{m}_k$, and the precision is given by

$$\mathbf{L}_k = \frac{(\nu_k + 1 - D)\beta_k}{(1 + \beta_k)} \mathbf{W}_k \tag{10.82}$$

in which ν_k is given by (10.63). When the size N of the data set is large the predictive distribution (10.81) reduces to a mixture of Gaussians.

Exercise 10.20

10.2.4 Determining the number of components

We have seen that the variational lower bound can be used to determine a posterior distribution over the number K of components in the mixture model. There is, however, one subtlety that needs to be addressed. For any given setting of the parameters in a Gaussian mixture model (except for specific degenerate settings), there will exist other parameter settings for which the density over the observed variables will be identical. These parameter values differ only through a re-labelling of the components. For instance, consider a mixture of two Gaussians and a single observed variable x, in which the parameters have the values $\pi_1 = a$, $\pi_2 = b$, $\mu_1 = c$, $\mu_2 = d$, $\sigma_1 = e$, $\sigma_2 = f$. Then the parameter values $\pi_1 = b$, $\pi_2 = a$, $\mu_1 = d$, $\mu_2 = c$, $\sigma_1 = f$, $\sigma_2 = e$, in which the two components have been exchanged, will by symmetry give rise to the same value of $p(x)$. If we have a mixture model comprising K components, then each parameter setting will be a member of a family of $K!$ equivalent settings.

Section 10.1.4

Exercise 10.21

In the context of maximum likelihood, this redundancy is irrelevant because the parameter optimization algorithm (for example EM) will, depending on the initialization of the parameters, find one specific solution, and the other equivalent solutions play no role. In a Bayesian setting, however, we marginalize over all possible

Figure 10.7 Plot of the variational lower bound $\mathcal{L}$ versus the number K of components in the Gaussian mixture model, for the Old Faithful data, showing a distinct peak at $K = 2$ components. For each value of K, the model is trained from 100 different random starts, and the results shown as '$+$' symbols plotted with small random horizontal perturbations so that they can be distinguished. Note that some solutions find suboptimal local maxima, but that this happens infrequently.

parameter values. We have seen in Figure 10.2 that if the true posterior distribution is multimodal, variational inference based on the minimization of $\mathrm{KL}(q\|p)$ will tend to approximate the distribution in the neighbourhood of one of the modes and ignore the others. Again, because equivalent modes have equivalent predictive densities, this is of no concern provided we are considering a model having a specific number K of components. If, however, we wish to compare different values of K, then we need to take account of this multimodality. A simple approximate solution is to add a term $\ln K!$ onto the lower bound when used for model comparison and averaging.

Exercise 10.22

Figure 10.7 shows a plot of the lower bound, including the multimodality factor, versus the number K of components for the Old Faithful data set. It is worth emphasizing once again that maximum likelihood would lead to values of the likelihood function that increase monotonically with K (assuming the singular solutions have been avoided, and discounting the effects of local maxima) and so cannot be used to determine an appropriate model complexity. By contrast, Bayesian inference automatically makes the trade-off between model complexity and fitting the data.

Section 3.4

This approach to the determination of K requires that a range of models having different K values be trained and compared. An alternative approach to determining a suitable value for K is to treat the mixing coefficients $\boldsymbol{\pi}$ as parameters and make point estimates of their values by maximizing the lower bound (Corduneanu and Bishop, 2001) with respect to $\boldsymbol{\pi}$ instead of maintaining a probability distribution over them as in the fully Bayesian approach. This leads to the re-estimation equation

Exercise 10.23

$$\pi_k = \frac{1}{N}\sum_{n=1}^{N} r_{nk} \tag{10.83}$$

and this maximization is interleaved with the variational updates for the q distribution over the remaining parameters. Components that provide insufficient contribution

to explaining the data will have their mixing coefficients driven to zero during the optimization, and so they are effectively removed from the model through *automatic relevance determination*. This allows us to make a single training run in which we start with a relatively large initial value of K, and allow surplus components to be pruned out of the model. The origins of the sparsity when optimizing with respect to hyperparameters is discussed in detail in the context of the relevance vector machine.

Section 7.2.2

10.2.5 Induced factorizations

In deriving these variational update equations for the Gaussian mixture model, we assumed a particular factorization of the variational posterior distribution given by (10.42). However, the optimal solutions for the various factors exhibit additional factorizations. In particular, the solution for $q^\star(\boldsymbol{\mu}, \boldsymbol{\Lambda})$ is given by the product of an independent distribution $q^\star(\boldsymbol{\mu}_k, \boldsymbol{\Lambda}_k)$ over each of the components k of the mixture, whereas the variational posterior distribution $q^\star(\mathbf{Z})$ over the latent variables, given by (10.48), factorizes into an independent distribution $q^\star(\mathbf{z}_n)$ for each observation n (note that it does not further factorize with respect to k because, for each value of n, the z_{nk} are constrained to sum to one over k). These additional factorizations are a consequence of the interaction between the assumed factorization and the conditional independence properties of the true distribution, as characterized by the directed graph in Figure 10.5.

We shall refer to these additional factorizations as *induced factorizations* because they arise from an interaction between the factorization assumed in the variational posterior distribution and the conditional independence properties of the true joint distribution. In a numerical implementation of the variational approach it is important to take account of such additional factorizations. For instance, it would be very inefficient to maintain a full precision matrix for the Gaussian distribution over a set of variables if the optimal form for that distribution always had a diagonal precision matrix (corresponding to a factorization with respect to the individual variables described by that Gaussian).

Such induced factorizations can easily be detected using a simple graphical test based on d-separation as follows. We partition the latent variables into three disjoint groups $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and then let us suppose that we are assuming a factorization between $\mathbf{C}$ and the remaining latent variables, so that

$$q(\mathbf{A}, \mathbf{B}, \mathbf{C}) = q(\mathbf{A}, \mathbf{B})q(\mathbf{C}). \tag{10.84}$$

Using the general result (10.9), together with the product rule for probabilities, we see that the optimal solution for $q(\mathbf{A}, \mathbf{B})$ is given by

$$\begin{aligned} \ln q^\star(\mathbf{A}, \mathbf{B}) &= \mathbb{E}_{\mathbf{C}}[\ln p(\mathbf{X}, \mathbf{A}, \mathbf{B}, \mathbf{C})] + \text{const} \\ &= \mathbb{E}_{\mathbf{C}}[\ln p(\mathbf{A}, \mathbf{B}|\mathbf{X}, \mathbf{C})] + \text{const}. \end{aligned} \tag{10.85}$$

We now ask whether this resulting solution will factorize between $\mathbf{A}$ and $\mathbf{B}$, in other words whether $q^\star(\mathbf{A}, \mathbf{B}) = q^\star(\mathbf{A})q^\star(\mathbf{B})$. This will happen if, and only if, $\ln p(\mathbf{A}, \mathbf{B}|\mathbf{X}, \mathbf{C}) = \ln p(\mathbf{A}|\mathbf{X}, \mathbf{C}) + \ln p(\mathbf{B}|\mathbf{X}, \mathbf{C})$, that is, if the conditional independence relation

$$\mathbf{A} \perp\!\!\!\perp \mathbf{B} \mid \mathbf{X}, \mathbf{C} \tag{10.86}$$

is satisfied. We can test to see if this relation does hold, for any choice of $\mathbf{A}$ and $\mathbf{B}$ by making use of the d-separation criterion.

To illustrate this, consider again the Bayesian mixture of Gaussians represented by the directed graph in Figure 10.5, in which we are assuming a variational factorization given by (10.42). We can see immediately that the variational posterior distribution over the parameters must factorize between $\boldsymbol{\pi}$ and the remaining parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Lambda}$ because all paths connecting $\boldsymbol{\pi}$ to either $\boldsymbol{\mu}$ or $\boldsymbol{\Lambda}$ must pass through one of the nodes $\mathbf{z}_n$ all of which are in the conditioning set for our conditional independence test and all of which are head-to-tail with respect to such paths.

10.3. Variational Linear Regression

As a second illustration of variational inference, we return to the Bayesian linear regression model of Section 3.3. In the evidence framework, we approximated the integration over α and β by making point estimates obtained by maximizing the log marginal likelihood. A fully Bayesian approach would integrate over the hyperparameters as well as over the parameters. Although exact integration is intractable, we can use variational methods to find a tractable approximation. In order to simplify the discussion, we shall suppose that the noise precision parameter β is known, and is fixed to its true value, although the framework is easily extended to include the distribution over β. For the linear regression model, the variational treatment will turn out to be equivalent to the evidence framework. Nevertheless, it provides a good exercise in the use of variational methods and will also lay the foundation for variational treatment of Bayesian logistic regression in Section 10.6.

Exercise 10.26

Recall that the likelihood function for $\mathbf{w}$, and the prior over $\mathbf{w}$, are given by

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N} \mathcal{N}(t_n|\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n, \beta^{-1}) \tag{10.87}$$

$$p(\mathbf{w}|\alpha) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I}) \tag{10.88}$$

where $\boldsymbol{\phi}_n = \boldsymbol{\phi}(\mathbf{x}_n)$. We now introduce a prior distribution over α. From our discussion in Section 2.3.6, we know that the conjugate prior for the precision of a Gaussian is given by a gamma distribution, and so we choose

$$p(\alpha) = \mathrm{Gam}(\alpha|a_0, b_0) \tag{10.89}$$

where $\mathrm{Gam}(\cdot|\cdot,\cdot)$ is defined by (B.26). Thus the joint distribution of all the variables is given by

$$p(\mathbf{t}, \mathbf{w}, \alpha) = p(\mathbf{t}|\mathbf{w})p(\mathbf{w}|\alpha)p(\alpha). \tag{10.90}$$

This can be represented as a directed graphical model as shown in Figure 10.8.

10.3.1 Variational distribution

Our first goal is to find an approximation to the posterior distribution $p(\mathbf{w}, \alpha|\mathbf{t})$. To do this, we employ the variational framework of Section 10.1, with a variational

Figure 10.8 Probabilistic graphical model representing the joint distribution (10.90) for the Bayesian linear regression model.

posterior distribution given by the factorized expression

$$q(\mathbf{w}, \alpha) = q(\mathbf{w})q(\alpha). \tag{10.91}$$

We can find re-estimation equations for the factors in this distribution by making use of the general result (10.9). Recall that for each factor, we take the log of the joint distribution over all variables and then average with respect to those variables not in that factor. Consider first the distribution over α. Keeping only terms that have a functional dependence on α, we have

$$\begin{aligned} \ln q^\star(\alpha) &= \ln p(\alpha) + \mathbb{E}_{\mathbf{w}}\left[\ln p(\mathbf{w}|\alpha)\right] + \text{const} \\ &= (a_0 - 1)\ln\alpha - b_0\alpha + \frac{M}{2}\ln\alpha - \frac{\alpha}{2}\mathbb{E}[\mathbf{w}^{\mathrm{T}}\mathbf{w}] + \text{const}. \end{aligned} \tag{10.92}$$

We recognize this as the log of a gamma distribution, and so identifying the coefficients of α and $\ln\alpha$ we obtain

$$q^\star(\alpha) = \mathrm{Gam}(\alpha|a_N, b_N) \tag{10.93}$$

where

$$a_N = a_0 + \frac{M}{2} \tag{10.94}$$

$$b_N = b_0 + \frac{1}{2}\mathbb{E}[\mathbf{w}^{\mathrm{T}}\mathbf{w}]. \tag{10.95}$$

Similarly, we can find the variational re-estimation equation for the posterior distribution over $\mathbf{w}$. Again, using the general result (10.9), and keeping only those terms that have a functional dependence on $\mathbf{w}$, we have

$$\ln q^\star(\mathbf{w}) = \ln p(\mathbf{t}|\mathbf{w}) + \mathbb{E}_\alpha\left[\ln p(\mathbf{w}|\alpha)\right] + \text{const} \tag{10.96}$$

$$= -\frac{\beta}{2}\sum_{n=1}^{N}\{\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n - t_n\}^2 - \frac{1}{2}\mathbb{E}[\alpha]\mathbf{w}^{\mathrm{T}}\mathbf{w} + \text{const} \tag{10.97}$$

$$= -\frac{1}{2}\mathbf{w}^{\mathrm{T}}\left(\mathbb{E}[\alpha]\mathbf{I} + \beta\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)\mathbf{w} + \beta\mathbf{w}^{\mathrm{T}}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t} + \text{const}. \tag{10.98}$$

Because this is a quadratic form, the distribution $q^\star(\mathbf{w})$ is Gaussian, and so we can complete the square in the usual way to identify the mean and covariance, giving

$$q^\star(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{m}_N, \mathbf{S}_N) \tag{10.99}$$

where

$$\mathbf{m}_N = \beta \mathbf{S}_N \boldsymbol{\Phi}^{\mathrm{T}} \mathbf{t} \tag{10.100}$$

$$\mathbf{S}_N = \left(\mathbb{E}[\alpha]\mathbf{I} + \beta \boldsymbol{\Phi}^{\mathrm{T}} \boldsymbol{\Phi}\right)^{-1}. \tag{10.101}$$

Note the close similarity to the posterior distribution (3.52) obtained when α was treated as a fixed parameter. The difference is that here α is replaced by its expectation $\mathbb{E}[\alpha]$ under the variational distribution. Indeed, we have chosen to use the same notation for the covariance matrix $\mathbf{S}_N$ in both cases.

Using the standard results (B.27), (B.38), and (B.39), we can obtain the required moments as follows

$$\mathbb{E}[\alpha] = a_N / b_N \tag{10.102}$$

$$\mathbb{E}[\mathbf{w}\mathbf{w}^{\mathrm{T}}] = \mathbf{m}_N \mathbf{m}_N^{\mathrm{T}} + \mathbf{S}_N. \tag{10.103}$$

The evaluation of the variational posterior distribution begins by initializing the parameters of one of the distributions $q(\mathbf{w})$ or $q(\alpha)$, and then alternately re-estimates these factors in turn until a suitable convergence criterion is satisfied (usually specified in terms of the lower bound to be discussed shortly).

It is instructive to relate the variational solution to that found using the evidence framework in Section 3.5. To do this consider the case $a_0 = b_0 = 0$, corresponding to the limit of an infinitely broad prior over α. The mean of the variational posterior distribution $q(\alpha)$ is then given by

$$\mathbb{E}[\alpha] = \frac{a_N}{b_N} = \frac{M/2}{\mathbb{E}[\mathbf{w}^{\mathrm{T}}\mathbf{w}]/2} = \frac{M}{\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N + \mathrm{Tr}(\mathbf{S}_N)}. \tag{10.104}$$

Comparison with (9.63) shows that in the case of this particularly simple model, the variational approach gives precisely the same expression as that obtained by maximizing the evidence function using EM except that the point estimate for α is replaced by its expected value. Because the distribution $q(\mathbf{w})$ depends on $q(\alpha)$ only through the expectation $\mathbb{E}[\alpha]$, we see that the two approaches will give identical results for the case of an infinitely broad prior.

10.3.2 Predictive distribution

The predictive distribution over t, given a new input $\mathbf{x}$, is easily evaluated for this model using the Gaussian variational posterior for the parameters

$$\begin{aligned} p(t|\mathbf{x}, \mathbf{t}) &= \int p(t|\mathbf{x}, \mathbf{w}) p(\mathbf{w}|\mathbf{t}) \, \mathrm{d}\mathbf{w} \\ &\simeq \int p(t|\mathbf{x}, \mathbf{w}) q(\mathbf{w}) \, \mathrm{d}\mathbf{w} \\ &= \int \mathcal{N}(t|\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}), \beta^{-1}) \mathcal{N}(\mathbf{w}|\mathbf{m}_N, \mathbf{S}_N) \, \mathrm{d}\mathbf{w} \\ &= \mathcal{N}(t|\mathbf{m}_N^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}), \sigma^2(\mathbf{x})) \end{aligned} \tag{10.105}$$

where we have evaluated the integral by making use of the result (2.115) for the linear-Gaussian model. Here the input-dependent variance is given by

$$\sigma^2(\mathbf{x}) = \frac{1}{\beta} + \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}}\mathbf{S}_N\boldsymbol{\phi}(\mathbf{x}). \tag{10.106}$$

Note that this takes the same form as the result (3.59) obtained with fixed α except that now the expected value $\mathbb{E}[\alpha]$ appears in the definition of $\mathbf{S}_N$.

10.3.3 Lower bound

Another quantity of importance is the lower bound $\mathcal{L}$ defined by

$$\begin{aligned}\mathcal{L}(q) &= \mathbb{E}[\ln p(\mathbf{w},\alpha,\mathbf{t})] - \mathbb{E}[\ln q(\mathbf{w},\alpha)] \\ &= \mathbb{E}_{\mathbf{w}}[\ln p(\mathbf{t}|\mathbf{w})] + \mathbb{E}_{\mathbf{w},\alpha}[\ln p(\mathbf{w}|\alpha)] + \mathbb{E}_{\alpha}[\ln p(\alpha)] \\ &\quad -\mathbb{E}_{\alpha}[\ln q(\mathbf{w})]_{\mathbf{w}} - \mathbb{E}[\ln q(\alpha)]. \end{aligned} \tag{10.107}$$

Exercise 10.27

Evaluation of the various terms is straightforward, making use of results obtained in previous chapters, and gives

$$\begin{aligned}\mathbb{E}[\ln p(\mathbf{t}|\mathbf{w})]_{\mathbf{w}} &= \frac{N}{2}\ln\left(\frac{\beta}{2\pi}\right) - \frac{\beta}{2}\mathbf{t}^{\mathrm{T}}\mathbf{t} + \beta\mathbf{m}_N^{\mathrm{T}}\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{t} \\ &\quad -\frac{\beta}{2}\mathrm{Tr}\left[\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}(\mathbf{m}_N\mathbf{m}_N^{\mathrm{T}} + \mathbf{S}_N)\right] && (10.108)\\ \mathbb{E}[\ln p(\mathbf{w}|\alpha)]_{\mathbf{w},\alpha} &= -\frac{M}{2}\ln(2\pi) + \frac{M}{2}(\psi(a_N) - \ln b_N) \\ &\quad -\frac{a_N}{2b_N}\left[\mathbf{m}_N^{\mathrm{T}}\mathbf{m}_N + \mathrm{Tr}(\mathbf{S}_N)\right] && (10.109)\\ \mathbb{E}[\ln p(\alpha)]_{\alpha} &= a_0\ln b_0 + (a_0 - 1)\left[\psi(a_N) - \ln b_N\right] \\ &\quad -b_0\frac{a_N}{b_N} - \ln\Gamma(a_N) && (10.110)\\ -\mathbb{E}[\ln q(\mathbf{w})]_{\mathbf{w}} &= \frac{1}{2}\ln|\mathbf{S}_N| + \frac{M}{2}\left[1 + \ln(2\pi)\right] && (10.111)\\ -\mathbb{E}[\ln q(\alpha)]_{\alpha} &= \ln\Gamma(a_N) - (a_N - 1)\psi(a_N) - \ln b_N + a_N. && (10.112)\end{aligned}$$

Figure 10.9 shows a plot of the lower bound $\mathcal{L}(q)$ versus the degree of a polynomial model for a synthetic data set generated from a degree three polynomial. Here the prior parameters have been set to $a_0 = b_0 = 0$, corresponding to the noninformative prior $p(\alpha) \propto 1/\alpha$, which is uniform over $\ln\alpha$ as discussed in Section 2.3.6. As we saw in Section 10.1, the quantity $\mathcal{L}$ represents lower bound on the log marginal likelihood $p(\mathbf{t}|M)$ for the model. If we assign equal prior probabilities $p(M)$ to the different values of M, then we can interpret $\mathcal{L}$ as an approximation to the posterior model probability $p(M|\mathbf{t})$. Thus the variational framework assigns the highest probability to the model with $M = 3$. This should be contrasted with the maximum likelihood result, which assigns ever smaller residual error to models of increasing complexity until the residual error is driven to zero, causing maximum likelihood to favour severely over-fitted models.

Figure 10.9 Plot of the lower bound $\mathcal{L}$ versus the order M of the polynomial, for a polynomial model, in which a set of 10 data points is generated from a polynomial with $M = 3$ sampled over the interval $(-5, 5)$ with additive Gaussian noise of variance 0.09. The value of the bound gives the log probability of the model, and we see that the value of the bound peaks at $M = 3$, corresponding to the true model from which the data set was generated.

10.4. Exponential Family Distributions

In Chapter 2, we discussed the important role played by the exponential family of distributions and their conjugate priors. For many of the models discussed in this book, the complete-data likelihood is drawn from the exponential family. However, in general this will not be the case for the marginal likelihood function for the observed data. For example, in a mixture of Gaussians, the joint distribution of observations $\mathbf{x}_n$ and corresponding hidden variables $\mathbf{z}_n$ is a member of the exponential family, whereas the marginal distribution of $\mathbf{x}_n$ is a mixture of Gaussians and hence is not.

Up to now we have grouped the variables in the model into observed variables and hidden variables. We now make a further distinction between latent variables, denoted $\mathbf{Z}$, and parameters, denoted $\boldsymbol{\theta}$, where parameters are *intensive* (fixed in number independent of the size of the data set), whereas latent variables are *extensive* (scale in number with the size of the data set). For example, in a Gaussian mixture model, the indicator variables z_{kn} (which specify which component k is responsible for generating data point $\mathbf{x}_n$) represent the latent variables, whereas the means $\boldsymbol{\mu}_k$, precisions $\boldsymbol{\Lambda}_k$ and mixing proportions π_k represent the parameters.

Consider the case of independent identically distributed data. We denote the data values by $\mathbf{X} = \{\mathbf{x}_n\}$, where $n = 1, \ldots N$, with corresponding latent variables $\mathbf{Z} = \{\mathbf{z}_n\}$. Now suppose that the joint distribution of observed and latent variables is a member of the exponential family, parameterized by natural parameters $\boldsymbol{\eta}$ so that

$$p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\eta}) = \prod_{n=1}^{N} h(\mathbf{x}_n, \mathbf{z}_n) g(\boldsymbol{\eta}) \exp\left\{\boldsymbol{\eta}^{\mathrm{T}} \mathbf{u}(\mathbf{x}_n, \mathbf{z}_n)\right\}. \tag{10.113}$$

We shall also use a conjugate prior for $\boldsymbol{\eta}$, which can be written as

$$p(\boldsymbol{\eta}|\nu_0, \mathbf{v}_0) = f(\nu_0, \boldsymbol{\chi}_0) g(\boldsymbol{\eta})^{\nu_0} \exp\left\{\nu_o \boldsymbol{\eta}^{\mathrm{T}} \boldsymbol{\chi}_0\right\}. \tag{10.114}$$

Recall that the conjugate prior distribution can be interpreted as a prior number ν_0 of observations all having the value $\boldsymbol{\chi}_0$ for the $\mathbf{u}$ vector. Now consider a variational

distribution that factorizes between the latent variables and the parameters, so that $q(\mathbf{Z}, \boldsymbol{\eta}) = q(\mathbf{Z})q(\boldsymbol{\eta})$. Using the general result (10.9), we can solve for the two factors as follows

$$\begin{aligned} \ln q^\star(\mathbf{Z}) &= \mathbb{E}_{\boldsymbol{\eta}}[\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\eta})] + \text{const} \\ &= \sum_{n=1}^{N} \left\{ \ln h(\mathbf{x}_n, \mathbf{z}_n) + \mathbb{E}[\boldsymbol{\eta}^{\mathrm{T}}]\mathbf{u}(\mathbf{x}_n, \mathbf{z}_n) \right\} + \text{const}. \end{aligned} \tag{10.115}$$

Thus we see that this decomposes into a sum of independent terms, one for each value of n, and hence the solution for $q^\star(\mathbf{Z})$ will factorize over n so that $q^\star(\mathbf{Z}) = \prod_n q^\star(\mathbf{z}_n)$. This is an example of an induced factorization. Taking the exponential of both sides, we have

Section 10.2.5

$$q^\star(\mathbf{z}_n) = h(\mathbf{x}_n, \mathbf{z}_n) g\left(\mathbb{E}[\boldsymbol{\eta}]\right) \exp\left\{ \mathbb{E}[\boldsymbol{\eta}^{\mathrm{T}}]\mathbf{u}(\mathbf{x}_n, \mathbf{z}_n) \right\} \tag{10.116}$$

where the normalization coefficient has been re-instated by comparison with the standard form for the exponential family.

Similarly, for the variational distribution over the parameters, we have

$$\ln q^\star(\boldsymbol{\eta}) = \ln p(\boldsymbol{\eta}|\nu_0, \boldsymbol{\chi}_0) + \mathbb{E}_{\mathbf{Z}}[\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\eta})] + \text{const} \tag{10.117}$$

$$= \nu_0 \ln g(\boldsymbol{\eta}) + \boldsymbol{\eta}^{\mathrm{T}}\boldsymbol{\chi}_0 + \sum_{n=1}^{N} \left\{ \ln g(\boldsymbol{\eta}) + \boldsymbol{\eta}^{\mathrm{T}}\mathbb{E}_{\mathbf{z}_n}[\mathbf{u}(\mathbf{x}_n, \mathbf{z}_n)] \right\} + \text{const}. \tag{10.118}$$

Again, taking the exponential of both sides, and re-instating the normalization coefficient by inspection, we have

$$q^\star(\boldsymbol{\eta}) = f(\nu_N, \boldsymbol{\chi}_N) g(\boldsymbol{\eta})^{\nu_N} \exp\left\{ \boldsymbol{\eta}^{\mathrm{T}}\boldsymbol{\chi}_N \right\} \tag{10.119}$$

where we have defined

$$\nu_N = \nu_0 + N \tag{10.120}$$

$$\boldsymbol{\chi}_N = \boldsymbol{\chi}_0 + \sum_{n=1}^{N} \mathbb{E}_{\mathbf{z}_n}[\mathbf{u}(\mathbf{x}_n, \mathbf{z}_n)]. \tag{10.121}$$

Note that the solutions for $q^\star(\mathbf{z}_n)$ and $q^\star(\boldsymbol{\eta})$ are coupled, and so we solve them iteratively in a two-stage procedure. In the variational E step, we evaluate the expected sufficient statistics $\mathbb{E}[\mathbf{u}(\mathbf{x}_n, \mathbf{z}_n)]$ using the current posterior distribution $q(\mathbf{z}_n)$ over the latent variables and use this to compute a revised posterior distribution $q(\boldsymbol{\eta})$ over the parameters. Then in the subsequent variational M step, we use this revised parameter posterior distribution to find the expected natural parameters $\mathbb{E}[\boldsymbol{\eta}^{\mathrm{T}}]$, which gives rise to a revised variational distribution over the latent variables.

10.4.1 Variational message passing

We have illustrated the application of variational methods by considering a specific model, the Bayesian mixture of Gaussians, in some detail. This model can be

described by the directed graph shown in Figure 10.5. Here we consider more generally the use of variational methods for models described by directed graphs and derive a number of widely applicable results.

The joint distribution corresponding to a directed graph can be written using the decomposition

$$p(\mathbf{x}) = \prod_i p(\mathbf{x}_i|\mathrm{pa}_i) \tag{10.122}$$

where $\mathbf{x}_i$ denotes the variable(s) associated with node i, and pa_i denotes the parent set corresponding to node i. Note that $\mathbf{x}_i$ may be a latent variable or it may belong to the set of observed variables. Now consider a variational approximation in which the distribution $q(\mathbf{x})$ is assumed to factorize with respect to the $\mathbf{x}_i$ so that

$$q(\mathbf{x}) = \prod_i q_i(\mathbf{x}_i). \tag{10.123}$$

Note that for observed nodes, there is no factor $q(\mathbf{x}_i)$ in the variational distribution. We now substitute (10.122) into our general result (10.9) to give

$$\ln q_j^\star(\mathbf{x}_j) = \mathbb{E}_{i \neq j}\left[\sum_i \ln p(\mathbf{x}_i|\mathrm{pa}_i)\right] + \mathrm{const}. \tag{10.124}$$

Any terms on the right-hand side that do not depend on $\mathbf{x}_j$ can be absorbed into the additive constant. In fact, the only terms that do depend on $\mathbf{x}_j$ are the conditional distribution for $\mathbf{x}_j$ given by $p(\mathbf{x}_j|\mathrm{pa}_j)$ together with any other conditional distributions that have $\mathbf{x}_j$ in the conditioning set. By definition, these conditional distributions correspond to the children of node j, and they therefore also depend on the *co-parents* of the child nodes, i.e., the other parents of the child nodes besides node $\mathbf{x}_j$ itself. We see that the set of all nodes on which $q^\star(\mathbf{x}_j)$ depends corresponds to the Markov blanket of node $\mathbf{x}_j$, as illustrated in Figure 8.26. Thus the update of the factors in the variational posterior distribution represents a local calculation on the graph. This makes possible the construction of general purpose software for variational inference in which the form of the model does not need to be specified in advance (Bishop *et al.*, 2003).

If we now specialize to the case of a model in which all of the conditional distributions have a conjugate-exponential structure, then the variational update procedure can be cast in terms of a local message passing algorithm (Winn and Bishop, 2005). In particular, the distribution associated with a particular node can be updated once that node has received messages from all of its parents and all of its children. This in turn requires that the children have already received messages from their co-parents. The evaluation of the lower bound can also be simplified because many of the required quantities are already evaluated as part of the message passing scheme. This distributed message passing formulation has good scaling properties and is well suited to large networks.

10.5. Local Variational Methods

The variational framework discussed in Sections 10.1 and 10.2 can be considered a 'global' method in the sense that it directly seeks an approximation to the full posterior distribution over all random variables. An alternative 'local' approach involves finding bounds on functions over individual variables or groups of variables within a model. For instance, we might seek a bound on a conditional distribution $p(y|x)$, which is itself just one factor in a much larger probabilistic model specified by a directed graph. The purpose of introducing the bound of course is to simplify the resulting distribution. This local approximation can be applied to multiple variables in turn until a tractable approximation is obtained, and in Section 10.6.1 we shall give a practical example of this approach in the context of logistic regression. Here we focus on developing the bounds themselves.

We have already seen in our discussion of the Kullback-Leibler divergence that the convexity of the logarithm function played a key role in developing the lower bound in the global variational approach. We have defined a (strictly) convex function as one for which every chord lies above the function. Convexity also plays a central role in the local variational framework. Note that our discussion will apply equally to concave functions with 'min' and 'max' interchanged and with lower bounds replaced by upper bounds.

Section 1.6.1

Let us begin by considering a simple example, namely the function $f(x) = \exp(-x)$, which is a convex function of x, and which is shown in the left-hand plot of Figure 10.10. Our goal is to approximate $f(x)$ by a simpler function, in particular a linear function of x. From Figure 10.10, we see that this linear function will be a lower bound on $f(x)$ if it corresponds to a tangent. We can obtain the tangent line $y(x)$ at a specific value of x, say $x = \xi$, by making a first order Taylor expansion

$$y(x) = f(\xi) + f'(\xi)(x - \xi) \tag{10.125}$$

so that $y(x) \leqslant f(x)$ with equality when $x = \xi$. For our example function $f(x) =$

Figure 10.10 In the left-hand figure the red curve shows the function $\exp(-x)$, and the blue line shows the tangent at $x = \xi$ defined by (10.125) with $\xi = 1$. This line has slope $\lambda = f'(\xi) = -\exp(-\xi)$. Note that any other tangent line, for example the ones shown in green, will have a smaller value of y at $x = \xi$. The right-hand figure shows the corresponding plot of the function $\lambda\xi - g(\lambda)$, where $g(\lambda)$ is given by (10.131), versus λ for $\xi = 1$, in which the maximum corresponds to $\lambda = -\exp(-\xi) = -1/e$.

Figure 10.11 In the left-hand plot the red curve shows a convex function $f(x)$, and the blue line represents the linear function λx, which is a lower bound on $f(x)$ because $f(x) > \lambda x$ for all x. For the given value of slope λ the contact point of the tangent line having the same slope is found by minimizing with respect to x the discrepancy (shown by the green dashed lines) given by $f(x) - \lambda x$. This defines the dual function $g(\lambda)$, which corresponds to the (negative of the) intercept of the tangent line having slope λ.

$\exp(-x)$, we therefore obtain the tangent line in the form

$$y(x) = \exp(-\xi) - \exp(-\xi)(x - \xi) \tag{10.126}$$

which is a linear function parameterized by ξ. For consistency with subsequent discussion, let us define $\lambda = -\exp(-\xi)$ so that

$$y(x, \lambda) = \lambda x - \lambda + \lambda \ln(-\lambda). \tag{10.127}$$

Different values of λ correspond to different tangent lines, and because all such lines are lower bounds on the function, we have $f(x) \geqslant y(x, \lambda)$. Thus we can write the function in the form

$$f(x) = \max_{\lambda} \{\lambda x - \lambda + \lambda \ln(-\lambda)\}. \tag{10.128}$$

We have succeeded in approximating the convex function $f(x)$ by a simpler, linear function $y(x, \lambda)$. The price we have paid is that we have introduced a variational parameter λ, and to obtain the tightest bound we must optimize with respect to λ.

We can formulate this approach more generally using the framework of *convex duality* (Rockafellar, 1972; Jordan *et al.*, 1999). Consider the illustration of a convex function $f(x)$ shown in the left-hand plot in Figure 10.11. In this example, the function λx is a lower bound on $f(x)$ but it is not the best lower bound that can be achieved by a linear function having slope λ, because the tightest bound is given by the tangent line. Let us write the equation of the tangent line, having slope λ as $\lambda x - g(\lambda)$ where the (negative) intercept $g(\lambda)$ clearly depends on the slope λ of the tangent. To determine the intercept, we note that the line must be moved vertically by an amount equal to the smallest vertical distance between the line and the function, as shown in Figure 10.11. Thus

$$\begin{aligned} g(\lambda) &= -\min_{x} \{f(x) - \lambda x\} \\ &= \max_{x} \{\lambda x - f(x)\}. \end{aligned} \tag{10.129}$$

Now, instead of fixing λ and varying x, we can consider a particular x and then adjust λ until the tangent plane is tangent at that particular x. Because the y value of the tangent line at a particular x is maximized when that value coincides with its contact point, we have

$$f(x) = \max_{\lambda} \{\lambda x - g(\lambda)\}. \tag{10.130}$$

We see that the functions $f(x)$ and $g(\lambda)$ play a dual role, and are related through (10.129) and (10.130).

Let us apply these duality relations to our simple example $f(x) = \exp(-x)$. From (10.129) we see that the maximizing value of x is given by $\xi = -\ln(-\lambda)$, and back-substituting we obtain the conjugate function $g(\lambda)$ in the form

$$g(\lambda) = \lambda - \lambda \ln(-\lambda) \tag{10.131}$$

as obtained previously. The function $\lambda\xi - g(\lambda)$ is shown, for $\xi = 1$ in the right-hand plot in Figure 10.10. As a check, we can substitute (10.131) into (10.130), which gives the maximizing value of $\lambda = -\exp(-x)$, and back-substituting then recovers the original function $f(x) = \exp(-x)$.

For concave functions, we can follow a similar argument to obtain upper bounds, in which $\max$' is replaced with '$\min$', so that

$$f(x) = \min_{\lambda} \{\lambda x - g(\lambda)\} \tag{10.132}$$

$$g(\lambda) = \min_{x} \{\lambda x - f(x)\}. \tag{10.133}$$

If the function of interest is not convex (or concave), then we cannot directly apply the method above to obtain a bound. However, we can first seek invertible transformations either of the function or of its argument which change it into a convex form. We then calculate the conjugate function and then transform back to the original variables.

An important example, which arises frequently in pattern recognition, is the logistic sigmoid function defined by

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{10.134}$$

As it stands this function is neither convex nor concave. However, if we take the logarithm we obtain a function which is concave, as is easily verified by finding the second derivative. From (10.133) the corresponding conjugate function then takes the form

Exercise 10.30

$$g(\lambda) = \min_{x} \{\lambda x - f(x)\} = -\lambda \ln \lambda - (1 - \lambda) \ln(1 - \lambda) \tag{10.135}$$

which we recognize as the binary entropy function for a variable whose probability of having the value 1 is λ. Using (10.132), we then obtain an upper bound on the log sigmoid

Appendix B

$$\ln \sigma(x) \leqslant \lambda x - g(\lambda) \tag{10.136}$$

Figure 10.12 The left-hand plot shows the logistic sigmoid function $\sigma(x)$ defined by (10.134) in red, together with two examples of the exponential upper bound (10.137) shown in blue. The right-hand plot shows the logistic sigmoid again in red together with the Gaussian lower bound (10.144) shown in blue. Here the parameter $\xi = 2.5$, and the bound is exact at $x = \xi$ and $x = -\xi$, denoted by the dashed green lines.

and taking the exponential, we obtain an upper bound on the logistic sigmoid itself of the form

$$\sigma(x) \leqslant \exp(\lambda x - g(\lambda)) \tag{10.137}$$

which is plotted for two values of λ on the left-hand plot in Figure 10.12.

We can also obtain a lower bound on the sigmoid having the functional form of a Gaussian. To do this, we follow Jaakkola and Jordan (2000) and make transformations both of the input variable and of the function itself. First we take the log of the logistic function and then decompose it so that

$$\begin{aligned} \ln \sigma(x) &= -\ln(1+e^{-x}) = -\ln\left\{e^{-x/2}(e^{x/2}+e^{-x/2})\right\} \\ &= x/2 - \ln(e^{x/2}+e^{-x/2}). \end{aligned} \tag{10.138}$$

Exercise 10.31

We now note that the function $f(x) = -\ln(e^{x/2}+e^{-x/2})$ is a convex function of the variable x^2, as can again be verified by finding the second derivative. This leads to a lower bound on $f(x)$, which is a linear function of x^2 whose conjugate function is given by

$$g(\lambda) = \max_{x^2}\left\{\lambda x^2 - f\left(\sqrt{x^2}\right)\right\}. \tag{10.139}$$

The stationarity condition leads to

$$0 = \lambda - \frac{dx}{dx^2}\frac{d}{dx}f(x) = \lambda + \frac{1}{4x}\tanh\left(\frac{x}{2}\right). \tag{10.140}$$

If we denote this value of x, corresponding to the contact point of the tangent line for this particular value of λ, by ξ, then we have

$$\lambda(\xi) = -\frac{1}{4\xi}\tanh\left(\frac{\xi}{2}\right) = -\frac{1}{2\xi}\left[\sigma(\xi) - \frac{1}{2}\right]. \tag{10.141}$$

Instead of thinking of λ as the variational parameter, we can let ξ play this role as this leads to simpler expressions for the conjugate function, which is then given by

$$g(\lambda) = \lambda(\xi)\xi^2 - f(\xi) = \lambda(\xi)\xi^2 + \ln(e^{\xi/2} + e^{-\xi/2}). \tag{10.142}$$

Hence the bound on $f(x)$ can be written as

$$f(x) \geqslant \lambda x^2 - g(\lambda) = \lambda x^2 - \lambda\xi^2 - \ln(e^{\xi/2} + e^{-\xi/2}). \tag{10.143}$$

The bound on the sigmoid then becomes

$$\sigma(x) \geqslant \sigma(\xi) \exp\left\{(x-\xi)/2 - \lambda(\xi)(x^2 - \xi^2)\right\} \tag{10.144}$$

where $\lambda(\xi)$ is defined by (10.141). This bound is illustrated in the right-hand plot of Figure 10.12. We see that the bound has the form of the exponential of a quadratic function of x, which will prove useful when we seek Gaussian representations of posterior distributions defined through logistic sigmoid functions.

Section 4.5

The logistic sigmoid arises frequently in probabilistic models over binary variables because it is the function that transforms a log odds ratio into a posterior probability. The corresponding transformation for a multiclass distribution is given by the softmax function. Unfortunately, the lower bound derived here for the logistic sigmoid does not directly extend to the softmax. Gibbs (1997) proposes a method for constructing a Gaussian distribution that is conjectured to be a bound (although no rigorous proof is given), which may be used to apply local variational methods to multiclass problems.

Section 4.3

We shall see an example of the use of local variational bounds in Sections 10.6.1. For the moment, however, it is instructive to consider in general terms how these bounds can be used. Suppose we wish to evaluate an integral of the form

$$I = \int \sigma(a)p(a)\,\mathrm{d}a \tag{10.145}$$

where $\sigma(a)$ is the logistic sigmoid, and $p(a)$ is a Gaussian probability density. Such integrals arise in Bayesian models when, for instance, we wish to evaluate the predictive distribution, in which case $p(a)$ represents a posterior parameter distribution. Because the integral is intractable, we employ the variational bound (10.144), which we write in the form $\sigma(a) \geqslant f(a,\xi)$ where ξ is a variational parameter. The integral now becomes the product of two exponential-quadratic functions and so can be integrated analytically to give a bound on I

$$I \geqslant \int f(a,\xi)p(a)\,\mathrm{d}a = F(\xi). \tag{10.146}$$

We now have the freedom to choose the variational parameter ξ, which we do by finding the value $\xi^\star$ that maximizes the function $F(\xi)$. The resulting value $F(\xi^\star)$ represents the tightest bound within this family of bounds and can be used as an approximation to I. This optimized bound, however, will in general not be exact.

Although the bound $\sigma(a) \geqslant f(a, \xi)$ on the logistic sigmoid can be optimized exactly, the required choice for ξ depends on the value of a, so that the bound is exact for one value of a only. Because the quantity $F(\xi)$ is obtained by integrating over all values of a, the value of $\xi^\star$ represents a compromise, weighted by the distribution $p(a)$.

10.6. Variational Logistic Regression

We now illustrate the use of local variational methods by returning to the Bayesian logistic regression model studied in Section 4.5. There we focussed on the use of the Laplace approximation, while here we consider a variational treatment based on the approach of Jaakkola and Jordan (2000). Like the Laplace method, this also leads to a Gaussian approximation to the posterior distribution. However, the greater flexibility of the variational approximation leads to improved accuracy compared to the Laplace method. Furthermore (unlike the Laplace method), the variational approach is optimizing a well defined objective function given by a rigourous bound on the model evidence. Logistic regression has also been treated by Dybowski and Roberts (2005) from a Bayesian perspective using Monte Carlo sampling techniques.

10.6.1 Variational posterior distribution

Here we shall make use of a variational approximation based on the local bounds introduced in Section 10.5. This allows the likelihood function for logistic regression, which is governed by the logistic sigmoid, to be approximated by the exponential of a quadratic form. It is therefore again convenient to choose a conjugate Gaussian prior of the form (4.140). For the moment, we shall treat the hyperparameters $\mathbf{m}_0$ and $\mathbf{S}_0$ as fixed constants. In Section 10.6.3, we shall demonstrate how the variational formalism can be extended to the case where there are unknown hyperparameters whose values are to be inferred from the data.

In the variational framework, we seek to maximize a lower bound on the marginal likelihood. For the Bayesian logistic regression model, the marginal likelihood takes the form

$$p(\mathbf{t}) = \int p(\mathbf{t}|\mathbf{w})p(\mathbf{w})\,\mathrm{d}\mathbf{w} = \int \left[\prod_{n=1}^{N} p(t_n|\mathbf{w})\right] p(\mathbf{w})\,\mathrm{d}\mathbf{w}. \tag{10.147}$$

We first note that the conditional distribution for t can be written as

$$\begin{aligned} p(t|\mathbf{w}) &= \sigma(a)^t \left\{1-\sigma(a)\right\}^{1-t} \\ &= \left(\frac{1}{1+e^{-a}}\right)^t \left(1-\frac{1}{1+e^{-a}}\right)^{1-t} \\ &= e^{at}\frac{e^{-a}}{1+e^{-a}} = e^{at}\sigma(-a) \end{aligned} \tag{10.148}$$

where $a = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}$. In order to obtain a lower bound on $p(\mathbf{t})$, we make use of the variational lower bound on the logistic sigmoid function given by (10.144), which

we reproduce here for convenience

$$\sigma(z) \geqslant \sigma(\xi) \exp\left\{(z-\xi)/2 - \lambda(\xi)(z^2-\xi^2)\right\} \tag{10.149}$$

where

$$\lambda(\xi) = \frac{1}{2\xi}\left[\sigma(\xi) - \frac{1}{2}\right]. \tag{10.150}$$

We can therefore write

$$p(t|\mathbf{w}) = e^{at}\sigma(-a) \geqslant e^{at}\sigma(\xi)\exp\left\{-(a+\xi)/2 - \lambda(\xi)(a^2-\xi^2)\right\}. \tag{10.151}$$

Note that because this bound is applied to each of the terms in the likelihood function separately, there is a variational parameter ξ_n corresponding to each training set observation $(\boldsymbol{\phi}_n, t_n)$. Using $a = \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}$, and multiplying by the prior distribution, we obtain the following bound on the joint distribution of $\mathbf{t}$ and $\mathbf{w}$

$$p(\mathbf{t}, \mathbf{w}) = p(\mathbf{t}|\mathbf{w})p(\mathbf{w}) \geqslant h(\mathbf{w}, \boldsymbol{\xi})p(\mathbf{w}) \tag{10.152}$$

where $\boldsymbol{\xi}$ denotes the set $\{\xi_n\}$ of variational parameters, and

$$\begin{aligned} h(\mathbf{w}, \boldsymbol{\xi}) \quad = \quad & \prod_{n=1}^{N} \sigma(\xi_n) \exp\left\{\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n t_n - (\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n + \xi_n)/2 \right. \\ & \left. - \lambda(\xi_n)([\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n]^2 - \xi_n^2)\right\}. \end{aligned} \tag{10.153}$$

Evaluation of the exact posterior distribution would require normalization of the left-hand side of this inequality. Because this is intractable, we work instead with the right-hand side. Note that the function on the right-hand side cannot be interpreted as a probability density because it is not normalized. Once it is normalized to give a variational posterior distribution $q(\mathbf{w})$, however, it no longer represents a bound.

Because the logarithm function is monotonically increasing, the inequality $A \geqslant B$ implies $\ln A \geqslant \ln B$. This gives a lower bound on the log of the joint distribution of $\mathbf{t}$ and $\mathbf{w}$ of the form

$$\begin{aligned} \ln\{p(\mathbf{t}|\mathbf{w})p(\mathbf{w})\} \geqslant \ln p(\mathbf{w}) + \sum_{n=1}^{N} & \left\{\ln\sigma(\xi_n) + \mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n t_n \right. \\ & \left. - (\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n + \xi_n)/2 - \lambda(\xi_n)([\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n]^2 - \xi_n^2)\right\}. \end{aligned} \tag{10.154}$$

Substituting for the prior $p(\mathbf{w})$, the right-hand side of this inequality becomes, as a function of $\mathbf{w}$

$$\begin{aligned} & -\frac{1}{2}(\mathbf{w}-\mathbf{m}_0)^{\mathrm{T}}\mathbf{S}_0^{-1}(\mathbf{w}-\mathbf{m}_0) \\ & + \sum_{n=1}^{N}\left\{\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n(t_n - 1/2) - \lambda(\xi_n)\mathbf{w}^{\mathrm{T}}(\boldsymbol{\phi}_n\boldsymbol{\phi}_n^{\mathrm{T}})\mathbf{w}\right\} + \text{const.} \end{aligned} \tag{10.155}$$

This is a quadratic function of $\mathbf{w}$, and so we can obtain the corresponding variational approximation to the posterior distribution by identifying the linear and quadratic terms in $\mathbf{w}$, giving a Gaussian variational posterior of the form

$$q(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{m}_N, \mathbf{S}_N) \tag{10.156}$$

where

$$\mathbf{m}_N = \mathbf{S}_N \left(\mathbf{S}_0^{-1}\mathbf{m}_0 + \sum_{n=1}^{N} (t_n - 1/2)\boldsymbol{\phi}_n \right) \tag{10.157}$$

$$\mathbf{S}_N^{-1} = \mathbf{S}_0^{-1} + 2\sum_{n=1}^{N} \lambda(\xi_n)\boldsymbol{\phi}_n\boldsymbol{\phi}_n^{\mathrm{T}}. \tag{10.158}$$

As with the Laplace framework, we have again obtained a Gaussian approximation to the posterior distribution. However, the additional flexibility provided by the variational parameters $\{\xi_n\}$ leads to improved accuracy in the approximation (Jaakkola and Jordan, 2000).

Here we have considered a batch learning context in which all of the training data is available at once. However, Bayesian methods are intrinsically well suited to sequential learning in which the data points are processed one at a time and then discarded. The formulation of this variational approach for the sequential case is straightforward.

Exercise 10.32

Note that the bound given by (10.149) applies only to the two-class problem and so this approach does not directly generalize to classification problems with $K > 2$ classes. An alternative bound for the multiclass case has been explored by Gibbs (1997).

10.6.2 Optimizing the variational parameters

We now have a normalized Gaussian approximation to the posterior distribution, which we shall use shortly to evaluate the predictive distribution for new data points. First, however, we need to determine the variational parameters $\{\xi_n\}$ by maximizing the lower bound on the marginal likelihood.

To do this, we substitute the inequality (10.152) back into the marginal likelihood to give

$$\ln p(\mathbf{t}) = \ln \int p(\mathbf{t}|\mathbf{w})p(\mathbf{w})\,\mathrm{d}\mathbf{w} \geqslant \ln \int h(\mathbf{w}, \boldsymbol{\xi})p(\mathbf{w})\,\mathrm{d}\mathbf{w} = \mathcal{L}(\boldsymbol{\xi}). \tag{10.159}$$

As with the optimization of the hyperparameter α in the linear regression model of Section 3.5, there are two approaches to determining the ξ_n. In the first approach, we recognize that the function $\mathcal{L}(\boldsymbol{\xi})$ is defined by an integration over $\mathbf{w}$ and so we can view $\mathbf{w}$ as a latent variable and invoke the EM algorithm. In the second approach, we integrate over $\mathbf{w}$ analytically and then perform a direct maximization over $\boldsymbol{\xi}$. Let us begin by considering the EM approach.

The EM algorithm starts by choosing some initial values for the parameters $\{\xi_n\}$, which we denote collectively by $\boldsymbol{\xi}^{\mathrm{old}}$. In the E step of the EM algorithm,

we then use these parameter values to find the posterior distribution over $\mathbf{w}$, which is given by (10.156). In the M step, we then maximize the expected complete-data log likelihood which is given by

$$Q(\boldsymbol{\xi}, \boldsymbol{\xi}^{\text{old}}) = \mathbb{E}\left[\ln h(\mathbf{w}, \boldsymbol{\xi}) p(\mathbf{w})\right] \tag{10.160}$$

where the expectation is taken with respect to the posterior distribution $q(\mathbf{w})$ evaluated using $\boldsymbol{\xi}^{\text{old}}$. Noting that $p(\mathbf{w})$ does not depend on $\boldsymbol{\xi}$, and substituting for $h(\mathbf{w}, \boldsymbol{\xi})$ we obtain

$$Q(\boldsymbol{\xi}, \boldsymbol{\xi}^{\text{old}}) = \sum_{n=1}^{N} \left\{\ln \sigma(\xi_n) - \xi_n/2 - \lambda(\xi_n)(\boldsymbol{\phi}_n^{\mathrm{T}} \mathbb{E}[\mathbf{w}\mathbf{w}^{\mathrm{T}}]\boldsymbol{\phi}_n - \xi_n^2)\right\} + \text{const} \tag{10.161}$$

where 'const' denotes terms that are independent of $\boldsymbol{\xi}$. We now set the derivative with respect to ξ_n equal to zero. A few lines of algebra, making use of the definitions of $\sigma(\xi)$ and $\lambda(\xi)$, then gives

$$0 = \lambda'(\xi_n)(\boldsymbol{\phi}_n^{\mathrm{T}} \mathbb{E}[\mathbf{w}\mathbf{w}^{\mathrm{T}}]\boldsymbol{\phi}_n - \xi_n^2). \tag{10.162}$$

We now note that $\lambda'(\xi)$ is a monotonic function of ξ for $\xi \geqslant 0$, and that we can restrict attention to nonnegative values of ξ without loss of generality due to the symmetry of the bound around $\xi = 0$. Thus $\lambda'(\xi) \neq 0$, and hence we obtain the following re-estimation equations

Exercise 10.33

$$(\xi_n^{\text{new}})^2 = \boldsymbol{\phi}_n^{\mathrm{T}} \mathbb{E}[\mathbf{w}\mathbf{w}^{\mathrm{T}}]\boldsymbol{\phi}_n = \boldsymbol{\phi}_n^{\mathrm{T}} \left(\mathbf{S}_N + \mathbf{m}_N \mathbf{m}_N^{\mathrm{T}}\right) \boldsymbol{\phi}_n \tag{10.163}$$

where we have used (10.156).

Let us summarize the EM algorithm for finding the variational posterior distribution. We first initialize the variational parameters $\boldsymbol{\xi}^{\text{old}}$. In the E step, we evaluate the posterior distribution over $\mathbf{w}$ given by (10.156), in which the mean and covariance are defined by (10.157) and (10.158). In the M step, we then use this variational posterior to compute a new value for $\boldsymbol{\xi}$ given by (10.163). The E and M steps are repeated until a suitable convergence criterion is satisfied, which in practice typically requires only a few iterations.

An alternative approach to obtaining re-estimation equations for $\boldsymbol{\xi}$ is to note that in the integral over $\mathbf{w}$ in the definition (10.159) of the lower bound $\mathcal{L}(\boldsymbol{\xi})$, the integrand has a Gaussian-like form and so the integral can be evaluated analytically. Having evaluated the integral, we can then differentiate with respect to ξ_n. It turns out that this gives rise to exactly the same re-estimation equations as does the EM approach given by (10.163).

Exercise 10.34

As we have emphasized already, in the application of variational methods it is useful to be able to evaluate the lower bound $\mathcal{L}(\boldsymbol{\xi})$ given by (10.159). The integration over $\mathbf{w}$ can be performed analytically by noting that $p(\mathbf{w})$ is Gaussian and $h(\mathbf{w}, \boldsymbol{\xi})$ is the exponential of a quadratic function of $\mathbf{w}$. Thus, by completing the square and making use of the standard result for the normalization coefficient of a Gaussian distribution, we can obtain a closed form solution which takes the form

Exercise 10.35

Figure 10.13 Illustration of the Bayesian approach to logistic regression for a simple linearly separable data set. The plot on the left shows the predictive distribution obtained using variational inference. We see that the decision boundary lies roughly mid way between the clusters of data points, and that the contours of the predictive distribution splay out away from the data reflecting the greater uncertainty in the classification of such regions. The plot on the right shows the decision boundaries corresponding to five samples of the parameter vector $\mathbf{w}$ drawn from the posterior distribution $p(\mathbf{w}|\mathbf{t})$.

$$\begin{aligned}\mathcal{L}(\boldsymbol{\xi}) &= \frac{1}{2}\ln\frac{|\mathbf{S}_N|}{|\mathbf{S}_0|} - \frac{1}{2}\mathbf{m}_N^{\mathrm{T}}\mathbf{S}_N^{-1}\mathbf{m}_N + \frac{1}{2}\mathbf{m}_0^{\mathrm{T}}\mathbf{S}_0^{-1}\mathbf{m}_0 \\ &\quad + \sum_{n=1}^{N}\left\{\ln\sigma(\xi_n) - \frac{1}{2}\xi_n - \lambda(\xi_n)\xi_n^2\right\}. \qquad (10.164)\end{aligned}$$

This variational framework can also be applied to situations in which the data is arriving sequentially (Jaakkola and Jordan, 2000). In this case we maintain a Gaussian posterior distribution over $\mathbf{w}$, which is initialized using the prior $p(\mathbf{w})$. As each data point arrives, the posterior is updated by making use of the bound (10.151) and then normalized to give an updated posterior distribution.

The predictive distribution is obtained by marginalizing over the posterior distribution, and takes the same form as for the Laplace approximation discussed in Section 4.5.2. Figure 10.13 shows the variational predictive distributions for a synthetic data set. This example provides interesting insights into the concept of 'large margin', which was discussed in Section 7.1 and which has qualitatively similar behaviour to the Bayesian solution.

10.6.3 Inference of hyperparameters

So far, we have treated the hyperparameter α in the prior distribution as a known constant. We now extend the Bayesian logistic regression model to allow the value of this parameter to be inferred from the data set. This can be achieved by combining the global and local variational approximations into a single framework, so as to maintain a lower bound on the marginal likelihood at each stage. Such a combined approach was adopted by Bishop and Svensén (2003) in the context of a Bayesian treatment of the hierarchical mixture of experts model.

Specifically, we consider once again a simple isotropic Gaussian prior distribution of the form

$$p(\mathbf{w}|\alpha) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I}). \tag{10.165}$$

Our analysis is readily extended to more general Gaussian priors, for instance if we wish to associate a different hyperparameter with different subsets of the parameters w_j. As usual, we consider a conjugate hyperprior over α given by a gamma distribution

$$p(\alpha) = \mathrm{Gam}(\alpha|a_0, b_0) \tag{10.166}$$

governed by the constants a_0 and b_0.

The marginal likelihood for this model now takes the form

$$p(\mathbf{t}) = \iint p(\mathbf{w}, \alpha, \mathbf{t})\, \mathrm{d}\mathbf{w}\, \mathrm{d}\alpha \tag{10.167}$$

where the joint distribution is given by

$$p(\mathbf{w}, \alpha, \mathbf{t}) = p(\mathbf{t}|\mathbf{w})p(\mathbf{w}|\alpha)p(\alpha). \tag{10.168}$$

We are now faced with an analytically intractable integration over $\mathbf{w}$ and α, which we shall tackle by using both the local and global variational approaches in the same model

To begin with, we introduce a variational distribution $q(\mathbf{w}, \alpha)$, and then apply the decomposition (10.2), which in this instance takes the form

$$\ln p(\mathbf{t}) = \mathcal{L}(q) + \mathrm{KL}(q\|p) \tag{10.169}$$

where the lower bound $\mathcal{L}(q)$ and the Kullback-Leibler divergence $\mathrm{KL}(q\|p)$ are defined by

$$\mathcal{L}(q) = \iint q(\mathbf{w}, \alpha) \ln \left\{ \frac{p(\mathbf{w}, \alpha, \mathbf{t})}{q(\mathbf{w}, \alpha)} \right\} \mathrm{d}\mathbf{w}\, \mathrm{d}\alpha \tag{10.170}$$

$$\mathrm{KL}(q\|p) = -\iint q(\mathbf{w}, \alpha) \ln \left\{ \frac{p(\mathbf{w}, \alpha|\mathbf{t}))}{q(\mathbf{w}, \alpha)} \right\} \mathrm{d}\mathbf{w}\, \mathrm{d}\alpha. \tag{10.171}$$

At this point, the lower bound $\mathcal{L}(q)$ is still intractable due to the form of the likelihood factor $p(\mathbf{t}|\mathbf{w})$. We therefore apply the local variational bound to each of the logistic sigmoid factors as before. This allows us to use the inequality (10.152) and place a lower bound on $\mathcal{L}(q)$, which will therefore also be a lower bound on the log marginal likelihood

$$\begin{aligned} \ln p(\mathbf{t}) &\geqslant \mathcal{L}(q) \geqslant \widetilde{\mathcal{L}}(q, \boldsymbol{\xi}) \\ &= \iint q(\mathbf{w}, \alpha) \ln \left\{ \frac{h(\mathbf{w}, \boldsymbol{\xi})p(\mathbf{w}|\alpha)p(\alpha)}{q(\mathbf{w}, \alpha)} \right\} \mathrm{d}\mathbf{w}\, \mathrm{d}\alpha. \end{aligned} \tag{10.172}$$

Next we assume that the variational distribution factorizes between parameters and hyperparameters so that

$$q(\mathbf{w}, \alpha) = q(\mathbf{w})q(\alpha). \tag{10.173}$$

With this factorization we can appeal to the general result (10.9) to find expressions for the optimal factors. Consider first the distribution $q(\mathbf{w})$. Discarding terms that are independent of $\mathbf{w}$, we have

$$\begin{aligned}\ln q(\mathbf{w}) &= \mathbb{E}_\alpha\left[\ln\left\{h(\mathbf{w},\boldsymbol{\xi})p(\mathbf{w}|\alpha)p(\alpha)\right\}\right] + \text{const} \\ &= \ln h(\mathbf{w},\boldsymbol{\xi}) + \mathbb{E}_\alpha\left[\ln p(\mathbf{w}|\alpha)\right] + \text{const}.\end{aligned}$$

We now substitute for $\ln h(\mathbf{w},\boldsymbol{\xi})$ using (10.153), and for $\ln p(\mathbf{w}|\alpha)$ using (10.165), giving

$$\ln q(\mathbf{w}) = -\frac{\mathbb{E}[\alpha]}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} + \sum_{n=1}^{N}\left\{(t_n - 1/2)\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n - \lambda(\xi_n)\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}_n\boldsymbol{\phi}_n^{\mathrm{T}}\mathbf{w}\right\} + \text{const}.$$

We see that this is a quadratic function of $\mathbf{w}$ and so the solution for $q(\mathbf{w})$ will be Gaussian. Completing the square in the usual way, we obtain

$$q(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\boldsymbol{\mu}_N, \boldsymbol{\Sigma}_N) \tag{10.174}$$

where we have defined

$$\boldsymbol{\Sigma}_N^{-1}\boldsymbol{\mu}_N = \sum_{n=1}^{N}(t_n - 1/2)\boldsymbol{\phi}_n \tag{10.175}$$

$$\boldsymbol{\Sigma}_N^{-1} = \mathbb{E}[\alpha]\mathbf{I} + 2\sum_{n=1}^{N}\lambda(\xi_n)\boldsymbol{\phi}_n\boldsymbol{\phi}_n^{\mathrm{T}}. \tag{10.176}$$

Similarly, the optimal solution for the factor $q(\alpha)$ is obtained from

$$\ln q(\alpha) = \mathbb{E}_{\mathbf{w}}\left[\ln p(\mathbf{w}|\alpha)\right] + \ln p(\alpha) + \text{const}.$$

Substituting for $\ln p(\mathbf{w}|\alpha)$ using (10.165), and for $\ln p(\alpha)$ using (10.166), we obtain

$$\ln q(\alpha) = \frac{M}{2}\ln\alpha - \frac{\alpha}{2}\mathbb{E}\left[\mathbf{w}^{\mathrm{T}}\mathbf{w}\right] + (a_0 - 1)\ln\alpha - b_0\alpha + \text{const}.$$

We recognize this as the log of a gamma distribution, and so we obtain

$$q(\alpha) = \mathrm{Gam}(\alpha|a_N, b_N) = \frac{1}{\Gamma(a_0)}a_0^{b_0}\alpha^{a_0-1}e^{-b_0\alpha} \tag{10.177}$$

where

$$a_N = a_0 + \frac{M}{2} \tag{10.178}$$

$$b_N = b_0 + \frac{1}{2}\mathbb{E}_{\mathbf{w}}\left[\mathbf{w}^{\mathrm{T}}\mathbf{w}\right]. \tag{10.179}$$

We also need to optimize the variational parameters ξ_n, and this is also done by maximizing the lower bound $\widetilde{\mathcal{L}}(q, \boldsymbol{\xi})$. Omitting terms that are independent of $\boldsymbol{\xi}$, and integrating over α, we have

$$\widetilde{\mathcal{L}}(q, \boldsymbol{\xi}) = \int q(\mathbf{w}) \ln h(\mathbf{w}, \boldsymbol{\xi}) \, \mathrm{d}\mathbf{w} + \text{const}. \tag{10.180}$$

Note that this has precisely the same form as (10.159), and so we can again appeal to our earlier result (10.163), which can be obtained by direct optimization of the marginal likelihood function, leading to re-estimation equations of the form

$$(\xi_n^{\text{new}})^2 = \boldsymbol{\phi}_n^{\mathrm{T}} \left(\boldsymbol{\Sigma}_N + \boldsymbol{\mu}_N \boldsymbol{\mu}_N^{\mathrm{T}} \right) \boldsymbol{\phi}_n. \tag{10.181}$$

We have obtained re-estimation equations for the three quantities $q(\mathbf{w})$, $q(\alpha)$, and $\boldsymbol{\xi}$, and so after making suitable initializations, we can cycle through these quantities, updating each in turn. The required moments are given by

Appendix B

$$\mathbb{E}[\alpha] = \frac{a_N}{b_N} \tag{10.182}$$

$$\mathbb{E}\left[\mathbf{w}^{\mathrm{T}}\mathbf{w}\right] = \boldsymbol{\Sigma}_N + \boldsymbol{\mu}_N^{\mathrm{T}} \boldsymbol{\mu}_N. \tag{10.183}$$

10.7. Expectation Propagation

We conclude this chapter by discussing an alternative form of deterministic approximate inference, known as *expectation propagation* or *EP* (Minka, 2001a; Minka, 2001b). As with the variational Bayes methods discussed so far, this too is based on the minimization of a Kullback-Leibler divergence but now of the reverse form, which gives the approximation rather different properties.

Consider for a moment the problem of minimizing $\mathrm{KL}(p\|q)$ with respect to $q(\mathbf{z})$ when $p(\mathbf{z})$ is a fixed distribution and $q(\mathbf{z})$ is a member of the exponential family and so, from (2.194), can be written in the form

$$q(\mathbf{z}) = h(\mathbf{z}) g(\boldsymbol{\eta}) \exp\left\{ \boldsymbol{\eta}^{\mathrm{T}} \mathbf{u}(\mathbf{z}) \right\}. \tag{10.184}$$

As a function of $\boldsymbol{\eta}$, the Kullback-Leibler divergence then becomes

$$\mathrm{KL}(p\|q) = -\ln g(\boldsymbol{\eta}) - \boldsymbol{\eta}^{\mathrm{T}} \mathbb{E}_{p(\mathbf{z})}[\mathbf{u}(\mathbf{z})] + \text{const} \tag{10.185}$$

where the constant terms are independent of the natural parameters $\boldsymbol{\eta}$. We can minimize $\mathrm{KL}(p\|q)$ within this family of distributions by setting the gradient with respect to $\boldsymbol{\eta}$ to zero, giving

$$-\nabla \ln g(\boldsymbol{\eta}) = \mathbb{E}_{p(\mathbf{z})}[\mathbf{u}(\mathbf{z})]. \tag{10.186}$$

However, we have already seen in (2.226) that the negative gradient of $\ln g(\boldsymbol{\eta})$ is given by the expectation of $\mathbf{u}(\mathbf{z})$ under the distribution $q(\mathbf{z})$. Equating these two results, we obtain

$$\mathbb{E}_{q(\mathbf{z})}[\mathbf{u}(\mathbf{z})] = \mathbb{E}_{p(\mathbf{z})}[\mathbf{u}(\mathbf{z})]. \tag{10.187}$$

We see that the optimum solution simply corresponds to matching the expected sufficient statistics. So, for instance, if $q(\mathbf{z})$ is a Gaussian $\mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ then we minimize the Kullback-Leibler divergence by setting the mean $\boldsymbol{\mu}$ of $q(\mathbf{z})$ equal to the mean of the distribution $p(\mathbf{z})$ and the covariance $\boldsymbol{\Sigma}$ equal to the covariance of $p(\mathbf{z})$. This is sometimes called *moment matching*. An example of this was seen in Figure 10.3(a).

Now let us exploit this result to obtain a practical algorithm for approximate inference. For many probabilistic models, the joint distribution of data $\mathcal{D}$ and hidden variables (including parameters) $\boldsymbol{\theta}$ comprises a product of factors in the form

$$p(\mathcal{D}, \boldsymbol{\theta}) = \prod_i f_i(\boldsymbol{\theta}). \tag{10.188}$$

This would arise, for example, in a model for independent, identically distributed data in which there is one factor $f_n(\boldsymbol{\theta}) = p(\mathbf{x}_n|\boldsymbol{\theta})$ for each data point $\mathbf{x}_n$, along with a factor $f_0(\boldsymbol{\theta}) = p(\boldsymbol{\theta})$ corresponding to the prior. More generally, it would also apply to any model defined by a directed probabilistic graph in which each factor is a conditional distribution corresponding to one of the nodes, or an undirected graph in which each factor is a clique potential. We are interested in evaluating the posterior distribution $p(\boldsymbol{\theta}|\mathcal{D})$ for the purpose of making predictions, as well as the model evidence $p(\mathcal{D})$ for the purpose of model comparison. From (10.188) the posterior is given by

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{1}{p(\mathcal{D})} \prod_i f_i(\boldsymbol{\theta}) \tag{10.189}$$

and the model evidence is given by

$$p(\mathcal{D}) = \int \prod_i f_i(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}. \tag{10.190}$$

Here we are considering continuous variables, but the following discussion applies equally to discrete variables with integrals replaced by summations. We shall suppose that the marginalization over $\boldsymbol{\theta}$, along with the marginalizations with respect to the posterior distribution required to make predictions, are intractable so that some form of approximation is required.

Expectation propagation is based on an approximation to the posterior distribution which is also given by a product of factors

$$q(\boldsymbol{\theta}) = \frac{1}{Z} \prod_i \widetilde{f}_i(\boldsymbol{\theta}) \tag{10.191}$$

in which each factor $\widetilde{f}_i(\boldsymbol{\theta})$ in the approximation corresponds to one of the factors $f_i(\boldsymbol{\theta})$ in the true posterior (10.189), and the factor $1/Z$ is the normalizing constant needed to ensure that the left-hand side of (10.191) integrates to unity. In order to obtain a practical algorithm, we need to constrain the factors $\widetilde{f}_i(\boldsymbol{\theta})$ in some way, and in particular we shall assume that they come from the exponential family. The product of the factors will therefore also be from the exponential family and so can

be described by a finite set of sufficient statistics. For example, if each of the $\widetilde{f}_i(\boldsymbol{\theta})$ is a Gaussian, then the overall approximation $q(\boldsymbol{\theta})$ will also be Gaussian.

Ideally we would like to determine the $\widetilde{f}_i(\boldsymbol{\theta})$ by minimizing the Kullback-Leibler divergence between the true posterior and the approximation given by

$$\mathrm{KL}\left(p\|q\right) = \mathrm{KL}\left(\frac{1}{p(\mathcal{D})}\prod_i f_i(\boldsymbol{\theta})\middle\|\frac{1}{Z}\prod_i \widetilde{f}_i(\boldsymbol{\theta})\right). \tag{10.192}$$

Note that this is the reverse form of KL divergence compared with that used in variational inference. In general, this minimization will be intractable because the KL divergence involves averaging with respect to the true distribution. As a rough approximation, we could instead minimize the KL divergences between the corresponding pairs $f_i(\boldsymbol{\theta})$ and $\widetilde{f}_i(\boldsymbol{\theta})$ of factors. This represents a much simpler problem to solve, and has the advantage that the algorithm is noniterative. However, because each factor is individually approximated, the product of the factors could well give a poor approximation.

Expectation propagation makes a much better approximation by optimizing each factor in turn in the context of all of the remaining factors. It starts by initializing the factors $\widetilde{f}_i(\boldsymbol{\theta})$, and then cycles through the factors refining them one at a time. This is similar in spirit to the update of factors in the variational Bayes framework considered earlier. Suppose we wish to refine factor $\widetilde{f}_j(\boldsymbol{\theta})$. We first remove this factor from the product to give $\prod_{i\neq j}\widetilde{f}_i(\boldsymbol{\theta})$. Conceptually, we will now determine a revised form of the factor $\widetilde{f}_j(\boldsymbol{\theta})$ by ensuring that the product

$$q^{\mathrm{new}}(\boldsymbol{\theta}) \propto \widetilde{f}_j(\boldsymbol{\theta})\prod_{i\neq j}\widetilde{f}_i(\boldsymbol{\theta}) \tag{10.193}$$

is as close as possible to

$$f_j(\boldsymbol{\theta})\prod_{i\neq j}\widetilde{f}_i(\boldsymbol{\theta}) \tag{10.194}$$

in which we keep fixed all of the factors $\widetilde{f}_i(\boldsymbol{\theta})$ for $i \neq j$. This ensures that the approximation is most accurate in the regions of high posterior probability as defined by the remaining factors. We shall see an example of this effect when we apply EP to the 'clutter problem'. To achieve this, we first remove the factor $\widetilde{f}_j(\boldsymbol{\theta})$ from the current approximation to the posterior by defining the unnormalized distribution

Section 10.7.1

$$q^{\backslash j}(\boldsymbol{\theta}) = \frac{q(\boldsymbol{\theta})}{\widetilde{f}_j(\boldsymbol{\theta})}. \tag{10.195}$$

Note that we could instead find $q^{\backslash j}(\boldsymbol{\theta})$ from the product of factors $i \neq j$, although in practice division is usually easier. This is now combined with the factor $f_j(\boldsymbol{\theta})$ to give a distribution

$$\frac{1}{Z_j}f_j(\boldsymbol{\theta})q^{\backslash j}(\boldsymbol{\theta}) \tag{10.196}$$

Figure 10.14 Illustration of the expectation propagation approximation using a Gaussian distribution for the example considered earlier in Figures 4.14 and 10.1. The left-hand plot shows the original distribution (yellow) along with the Laplace (red), global variational (green), and EP (blue) approximations, and the right-hand plot shows the corresponding negative logarithms of the distributions. Note that the EP distribution is broader than that variational inference, as a consequence of the different form of KL divergence.

where Z_j is the normalization constant given by

$$Z_j = \int f_j(\boldsymbol{\theta}) q^{\backslash j}(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}. \tag{10.197}$$

We now determine a revised factor $\widetilde{f}_j(\boldsymbol{\theta})$ by minimizing the Kullback-Leibler divergence

$$\mathrm{KL}\left(\frac{f_j(\boldsymbol{\theta}) q^{\backslash j}(\boldsymbol{\theta})}{Z_j} \middle\| q^{\mathrm{new}}(\boldsymbol{\theta}) \right). \tag{10.198}$$

This is easily solved because the approximating distribution $q^{\mathrm{new}}(\boldsymbol{\theta})$ is from the exponential family, and so we can appeal to the result (10.187), which tells us that the parameters of $q^{\mathrm{new}}(\boldsymbol{\theta})$ are obtained by matching its expected sufficient statistics to the corresponding moments of (10.196). We shall assume that this is a tractable operation. For example, if we choose $q(\boldsymbol{\theta})$ to be a Gaussian distribution $\mathcal{N}(\boldsymbol{\theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, then $\boldsymbol{\mu}$ is set equal to the mean of the (unnormalized) distribution $f_j(\boldsymbol{\theta}) q^{\backslash j}(\boldsymbol{\theta})$, and $\boldsymbol{\Sigma}$ is set to its covariance. More generally, it is straightforward to obtain the required expectations for any member of the exponential family, provided it can be normalized, because the expected statistics can be related to the derivatives of the normalization coefficient, as given by (2.226). The EP approximation is illustrated in Figure 10.14.

From (10.193), we see that the revised factor $\widetilde{f}_j(\boldsymbol{\theta})$ can be found by taking $q^{\mathrm{new}}(\boldsymbol{\theta})$ and dividing out the remaining factors so that

$$\widetilde{f}_j(\boldsymbol{\theta}) = K \frac{q^{\mathrm{new}}(\boldsymbol{\theta})}{q^{\backslash j}(\boldsymbol{\theta})} \tag{10.199}$$

where we have used (10.195). The coefficient K is determined by multiplying both

sides of (10.199) by $q^{\backslash i}(\boldsymbol{\theta})$ and integrating to give

$$K = \int \widetilde{f}_j(\boldsymbol{\theta}) q^{\backslash j}(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta} \tag{10.200}$$

where we have used the fact that $q^{\mathrm{new}}(\boldsymbol{\theta})$ is normalized. The value of K can therefore be found by matching zeroth-order moments

$$\int \widetilde{f}_j(\boldsymbol{\theta}) q^{\backslash j}(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta} = \int f_j(\boldsymbol{\theta}) q^{\backslash j}(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}. \tag{10.201}$$

Combining this with (10.197), we then see that $K = Z_j$ and so can be found by evaluating the integral in (10.197).

In practice, several passes are made through the set of factors, revising each factor in turn. The posterior distribution $p(\boldsymbol{\theta}|\mathcal{D})$ is then approximated using (10.191), and the model evidence $p(\mathcal{D})$ can be approximated by using (10.190) with the factors $f_i(\boldsymbol{\theta})$ replaced by their approximations $\widetilde{f}_i(\boldsymbol{\theta})$.

Expectation Propagation

We are given a joint distribution over observed data $\mathcal{D}$ and stochastic variables $\boldsymbol{\theta}$ in the form of a product of factors

$$p(\mathcal{D}, \boldsymbol{\theta}) = \prod_i f_i(\boldsymbol{\theta}) \tag{10.202}$$

and we wish to approximate the posterior distribution $p(\boldsymbol{\theta}|\mathcal{D})$ by a distribution of the form

$$q(\boldsymbol{\theta}) = \frac{1}{Z} \prod_i \widetilde{f}_i(\boldsymbol{\theta}). \tag{10.203}$$

We also wish to approximate the model evidence $p(\mathcal{D})$.

1. Initialize all of the approximating factors $\widetilde{f}_i(\boldsymbol{\theta})$.
2. Initialize the posterior approximation by setting

$$q(\boldsymbol{\theta}) \propto \prod_i \widetilde{f}_i(\boldsymbol{\theta}). \tag{10.204}$$

3. Until convergence:
   (a) Choose a factor $\widetilde{f}_j(\boldsymbol{\theta})$ to refine.
   (b) Remove $\widetilde{f}_j(\boldsymbol{\theta})$ from the posterior by division

$$q^{\backslash j}(\boldsymbol{\theta}) = \frac{q(\boldsymbol{\theta})}{\widetilde{f}_j(\boldsymbol{\theta})}. \tag{10.205}$$

(c) Evaluate the new posterior by setting the sufficient statistics (moments) of $q^{\text{new}}(\boldsymbol{\theta})$ equal to those of $q^{\backslash j}(\boldsymbol{\theta})f_j(\boldsymbol{\theta})$, including evaluation of the normalization constant

$$Z_j = \int q^{\backslash j}(\boldsymbol{\theta})f_j(\boldsymbol{\theta})\,\mathrm{d}\boldsymbol{\theta}. \tag{10.206}$$

(d) Evaluate and store the new factor

$$\widetilde{f}_j(\boldsymbol{\theta}) = Z_j \frac{q^{\text{new}}(\boldsymbol{\theta})}{q^{\backslash j}(\boldsymbol{\theta})}. \tag{10.207}$$

4. Evaluate the approximation to the model evidence

$$p(\mathcal{D}) \simeq \int \prod_i \widetilde{f}_i(\boldsymbol{\theta})\,\mathrm{d}\boldsymbol{\theta}. \tag{10.208}$$

A special case of EP, known as *assumed density filtering* (ADF) or *moment matching* (Maybeck, 1982; Lauritzen, 1992; Boyen and Koller, 1998; Opper and Winther, 1999), is obtained by initializing all of the approximating factors except the first to unity and then making one pass through the factors updating each of them once. Assumed density filtering can be appropriate for on-line learning in which data points are arriving in a sequence and we need to learn from each data point and then discard it before considering the next point. However, in a batch setting we have the opportunity to re-use the data points many times in order to achieve improved accuracy, and it is this idea that is exploited in expectation propagation. Furthermore, if we apply ADF to batch data, the results will have an undesirable dependence on the (arbitrary) order in which the data points are considered, which again EP can overcome.

One disadvantage of expectation propagation is that there is no guarantee that the iterations will converge. However, for approximations $q(\boldsymbol{\theta})$ in the exponential family, if the iterations do converge, the resulting solution will be a stationary point of a particular energy function (Minka, 2001a), although each iteration of EP does not necessarily decrease the value of this energy function. This is in contrast to variational Bayes, which iteratively maximizes a lower bound on the log marginal likelihood, in which each iteration is guaranteed not to decrease the bound. It is possible to optimize the EP cost function directly, in which case it is guaranteed to converge, although the resulting algorithms can be slower and more complex to implement.

Another difference between variational Bayes and EP arises from the form of KL divergence that is minimized by the two algorithms, because the former minimizes $\mathrm{KL}(q\|p)$ whereas the latter minimizes $\mathrm{KL}(p\|q)$. As we saw in Figure 10.3, for distributions $p(\boldsymbol{\theta})$ which are multimodal, minimizing $\mathrm{KL}(p\|q)$ can lead to poor approximations. In particular, if EP is applied to mixtures the results are not sensible because the approximation tries to capture all of the modes of the posterior distribution. Conversely, in logistic-type models, EP often out-performs both local variational methods and the Laplace approximation (Kuss and Rasmussen, 2006).

Figure 10.15 Illustration of the clutter problem for a data space dimensionality of $D = 1$. Training data points, denoted by the crosses, are drawn from a mixture of two Gaussians with components shown in red and green. The goal is to infer the mean of the green Gaussian from the observed data.

10.7.1 Example: The clutter problem

Following Minka (2001b), we illustrate the EP algorithm using a simple example in which the goal is to infer the mean $\boldsymbol{\theta}$ of a multivariate Gaussian distribution over a variable $\mathbf{x}$ given a set of observations drawn from that distribution. To make the problem more interesting, the observations are embedded in background clutter, which itself is also Gaussian distributed, as illustrated in Figure 10.15. The distribution of observed values $\mathbf{x}$ is therefore a mixture of Gaussians, which we take to be of the form

$$p(\mathbf{x}|\boldsymbol{\theta}) = (1-w)\mathcal{N}(\mathbf{x}|\boldsymbol{\theta}, \mathbf{I}) + w\mathcal{N}(\mathbf{x}|\mathbf{0}, a\mathbf{I}) \tag{10.209}$$

where w is the proportion of background clutter and is assumed to be known. The prior over $\boldsymbol{\theta}$ is taken to be Gaussian

$$p(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\mathbf{0}, b\mathbf{I}) \tag{10.210}$$

and Minka (2001a) chooses the parameter values $a = 10$, $b = 100$ and $w = 0.5$. The joint distribution of N observations $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ and $\boldsymbol{\theta}$ is given by

$$p(\mathcal{D}, \boldsymbol{\theta}) = p(\boldsymbol{\theta}) \prod_{n=1}^{N} p(\mathbf{x}_n|\boldsymbol{\theta}) \tag{10.211}$$

and so the posterior distribution comprises a mixture of 2^N Gaussians. Thus the computational cost of solving this problem exactly would grow exponentially with the size of the data set, and so an exact solution is intractable for moderately large N.

To apply EP to the clutter problem, we first identify the factors $f_0(\boldsymbol{\theta}) = p(\boldsymbol{\theta})$ and $f_n(\boldsymbol{\theta}) = p(\mathbf{x}_n|\boldsymbol{\theta})$. Next we select an approximating distribution from the exponential family, and for this example it is convenient to choose a spherical Gaussian

$$q(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}|\mathbf{m}, v\mathbf{I}). \tag{10.212}$$

The factor approximations will therefore take the form of exponential-quadratic functions of the form

$$\widetilde{f}_n(\boldsymbol{\theta}) = s_n \mathcal{N}(\boldsymbol{\theta}|\mathbf{m}_n, v_n\mathbf{I}) \tag{10.213}$$

where $n = 1, \ldots, N$, and we set $\widetilde{f}_0(\boldsymbol{\theta})$ equal to the prior $p(\boldsymbol{\theta})$. Note that the use of $\mathcal{N}(\boldsymbol{\theta}|\cdot,\cdot)$ does not imply that the right-hand side is a well-defined Gaussian density (in fact, as we shall see, the variance parameter v_n can be negative) but is simply a convenient shorthand notation. The approximations $\widetilde{f}_n(\boldsymbol{\theta})$, for $n = 1, \ldots, N$, can be initialized to unity, corresponding to $s_n = (2\pi v_n)^{D/2}$, $v_n \to \infty$ and $\mathbf{m}_n = \mathbf{0}$, where D is the dimensionality of $\mathbf{x}$ and hence of $\boldsymbol{\theta}$. The initial $q(\boldsymbol{\theta})$, defined by (10.191), is therefore equal to the prior.

We then iteratively refine the factors by taking one factor $f_n(\boldsymbol{\theta})$ at a time and applying (10.205), (10.206), and (10.207). Note that we do not need to revise the term $f_0(\boldsymbol{\theta})$ because an EP update will leave this term unchanged. Here we state the results and leave the reader to fill in the details.

Exercise 10.37

First we remove the current estimate $\widetilde{f}_n(\boldsymbol{\theta})$ from $q(\boldsymbol{\theta})$ by division using (10.205) to give $q^{\backslash n}(\boldsymbol{\theta})$, which has mean and inverse variance given by

Exercise 10.38

$$\mathbf{m}^{\backslash n} = \mathbf{m} + v^{\backslash n} v_n^{-1}(\mathbf{m} - \mathbf{m}_n) \tag{10.214}$$

$$(v^{\backslash n})^{-1} = v^{-1} - v_n^{-1}. \tag{10.215}$$

Next we evaluate the normalization constant Z_n using (10.206) to give

$$Z_n = (1-w)\mathcal{N}(\mathbf{x}_n|\mathbf{m}^{\backslash n}, (v^{\backslash n}+1)\mathbf{I}) + w\mathcal{N}(\mathbf{x}_n|\mathbf{0}, a\mathbf{I}). \tag{10.216}$$

Similarly, we compute the mean and variance of $q^{\text{new}}(\boldsymbol{\theta})$ by finding the mean and variance of $q^{\backslash n}(\boldsymbol{\theta}) f_n(\boldsymbol{\theta})$ to give

Exercise 10.39

$$\mathbf{m} = \mathbf{m}^{\backslash n} + \rho_n \frac{v^{\backslash n}}{v^{\backslash n}+1}(\mathbf{x}_n - \mathbf{m}^{\backslash n}) \tag{10.217}$$

$$v = v^{\backslash n} - \rho_n \frac{(v^{\backslash n})^2}{v^{\backslash n}+1} + \rho_n(1-\rho_n)\frac{(v^{\backslash n})^2\|\mathbf{x}_n - \mathbf{m}^{\backslash n}\|^2}{D(v^{\backslash n}+1)^2} \tag{10.218}$$

where the quantity

$$\rho_n = 1 - \frac{w}{Z_n}\mathcal{N}(\mathbf{x}_n|\mathbf{0}, a\mathbf{I}) \tag{10.219}$$

has a simple interpretation as the probability of the point $\mathbf{x}_n$ not being clutter. Then we use (10.207) to compute the refined factor $\widetilde{f}_n(\boldsymbol{\theta})$ whose parameters are given by

$$v_n^{-1} = (v^{\text{new}})^{-1} - (v^{\backslash n})^{-1} \tag{10.220}$$

$$\mathbf{m}_n = \mathbf{m}^{\backslash n} + (v_n + v^{\backslash n})(v^{\backslash n})^{-1}(\mathbf{m}^{\text{new}} - \mathbf{m}^{\backslash n}) \tag{10.221}$$

$$s_n = \frac{Z_n}{(2\pi v_n)^{D/2}\mathcal{N}(\mathbf{m}_n|\mathbf{m}^{\backslash n}, (v_n + v^{\backslash n})\mathbf{I})}. \tag{10.222}$$

This refinement process is repeated until a suitable termination criterion is satisfied, for instance that the maximum change in parameter values resulting from a complete

Figure 10.16 Examples of the approximation of specific factors for a one-dimensional version of the clutter problem, showing $f_n(\theta)$ in blue, $\widetilde{f}_n(\theta)$ in red, and $q^{\backslash n}(\theta)$ in green. Notice that the current form for $q^{\backslash n}(\theta)$ controls the range of θ over which $\widetilde{f}_n(\theta)$ will be a good approximation to $f_n(\theta)$.

pass through all factors is less than some threshold. Finally, we use (10.208) to evaluate the approximation to the model evidence, given by

$$p(\mathcal{D}) \simeq (2\pi v^{\text{new}})^{D/2} \exp(B/2) \prod_{n=1}^{N} \left\{ s_n (2\pi v_n)^{-D/2} \right\} \tag{10.223}$$

where

$$B = \frac{(\mathbf{m}^{\text{new}})^{\mathrm{T}} \mathbf{m}^{\text{new}}}{v} - \sum_{n=1}^{N} \frac{\mathbf{m}_n^{\mathrm{T}} \mathbf{m}_n}{v_n}. \tag{10.224}$$

Examples factor approximations for the clutter problem with a one-dimensional parameter space θ are shown in Figure 10.16. Note that the factor approximations can have infinite or even negative values for the 'variance' parameter v_n. This simply corresponds to approximations that curve upwards instead of downwards and are not necessarily problematic provided the overall approximate posterior $q(\boldsymbol{\theta})$ has positive variance. Figure 10.17 compares the performance of EP with variational Bayes (mean field theory) and the Laplace approximation on the clutter problem.

10.7.2 Expectation propagation on graphs

So far in our general discussion of EP, we have allowed the factors $f_i(\boldsymbol{\theta})$ in the distribution $p(\boldsymbol{\theta})$ to be functions of all of the components of $\boldsymbol{\theta}$, and similarly for the approximating factors $\widetilde{f}(\boldsymbol{\theta})$ in the approximating distribution $q(\boldsymbol{\theta})$. We now consider situations in which the factors depend only on subsets of the variables. Such restrictions can be conveniently expressed using the framework of probabilistic graphical models, as discussed in Chapter 8. Here we use a factor graph representation because this encompasses both directed and undirected graphs.

Figure 10.17 Comparison of expectation propagation, variational inference, and the Laplace approximation on the clutter problem. The left-hand plot shows the error in the predicted posterior mean versus the number of floating point operations, and the right-hand plot shows the corresponding results for the model evidence.

We shall focus on the case in which the approximating distribution is fully factorized, and we shall show that in this case expectation propagation reduces to loopy belief propagation (Minka, 2001a). To start with, we show this in the context of a simple example, and then we shall explore the general case.

First of all, recall from (10.17) that if we minimize the Kullback-Leibler divergence $\mathrm{KL}(p\|q)$ with respect to a factorized distribution q, then the optimal solution for each factor is simply the corresponding marginal of p.

Section 8.4.4

Now consider the factor graph shown on the left in Figure 10.18, which was introduced earlier in the context of the sum-product algorithm. The joint distribution is given by

$$p(\mathbf{x}) = f_a(x_1, x_2) f_b(x_2, x_3) f_c(x_2, x_4). \tag{10.225}$$

We seek an approximation $q(\mathbf{x})$ that has the same factorization, so that

$$q(\mathbf{x}) \propto \widetilde{f}_a(x_1, x_2) \widetilde{f}_b(x_2, x_3) \widetilde{f}_c(x_2, x_4). \tag{10.226}$$

Note that normalization constants have been omitted, and these can be re-instated at the end by local normalization, as is generally done in belief propagation. Now suppose we restrict attention to approximations in which the factors themselves factorize with respect to the individual variables so that

$$q(\mathbf{x}) \propto \widetilde{f}_{a1}(x_1) \widetilde{f}_{a2}(x_2) \widetilde{f}_{b2}(x_2) \widetilde{f}_{b3}(x_3) \widetilde{f}_{c2}(x_2) \widetilde{f}_{c4}(x_4) \tag{10.227}$$

which corresponds to the factor graph shown on the right in Figure 10.18. Because the individual factors are factorized, the overall distribution $q(\mathbf{x})$ is itself fully factorized.

Now we apply the EP algorithm using the fully factorized approximation. Suppose that we have initialized all of the factors and that we choose to refine factor

Figure 10.18 On the left is a simple factor graph from Figure 8.51 and reproduced here for convenience. On the right is the corresponding factorized approximation.

$\widetilde{f}_b(x_2, x_3) = \widetilde{f}_{b2}(x_2)\widetilde{f}_{b3}(x_3)$. We first remove this factor from the approximating distribution to give

$$q^{\backslash b}(\mathbf{x}) = \widetilde{f}_{a1}(x_1)\widetilde{f}_{a2}(x_2)\widetilde{f}_{c2}(x_2)\widetilde{f}_{c4}(x_4) \tag{10.228}$$

and we then multiply this by the exact factor $f_b(x_2, x_3)$ to give

$$\widehat{p}(\mathbf{x}) = q^{\backslash b}(\mathbf{x})f_b(x_2, x_3) = \widetilde{f}_{a1}(x_1)\widetilde{f}_{a2}(x_2)\widetilde{f}_{c2}(x_2)\widetilde{f}_{c4}(x_4)f_b(x_2, x_3). \tag{10.229}$$

We now find $q^{\text{new}}(\mathbf{x})$ by minimizing the Kullback-Leibler divergence $\text{KL}(\widehat{p}\|q^{\text{new}})$. The result, as noted above, is that $q^{\text{new}}(\mathbf{z})$ comprises the product of factors, one for each variable x_i, in which each factor is given by the corresponding marginal of $\widehat{p}(\mathbf{x})$. These four marginals are given by

$$\widehat{p}(x_1) \propto \widetilde{f}_{a1}(x_1) \tag{10.230}$$

$$\widehat{p}(x_2) \propto \widetilde{f}_{a2}(x_2)\widetilde{f}_{c2}(x_2)\sum_{x_3} f_b(x_2, x_3) \tag{10.231}$$

$$\widehat{p}(x_3) \propto \sum_{x_2}\left\{f_b(x_2, x_3)\widetilde{f}_{a2}(x_2)\widetilde{f}_{c2}(x_2)\right\} \tag{10.232}$$

$$\widehat{p}(x_4) \propto \widetilde{f}_{c4}(x_4) \tag{10.233}$$

and $q^{\text{new}}(\mathbf{x})$ is obtained by multiplying these marginals together. We see that the only factors in $q(\mathbf{x})$ that change when we update $\widetilde{f}_b(x_2, x_3)$ are those that involve the variables in f_b namely x_2 and x_3. To obtain the refined factor $\widetilde{f}_b(x_2, x_3) = \widetilde{f}_{b2}(x_2)\widetilde{f}_{b3}(x_3)$ we simply divide $q^{\text{new}}(\mathbf{x})$ by $q^{\backslash b}(\mathbf{x})$, which gives

$$\widetilde{f}_{b2}(x_2) \propto \sum_{x_3} f_b(x_2, x_3) \tag{10.234}$$

$$\widetilde{f}_{b3}(x_3) \propto \sum_{x_2}\left\{f_b(x_2, x_3)\widetilde{f}_{a2}(x_2)\widetilde{f}_{c2}(x_2)\right\}. \tag{10.235}$$

Section 8.4.4

These are precisely the messages obtained using belief propagation in which messages from variable nodes to factor nodes have been folded into the messages from factor nodes to variable nodes. In particular, $\widetilde{f}_{b2}(x_2)$ corresponds to the message $\mu_{f_b \to x_2}(x_2)$ sent by factor node f_b to variable node x_2 and is given by (8.81). Similarly, if we substitute (8.78) into (8.79), we obtain (10.235) in which $\widetilde{f}_{a2}(x_2)$ corresponds to $\mu_{f_a \to x_2}(x_2)$ and $\widetilde{f}_{c2}(x_2)$ corresponds to $\mu_{f_c \to x_2}(x_2)$, giving the message $\widetilde{f}_{b3}(x_3)$ which corresponds to $\mu_{f_b \to x_3}(x_3)$.

This result differs slightly from standard belief propagation in that messages are passed in both directions at the same time. We can easily modify the EP procedure to give the standard form of the sum-product algorithm by updating just one of the factors at a time, for instance if we refine only $\widetilde{f}_{b3}(x_3)$, then $\widetilde{f}_{b2}(x_2)$ is unchanged by definition, while the refined version of $\widetilde{f}_{b3}(x_3)$ is again given by (10.235). If we are refining only one term at a time, then we can choose the order in which the refinements are done as we wish. In particular, for a tree-structured graph we can follow a two-pass update scheme, corresponding to the standard belief propagation schedule, which will result in exact inference of the variable and factor marginals. The initialization of the approximation factors in this case is unimportant.

Now let us consider a general factor graph corresponding to the distribution

$$p(\boldsymbol{\theta}) = \prod_i f_i(\boldsymbol{\theta}_i) \tag{10.236}$$

where $\boldsymbol{\theta}_i$ represents the subset of variables associated with factor f_i. We approximate this using a fully factorized distribution of the form

$$q(\boldsymbol{\theta}) \propto \prod_i \prod_k \widetilde{f}_{ik}(\theta_k) \tag{10.237}$$

where θ_k corresponds to an individual variable node. Suppose that we wish to refine the particular term $\widetilde{f}_{jl}(\theta_l)$ keeping all other terms fixed. We first remove the term $\widetilde{f}_j(\boldsymbol{\theta}_j)$ from $q(\boldsymbol{\theta})$ to give

$$q^{\setminus j}(\boldsymbol{\theta}) \propto \prod_{i \neq j} \prod_k \widetilde{f}_{ik}(\theta_k) \tag{10.238}$$

and then multiply by the exact factor $f_j(\boldsymbol{\theta}_j)$. To determine the refined term $\widetilde{f}_{jl}(\theta_l)$, we need only consider the functional dependence on θ_l, and so we simply find the corresponding marginal of

$$q^{\setminus j}(\boldsymbol{\theta}) f_j(\boldsymbol{\theta}_j). \tag{10.239}$$

Up to a multiplicative constant, this involves taking the marginal of $f_j(\boldsymbol{\theta}_j)$ multiplied by any terms from $q^{\setminus j}(\boldsymbol{\theta})$ that are functions of any of the variables in $\boldsymbol{\theta}_j$. Terms that correspond to other factors $\widetilde{f}_i(\boldsymbol{\theta}_i)$ for $i \neq j$ will cancel between numerator and denominator when we subsequently divide by $q^{\setminus j}(\boldsymbol{\theta})$. We therefore obtain

$$\widetilde{f}_{jl}(\theta_l) \propto \sum_{\theta_{m \neq l} \in \boldsymbol{\theta}_j} f_j(\boldsymbol{\theta}_j) \prod_k \prod_{m \neq l} \widetilde{f}_{km}(\theta_m). \tag{10.240}$$

We recognize this as the sum-product rule in the form in which messages from variable nodes to factor nodes have been eliminated, as illustrated by the example shown in Figure 8.50. The quantity $\widetilde{f}_{jm}(\theta_m)$ corresponds to the message $\mu_{f_j \to \theta_m}(\theta_m)$, which factor node j sends to variable node m, and the product over k in (10.240) is over all factors that depend on the variables θ_m that have variables (other than variable θ_l) in common with factor $f_j(\boldsymbol{\theta}_j)$. In other words, to compute the outgoing message from a factor node, we take the product of all the incoming messages from other factor nodes, multiply by the local factor, and then marginalize.

Thus, the sum-product algorithm arises as a special case of expectation propagation if we use an approximating distribution that is fully factorized. This suggests that more flexible approximating distributions, corresponding to partially disconnected graphs, could be used to achieve higher accuracy. Another generalization is to group factors $f_i(\boldsymbol{\theta}_i)$ together into sets and to refine all the factors in a set together at each iteration. Both of these approaches can lead to improvements in accuracy (Minka, 2001b). In general, the problem of choosing the best combination of grouping and disconnection is an open research issue.

We have seen that variational message passing and expectation propagation optimize two different forms of the Kullback-Leibler divergence. Minka (2005) has shown that a broad range of message passing algorithms can be derived from a common framework involving minimization of members of the alpha family of divergences, given by (10.19). These include variational message passing, loopy belief propagation, and expectation propagation, as well as a range of other algorithms, which we do not have space to discuss here, such as *tree-reweighted message passing* (Wainwright *et al.*, 2005), *fractional belief propagation* (Wiegerinck and Heskes, 2003), and *power EP* (Minka, 2004).

Exercises

10.1 ($\star$) **www** Verify that the log marginal distribution of the observed data $\ln p(\mathbf{X})$ can be decomposed into two terms in the form (10.2) where $\mathcal{L}(q)$ is given by (10.3) and $\mathrm{KL}(q\|p)$ is given by (10.4).

10.2 ($\star$) Use the properties $\mathbb{E}[z_1] = m_1$ and $\mathbb{E}[z_2] = m_2$ to solve the simultaneous equations (10.13) and (10.15), and hence show that, provided the original distribution $p(\mathbf{z})$ is nonsingular, the unique solution for the means of the factors in the approximation distribution is given by $\mathbb{E}[z_1] = \mu_1$ and $\mathbb{E}[z_2] = \mu_2$.

10.3 ($\star\star$) **www** Consider a factorized variational distribution $q(\mathbf{Z})$ of the form (10.5). By using the technique of Lagrange multipliers, verify that minimization of the Kullback-Leibler divergence $\mathrm{KL}(p\|q)$ with respect to one of the factors $q_i(\mathbf{Z}_i)$, keeping all other factors fixed, leads to the solution (10.17).

10.4 ($\star\star$) Suppose that $p(\mathbf{x})$ is some fixed distribution and that we wish to approximate it using a Gaussian distribution $q(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$. By writing down the form of the KL divergence $\mathrm{KL}(p\|q)$ for a Gaussian $q(\mathbf{x})$ and then differentiating, show that

minimization of $\mathrm{KL}(p\|q)$ with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ leads to the result that $\boldsymbol{\mu}$ is given by the expectation of $\mathbf{x}$ under $p(\mathbf{x})$ and that $\boldsymbol{\Sigma}$ is given by the covariance.

10.5 ($\star\star$) **www** Consider a model in which the set of all hidden stochastic variables, denoted collectively by $\mathbf{Z}$, comprises some latent variables $\mathbf{z}$ together with some model parameters $\boldsymbol{\theta}$. Suppose we use a variational distribution that factorizes between latent variables and parameters so that $q(\mathbf{z}, \boldsymbol{\theta}) = q_{\mathbf{z}}(\mathbf{z}) q_{\boldsymbol{\theta}}(\boldsymbol{\theta})$, in which the distribution $q_{\boldsymbol{\theta}}(\boldsymbol{\theta})$ is approximated by a point estimate of the form $q_{\boldsymbol{\theta}}(\boldsymbol{\theta}) = \delta(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$ where $\boldsymbol{\theta}_0$ is a vector of free parameters. Show that variational optimization of this factorized distribution is equivalent to an EM algorithm, in which the E step optimizes $q_{\mathbf{z}}(\mathbf{z})$, and the M step maximizes the expected complete-data log posterior distribution of $\boldsymbol{\theta}$ with respect to $\boldsymbol{\theta}_0$.

10.6 ($\star\star$) The alpha family of divergences is defined by (10.19). Show that the Kullback-Leibler divergence $\mathrm{KL}(p\|q)$ corresponds to $\alpha \to 1$. This can be done by writing $p^{\epsilon} = \exp(\epsilon \ln p) = 1 + \epsilon \ln p + O(\epsilon^2)$ and then taking $\epsilon \to 0$. Similarly show that $\mathrm{KL}(q\|p)$ corresponds to $\alpha \to -1$.

10.7 ($\star\star$) Consider the problem of inferring the mean and precision of a univariate Gaussian using a factorized variational approximation, as considered in Section 10.1.3. Show that the factor $q_\mu(\mu)$ is a Gaussian of the form $\mathcal{N}(\mu|\mu_N, \lambda_N^{-1})$ with mean and precision given by (10.26) and (10.27), respectively. Similarly show that the factor $q_\tau(\tau)$ is a gamma distribution of the form $\mathrm{Gam}(\tau|a_N, b_N)$ with parameters given by (10.29) and (10.30).

10.8 ($\star$) Consider the variational posterior distribution for the precision of a univariate Gaussian whose parameters are given by (10.29) and (10.30). By using the standard results for the mean and variance of the gamma distribution given by (B.27) and (B.28), show that if we let $N \to \infty$, this variational posterior distribution has a mean given by the inverse of the maximum likelihood estimator for the variance of the data, and a variance that goes to zero.

10.9 ($\star\star$) By making use of the standard result $\mathbb{E}[\tau] = a_N/b_N$ for the mean of a gamma distribution, together with (10.26), (10.27), (10.29), and (10.30), derive the result (10.33) for the reciprocal of the expected precision in the factorized variational treatment of a univariate Gaussian.

10.10 ($\star$) **www** Derive the decomposition given by (10.34) that is used to find approximate posterior distributions over models using variational inference.

10.11 ($\star\star$) **www** By using a Lagrange multiplier to enforce the normalization constraint on the distribution $q(m)$, show that the maximum of the lower bound (10.35) is given by (10.36).

10.12 ($\star\star$) Starting from the joint distribution (10.41), and applying the general result (10.9), show that the optimal variational distribution $q^\star(\mathbf{Z})$ over the latent variables for the Bayesian mixture of Gaussians is given by (10.48) by verifying the steps given in the text.

10.13 ($\star\star$) **www** Starting from (10.54), derive the result (10.59) for the optimum variational posterior distribution over $\boldsymbol{\mu}_k$ and $\boldsymbol{\Lambda}_k$ in the Bayesian mixture of Gaussians, and hence verify the expressions for the parameters of this distribution given by (10.60)–(10.63).

10.14 ($\star\star$) Using the distribution (10.59), verify the result (10.64).

10.15 ($\star$) Using the result (B.17), show that the expected value of the mixing coefficients in the variational mixture of Gaussians is given by (10.69).

10.16 ($\star\star$) **www** Verify the results (10.71) and (10.72) for the first two terms in the lower bound for the variational Gaussian mixture model given by (10.70).

10.17 ($\star\star\star$) Verify the results (10.73)–(10.77) for the remaining terms in the lower bound for the variational Gaussian mixture model given by (10.70).

10.18 ($\star\star\star$) In this exercise, we shall derive the variational re-estimation equations for the Gaussian mixture model by direct differentiation of the lower bound. To do this we assume that the variational distribution has the factorization defined by (10.42) and (10.55) with factors given by (10.48), (10.57), and (10.59). Substitute these into (10.70) and hence obtain the lower bound as a function of the parameters of the variational distribution. Then, by maximizing the bound with respect to these parameters, derive the re-estimation equations for the factors in the variational distribution, and show that these are the same as those obtained in Section 10.2.1.

10.19 ($\star\star$) Derive the result (10.81) for the predictive distribution in the variational treatment of the Bayesian mixture of Gaussians model.

10.20 ($\star\star$) **www** This exercise explores the variational Bayes solution for the mixture of Gaussians model when the size N of the data set is large and shows that it reduces (as we would expect) to the maximum likelihood solution based on EM derived in Chapter 9. Note that results from Appendix B may be used to help answer this exercise. First show that the posterior distribution $q^\star(\boldsymbol{\Lambda}_k)$ of the precisions becomes sharply peaked around the maximum likelihood solution. Do the same for the posterior distribution of the means $q^\star(\boldsymbol{\mu}_k|\boldsymbol{\Lambda}_k)$. Next consider the posterior distribution $q^\star(\boldsymbol{\pi})$ for the mixing coefficients and show that this too becomes sharply peaked around the maximum likelihood solution. Similarly, show that the responsibilities become equal to the corresponding maximum likelihood values for large N, by making use of the following asymptotic result for the digamma function for large x

$$\psi(x) = \ln x + O\left(1/x\right). \tag{10.241}$$

Finally, by making use of (10.80), show that for large N, the predictive distribution becomes a mixture of Gaussians.

10.21 ($\star$) Show that the number of equivalent parameter settings due to interchange symmetries in a mixture model with K components is $K!$.

10.22 ($\star\star$) We have seen that each mode of the posterior distribution in a Gaussian mixture model is a member of a family of $K!$ equivalent modes. Suppose that the result of running the variational inference algorithm is an approximate posterior distribution q that is localized in the neighbourhood of one of the modes. We can then approximate the full posterior distribution as a mixture of $K!$ such q distributions, once centred on each mode and having equal mixing coefficients. Show that if we assume negligible overlap between the components of the q mixture, the resulting lower bound differs from that for a single component q distribution through the addition of an extra term $\ln K!$.

10.23 ($\star\star$) **www** Consider a variational Gaussian mixture model in which there is no prior distribution over mixing coefficients $\{\pi_k\}$. Instead, the mixing coefficients are treated as parameters, whose values are to be found by maximizing the variational lower bound on the log marginal likelihood. Show that maximizing this lower bound with respect to the mixing coefficients, using a Lagrange multiplier to enforce the constraint that the mixing coefficients sum to one, leads to the re-estimation result (10.83). Note that there is no need to consider all of the terms in the lower bound but only the dependence of the bound on the $\{\pi_k\}$.

10.24 ($\star\star$) **www** We have seen in Section 10.2 that the singularities arising in the maximum likelihood treatment of Gaussian mixture models do not arise in a Bayesian treatment. Discuss whether such singularities would arise if the Bayesian model were solved using maximum posterior (MAP) estimation.

10.25 ($\star\star$) The variational treatment of the Bayesian mixture of Gaussians, discussed in Section 10.2, made use of a factorized approximation (10.5) to the posterior distribution. As we saw in Figure 10.2, the factorized assumption causes the variance of the posterior distribution to be under-estimated for certain directions in parameter space. Discuss qualitatively the effect this will have on the variational approximation to the model evidence, and how this effect will vary with the number of components in the mixture. Hence explain whether the variational Gaussian mixture will tend to under-estimate or over-estimate the optimal number of components.

10.26 ($\star\star\star$) Extend the variational treatment of Bayesian linear regression to include a gamma hyperprior $\mathrm{Gam}(\beta|c_0, d_0)$ over β and solve variationally, by assuming a factorized variational distribution of the form $q(\mathbf{w})q(\alpha)q(\beta)$. Derive the variational update equations for the three factors in the variational distribution and also obtain an expression for the lower bound and for the predictive distribution.

10.27 ($\star\star$) By making use of the formulae given in Appendix B show that the variational lower bound for the linear basis function regression model, defined by (10.107), can be written in the form (10.107) with the various terms defined by (10.108)–(10.112).

10.28 ($\star\star\star$) Rewrite the model for the Bayesian mixture of Gaussians, introduced in Section 10.2, as a conjugate model from the exponential family, as discussed in Section 10.4. Hence use the general results (10.115) and (10.119) to derive the specific results (10.48), (10.57), and (10.59).

10.29 ($\star$) **www** Show that the function $f(x) = \ln(x)$ is concave for $0 < x < \infty$ by computing its second derivative. Determine the form of the dual function $g(\lambda)$ defined by (10.133), and verify that minimization of $\lambda x - g(\lambda)$ with respect to λ according to (10.132) indeed recovers the function $\ln(x)$.

10.30 ($\star$) By evaluating the second derivative, show that the log logistic function $f(x) = -\ln(1 + e^{-x})$ is concave. Derive the variational upper bound (10.137) directly by making a second order Taylor expansion of the log logistic function around a point $x = \xi$.

10.31 ($\star\star$) By finding the second derivative with respect to x, show that the function $f(x) = -\ln(e^{x/2} + e^{-x/2})$ is a concave function of x. Now consider the second derivatives with respect to the variable x^2 and hence show that it is a convex function of x^2. Plot graphs of $f(x)$ against x and against x^2. Derive the lower bound (10.144) on the logistic sigmoid function directly by making a first order Taylor series expansion of the function $f(x)$ in the variable x^2 centred on the value ξ^2.

10.32 ($\star\star$) **www** Consider the variational treatment of logistic regression with sequential learning in which data points are arriving one at a time and each must be processed and discarded before the next data point arrives. Show that a Gaussian approximation to the posterior distribution can be maintained through the use of the lower bound (10.151), in which the distribution is initialized using the prior, and as each data point is absorbed its corresponding variational parameter ξ_n is optimized.

10.33 ($\star$) By differentiating the quantity $Q(\boldsymbol{\xi}, \boldsymbol{\xi}^{\text{old}})$ defined by (10.161) with respect to the variational parameter ξ_n show that the update equation for ξ_n for the Bayesian logistic regression model is given by (10.163).

10.34 ($\star\star$) In this exercise we derive re-estimation equations for the variational parameters $\boldsymbol{\xi}$ in the Bayesian logistic regression model of Section 4.5 by direct maximization of the lower bound given by (10.164). To do this set the derivative of $\mathcal{L}(\boldsymbol{\xi})$ with respect to ξ_n equal to zero, making use of the result (3.117) for the derivative of the log of a determinant, together with the expressions (10.157) and (10.158) which define the mean and covariance of the variational posterior distribution $q(\mathbf{w})$.

10.35 ($\star\star$) Derive the result (10.164) for the lower bound $\mathcal{L}(\boldsymbol{\xi})$ in the variational logistic regression model. This is most easily done by substituting the expressions for the Gaussian prior $q(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{m}_0, \mathbf{S}_0)$, together with the lower bound $h(\mathbf{w}, \boldsymbol{\xi})$ on the likelihood function, into the integral (10.159) which defines $\mathcal{L}(\boldsymbol{\xi})$. Next gather together the terms which depend on $\mathbf{w}$ in the exponential and complete the square to give a Gaussian integral, which can then be evaluated by invoking the standard result for the normalization coefficient of a multivariate Gaussian. Finally take the logarithm to obtain (10.164).

10.36 ($\star\star$) Consider the ADF approximation scheme discussed in Section 10.7, and show that inclusion of the factor $f_j(\boldsymbol{\theta})$ leads to an update of the model evidence of the form

$$p_j(\mathcal{D}) \simeq p_{j-1}(\mathcal{D}) Z_j \tag{10.242}$$

where Z_j is the normalization constant defined by (10.197). By applying this result recursively, and initializing with $p_0(\mathcal{D}) = 1$, derive the result

$$p(\mathcal{D}) \simeq \prod_j Z_j. \tag{10.243}$$

10.37 ($\star$) **WWW** Consider the expectation propagation algorithm from Section 10.7, and suppose that one of the factors $f_0(\boldsymbol{\theta})$ in the definition (10.188) has the same exponential family functional form as the approximating distribution $q(\boldsymbol{\theta})$. Show that if the factor $\widetilde{f}_0(\boldsymbol{\theta})$ is initialized to be $f_0(\boldsymbol{\theta})$, then an EP update to refine $\widetilde{f}_0(\boldsymbol{\theta})$ leaves $\widetilde{f}_0(\boldsymbol{\theta})$ unchanged. This situation typically arises when one of the factors is the prior $p(\boldsymbol{\theta})$, and so we see that the prior factor can be incorporated once exactly and does not need to be refined.

10.38 ($\star\star\star$) In this exercise and the next, we shall verify the results (10.214)–(10.224) for the expectation propagation algorithm applied to the clutter problem. Begin by using the division formula (10.205) to derive the expressions (10.214) and (10.215) by completing the square inside the exponential to identify the mean and variance. Also, show that the normalization constant Z_n, defined by (10.206), is given for the clutter problem by (10.216). This can be done by making use of the general result (2.115).

10.39 ($\star\star\star$) Show that the mean and variance of $q^{\text{new}}(\boldsymbol{\theta})$ for EP applied to the clutter problem are given by (10.217) and (10.218). To do this, first prove the following results for the expectations of $\boldsymbol{\theta}$ and $\boldsymbol{\theta}\boldsymbol{\theta}^{\mathrm{T}}$ under $q^{\text{new}}(\boldsymbol{\theta})$

$$\mathbb{E}[\boldsymbol{\theta}] = \mathbf{m}^{\backslash n} + v^{\backslash n}\nabla_{\mathbf{m}^{\backslash n}} \ln Z_n \tag{10.244}$$

$$\mathbb{E}[\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{\theta}] = 2(v^{\backslash n})^2\nabla_{v^{\backslash n}} \ln Z_n + 2\mathbb{E}[\boldsymbol{\theta}]^{\mathrm{T}}\mathbf{m}^{\backslash n} - \|\mathbf{m}^{\backslash n}\|^2 \tag{10.245}$$

and then make use of the result (10.216) for Z_n. Next, prove the results (10.220)–(10.222) by using (10.207) and completing the square in the exponential. Finally, use (10.208) to derive the result (10.223).

11 Sampling Methods

For most probabilistic models of practical interest, exact inference is intractable, and so we have to resort to some form of approximation. In Chapter 10, we discussed inference algorithms based on deterministic approximations, which include methods such as variational Bayes and expectation propagation. Here we consider approximate inference methods based on numerical sampling, also known as *Monte Carlo* techniques.

Although for some applications the posterior distribution over unobserved variables will be of direct interest in itself, for most situations the posterior distribution is required primarily for the purpose of evaluating expectations, for example in order to make predictions. The fundamental problem that we therefore wish to address in this chapter involves finding the expectation of some function $f(\mathbf{z})$ with respect to a probability distribution $p(\mathbf{z})$. Here, the components of $\mathbf{z}$ might comprise discrete or continuous variables or some combination of the two. Thus in the case of continuous

Figure 11.1 Schematic illustration of a function $f(z)$ whose expectation is to be evaluated with respect to a distribution $p(z)$.

variables, we wish to evaluate the expectation

$$\mathbb{E}[f] = \int f(\mathbf{z})p(\mathbf{z})\,\mathrm{d}\mathbf{z} \tag{11.1}$$

where the integral is replaced by summation in the case of discrete variables. This is illustrated schematically for a single continuous variable in Figure 11.1. We shall suppose that such expectations are too complex to be evaluated exactly using analytical techniques.

The general idea behind sampling methods is to obtain a set of samples $\mathbf{z}^{(l)}$ (where $l = 1, \ldots, L$) drawn independently from the distribution $p(\mathbf{z})$. This allows the expectation (11.1) to be approximated by a finite sum

$$\widehat{f} = \frac{1}{L}\sum_{l=1}^{L} f(\mathbf{z}^{(l)}). \tag{11.2}$$

As long as the samples $\mathbf{z}^{(l)}$ are drawn from the distribution $p(\mathbf{z})$, then $\mathbb{E}[\widehat{f}] = \mathbb{E}[f]$ and so the estimator $\widehat{f}$ has the correct mean. The variance of the estimator is given by

Exercise 11.1

$$\mathrm{var}[\widehat{f}] = \frac{1}{L}\mathbb{E}\left[(f - \mathbb{E}[f])^2\right] \tag{11.3}$$

is the variance of the function $f(\mathbf{z})$ under the distribution $p(\mathbf{z})$. It is worth emphasizing that the accuracy of the estimator therefore does not depend on the dimensionality of $\mathbf{z}$, and that, in principle, high accuracy may be achievable with a relatively small number of samples $\mathbf{z}^{(l)}$. In practice, ten or twenty independent samples may suffice to estimate an expectation to sufficient accuracy.

The problem, however, is that the samples $\{\mathbf{z}^{(l)}\}$ might not be independent, and so the effective sample size might be much smaller than the apparent sample size. Also, referring back to Figure 11.1, we note that if $f(\mathbf{z})$ is small in regions where $p(\mathbf{z})$ is large, and vice versa, then the expectation may be dominated by regions of small probability, implying that relatively large sample sizes will be required to achieve sufficient accuracy.

For many models, the joint distribution $p(\mathbf{z})$ is conveniently specified in terms of a graphical model. In the case of a directed graph with no observed variables, it is

straightforward to sample from the joint distribution (assuming that it is possible to sample from the conditional distributions at each node) using the following *ancestral sampling* approach, discussed briefly in Section 8.1.2. The joint distribution is specified by

$$p(\mathbf{z}) = \prod_{i=1}^{M} p(\mathbf{z}_i|\mathrm{pa}_i) \tag{11.4}$$

where $\mathbf{z}_i$ are the set of variables associated with node i, and pa_i denotes the set of variables associated with the parents of node i. To obtain a sample from the joint distribution, we make one pass through the set of variables in the order $\mathbf{z}_1, \ldots, \mathbf{z}_M$ sampling from the conditional distributions $p(\mathbf{z}_i|\mathrm{pa}_i)$. This is always possible because at each step all of the parent values will have been instantiated. After one pass through the graph, we will have obtained a sample from the joint distribution.

Now consider the case of a directed graph in which some of the nodes are instantiated with observed values. We can in principle extend the above procedure, at least in the case of nodes representing discrete variables, to give the following *logic sampling* approach (Henrion, 1988), which can be seen as a special case of *importance sampling* discussed in Section 11.1.4. At each step, when a sampled value is obtained for a variable $\mathbf{z}_i$ whose value is observed, the sampled value is compared to the observed value, and if they agree then the sample value is retained and the algorithm proceeds to the next variable in turn. However, if the sampled value and the observed value disagree, then the whole sample so far is discarded and the algorithm starts again with the first node in the graph. This algorithm samples correctly from the posterior distribution because it corresponds simply to drawing samples from the joint distribution of hidden variables and data variables and then discarding those samples that disagree with the observed data (with the slight saving of not continuing with the sampling from the joint distribution as soon as one contradictory value is observed). However, the overall probability of accepting a sample from the posterior decreases rapidly as the number of observed variables increases and as the number of states that those variables can take increases, and so this approach is rarely used in practice.

In the case of probability distributions defined by an undirected graph, there is no one-pass sampling strategy that will sample even from the prior distribution with no observed variables. Instead, computationally more expensive techniques must be employed, such as Gibbs sampling, which is discussed in Section 11.3.

As well as sampling from conditional distributions, we may also require samples from a marginal distribution. If we already have a strategy for sampling from a joint distribution $p(\mathbf{u}, \mathbf{v})$, then it is straightforward to obtain samples from the marginal distribution $p(\mathbf{u})$ simply by ignoring the values for $\mathbf{v}$ in each sample.

There are numerous texts dealing with Monte Carlo methods. Those of particular interest from the statistical inference perspective include Chen *et al.* (2001), Gamerman (1997), Gilks *et al.* (1996), Liu (2001), Neal (1996), and Robert and Casella (1999). Also there are review articles by Besag *et al.* (1995), Brooks (1998), Diaconis and Saloff-Coste (1998), Jerrum and Sinclair (1996), Neal (1993), Tierney (1994), and Andrieu *et al.* (2003) that provide additional information on sampling

methods for statistical inference.

Diagnostic tests for convergence of Markov chain Monte Carlo algorithms are summarized in Robert and Casella (1999), and some practical guidance on the use of sampling methods in the context of machine learning is given in Bishop and Nabney (2008).

11.1. Basic Sampling Algorithms

In this section, we consider some simple strategies for generating random samples from a given distribution. Because the samples will be generated by a computer algorithm they will in fact be *pseudo-random* numbers, that is, they will be deterministically calculated, but must nevertheless pass appropriate tests for randomness. Generating such numbers raises several subtleties (Press *et al.*, 1992) that lie outside the scope of this book. Here we shall assume that an algorithm has been provided that generates pseudo-random numbers distributed uniformly over $(0, 1)$, and indeed most software environments have such a facility built in.

11.1.1 Standard distributions

We first consider how to generate random numbers from simple nonuniform distributions, assuming that we already have available a source of uniformly distributed random numbers. Suppose that z is uniformly distributed over the interval $(0, 1)$, and that we transform the values of z using some function $f(\cdot)$ so that $y = f(z)$. The distribution of y will be governed by

$$p(y) = p(z) \left| \frac{dz}{dy} \right| \tag{11.5}$$

where, in this case, $p(z) = 1$. Our goal is to choose the function $f(z)$ such that the resulting values of y have some specific desired distribution $p(y)$. Integrating (11.5) we obtain

$$z = h(y) \equiv \int_{-\infty}^{y} p(\widehat{y})\, \mathrm{d}\widehat{y} \tag{11.6}$$

Exercise 11.2

which is the indefinite integral of $p(y)$. Thus, $y = h^{-1}(z)$, and so we have to transform the uniformly distributed random numbers using a function which is the inverse of the indefinite integral of the desired distribution. This is illustrated in Figure 11.2.

Consider for example the *exponential distribution*

$$p(y) = \lambda \exp(-\lambda y) \tag{11.7}$$

where $0 \leqslant y < \infty$. In this case the lower limit of the integral in (11.6) is 0, and so $h(y) = 1 - \exp(-\lambda y)$. Thus, if we transform our uniformly distributed variable z using $y = -\lambda^{-1} \ln(1 - z)$, then y will have an exponential distribution.

Figure 11.2 Geometrical interpretation of the transformation method for generating nonuniformly distributed random numbers. $h(y)$ is the indefinite integral of the desired distribution $p(y)$. If a uniformly distributed random variable z is transformed using $y = h^{-1}(z)$, then y will be distributed according to $p(y)$.

Another example of a distribution to which the transformation method can be applied is given by the Cauchy distribution

$$p(y) = \frac{1}{\pi}\frac{1}{1+y^2}. \tag{11.8}$$

Exercise 11.3 In this case, the inverse of the indefinite integral can be expressed in terms of the 'tan' function.

The generalization to multiple variables is straightforward and involves the Jacobian of the change of variables, so that

$$p(y_1, \ldots, y_M) = p(z_1, \ldots, z_M)\left|\frac{\partial(z_1, \ldots, z_M)}{\partial(y_1, \ldots, y_M)}\right|. \tag{11.9}$$

As a final example of the transformation method we consider the Box-Muller method for generating samples from a Gaussian distribution. First, suppose we generate pairs of uniformly distributed random numbers $z_1, z_2 \in (-1, 1)$, which we can do by transforming a variable distributed uniformly over $(0, 1)$ using $z \to 2z - 1$. Next we discard each pair unless it satisfies $z_1^2 + z_2^2 \leqslant 1$. This leads to a uniform distribution of points inside the unit circle with $p(z_1, z_2) = 1/\pi$, as illustrated in Figure 11.3. Then, for each pair z_1, z_2 we evaluate the quantities

Figure 11.3 The Box-Muller method for generating Gaussian distributed random numbers starts by generating samples from a uniform distribution inside the unit circle.

$$y_1 = z_1 \left(\frac{-2 \ln z_1}{r^2} \right)^{1/2} \tag{11.10}$$

$$y_2 = z_2 \left(\frac{-2 \ln z_2}{r^2} \right)^{1/2} \tag{11.11}$$

Exercise 11.4 where $r^2 = z_1^2 + z_2^2$. Then the joint distribution of y_1 and y_2 is given by

$$\begin{aligned} p(y_1, y_2) &= p(z_1, z_2) \left| \frac{\partial(z_1, z_2)}{\partial(y_1, y_2)} \right| \\ &= \left[\frac{1}{\sqrt{2\pi}} \exp(-y_1^2/2) \right] \left[\frac{1}{\sqrt{2\pi}} \exp(-y_2^2/2) \right] \end{aligned} \tag{11.12}$$

and so y_1 and y_2 are independent and each has a Gaussian distribution with zero mean and unit variance.

If y has a Gaussian distribution with zero mean and unit variance, then $\sigma y + \mu$ will have a Gaussian distribution with mean μ and variance σ^2. To generate vector-valued variables having a multivariate Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$, we can make use of the *Cholesky decomposition*, which takes the form $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$ (Press *et al.*, 1992). Then, if $\mathbf{z}$ is a vector valued random variable whose components are independent and Gaussian distributed with zero mean and unit variance, then $\mathbf{y} = \boldsymbol{\mu} + \mathbf{L}\mathbf{z}$ will have mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$.

Exercise 11.5

Obviously, the transformation technique depends for its success on the ability to calculate and then invert the indefinite integral of the required distribution. Such operations will only be feasible for a limited number of simple distributions, and so we must turn to alternative approaches in search of a more general strategy. Here we consider two techniques called *rejection sampling* and *importance sampling*. Although mainly limited to univariate distributions and thus not directly applicable to complex problems in many dimensions, they do form important components in more general strategies.

11.1.2 Rejection sampling

The rejection sampling framework allows us to sample from relatively complex distributions, subject to certain constraints. We begin by considering univariate distributions and discuss the extension to multiple dimensions subsequently.

Suppose we wish to sample from a distribution $p(\mathbf{z})$ that is not one of the simple, standard distributions considered so far, and that sampling directly from $p(\mathbf{z})$ is difficult. Furthermore suppose, as is often the case, that we are easily able to evaluate $p(\mathbf{z})$ for any given value of $\mathbf{z}$, up to some normalizing constant Z, so that

$$p(z) = \frac{1}{Z_p} \widetilde{p}(z) \tag{11.13}$$

where $\widetilde{p}(z)$ can readily be evaluated, but Z_p is unknown.

In order to apply rejection sampling, we need some simpler distribution $q(z)$, sometimes called a *proposal distribution*, from which we can readily draw samples.

Figure 11.4 In the rejection sampling method, samples are drawn from a simple distribution $q(z)$ and rejected if they fall in the grey area between the unnormalized distribution $\widetilde{p}(z)$ and the scaled distribution $kq(z)$. The resulting samples are distributed according to $p(z)$, which is the normalized version of $\widetilde{p}(z)$.

We next introduce a constant k whose value is chosen such that $kq(z) \geqslant \widetilde{p}(z)$ for all values of z. The function $kq(z)$ is called the comparison function and is illustrated for a univariate distribution in Figure 11.4. Each step of the rejection sampler involves generating two random numbers. First, we generate a number z_0 from the distribution $q(z)$. Next, we generate a number u_0 from the uniform distribution over $[0, kq(z_0)]$. This pair of random numbers has uniform distribution under the curve of the function $kq(z)$. Finally, if $u_0 > \widetilde{p}(z_0)$ then the sample is rejected, otherwise u_0 is retained. Thus the pair is rejected if it lies in the grey shaded region in Figure 11.4. The remaining pairs then have uniform distribution under the curve of $\widetilde{p}(z)$, and hence the corresponding z values are distributed according to $p(z)$, as desired.

Exercise 11.6

The original values of z are generated from the distribution $q(z)$, and these samples are then accepted with probability $\widetilde{p}(z)/kq(z)$, and so the probability that a sample will be accepted is given by

$$\begin{aligned} p(\text{accept}) &= \int \{\widetilde{p}(z)/kq(z)\}\, q(z)\, \mathrm{d}z \\ &= \frac{1}{k} \int \widetilde{p}(z)\, \mathrm{d}z. \end{aligned} \tag{11.14}$$

Thus the fraction of points that are rejected by this method depends on the ratio of the area under the unnormalized distribution $\widetilde{p}(z)$ to the area under the curve $kq(z)$. We therefore see that the constant k should be as small as possible subject to the limitation that $kq(z)$ must be nowhere less than $\widetilde{p}(z)$.

As an illustration of the use of rejection sampling, consider the task of sampling from the gamma distribution

$$\mathrm{Gam}(z|a, b) = \frac{b^a z^{a-1} \exp(-bz)}{\Gamma(a)} \tag{11.15}$$

which, for $a > 1$, has a bell-shaped form, as shown in Figure 11.5. A suitable proposal distribution is therefore the Cauchy (11.8) because this too is bell-shaped and because we can use the transformation method, discussed earlier, to sample from it. We need to generalize the Cauchy slightly to ensure that it nowhere has a smaller value than the gamma distribution. This can be achieved by transforming a uniform random variable y using $z = b \tan y + c$, which gives random numbers distributed according to.

Exercise 11.7

Figure 11.5 Plot showing the gamma distribution given by (11.15) as the green curve, with a scaled Cauchy proposal distribution shown by the red curve. Samples from the gamma distribution can be obtained by sampling from the Cauchy and then applying the rejection sampling criterion.

$$q(z) = \frac{k}{1 + (z - c)^2/b^2}. \tag{11.16}$$

The minimum reject rate is obtained by setting $c = a - 1$, $b^2 = 2a - 1$ and choosing the constant k to be as small as possible while still satisfying the requirement $kq(z) \geqslant \widetilde{p}(z)$. The resulting comparison function is also illustrated in Figure 11.5.

11.1.3 Adaptive rejection sampling

In many instances where we might wish to apply rejection sampling, it proves difficult to determine a suitable analytic form for the envelope distribution $q(z)$. An alternative approach is to construct the envelope function on the fly based on measured values of the distribution $p(z)$ (Gilks and Wild, 1992). Construction of an envelope function is particularly straightforward for cases in which $p(z)$ is log concave, in other words when $\ln p(z)$ has derivatives that are nonincreasing functions of z. The construction of a suitable envelope function is illustrated graphically in Figure 11.6.

The function $\ln p(z)$ and its gradient are evaluated at some initial set of grid points, and the intersections of the resulting tangent lines are used to construct the envelope function. Next a sample value is drawn from the envelope distribution. This is straightforward because the log of the envelope distribution is a succession

Exercise 11.9

Figure 11.6 In the case of distributions that are log concave, an envelope function for use in rejection sampling can be constructed using the tangent lines computed at a set of grid points. If a sample point is rejected, it is added to the set of grid points and used to refine the envelope distribution.

Figure 11.7 Illustrative example of rejection sampling involving sampling from a Gaussian distribution $p(z)$ shown by the green curve, by using rejection sampling from a proposal distribution $q(z)$ that is also Gaussian and whose scaled version $kq(z)$ is shown by the red curve.

of linear functions, and hence the envelope distribution itself comprises a piecewise exponential distribution of the form

$$q(z) = k_i \lambda_i \exp\{-\lambda_i(z - z_{i-1})\} \qquad z_{i-1} < z \leqslant z_i. \tag{11.17}$$

Once a sample has been drawn, the usual rejection criterion can be applied. If the sample is accepted, then it will be a draw from the desired distribution. If, however, the sample is rejected, then it is incorporated into the set of grid points, a new tangent line is computed, and the envelope function is thereby refined. As the number of grid points increases, so the envelope function becomes a better approximation of the desired distribution $p(z)$ and the probability of rejection decreases.

A variant of the algorithm exists that avoids the evaluation of derivatives (Gilks, 1992). The adaptive rejection sampling framework can also be extended to distributions that are not log concave, simply by following each rejection sampling step with a Metropolis-Hastings step (to be discussed in Section 11.2.2), giving rise to *adaptive rejection Metropolis* sampling (Gilks *et al.*, 1995).

Clearly for rejection sampling to be of practical value, we require that the comparison function be close to the required distribution so that the rate of rejection is kept to a minimum. Now let us examine what happens when we try to use rejection sampling in spaces of high dimensionality. Consider, for the sake of illustration, a somewhat artificial problem in which we wish to sample from a zero-mean multivariate Gaussian distribution with covariance $\sigma_p^2\mathbf{I}$, where $\mathbf{I}$ is the unit matrix, by rejection sampling from a proposal distribution that is itself a zero-mean Gaussian distribution having covariance $\sigma_q^2\mathbf{I}$. Obviously, we must have $\sigma_q^2 \geqslant \sigma_p^2$ in order that there exists a k such that $kq(z) \geqslant p(z)$. In D-dimensions the optimum value of k is given by $k = (\sigma_q/\sigma_p)^D$, as illustrated for $D = 1$ in Figure 11.7. The acceptance rate will be the ratio of volumes under $p(z)$ and $kq(z)$, which, because both distributions are normalized, is just $1/k$. Thus the acceptance rate diminishes exponentially with dimensionality. Even if σ_q exceeds σ_p by just one percent, for $D = 1,000$ the acceptance ratio will be approximately $1/20,000$. In this illustrative example the comparison function is close to the required distribution. For more practical examples, where the desired distribution may be multimodal and sharply peaked, it will be extremely difficult to find a good proposal distribution and comparison function.

Figure 11.8 Importance sampling addresses the problem of evaluating the expectation of a function $f(z)$ with respect to a distribution $p(z)$ from which it is difficult to draw samples directly. Instead, samples $\{z^{(l)}\}$ are drawn from a simpler distribution $q(z)$, and the corresponding terms in the summation are weighted by the ratios $p(z^{(l)})/q(z^{(l)})$.

Furthermore, the exponential decrease of acceptance rate with dimensionality is a generic feature of rejection sampling. Although rejection can be a useful technique in one or two dimensions it is unsuited to problems of high dimensionality. It can, however, play a role as a subroutine in more sophisticated algorithms for sampling in high dimensional spaces.

11.1.4 Importance sampling

One of the principal reasons for wishing to sample from complicated probability distributions is to be able to evaluate expectations of the form (11.1). The technique of *importance sampling* provides a framework for approximating expectations directly but does not itself provide a mechanism for drawing samples from distribution $p(\mathbf{z})$.

The finite sum approximation to the expectation, given by (11.2), depends on being able to draw samples from the distribution $p(\mathbf{z})$. Suppose, however, that it is impractical to sample directly from $p(\mathbf{z})$ but that we can evaluate $p(\mathbf{z})$ easily for any given value of $\mathbf{z}$. One simplistic strategy for evaluating expectations would be to discretize $\mathbf{z}$-space into a uniform grid and to evaluate the integrand as a sum of the form

$$\mathbb{E}[f] \simeq \sum_{l=1}^{L} p(\mathbf{z}^{(l)}) f(\mathbf{z}^{(l)}). \tag{11.18}$$

An obvious problem with this approach is that the number of terms in the summation grows exponentially with the dimensionality of $\mathbf{z}$. Furthermore, as we have already noted, the kinds of probability distributions of interest will often have much of their mass confined to relatively small regions of $\mathbf{z}$ space and so uniform sampling will be very inefficient because in high-dimensional problems, only a very small proportion of the samples will make a significant contribution to the sum. We would really like to choose the sample points to fall in regions where $p(\mathbf{z})$ is large, or ideally where the product $p(\mathbf{z})f(\mathbf{z})$ is large.

As in the case of rejection sampling, importance sampling is based on the use of a proposal distribution $q(\mathbf{z})$ from which it is easy to draw samples, as illustrated in Figure 11.8. We can then express the expectation in the form of a finite sum over

samples $\{\mathbf{z}^{(l)}\}$ drawn from $q(\mathbf{z})$

$$\begin{aligned}\mathbb{E}[f] &= \int f(\mathbf{z})p(\mathbf{z})\,\mathrm{d}\mathbf{z} \\ &= \int f(\mathbf{z})\frac{p(\mathbf{z})}{q(\mathbf{z})}q(\mathbf{z})\,\mathrm{d}\mathbf{z} \\ &\simeq \frac{1}{L}\sum_{l=1}^{L}\frac{p(\mathbf{z}^{(l)})}{q(\mathbf{z}^{(l)})}f(\mathbf{z}^{(l)}).\end{aligned} \tag{11.19}$$

The quantities $r_l = p(\mathbf{z}^{(l)})/q(\mathbf{z}^{(l)})$ are known as *importance weights*, and they correct the bias introduced by sampling from the wrong distribution. Note that, unlike rejection sampling, all of the samples generated are retained.

It will often be the case that the distribution $p(\mathbf{z})$ can only be evaluated up to a normalization constant, so that $p(\mathbf{z}) = \widetilde{p}(\mathbf{z})/Z_p$ where $\widetilde{p}(\mathbf{z})$ can be evaluated easily, whereas Z_p is unknown. Similarly, we may wish to use an importance sampling distribution $q(\mathbf{z}) = \widetilde{q}(\mathbf{z})/Z_q$, which has the same property. We then have

$$\begin{aligned}\mathbb{E}[f] &= \int f(\mathbf{z})p(\mathbf{z})\,\mathrm{d}\mathbf{z} \\ &= \frac{Z_q}{Z_p}\int f(\mathbf{z})\frac{\widetilde{p}(\mathbf{z})}{\widetilde{q}(\mathbf{z})}q(\mathbf{z})\,\mathrm{d}\mathbf{z} \\ &\simeq \frac{Z_q}{Z_p}\frac{1}{L}\sum_{l=1}^{L}\widetilde{r}_l f(\mathbf{z}^{(l)}).\end{aligned} \tag{11.20}$$

where $\widetilde{r}_l = \widetilde{p}(\mathbf{z}^{(l)})/\widetilde{q}(\mathbf{z}^{(l)})$. We can use the same sample set to evaluate the ratio Z_p/Z_q with the result

$$\begin{aligned}\frac{Z_p}{Z_q} &= \frac{1}{Z_q}\int \widetilde{p}(\mathbf{z})\,\mathrm{d}\mathbf{z} = \int \frac{\widetilde{p}(\mathbf{z})}{\widetilde{q}(\mathbf{z})}q(\mathbf{z})\,\mathrm{d}\mathbf{z} \\ &\simeq \frac{1}{L}\sum_{l=1}^{L}\widetilde{r}_l\end{aligned} \tag{11.21}$$

and hence

$$\mathbb{E}[f] \simeq \sum_{l=1}^{L} w_l f(\mathbf{z}^{(l)}) \tag{11.22}$$

where we have defined

$$w_l = \frac{\widetilde{r}_l}{\sum_m \widetilde{r}_m} = \frac{\widetilde{p}(\mathbf{z}^{(l)})/q(\mathbf{z}^{(l)})}{\sum_m \widetilde{p}(\mathbf{z}^{(m)})/q(\mathbf{z}^{(m)})}. \tag{11.23}$$

As with rejection sampling, the success of the importance sampling approach depends crucially on how well the sampling distribution $q(\mathbf{z})$ matches the desired

distribution $p(\mathbf{z})$. If, as is often the case, $p(\mathbf{z})f(\mathbf{z})$ is strongly varying and has a significant proportion of its mass concentrated over relatively small regions of $\mathbf{z}$ space, then the set of importance weights $\{r_l\}$ may be dominated by a few weights having large values, with the remaining weights being relatively insignificant. Thus the effective sample size can be much smaller than the apparent sample size L. The problem is even more severe if none of the samples falls in the regions where $p(\mathbf{z})f(\mathbf{z})$ is large. In that case, the apparent variances of r_l and $r_l f(\mathbf{z}^{(l)})$ may be small even though the estimate of the expectation may be severely wrong. Hence a major drawback of the importance sampling method is the potential to produce results that are arbitrarily in error and with no diagnostic indication. This also highlights a key requirement for the sampling distribution $q(\mathbf{z})$, namely that it should not be small or zero in regions where $p(\mathbf{z})$ may be significant.

For distributions defined in terms of a graphical model, we can apply the importance sampling technique in various ways. For discrete variables, a simple approach is called *uniform sampling*. The joint distribution for a directed graph is defined by (11.4). Each sample from the joint distribution is obtained by first setting those variables $\mathbf{z}_i$ that are in the evidence set equal to their observed values. Each of the remaining variables is then sampled independently from a uniform distribution over the space of possible instantiations. To determine the corresponding weight associated with a sample $\mathbf{z}^{(l)}$, we note that the sampling distribution $\widetilde{q}(\mathbf{z})$ is uniform over the possible choices for $\mathbf{z}$, and that $\widetilde{p}(\mathbf{z}|\mathbf{x}) = \widetilde{p}(\mathbf{z})$, where $\mathbf{x}$ denotes the subset of variables that are observed, and the equality follows from the fact that every sample $\mathbf{z}$ that is generated is necessarily consistent with the evidence. Thus the weights r_l are simply proportional to $p(\mathbf{z})$. Note that the variables can be sampled in any order. This approach can yield poor results if the posterior distribution is far from uniform, as is often the case in practice.

An improvement on this approach is called *likelihood weighted sampling* (Fung and Chang, 1990; Shachter and Peot, 1990) and is based on ancestral sampling of the variables. For each variable in turn, if that variable is in the evidence set, then it is just set to its instantiated value. If it is not in the evidence set, then it is sampled from the conditional distribution $p(\mathbf{z}_i|\mathrm{pa}_i)$ in which the conditioning variables are set to their currently sampled values. The weighting associated with the resulting sample $\mathbf{z}$ is then given by

$$r(\mathbf{z}) = \prod_{\mathbf{z}_i \notin \mathbf{e}} \frac{p(\mathbf{z}_i|\mathrm{pa}_i)}{p(\mathbf{z}_i|\mathrm{pa}_i)} \prod_{\mathbf{z}_i \in \mathbf{e}} \frac{p(\mathbf{z}_i|\mathrm{pa}_i)}{1} = \prod_{\mathbf{z}_i \in \mathbf{e}} p(\mathbf{z}_i|\mathrm{pa}_i). \tag{11.24}$$

This method can be further extended using *self-importance sampling* (Shachter and Peot, 1990) in which the importance sampling distribution is continually updated to reflect the current estimated posterior distribution.

11.1.5 Sampling-importance-resampling

The rejection sampling method discussed in Section 11.1.2 depends in part for its success on the determination of a suitable value for the constant k. For many pairs of distributions $p(\mathbf{z})$ and $q(\mathbf{z})$, it will be impractical to determine a suitable

value for k in that any value that is sufficiently large to guarantee a bound on the desired distribution will lead to impractically small acceptance rates.

As in the case of rejection sampling, the *sampling-importance-resampling* (SIR) approach also makes use of a sampling distribution $q(\mathbf{z})$ but avoids having to determine the constant k. There are two stages to the scheme. In the first stage, L samples $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(L)}$ are drawn from $q(\mathbf{z})$. Then in the second stage, weights $w_1, \ldots, w_L$ are constructed using (11.23). Finally, a second set of L samples is drawn from the discrete distribution $(\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(L)})$ with probabilities given by the weights $(w_1, \ldots, w_L)$.

The resulting L samples are only approximately distributed according to $p(\mathbf{z})$, but the distribution becomes correct in the limit $L \to \infty$. To see this, consider the univariate case, and note that the cumulative distribution of the resampled values is given by

$$\begin{aligned} p(z \leqslant a) &= \sum_{l: z^{(l)} \leqslant a} w_l \\ &= \frac{\sum_l I(z^{(l)} \leqslant a) \widetilde{p}(z^{(l)})/q(z^{(l)})}{\sum_l \widetilde{p}(z^{(l)})/q(z^{(l)})} \end{aligned} \tag{11.25}$$

where $I(.)$ is the indicator function (which equals 1 if its argument is true and 0 otherwise). Taking the limit $L \to \infty$, and assuming suitable regularity of the distributions, we can replace the sums by integrals weighted according to the original sampling distribution $q(z)$

$$\begin{aligned} p(z \leqslant a) &= \frac{\displaystyle\int I(z \leqslant a) \left\{\widetilde{p}(z)/q(z)\right\} q(z)\, \mathrm{d}z}{\displaystyle\int \left\{\widetilde{p}(z)/q(z)\right\} q(z)\, \mathrm{d}z} \\ &= \frac{\displaystyle\int I(z \leqslant a) \widetilde{p}(z)\, \mathrm{d}z}{\displaystyle\int \widetilde{p}(z)\, \mathrm{d}z} \\ &= \int I(z \leqslant a) p(z)\, \mathrm{d}z \end{aligned} \tag{11.26}$$

which is the cumulative distribution function of $p(z)$. Again, we see that the normalization of $p(z)$ is not required.

For a finite value of L, and a given initial sample set, the resampled values will only approximately be drawn from the desired distribution. As with rejection sampling, the approximation improves as the sampling distribution $q(\mathbf{z})$ gets closer to the desired distribution $p(\mathbf{z})$. When $q(\mathbf{z}) = p(\mathbf{z})$, the initial samples $(\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(L)})$ have the desired distribution, and the weights $w_n = 1/L$ so that the resampled values also have the desired distribution.

If moments with respect to the distribution $p(\mathbf{z})$ are required, then they can be

evaluated directly using the original samples together with the weights, because

$$\begin{aligned}\mathbb{E}[f(\mathbf{z})] &= \int f(\mathbf{z})p(\mathbf{z})\,\mathrm{d}\mathbf{z} \\ &= \frac{\displaystyle\int f(\mathbf{z})[\widetilde{p}(\mathbf{z})/q(\mathbf{z})]q(\mathbf{z})\,\mathrm{d}\mathbf{z}}{\displaystyle\int [\widetilde{p}(\mathbf{z})/q(\mathbf{z})]q(\mathbf{z})\,\mathrm{d}\mathbf{z}} \\ &\simeq \sum_{l=1}^{L} w_l f(\mathbf{z}_l). \end{aligned} \tag{11.27}$$

11.1.6 Sampling and the EM algorithm

In addition to providing a mechanism for direct implementation of the Bayesian framework, Monte Carlo methods can also play a role in the frequentist paradigm, for example to find maximum likelihood solutions. In particular, sampling methods can be used to approximate the E step of the EM algorithm for models in which the E step cannot be performed analytically. Consider a model with hidden variables $\mathbf{Z}$, visible (observed) variables $\mathbf{X}$, and parameters $\boldsymbol{\theta}$. The function that is optimized with respect to $\boldsymbol{\theta}$ in the M step is the expected complete-data log likelihood, given by

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\mathrm{old}}) = \int p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\mathrm{old}}) \ln p(\mathbf{Z}, \mathbf{X}|\boldsymbol{\theta})\,\mathrm{d}\mathbf{Z}. \tag{11.28}$$

We can use sampling methods to approximate this integral by a finite sum over samples $\{\mathbf{Z}^{(l)}\}$, which are drawn from the current estimate for the posterior distribution $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\mathrm{old}})$, so that

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\mathrm{old}}) \simeq \frac{1}{L}\sum_{l=1}^{L} \ln p(\mathbf{Z}^{(l)}, \mathbf{X}|\boldsymbol{\theta}). \tag{11.29}$$

The Q function is then optimized in the usual way in the M step. This procedure is called the *Monte Carlo EM algorithm*.

It is straightforward to extend this to the problem of finding the mode of the posterior distribution over $\boldsymbol{\theta}$ (the MAP estimate) when a prior distribution $p(\boldsymbol{\theta})$ has been defined, simply by adding $\ln p(\boldsymbol{\theta})$ to the function $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\mathrm{old}})$ before performing the M step.

A particular instance of the Monte Carlo EM algorithm, called *stochastic EM*, arises if we consider a finite mixture model, and draw just one sample at each E step. Here the latent variable $\mathbf{Z}$ characterizes which of the K components of the mixture is responsible for generating each data point. In the E step, a sample of $\mathbf{Z}$ is taken from the posterior distribution $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\mathrm{old}})$ where $\mathbf{X}$ is the data set. This effectively makes a hard assignment of each data point to one of the components in the mixture. In the M step, this sampled approximation to the posterior distribution is used to update the model parameters in the usual way.

Now suppose we move from a maximum likelihood approach to a full Bayesian treatment in which we wish to sample from the posterior distribution over the parameter vector $\boldsymbol{\theta}$. In principle, we would like to draw samples from the joint posterior $p(\boldsymbol{\theta}, \mathbf{Z}|\mathbf{X})$, but we shall suppose that this is computationally difficult. Suppose further that it is relatively straightforward to sample from the complete-data parameter posterior $p(\boldsymbol{\theta}|\mathbf{Z}, \mathbf{X})$. This inspires the *data augmentation* algorithm, which alternates between two steps known as the I-step (imputation step, analogous to an E step) and the P-step (posterior step, analogous to an M step).

IP Algorithm

I-step. We wish to sample from $p(\mathbf{Z}|\mathbf{X})$ but we cannot do this directly. We therefore note the relation

$$p(\mathbf{Z}|\mathbf{X}) = \int p(\mathbf{Z}|\boldsymbol{\theta}, \mathbf{X}) p(\boldsymbol{\theta}|\mathbf{X}) \, \mathrm{d}\boldsymbol{\theta} \tag{11.30}$$

and hence for $l = 1, \ldots, L$ we first draw a sample $\boldsymbol{\theta}^{(l)}$ from the current estimate for $p(\boldsymbol{\theta}|\mathbf{X})$, and then use this to draw a sample $\mathbf{Z}^{(l)}$ from $p(\mathbf{Z}|\boldsymbol{\theta}^{(l)}, \mathbf{X})$.

P-step. Given the relation

$$p(\boldsymbol{\theta}|\mathbf{X}) = \int p(\boldsymbol{\theta}|\mathbf{Z}, \mathbf{X}) p(\mathbf{Z}|\mathbf{X}) \, \mathrm{d}\mathbf{Z} \tag{11.31}$$

we use the samples $\{\mathbf{Z}^{(l)}\}$ obtained from the I-step to compute a revised estimate of the posterior distribution over $\boldsymbol{\theta}$ given by

$$p(\boldsymbol{\theta}|\mathbf{X}) \simeq \frac{1}{L} \sum_{l=1}^{L} p(\boldsymbol{\theta}|\mathbf{Z}^{(l)}, \mathbf{X}). \tag{11.32}$$

By assumption, it will be feasible to sample from this approximation in the I-step.

Note that we are making a (somewhat artificial) distinction between parameters $\boldsymbol{\theta}$ and hidden variables $\mathbf{Z}$. From now on, we blur this distinction and focus simply on the problem of drawing samples from a given posterior distribution.

11.2. Markov Chain Monte Carlo

In the previous section, we discussed the rejection sampling and importance sampling strategies for evaluating expectations of functions, and we saw that they suffer from severe limitations particularly in spaces of high dimensionality. We therefore turn in this section to a very general and powerful framework called Markov chain Monte Carlo (MCMC), which allows sampling from a large class of distributions,

and which scales well with the dimensionality of the sample space. Markov chain Monte Carlo methods have their origins in physics (Metropolis and Ulam, 1949), and it was only towards the end of the 1980s that they started to have a significant impact in the field of statistics.

As with rejection and importance sampling, we again sample from a proposal distribution. This time, however, we maintain a record of the current state $\mathbf{z}^{(\tau)}$, and the proposal distribution $q(\mathbf{z}|\mathbf{z}^{(\tau)})$ depends on this current state, and so the sequence of samples $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \ldots$ forms a Markov chain. Again, if we write $p(\mathbf{z}) = \widetilde{p}(\mathbf{z})/Z_p$, we will assume that $\widetilde{p}(\mathbf{z})$ can readily be evaluated for any given value of $\mathbf{z}$, although the value of Z_p may be unknown. The proposal distribution itself is chosen to be sufficiently simple that it is straightforward to draw samples from it directly. At each cycle of the algorithm, we generate a candidate sample $\mathbf{z}^\star$ from the proposal distribution and then accept the sample according to an appropriate criterion.

Section 11.2.1

In the basic *Metropolis* algorithm (Metropolis *et al.*, 1953), we assume that the proposal distribution is symmetric, that is $q(\mathbf{z}_A|\mathbf{z}_B) = q(\mathbf{z}_B|\mathbf{z}_A)$ for all values of $\mathbf{z}_A$ and $\mathbf{z}_B$. The candidate sample is then accepted with probability

$$A(\mathbf{z}^\star, \mathbf{z}^{(\tau)}) = \min\left(1, \frac{\widetilde{p}(\mathbf{z}^\star)}{\widetilde{p}(\mathbf{z}^{(\tau)})}\right). \tag{11.33}$$

This can be achieved by choosing a random number u with uniform distribution over the unit interval $(0, 1)$ and then accepting the sample if $A(\mathbf{z}^\star, \mathbf{z}^{(\tau)}) > u$. Note that if the step from $\mathbf{z}^{(\tau)}$ to $\mathbf{z}^\star$ causes an increase in the value of $p(\mathbf{z})$, then the candidate point is certain to be kept.

If the candidate sample is accepted, then $\mathbf{z}^{(\tau+1)} = \mathbf{z}^\star$, otherwise the candidate point $\mathbf{z}^\star$ is discarded, $\mathbf{z}^{(\tau+1)}$ is set to $\mathbf{z}^{(\tau)}$ and another candidate sample is drawn from the distribution $q(\mathbf{z}|\mathbf{z}^{(\tau+1)})$. This is in contrast to rejection sampling, where rejected samples are simply discarded. In the Metropolis algorithm when a candidate point is rejected, the previous sample is included instead in the final list of samples, leading to multiple copies of samples. Of course, in a practical implementation, only a single copy of each retained sample would be kept, along with an integer weighting factor recording how many times that state appears. As we shall see, as long as $q(\mathbf{z}_A|\mathbf{z}_B)$ is positive for any values of $\mathbf{z}_A$ and $\mathbf{z}_B$ (this is a sufficient but not necessary condition), the distribution of $\mathbf{z}^{(\tau)}$ tends to $p(\mathbf{z})$ as $\tau \to \infty$. It should be emphasized, however, that the sequence $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \ldots$ is not a set of independent samples from $p(\mathbf{z})$ because successive samples are highly correlated. If we wish to obtain independent samples, then we can discard most of the sequence and just retain every M^{th} sample. For M sufficiently large, the retained samples will for all practical purposes be independent. Figure 11.9 shows a simple illustrative example of sampling from a two-dimensional Gaussian distribution using the Metropolis algorithm in which the proposal distribution is an isotropic Gaussian.

Further insight into the nature of Markov chain Monte Carlo algorithms can be gleaned by looking at the properties of a specific example, namely a simple random

Figure 11.9 A simple illustration using Metropolis algorithm to sample from a Gaussian distribution whose one standard-deviation contour is shown by the ellipse. The proposal distribution is an isotropic Gaussian distribution whose standard deviation is 0.2. Steps that are accepted are shown as green lines, and rejected steps are shown in red. A total of 150 candidate samples are generated, of which 43 are rejected.

walk. Consider a state space z consisting of the integers, with probabilities

$$p(z^{(\tau+1)} = z^{(\tau)}) = 0.5 \tag{11.34}$$

$$p(z^{(\tau+1)} = z^{(\tau)} + 1) = 0.25 \tag{11.35}$$

$$p(z^{(\tau+1)} = z^{(\tau)} - 1) = 0.25 \tag{11.36}$$

where $z^{(\tau)}$ denotes the state at step τ. If the initial state is $z^{(1)} = 0$, then by symmetry the expected state at time τ will also be zero $\mathbb{E}[z^{(\tau)}] = 0$, and similarly it is easily seen that $\mathbb{E}[(z^{(\tau)})^2] = \tau/2$. Thus after τ steps, the random walk has only travelled a distance that on average is proportional to the square root of τ. This square root dependence is typical of random walk behaviour and shows that random walks are very inefficient in exploring the state space. As we shall see, a central goal in designing Markov chain Monte Carlo methods is to avoid random walk behaviour.

Exercise 11.10

11.2.1 Markov chains

Before discussing Markov chain Monte Carlo methods in more detail, it is useful to study some general properties of Markov chains in more detail. In particular, we ask under what circumstances will a Markov chain converge to the desired distribution. A first-order Markov chain is defined to be a series of random variables $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(M)}$ such that the following conditional independence property holds for $m \in \{1, \ldots, M-1\}$

$$p(\mathbf{z}^{(m+1)}|\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}) = p(\mathbf{z}^{(m+1)}|\mathbf{z}^{(m)}). \tag{11.37}$$

This of course can be represented as a directed graph in the form of a chain, an example of which is shown in Figure 8.38. We can then specify the Markov chain by giving the probability distribution for the initial variable $p(\mathbf{z}^{(0)})$ together with the

conditional probabilities for subsequent variables in the form of *transition probabilities* $T_m(\mathbf{z}^{(m)}, \mathbf{z}^{(m+1)}) \equiv p(\mathbf{z}^{(m+1)}|\mathbf{z}^{(m)})$. A Markov chain is called *homogeneous* if the transition probabilities are the same for all m.

The marginal probability for a particular variable can be expressed in terms of the marginal probability for the previous variable in the chain in the form

$$p(\mathbf{z}^{(m+1)}) = \sum_{\mathbf{z}^{(m)}} p(\mathbf{z}^{(m+1)}|\mathbf{z}^{(m)})p(\mathbf{z}^{(m)}). \tag{11.38}$$

A distribution is said to be invariant, or stationary, with respect to a Markov chain if each step in the chain leaves that distribution invariant. Thus, for a homogeneous Markov chain with transition probabilities $T(\mathbf{z}', \mathbf{z})$, the distribution $p^\star(\mathbf{z})$ is invariant if

$$p^\star(\mathbf{z}) = \sum_{\mathbf{z}'} T(\mathbf{z}', \mathbf{z})p^\star(\mathbf{z}'). \tag{11.39}$$

Note that a given Markov chain may have more than one invariant distribution. For instance, if the transition probabilities are given by the identity transformation, then any distribution will be invariant.

A sufficient (but not necessary) condition for ensuring that the required distribution $p(\mathbf{z})$ is invariant is to choose the transition probabilities to satisfy the property of *detailed balance*, defined by

$$p^\star(\mathbf{z})T(\mathbf{z}, \mathbf{z}') = p^\star(\mathbf{z}')T(\mathbf{z}', \mathbf{z}) \tag{11.40}$$

for the particular distribution $p^\star(\mathbf{z})$. It is easily seen that a transition probability that satisfies detailed balance with respect to a particular distribution will leave that distribution invariant, because

$$\sum_{\mathbf{z}'} p^\star(\mathbf{z}')T(\mathbf{z}', \mathbf{z}) = \sum_{\mathbf{z}'} p^\star(\mathbf{z})T(\mathbf{z}, \mathbf{z}') = p^\star(\mathbf{z}) \sum_{\mathbf{z}'} p(\mathbf{z}'|\mathbf{z}) = p^\star(\mathbf{z}). \tag{11.41}$$

A Markov chain that respects detailed balance is said to be *reversible*.

Our goal is to use Markov chains to sample from a given distribution. We can achieve this if we set up a Markov chain such that the desired distribution is invariant. However, we must also require that for $m \to \infty$, the distribution $p(\mathbf{z}^{(m)})$ converges to the required invariant distribution $p^\star(\mathbf{z})$, irrespective of the choice of initial distribution $p(\mathbf{z}^{(0)})$. This property is called *ergodicity*, and the invariant distribution is then called the *equilibrium* distribution. Clearly, an ergodic Markov chain can have only one equilibrium distribution. It can be shown that a homogeneous Markov chain will be ergodic, subject only to weak restrictions on the invariant distribution and the transition probabilities (Neal, 1993).

In practice we often construct the transition probabilities from a set of 'base' transitions $B_1, \ldots, B_K$. This can be achieved through a mixture distribution of the form

$$T(\mathbf{z}', \mathbf{z}) = \sum_{k=1}^{K} \alpha_k B_k(\mathbf{z}', \mathbf{z}) \tag{11.42}$$

for some set of mixing coefficients $\alpha_1, \ldots, \alpha_K$ satisfying $\alpha_k \geqslant 0$ and $\sum_k \alpha_k = 1$. Alternatively, the base transitions may be combined through successive application, so that

$$T(\mathbf{z}', \mathbf{z}) = \sum_{\mathbf{z}_1} \ldots \sum_{\mathbf{z}_{n-1}} B_1(\mathbf{z}', \mathbf{z}_1) \ldots B_{K-1}(\mathbf{z}_{K-2}, \mathbf{z}_{K-1}) B_K(\mathbf{z}_{K-1}, \mathbf{z}). \quad (11.43)$$

If a distribution is invariant with respect to each of the base transitions, then obviously it will also be invariant with respect to either of the $T(\mathbf{z}', \mathbf{z})$ given by (11.42) or (11.43). For the case of the mixture (11.42), if each of the base transitions satisfies detailed balance, then the mixture transition T will also satisfy detailed balance. This does not hold for the transition probability constructed using (11.43), although by symmetrizing the order of application of the base transitions, in the form $B_1, B_2, \ldots, B_K, B_K, \ldots, B_2, B_1$, detailed balance can be restored. A common example of the use of composite transition probabilities is where each base transition changes only a subset of the variables.

11.2.2 The Metropolis-Hastings algorithm

Earlier we introduced the basic Metropolis algorithm, without actually demonstrating that it samples from the required distribution. Before giving a proof, we first discuss a generalization, known as the *Metropolis-Hastings* algorithm (Hastings, 1970), to the case where the proposal distribution is no longer a symmetric function of its arguments. In particular at step τ of the algorithm, in which the current state is $\mathbf{z}^{(\tau)}$, we draw a sample $\mathbf{z}^\star$ from the distribution $q_k(\mathbf{z}|\mathbf{z}^{(\tau)})$ and then accept it with probability $A_k(\mathbf{z}^\star, \mathbf{z}_\tau)$ where

$$A_k(\mathbf{z}^\star, \mathbf{z}^{(\tau)}) = \min\left(1, \frac{\widetilde{p}(\mathbf{z}^\star) q_k(\mathbf{z}^{(\tau)}|\mathbf{z}^\star)}{\widetilde{p}(\mathbf{z}^{(\tau)}) q_k(\mathbf{z}^\star|\mathbf{z}^{(\tau)})}\right). \quad (11.44)$$

Here k labels the members of the set of possible transitions being considered. Again, the evaluation of the acceptance criterion does not require knowledge of the normalizing constant Z_p in the probability distribution $p(\mathbf{z}) = \widetilde{p}(\mathbf{z})/Z_p$. For a symmetric proposal distribution the Metropolis-Hastings criterion (11.44) reduces to the standard Metropolis criterion given by (11.33).

We can show that $p(\mathbf{z})$ is an invariant distribution of the Markov chain defined by the Metropolis-Hastings algorithm by showing that detailed balance, defined by (11.40), is satisfied. Using (11.44) we have

$$\begin{aligned} p(\mathbf{z}) q_k(\mathbf{z}|\mathbf{z}') A_k(\mathbf{z}', \mathbf{z}) &= \min\left(p(\mathbf{z}) q_k(\mathbf{z}|\mathbf{z}'), p(\mathbf{z}') q_k(\mathbf{z}'|\mathbf{z})\right) \\ &= \min\left(p(\mathbf{z}') q_k(\mathbf{z}'|\mathbf{z}), p(\mathbf{z}) q_k(\mathbf{z}|\mathbf{z}')\right) \\ &= p(\mathbf{z}') q_k(\mathbf{z}'|\mathbf{z}) A_k(\mathbf{z}, \mathbf{z}') \end{aligned} \quad (11.45)$$

as required.

The specific choice of proposal distribution can have a marked effect on the performance of the algorithm. For continuous state spaces, a common choice is a Gaussian centred on the current state, leading to an important trade-off in determining the variance parameter of this distribution. If the variance is small, then the

Figure 11.10 Schematic illustration of the use of an isotropic Gaussian proposal distribution (blue circle) to sample from a correlated multivariate Gaussian distribution (red ellipse) having very different standard deviations in different directions, using the Metropolis-Hastings algorithm. In order to keep the rejection rate low, the scale ρ of the proposal distribution should be on the order of the smallest standard deviation $\sigma_{\min}$, which leads to random walk behaviour in which the number of steps separating states that are approximately independent is of order $(\sigma_{\max}/\sigma_{\min})^2$ where $\sigma_{\max}$ is the largest standard deviation.

proportion of accepted transitions will be high, but progress through the state space takes the form of a slow random walk leading to long correlation times. However, if the variance parameter is large, then the rejection rate will be high because, in the kind of complex problems we are considering, many of the proposed steps will be to states for which the probability $p(\mathbf{z})$ is low. Consider a multivariate distribution $p(\mathbf{z})$ having strong correlations between the components of $\mathbf{z}$, as illustrated in Figure 11.10. The scale ρ of the proposal distribution should be as large as possible without incurring high rejection rates. This suggests that ρ should be of the same order as the smallest length scale $\sigma_{\min}$. The system then explores the distribution along the more extended direction by means of a random walk, and so the number of steps to arrive at a state that is more or less independent of the original state is of order $(\sigma_{\max}/\sigma_{\min})^2$. In fact in two dimensions, the increase in rejection rate as ρ increases is offset by the larger steps sizes of those transitions that are accepted, and more generally for a multivariate Gaussian the number of steps required to obtain independent samples scales like $(\sigma_{\max}/\sigma_2)^2$ where σ_2 is the second-smallest standard deviation (Neal, 1993). These details aside, it remains the case that if the length scales over which the distributions vary are very different in different directions, then the Metropolis Hastings algorithm can have very slow convergence.

11.3. Gibbs Sampling

Gibbs sampling (Geman and Geman, 1984) is a simple and widely applicable Markov chain Monte Carlo algorithm and can be seen as a special case of the Metropolis-Hastings algorithm.

Consider the distribution $p(\mathbf{z}) = p(z_1, \ldots, z_M)$ from which we wish to sample, and suppose that we have chosen some initial state for the Markov chain. Each step of the Gibbs sampling procedure involves replacing the value of one of the variables by a value drawn from the distribution of that variable conditioned on the values of the remaining variables. Thus we replace z_i by a value drawn from the distribution $p(z_i|\mathbf{z}_{\backslash i})$, where z_i denotes the ith component of $\mathbf{z}$, and $\mathbf{z}_{\backslash i}$ denotes $z_1, \ldots, z_M$ but with z_i omitted. This procedure is repeated either by cycling through the variables

in some particular order or by choosing the variable to be updated at each step at random from some distribution.

For example, suppose we have a distribution $p(z_1, z_2, z_3)$ over three variables, and at step τ of the algorithm we have selected values $z_1^{(\tau)}$, $z_2^{(\tau)}$ and $z_3^{(\tau)}$. We first replace $z_1^{(\tau)}$ by a new value $z_1^{(\tau+1)}$ obtained by sampling from the conditional distribution

$$p(z_1|z_2^{(\tau)}, z_3^{(\tau)}). \tag{11.46}$$

Next we replace $z_2^{(\tau)}$ by a value $z_2^{(\tau+1)}$ obtained by sampling from the conditional distribution

$$p(z_2|z_1^{(\tau+1)}, z_3^{(\tau)}) \tag{11.47}$$

so that the new value for z_1 is used straight away in subsequent sampling steps. Then we update z_3 with a sample $z_3^{(\tau+1)}$ drawn from

$$p(z_3|z_1^{(\tau+1)}, z_2^{(\tau+1)}) \tag{11.48}$$

and so on, cycling through the three variables in turn.

Gibbs Sampling

1. Initialize $\{z_i : i = 1, \ldots, M\}$
2. For $\tau = 1, \ldots, T$:
   - Sample $z_1^{(\tau+1)} \sim p(z_1|z_2^{(\tau)}, z_3^{(\tau)}, \ldots, z_M^{(\tau)})$.
   - Sample $z_2^{(\tau+1)} \sim p(z_2|z_1^{(\tau+1)}, z_3^{(\tau)}, \ldots, z_M^{(\tau)})$.

     $\vdots$
   - Sample $z_j^{(\tau+1)} \sim p(z_j|z_1^{(\tau+1)}, \ldots, z_{j-1}^{(\tau+1)}, z_{j+1}^{(\tau)}, \ldots, z_M^{(\tau)})$.

     $\vdots$
   - Sample $z_M^{(\tau+1)} \sim p(z_M|z_1^{(\tau+1)}, z_2^{(\tau+1)}, \ldots, z_{M-1}^{(\tau+1)})$.

Josiah Willard Gibbs

1839–1903

Gibbs spent almost his entire life living in a house built by his father in New Haven, Connecticut. In 1863, Gibbs was granted the first PhD in engineering in the United States, and in 1871 he was appointed to the first chair of mathematical physics in the United States at Yale, a post for which he received no salary because at the time he had no publications. He developed the field of vector analysis and made contributions to crystallography and planetary orbits. His most famous work, entitled *On the Equilibrium of Heterogeneous Substances*, laid the foundations for the science of physical chemistry.

To show that this procedure samples from the required distribution, we first of all note that the distribution $p(\mathbf{z})$ is an invariant of each of the Gibbs sampling steps individually and hence of the whole Markov chain. This follows from the fact that when we sample from $p(z_i|\{\mathbf{z}_{\backslash i})$, the marginal distribution $p(\mathbf{z}_{\backslash i})$ is clearly invariant because the value of $\mathbf{z}_{\backslash i}$ is unchanged. Also, each step by definition samples from the correct conditional distribution $p(z_i|\mathbf{z}_{\backslash i})$. Because these conditional and marginal distributions together specify the joint distribution, we see that the joint distribution is itself invariant.

The second requirement to be satisfied in order that the Gibbs sampling procedure samples from the correct distribution is that it be ergodic. A sufficient condition for ergodicity is that none of the conditional distributions be anywhere zero. If this is the case, then any point in z space can be reached from any other point in a finite number of steps involving one update of each of the component variables. If this requirement is not satisfied, so that some of the conditional distributions have zeros, then ergodicity, if it applies, must be proven explicitly.

The distribution of initial states must also be specified in order to complete the algorithm, although samples drawn after many iterations will effectively become independent of this distribution. Of course, successive samples from the Markov chain will be highly correlated, and so to obtain samples that are nearly independent it will be necessary to subsample the sequence.

We can obtain the Gibbs sampling procedure as a particular instance of the Metropolis-Hastings algorithm as follows. Consider a Metropolis-Hastings sampling step involving the variable z_k in which the remaining variables $\mathbf{z}_{\backslash k}$ remain fixed, and for which the transition probability from $\mathbf{z}$ to $\mathbf{z}^\star$ is given by $q_k(\mathbf{z}^\star|\mathbf{z}) = p(z_k^\star|\mathbf{z}_{\backslash k})$. We note that $\mathbf{z}^\star_{\backslash k} = \mathbf{z}_{\backslash k}$ because these components are unchanged by the sampling step. Also, $p(\mathbf{z}) = p(z_k|\mathbf{z}_{\backslash k})p(\mathbf{z}_{\backslash k})$. Thus the factor that determines the acceptance probability in the Metropolis-Hastings (11.44) is given by

$$A(\mathbf{z}^\star, \mathbf{z}) = \frac{p(\mathbf{z}^\star)q_k(\mathbf{z}|\mathbf{z}^\star)}{p(\mathbf{z})q_k(\mathbf{z}^\star|\mathbf{z})} = \frac{p(z_k^\star|\mathbf{z}^\star_{\backslash k})p(\mathbf{z}^\star_{\backslash k})p(z_k|\mathbf{z}^\star_{\backslash k})}{p(z_k|\mathbf{z}_{\backslash k})p(\mathbf{z}_{\backslash k})p(z_k^\star|\mathbf{z}_{\backslash k})} = 1 \tag{11.49}$$

where we have used $\mathbf{z}^\star_{\backslash k} = \mathbf{z}_{\backslash k}$. Thus the Metropolis-Hastings steps are always accepted.

As with the Metropolis algorithm, we can gain some insight into the behaviour of Gibbs sampling by investigating its application to a Gaussian distribution. Consider a correlated Gaussian in two variables, as illustrated in Figure 11.11, having conditional distributions of width l and marginal distributions of width L. The typical step size is governed by the conditional distributions and will be of order l. Because the state evolves according to a random walk, the number of steps needed to obtain independent samples from the distribution will be of order $(L/l)^2$. Of course if the Gaussian distribution were uncorrelated, then the Gibbs sampling procedure would be optimally efficient. For this simple problem, we could rotate the coordinate system in order to decorrelate the variables. However, in practical applications it will generally be infeasible to find such transformations.

One approach to reducing random walk behaviour in Gibbs sampling is called *over-relaxation* (Adler, 1981). In its original form, this applies to problems for which

Figure 11.11 Illustration of Gibbs sampling by alternate updates of two variables whose distribution is a correlated Gaussian. The step size is governed by the standard deviation of the conditional distribution (green curve), and is $O(l)$, leading to slow progress in the direction of elongation of the joint distribution (red ellipse). The number of steps needed to obtain an independent sample from the distribution is $O((L/l)^2)$.

the conditional distributions are Gaussian, which represents a more general class of distributions than the multivariate Gaussian because, for example, the non-Gaussian distribution $p(z, y) \propto \exp(-z^2 y^2)$ has Gaussian conditional distributions. At each step of the Gibbs sampling algorithm, the conditional distribution for a particular component z_i has some mean μ_i and some variance σ_i^2. In the over-relaxation framework, the value of z_i is replaced with

$$z_i' = \mu_i + \alpha(z_i - \mu_i) + \sigma_i(1 - \alpha_i^2)^{1/2}\nu \tag{11.50}$$

where ν is a Gaussian random variable with zero mean and unit variance, and α is a parameter such that $-1 < \alpha < 1$. For $\alpha = 0$, the method is equivalent to standard Gibbs sampling, and for $\alpha < 0$ the step is biased to the opposite side of the mean. This step leaves the desired distribution invariant because if z_i has mean μ_i and variance σ_i^2, then so too does z_i'. The effect of over-relaxation is to encourage directed motion through state space when the variables are highly correlated. The framework of *ordered over-relaxation* (Neal, 1999) generalizes this approach to non-Gaussian distributions.

The practical applicability of Gibbs sampling depends on the ease with which samples can be drawn from the conditional distributions $p(z_k|\mathbf{z}_{\setminus k})$. In the case of probability distributions specified using graphical models, the conditional distributions for individual nodes depend only on the variables in the corresponding Markov blankets, as illustrated in Figure 11.12. For directed graphs, a wide choice of conditional distributions for the individual nodes conditioned on their parents will lead to conditional distributions for Gibbs sampling that are log concave. The adaptive rejection sampling methods discussed in Section 11.1.3 therefore provide a framework for Monte Carlo sampling from directed graphs with broad applicability.

If the graph is constructed using distributions from the exponential family, and if the parent-child relationships preserve conjugacy, then the full conditional distributions arising in Gibbs sampling will have the same functional form as the orig-

Figure 11.12 The Gibbs sampling method requires samples to be drawn from the conditional distribution of a variable conditioned on the remaining variables. For graphical models, this conditional distribution is a function only of the states of the nodes in the Markov blanket. For an undirected graph this comprises the set of neighbours, as shown on the left, while for a directed graph the Markov blanket comprises the parents, the children, and the co-parents, as shown on the right.

inal conditional distributions (conditioned on the parents) defining each node, and so standard sampling techniques can be employed. In general, the full conditional distributions will be of a complex form that does not permit the use of standard sampling algorithms. However, if these conditionals are log concave, then sampling can be done efficiently using adaptive rejection sampling (assuming the corresponding variable is a scalar).

If, at each stage of the Gibbs sampling algorithm, instead of drawing a sample from the corresponding conditional distribution, we make a point estimate of the variable given by the maximum of the conditional distribution, then we obtain the iterated conditional modes (ICM) algorithm discussed in Section 8.3.3. Thus ICM can be seen as a greedy approximation to Gibbs sampling.

Because the basic Gibbs sampling technique considers one variable at a time, there are strong dependencies between successive samples. At the opposite extreme, if we could draw samples directly from the joint distribution (an operation that we are supposing is intractable), then successive samples would be independent. We can hope to improve on the simple Gibbs sampler by adopting an intermediate strategy in which we sample successively from groups of variables rather than individual variables. This is achieved in the *blocking Gibbs* sampling algorithm by choosing blocks of variables, not necessarily disjoint, and then sampling jointly from the variables in each block in turn, conditioned on the remaining variables (Jensen *et al.*, 1995).

11.4. Slice Sampling

We have seen that one of the difficulties with the Metropolis algorithm is the sensitivity to step size. If this is too small, the result is slow decorrelation due to random walk behaviour, whereas if it is too large the result is inefficiency due to a high rejection rate. The technique of *slice sampling* (Neal, 2003) provides an adaptive step size that is automatically adjusted to match the characteristics of the distribution. Again it requires that we are able to evaluate the unnormalized distribution $\widetilde{p}(\mathbf{z})$.

Consider first the univariate case. Slice sampling involves augmenting z with an additional variable u and then drawing samples from the joint (z, u) space. We shall see another example of this approach when we discuss hybrid Monte Carlo in Section 11.5. The goal is to sample uniformly from the area under the distribution

Figure 11.13 Illustration of slice sampling. (a) For a given value $z^{(\tau)}$, a value of u is chosen uniformly in the region $0 \leqslant u \leqslant \widetilde{p}(z^{(\tau)})$, which then defines a 'slice' through the distribution, shown by the solid horizontal lines. (b) Because it is infeasible to sample directly from a slice, a new sample of z is drawn from a region $z_{\min} \leqslant z \leqslant z_{\max}$, which contains the previous value $z^{(\tau)}$.

given by

$$\widehat{p}(z, u) = \begin{cases} 1/Z_p & \text{if } 0 \leqslant u \leqslant \widetilde{p}(z) \\ 0 & \text{otherwise} \end{cases} \tag{11.51}$$

where $Z_p = \int \widetilde{p}(z)\,\mathrm{d}z$. The marginal distribution over z is given by

$$\int \widehat{p}(z, u)\,\mathrm{d}u = \int_0^{\widetilde{p}(z)} \frac{1}{Z_p}\,\mathrm{d}u = \frac{\widetilde{p}(z)}{Z_p} = p(z) \tag{11.52}$$

and so we can sample from $p(z)$ by sampling from $\widehat{p}(z, u)$ and then ignoring the u values. This can be achieved by alternately sampling z and u. Given the value of z we evaluate $\widetilde{p}(z)$ and then sample u uniformly in the range $0 \leqslant u \leqslant \widetilde{p}(z)$, which is straightforward. Then we fix u and sample z uniformly from the 'slice' through the distribution defined by $\{z : \widetilde{p}(z) > u\}$. This is illustrated in Figure 11.13(a).

In practice, it can be difficult to sample directly from a slice through the distribution and so instead we define a sampling scheme that leaves the uniform distribution under $\widehat{p}(z, u)$ invariant, which can be achieved by ensuring that detailed balance is satisfied. Suppose the current value of z is denoted $z^{(\tau)}$ and that we have obtained a corresponding sample u. The next value of z is obtained by considering a region $z_{\min} \leqslant z \leqslant z_{\max}$ that contains $z^{(\tau)}$. It is in the choice of this region that the adaptation to the characteristic length scales of the distribution takes place. We want the region to encompass as much of the slice as possible so as to allow large moves in z space while having as little as possible of this region lying outside the slice, because this makes the sampling less efficient.

One approach to the choice of region involves starting with a region containing $z^{(\tau)}$ having some width w and then testing each of the end points to see if they lie within the slice. If either end point does not, then the region is extended in that direction by increments of value w until the end point lies outside the region. A candidate value z' is then chosen uniformly from this region, and if it lies within the slice, then it forms $z^{(\tau+1)}$. If it lies outside the slice, then the region is shrunk such that z' forms an end point and such that the region still contains $z^{(\tau)}$. Then another

candidate point is drawn uniformly from this reduced region and so on, until a value of z is found that lies within the slice.

Slice sampling can be applied to multivariate distributions by repeatedly sampling each variable in turn, in the manner of Gibbs sampling. This requires that we are able to compute, for each component z_i, a function that is proportional to $p(z_i|\mathbf{z}_{\setminus i})$.

11.5. The Hybrid Monte Carlo Algorithm

As we have already noted, one of the major limitations of the Metropolis algorithm is that it can exhibit random walk behaviour whereby the distance traversed through the state space grows only as the square root of the number of steps. The problem cannot be resolved simply by taking bigger steps as this leads to a high rejection rate.

In this section, we introduce a more sophisticated class of transitions based on an analogy with physical systems and that has the property of being able to make large changes to the system state while keeping the rejection probability small. It is applicable to distributions over continuous variables for which we can readily evaluate the gradient of the log probability with respect to the state variables. We will discuss the dynamical systems framework in Section 11.5.1, and then in Section 11.5.2 we explain how this may be combined with the Metropolis algorithm to yield the powerful hybrid Monte Carlo algorithm. A background in physics is not required as this section is self-contained and the key results are all derived from first principles.

11.5.1 Dynamical systems

The dynamical approach to stochastic sampling has its origins in algorithms for simulating the behaviour of physical systems evolving under Hamiltonian dynamics. In a Markov chain Monte Carlo simulation, the goal is to sample from a given probability distribution $p(\mathbf{z})$. The framework of *Hamiltonian dynamics* is exploited by casting the probabilistic simulation in the form of a Hamiltonian system. In order to remain in keeping with the literature in this area, we make use of the relevant dynamical systems terminology where appropriate, which will be defined as we go along.

The dynamics that we consider corresponds to the evolution of the state variable $\mathbf{z} = \{z_i\}$ under continuous time, which we denote by τ. Classical dynamics is described by Newton's second law of motion in which the acceleration of an object is proportional to the applied force, corresponding to a second-order differential equation over time. We can decompose a second-order equation into two coupled first-order equations by introducing intermediate *momentum* variables $\mathbf{r}$, corresponding to the rate of change of the state variables $\mathbf{z}$, having components

$$r_i = \frac{\mathrm{d}z_i}{\mathrm{d}\tau} \tag{11.53}$$

where the z_i can be regarded as *position* variables in this dynamics perspective. Thus

for each position variable there is a corresponding momentum variable, and the joint space of position and momentum variables is called *phase space*.

Without loss of generality, we can write the probability distribution $p(\mathbf{z})$ in the form

$$p(\mathbf{z}) = \frac{1}{Z_p} \exp\left(-E(\mathbf{z})\right) \tag{11.54}$$

where $E(\mathbf{z})$ is interpreted as the *potential energy* of the system when in state $\mathbf{z}$. The system acceleration is the rate of change of momentum and is given by the applied *force*, which itself is the negative gradient of the potential energy

$$\frac{\mathrm{d}r_i}{\mathrm{d}\tau} = -\frac{\partial E(\mathbf{z})}{\partial z_i}. \tag{11.55}$$

It is convenient to reformulate this dynamical system using the Hamiltonian framework. To do this, we first define the *kinetic energy* by

$$K(\mathbf{r}) = \frac{1}{2}\|\mathbf{r}\|^2 = \frac{1}{2}\sum_i r_i^2. \tag{11.56}$$

The total energy of the system is then the sum of its potential and kinetic energies

$$H(\mathbf{z}, \mathbf{r}) = E(\mathbf{z}) + K(\mathbf{r}) \tag{11.57}$$

where H is the *Hamiltonian* function. Using (11.53), (11.55), (11.56), and (11.57), we can now express the dynamics of the system in terms of the Hamiltonian equations given by

Exercise 11.15

$$\frac{\mathrm{d}z_i}{\mathrm{d}\tau} = \frac{\partial H}{\partial r_i} \tag{11.58}$$

$$\frac{\mathrm{d}r_i}{\mathrm{d}\tau} = -\frac{\partial H}{\partial z_i}. \tag{11.59}$$

William Hamilton

1805–1865

William Rowan Hamilton was an Irish mathematician and physicist, and child prodigy, who was appointed Professor of Astronomy at Trinity College, Dublin, in 1827, before he had even graduated. One of Hamilton's most important contributions was a new formulation of dynamics, which played a significant role in the later development of quantum mechanics. His other great achievement was the development of *quaternions*, which generalize the concept of complex numbers by introducing three distinct square roots of minus one, which satisfy $i^2 = j^2 = k^2 = ijk = -1$. It is said that these equations occurred to him while walking along the Royal Canal in Dublin with his wife, on 16 October 1843, and he promptly carved the equations into the side of Broome bridge. Although there is no longer any evidence of the carving, there is now a stone plaque on the bridge commemorating the discovery and displaying the quaternion equations.

During the evolution of this dynamical system, the value of the Hamiltonian H is constant, as is easily seen by differentiation

$$\begin{aligned}\frac{\mathrm{d}H}{\mathrm{d}\tau} &= \sum_i \left\{ \frac{\partial H}{\partial z_i}\frac{\mathrm{d}z_i}{\mathrm{d}\tau} + \frac{\partial H}{\partial r_i}\frac{\mathrm{d}r_i}{\mathrm{d}\tau} \right\} \\ &= \sum_i \left\{ \frac{\partial H}{\partial z_i}\frac{\partial H}{\partial r_i} - \frac{\partial H}{\partial r_i}\frac{\partial H}{\partial z_i} \right\} = 0. \end{aligned} \tag{11.60}$$

A second important property of Hamiltonian dynamical systems, known as *Liouville's Theorem*, is that they preserve volume in phase space. In other words, if we consider a region within the space of variables $(\mathbf{z}, \mathbf{r})$, then as this region evolves under the equations of Hamiltonian dynamics, its shape may change but its volume will not. This can be seen by noting that the flow field (rate of change of location in phase space) is given by

$$\mathbf{V} = \left(\frac{\mathrm{d}\mathbf{z}}{\mathrm{d}\tau}, \frac{\mathrm{d}\mathbf{r}}{\mathrm{d}\tau} \right) \tag{11.61}$$

and that the divergence of this field vanishes

$$\begin{aligned}\operatorname{div} \mathbf{V} &= \sum_i \left\{ \frac{\partial}{\partial z_i}\frac{\mathrm{d}z_i}{\mathrm{d}\tau} + \frac{\partial}{\partial r_i}\frac{\mathrm{d}r_i}{\mathrm{d}\tau} \right\} \\ &= \sum_i \left\{ -\frac{\partial}{\partial z_i}\frac{\partial H}{\partial r_i} + \frac{\partial}{\partial r_i}\frac{\partial H}{\partial z_i} \right\} = 0. \end{aligned} \tag{11.62}$$

Now consider the joint distribution over phase space whose total energy is the Hamiltonian, i.e., the distribution given by

$$p(\mathbf{z}, \mathbf{r}) = \frac{1}{Z_H} \exp(-H(\mathbf{z}, \mathbf{r})). \tag{11.63}$$

Using the two results of conservation of volume and conservation of H, it follows that the Hamiltonian dynamics will leave $p(\mathbf{z}, \mathbf{r})$ invariant. This can be seen by considering a small region of phase space over which H is approximately constant. If we follow the evolution of the Hamiltonian equations for a finite time, then the volume of this region will remain unchanged as will the value of H in this region, and hence the probability density, which is a function only of H, will also be unchanged.

Although H is invariant, the values of $\mathbf{z}$ and $\mathbf{r}$ will vary, and so by integrating the Hamiltonian dynamics over a finite time duration it becomes possible to make large changes to $\mathbf{z}$ in a systematic way that avoids random walk behaviour.

Evolution under the Hamiltonian dynamics will not, however, sample ergodically from $p(\mathbf{z}, \mathbf{r})$ because the value of H is constant. In order to arrive at an ergodic sampling scheme, we can introduce additional moves in phase space that change the value of H while also leaving the distribution $p(\mathbf{z}, \mathbf{r})$ invariant. The simplest way to achieve this is to replace the value of $\mathbf{r}$ with one drawn from its distribution conditioned on $\mathbf{z}$. This can be regarded as a Gibbs sampling step, and hence from

Section 11.3 we see that this also leaves the desired distribution invariant. Noting that $\mathbf{z}$ and $\mathbf{r}$ are independent in the distribution $p(\mathbf{z}, \mathbf{r})$, we see that the conditional distribution $p(\mathbf{r}|\mathbf{z})$ is a Gaussian from which it is straightforward to sample.

Exercise 11.16

In a practical application of this approach, we have to address the problem of performing a numerical integration of the Hamiltonian equations. This will necessarily introduce numerical errors and so we should devise a scheme that minimizes the impact of such errors. In fact, it turns out that integration schemes can be devised for which Liouville's theorem still holds exactly. This property will be important in the hybrid Monte Carlo algorithm, which is discussed in Section 11.5.2. One scheme for achieving this is called the *leapfrog* discretization and involves alternately updating discrete-time approximations $\widehat{\mathbf{z}}$ and $\widehat{\mathbf{r}}$ to the position and momentum variables using

$$\widehat{r}_i(\tau + \epsilon/2) = \widehat{r}_i(\tau) - \frac{\epsilon}{2}\frac{\partial E}{\partial z_i}(\widehat{\mathbf{z}}(\tau)) \tag{11.64}$$

$$\widehat{z}_i(\tau + \epsilon) = \widehat{z}_i(\tau) + \epsilon \widehat{r}_i(\tau + \epsilon/2) \tag{11.65}$$

$$\widehat{r}_i(\tau + \epsilon) = \widehat{r}_i(\tau + \epsilon/2) - \frac{\epsilon}{2}\frac{\partial E}{\partial z_i}(\widehat{\mathbf{z}}(\tau + \epsilon)). \tag{11.66}$$

We see that this takes the form of a half-step update of the momentum variables with step size $\epsilon/2$, followed by a full-step update of the position variables with step size ϵ, followed by a second half-step update of the momentum variables. If several leapfrog steps are applied in succession, it can be seen that half-step updates to the momentum variables can be combined into full-step updates with step size ϵ. The successive updates to position and momentum variables then leapfrog over each other. In order to advance the dynamics by a time interval τ, we need to take τ/ϵ steps. The error involved in the discretized approximation to the continuous time dynamics will go to zero, assuming a smooth function $E(\mathbf{z})$, in the limit $\epsilon \to 0$. However, for a nonzero ϵ as used in practice, some residual error will remain. We shall see in Section 11.5.2 how the effects of such errors can be eliminated in the hybrid Monte Carlo algorithm.

In summary then, the Hamiltonian dynamical approach involves alternating between a series of leapfrog updates and a resampling of the momentum variables from their marginal distribution.

Note that the Hamiltonian dynamics method, unlike the basic Metropolis algorithm, is able to make use of information about the gradient of the log probability distribution as well as about the distribution itself. An analogous situation is familiar from the domain of function optimization. In most cases where gradient information is available, it is highly advantageous to make use of it. Informally, this follows from the fact that in a space of dimension D, the additional computational cost of evaluating a gradient compared with evaluating the function itself will typically be a fixed factor independent of D, whereas the D-dimensional gradient vector conveys D pieces of information compared with the one piece of information given by the function itself.

11.5.2 Hybrid Monte Carlo

As we discussed in the previous section, for a nonzero step size ϵ, the discretization of the leapfrog algorithm will introduce errors into the integration of the Hamiltonian dynamical equations. *Hybrid Monte Carlo* (Duane *et al.*, 1987; Neal, 1996) combines Hamiltonian dynamics with the Metropolis algorithm and thereby removes any bias associated with the discretization.

Specifically, the algorithm uses a Markov chain consisting of alternate stochastic updates of the momentum variable $\mathbf{r}$ and Hamiltonian dynamical updates using the leapfrog algorithm. After each application of the leapfrog algorithm, the resulting candidate state is accepted or rejected according to the Metropolis criterion based on the value of the Hamiltonian H. Thus if $(\mathbf{z}, \mathbf{r})$ is the initial state and $(\mathbf{z}^\star, \mathbf{r}^\star)$ is the state after the leapfrog integration, then this candidate state is accepted with probability

$$\min\left(1, \exp\{H(\mathbf{z}, \mathbf{r}) - H(\mathbf{z}^\star, \mathbf{r}^\star)\}\right). \tag{11.67}$$

If the leapfrog integration were to simulate the Hamiltonian dynamics perfectly, then every such candidate step would automatically be accepted because the value of H would be unchanged. Due to numerical errors, the value of H may sometimes decrease, and we would like the Metropolis criterion to remove any bias due to this effect and ensure that the resulting samples are indeed drawn from the required distribution. In order for this to be the case, we need to ensure that the update equations corresponding to the leapfrog integration satisfy detailed balance (11.40). This is easily achieved by modifying the leapfrog scheme as follows.

Before the start of each leapfrog integration sequence, we choose at random, with equal probability, whether to integrate forwards in time (using step size ϵ) or backwards in time (using step size $-\epsilon$). We first note that the leapfrog integration scheme (11.64), (11.65), and (11.66) is time-reversible, so that integration for L steps using step size $-\epsilon$ will exactly undo the effect of integration for L steps using step size ϵ. Next we show that the leapfrog integration preserves phase-space volume exactly. This follows from the fact that each step in the leapfrog scheme updates either a z_i variable or an r_i variable by an amount that is a function only of the other variable. As shown in Figure 11.14, this has the effect of shearing a region of phase space while not altering its volume.

Finally, we use these results to show that detailed balance holds. Consider a small region $\mathcal{R}$ of phase space that, under a sequence of L leapfrog iterations of step size ϵ, maps to a region $\mathcal{R}'$. Using conservation of volume under the leapfrog iteration, we see that if $\mathcal{R}$ has volume δV then so too will $\mathcal{R}'$. If we choose an initial point from the distribution (11.63) and then update it using L leapfrog interactions, the probability of the transition going from $\mathcal{R}$ to $\mathcal{R}'$ is given by

$$\frac{1}{Z_H}\exp(-H(\mathcal{R}))\delta V \frac{1}{2}\min\left\{1, \exp(-H(\mathcal{R}) + H(\mathcal{R}'))\right\}. \tag{11.68}$$

where the factor of $1/2$ arises from the probability of choosing to integrate with a positive step size rather than a negative one. Similarly, the probability of starting in

Figure 11.14 Each step of the leapfrog algorithm (11.64)–(11.66) modifies either a position variable z_i or a momentum variable r_i. Because the change to one variable is a function only of the other, any region in phase space will be sheared without change of volume.

region $\mathcal{R}'$ and integrating backwards in time to end up in region $\mathcal{R}$ is given by

$$\frac{1}{Z_H}\exp(-H(\mathcal{R}'))\delta V\frac{1}{2}\min\left\{1,\exp(-H(\mathcal{R}')+H(\mathcal{R}))\right\}. \tag{11.69}$$

It is easily seen that the two probabilities (11.68) and (11.69) are equal, and hence detailed balance holds. Note that this proof ignores any overlap between the regions $\mathcal{R}$ and $\mathcal{R}'$ but is easily generalized to allow for such overlap.

Exercise 11.17

It is not difficult to construct examples for which the leapfrog algorithm returns to its starting position after a finite number of iterations. In such cases, the random replacement of the momentum values before each leapfrog integration will not be sufficient to ensure ergodicity because the position variables will never be updated. Such phenomena are easily avoided by choosing the magnitude of the step size at random from some small interval, before each leapfrog integration.

We can gain some insight into the behaviour of the hybrid Monte Carlo algorithm by considering its application to a multivariate Gaussian. For convenience, consider a Gaussian distribution $p(\mathbf{z})$ with independent components, for which the Hamiltonian is given by

$$H(\mathbf{z},\mathbf{r}) = \frac{1}{2}\sum_i \frac{1}{\sigma_i^2}z_i^2 + \frac{1}{2}\sum_i r_i^2. \tag{11.70}$$

Our conclusions will be equally valid for a Gaussian distribution having correlated components because the hybrid Monte Carlo algorithm exhibits rotational isotropy. During the leapfrog integration, each pair of phase-space variables z_i, r_i evolves independently. However, the acceptance or rejection of the candidate point is based on the value of H, which depends on the values of all of the variables. Thus, a significant integration error in any one of the variables could lead to a high probability of rejection. In order that the discrete leapfrog integration be a reasonably

good approximation to the true continuous-time dynamics, it is necessary for the leapfrog integration scale ϵ to be smaller than the shortest length-scale over which the potential is varying significantly. This is governed by the smallest value of σ_i, which we denote by $\sigma_{\min}$. Recall that the goal of the leapfrog integration in hybrid Monte Carlo is to move a substantial distance through phase space to a new state that is relatively independent of the initial state and still achieve a high probability of acceptance. In order to achieve this, the leapfrog integration must be continued for a number of iterations of order $\sigma_{\max}/\sigma_{\min}$.

By contrast, consider the behaviour of a simple Metropolis algorithm with an isotropic Gaussian proposal distribution of variance s^2, considered earlier. In order to avoid high rejection rates, the value of s must be of order $\sigma_{\min}$. The exploration of state space then proceeds by a random walk and takes of order $(\sigma_{\max}/\sigma_{\min})^2$ steps to arrive at a roughly independent state.

11.6. Estimating the Partition Function

As we have seen, most of the sampling algorithms considered in this chapter require only the functional form of the probability distribution up to a multiplicative constant. Thus if we write

$$p_E(\mathbf{z}) = \frac{1}{Z_E}\exp(-E(\mathbf{z})) \tag{11.71}$$

then the value of the normalization constant Z_E, also known as the partition function, is not needed in order to draw samples from $p(\mathbf{z})$. However, knowledge of the value of Z_E can be useful for Bayesian model comparison since it represents the model evidence (i.e., the probability of the observed data given the model), and so it is of interest to consider how its value might be obtained. We assume that direct evaluation by summing, or integrating, the function $\exp(-E(\mathbf{z}))$ over the state space of $\mathbf{z}$ is intractable.

For model comparison, it is actually the ratio of the partition functions for two models that is required. Multiplication of this ratio by the ratio of prior probabilities gives the ratio of posterior probabilities, which can then be used for model selection or model averaging.

One way to estimate a ratio of partition functions is to use importance sampling from a distribution with energy function $G(\mathbf{z})$

$$\begin{aligned}
\frac{Z_E}{Z_G} &= \frac{\sum_{\mathbf{z}}\exp(-E(\mathbf{z}))}{\sum_{\mathbf{z}}\exp(-G(\mathbf{z}))} \\
&= \frac{\sum_{\mathbf{z}}\exp(-E(\mathbf{z})+G(\mathbf{z}))\exp(-G(\mathbf{z}))}{\sum_{\mathbf{z}}\exp(-G(\mathbf{z}))} \\
&= \mathbb{E}_{G(\mathbf{z})}[\exp(-E+G)] \\
&\simeq \sum_l \exp(-E(\mathbf{z}^{(l)})+G(\mathbf{z}^{(l)}))
\end{aligned} \tag{11.72}$$

where $\{\mathbf{z}^{(l)}\}$ are samples drawn from the distribution defined by $p_G(\mathbf{z})$. If the distribution p_G is one for which the partition function can be evaluated analytically, for example a Gaussian, then the absolute value of Z_E can be obtained.

This approach will only yield accurate results if the importance sampling distribution p_G is closely matched to the distribution p_E, so that the ratio p_E/p_G does not have wide variations. In practice, suitable analytically specified importance sampling distributions cannot readily be found for the kinds of complex models considered in this book.

An alternative approach is therefore to use the samples obtained from a Markov chain to define the importance-sampling distribution. If the transition probability for the Markov chain is given by $T(\mathbf{z}, \mathbf{z}')$, and the sample set is given by $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(L)}$, then the sampling distribution can be written as

$$\frac{1}{Z_G} \exp\left(-G(\mathbf{z})\right) = \sum_{l=1}^{L} T(\mathbf{z}^{(l)}, \mathbf{z}) \tag{11.73}$$

which can be used directly in (11.72).

Methods for estimating the ratio of two partition functions require for their success that the two corresponding distributions be reasonably closely matched. This is especially problematic if we wish to find the absolute value of the partition function for a complex distribution because it is only for relatively simple distributions that the partition function can be evaluated directly, and so attempting to estimate the ratio of partition functions directly is unlikely to be successful. This problem can be tackled using a technique known as *chaining* (Neal, 1993; Barber and Bishop, 1997), which involves introducing a succession of intermediate distributions $p_2, \ldots, p_{M-1}$ that interpolate between a simple distribution $p_1(\mathbf{z})$ for which we can evaluate the normalization coefficient Z_1 and the desired complex distribution $p_M(\mathbf{z})$. We then have

$$\frac{Z_M}{Z_1} = \frac{Z_2}{Z_1}\frac{Z_3}{Z_2} \cdots \frac{Z_M}{Z_{M-1}} \tag{11.74}$$

in which the intermediate ratios can be determined using Monte Carlo methods as discussed above. One way to construct such a sequence of intermediate systems is to use an energy function containing a continuous parameter $0 \leqslant \alpha \leqslant 1$ that interpolates between the two distributions

$$E_\alpha(\mathbf{z}) = (1-\alpha)E_1(\mathbf{z}) + \alpha E_M(\mathbf{z}). \tag{11.75}$$

If the intermediate ratios in (11.74) are to be found using Monte Carlo, it may be more efficient to use a single Markov chain run than to restart the Markov chain for each ratio. In this case, the Markov chain is run initially for the system p_1 and then after some suitable number of steps moves on to the next distribution in the sequence. Note, however, that the system must remain close to the equilibrium distribution at each stage.

Exercises

11.1 ($\star$) **www** Show that the finite sample estimator $\widehat{f}$ defined by (11.2) has mean equal to $\mathbb{E}[f]$ and variance given by (11.3).

11.2 ($\star$) Suppose that z is a random variable with uniform distribution over $(0, 1)$ and that we transform z using $y = h^{-1}(z)$ where $h(y)$ is given by (11.6). Show that y has the distribution $p(y)$.

11.3 ($\star$) Given a random variable z that is uniformly distributed over $(0, 1)$, find a transformation $y = f(z)$ such that y has a Cauchy distribution given by (11.8).

11.4 ($\star\star$) Suppose that z_1 and z_2 are uniformly distributed over the unit circle, as shown in Figure 11.3, and that we make the change of variables given by (11.10) and (11.11). Show that (y_1, y_2) will be distributed according to (11.12).

11.5 ($\star$) **www** Let $\mathbf{z}$ be a D-dimensional random variable having a Gaussian distribution with zero mean and unit covariance matrix, and suppose that the positive definite symmetric matrix $\boldsymbol{\Sigma}$ has the Cholesky decomposition $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$ where $\mathbf{L}$ is a lower-triangular matrix (i.e., one with zeros above the leading diagonal). Show that the variable $\mathbf{y} = \boldsymbol{\mu} + \mathbf{L}\mathbf{z}$ has a Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$. This provides a technique for generating samples from a general multivariate Gaussian using samples from a univariate Gaussian having zero mean and unit variance.

11.6 ($\star\star$) **www** In this exercise, we show more carefully that rejection sampling does indeed draw samples from the desired distribution $p(\mathbf{z})$. Suppose the proposal distribution is $q(\mathbf{z})$ and show that the probability of a sample value $\mathbf{z}$ being accepted is given by $\widetilde{p}(\mathbf{z})/kq(\mathbf{z})$ where $\widetilde{p}$ is any unnormalized distribution that is proportional to $p(\mathbf{z})$, and the constant k is set to the smallest value that ensures $kq(\mathbf{z}) \geqslant \widetilde{p}(\mathbf{z})$ for all values of $\mathbf{z}$. Note that the probability of drawing a value $\mathbf{z}$ is given by the probability of drawing that value from $q(\mathbf{z})$ times the probability of accepting that value given that it has been drawn. Make use of this, along with the sum and product rules of probability, to write down the normalized form for the distribution over $\mathbf{z}$, and show that it equals $p(\mathbf{z})$.

11.7 ($\star$) Suppose that z has a uniform distribution over the interval $[0, 1]$. Show that the variable $y = b \tan z + c$ has a Cauchy distribution given by (11.16).

11.8 ($\star\star$) Determine expressions for the coefficients k_i in the envelope distribution (11.17) for adaptive rejection sampling using the requirements of continuity and normalization.

11.9 ($\star\star$) By making use of the technique discussed in Section 11.1.1 for sampling from a single exponential distribution, devise an algorithm for sampling from the piecewise exponential distribution defined by (11.17).

11.10 ($\star$) Show that the simple random walk over the integers defined by (11.34), (11.35), and (11.36) has the property that $\mathbb{E}[(z^{(\tau)})^2] = \mathbb{E}[(z^{(\tau-1)})^2] + 1/2$ and hence by induction that $\mathbb{E}[(z^{(\tau)})^2] = \tau/2$.

Figure 11.15 A probability distribution over two variables z_1 and z_2 that is uniform over the shaded regions and that is zero everywhere else.

11.11 ($\star\star$) **www** Show that the Gibbs sampling algorithm, discussed in Section 11.3, satisfies detailed balance as defined by (11.40).

11.12 ($\star$) Consider the distribution shown in Figure 11.15. Discuss whether the standard Gibbs sampling procedure for this distribution is ergodic, and therefore whether it would sample correctly from this distribution

11.13 ($\star\star$) Consider the simple 3-node graph shown in Figure 11.16 in which the observed node x is given by a Gaussian distribution $\mathcal{N}(x|\mu, \tau^{-1})$ with mean μ and precision τ. Suppose that the marginal distributions over the mean and precision are given by $\mathcal{N}(\mu|\mu_0, s_0)$ and $\mathrm{Gam}(\tau|a, b)$, where $\mathrm{Gam}(\cdot|\cdot,\cdot)$ denotes a gamma distribution. Write down expressions for the conditional distributions $p(\mu|x, \tau)$ and $p(\tau|x, \mu)$ that would be required in order to apply Gibbs sampling to the posterior distribution $p(\mu, \tau|x)$.

11.14 ($\star$) Verify that the over-relaxation update (11.50), in which z_i has mean μ_i and variance σ_i, and where ν has zero mean and unit variance, gives a value z_i' with mean μ_i and variance σ_i^2.

11.15 ($\star$) **www** Using (11.56) and (11.57), show that the Hamiltonian equation (11.58) is equivalent to (11.53). Similarly, using (11.57) show that (11.59) is equivalent to (11.55).

11.16 ($\star$) By making use of (11.56), (11.57), and (11.63), show that the conditional distribution $p(\mathbf{r}|\mathbf{z})$ is a Gaussian.

Figure 11.16 A graph involving an observed Gaussian variable x with prior distributions over its mean μ and precision τ.

11.17 ($\star$) **www** Verify that the two probabilities (11.68) and (11.69) are equal, and hence that detailed balance holds for the hybrid Monte Carlo algorithm.

12 Continuous Latent Variables

In Chapter 9, we discussed probabilistic models having discrete latent variables, such as the mixture of Gaussians. We now explore models in which some, or all, of the latent variables are continuous. An important motivation for such models is that many data sets have the property that the data points all lie close to a manifold of much lower dimensionality than that of the original data space. To see why this might arise, consider an artificial data set constructed by taking one of the off-line digits, represented by a 64×64 pixel grey-level image, and embedding it in a larger image of size 100×100 by padding with pixels having the value zero (corresponding to white pixels) in which the location and orientation of the digit is varied at random, as illustrated in Figure 12.1. Each of the resulting images is represented by a point in the $100 \times 100 = 10,000$-dimensional data space. However, across a data set of such images, there are only three *degrees of freedom* of variability, corresponding to the vertical and horizontal translations and the rotations. The data points will therefore live on a subspace of the data space whose *intrinsic dimensionality* is three. Note

Appendix A

Figure 12.1 A synthetic data set obtained by taking one of the off-line digit images and creating multiple copies in each of which the digit has undergone a random displacement and rotation within some larger image field. The resulting images each have $100 \times 100 = 10,000$ pixels.

that the manifold will be nonlinear because, for instance, if we translate the digit past a particular pixel, that pixel value will go from zero (white) to one (black) and back to zero again, which is clearly a nonlinear function of the digit position. In this example, the translation and rotation parameters are latent variables because we observe only the image vectors and are not told which values of the translation or rotation variables were used to create them.

For real digit image data, there will be a further degree of freedom arising from scaling. Moreover there will be multiple additional degrees of freedom associated with more complex deformations due to the variability in an individual's writing as well as the differences in writing styles between individuals. Nevertheless, the number of such degrees of freedom will be small compared to the dimensionality of the data set.

Appendix A

Another example is provided by the oil flow data set, in which (for a given geometrical configuration of the gas, water, and oil phases) there are only two degrees of freedom of variability corresponding to the fraction of oil in the pipe and the fraction of water (the fraction of gas then being determined). Although the data space comprises 12 measurements, a data set of points will lie close to a two-dimensional manifold embedded within this space. In this case, the manifold comprises several distinct segments corresponding to different flow regimes, each such segment being a (noisy) continuous two-dimensional manifold. If our goal is data compression, or density modelling, then there can be benefits in exploiting this manifold structure.

In practice, the data points will not be confined precisely to a smooth low-dimensional manifold, and we can interpret the departures of data points from the manifold as 'noise'. This leads naturally to a generative view of such models in which we first select a point within the manifold according to some latent variable distribution and then generate an observed data point by adding noise, drawn from some conditional distribution of the data variables given the latent variables.

The simplest continuous latent variable model assumes Gaussian distributions for both the latent and observed variables and makes use of a linear-Gaussian dependence of the observed variables on the state of the latent variables. This leads to a probabilistic formulation of the well-known technique of principal component analysis (PCA), as well as to a related model called factor analysis.

Section 8.1.4

Section 12.1

In this chapter w will begin with a standard, nonprobabilistic treatment of PCA, and then we show how PCA arises naturally as the maximum likelihood solution to

Figure 12.2 Principal component analysis seeks a space of lower dimensionality, known as the principal subspace and denoted by the magenta line, such that the orthogonal projection of the data points (red dots) onto this subspace maximizes the variance of the projected points (green dots). An alternative definition of PCA is based on minimizing the sum-of-squares of the projection errors, indicated by the blue lines.

Section 12.2 a particular form of linear-Gaussian latent variable model. This probabilistic reformulation brings many advantages, such as the use of EM for parameter estimation, principled extensions to mixtures of PCA models, and Bayesian formulations that allow the number of principal components to be determined automatically from the data. Finally, we discuss briefly several generalizations of the latent variable concept that go beyond the linear-Gaussian assumption including non-Gaussian latent variables, which leads to the framework of *independent component analysis*, as well as *Section 12.4* models having a nonlinear relationship between latent and observed variables.

12.1. Principal Component Analysis

Principal component analysis, or PCA, is a technique that is widely used for applications such as dimensionality reduction, lossy data compression, feature extraction, and data visualization (Jolliffe, 2002). It is also known as the *Karhunen-Loève* transform.

There are two commonly used definitions of PCA that give rise to the same algorithm. PCA can be defined as the orthogonal projection of the data onto a lower dimensional linear space, known as the *principal subspace*, such that the variance of the projected data is maximized (Hotelling, 1933). Equivalently, it can be defined as the linear projection that minimizes the average projection cost, defined as the mean squared distance between the data points and their projections (Pearson, 1901). The process of orthogonal projection is illustrated in Figure 12.2. We consider each of these definitions in turn.

12.1.1 Maximum variance formulation

Consider a data set of observations $\{\mathbf{x}_n\}$ where $n = 1, \ldots, N$, and $\mathbf{x}_n$ is a Euclidean variable with dimensionality D. Our goal is to project the data onto a space having dimensionality $M < D$ while maximizing the variance of the projected data. For the moment, we shall assume that the value of M is given. Later in this

chapter, we shall consider techniques to determine an appropriate value of M from the data.

To begin with, consider the projection onto a one-dimensional space ($M = 1$). We can define the direction of this space using a D-dimensional vector $\mathbf{u}_1$, which for convenience (and without loss of generality) we shall choose to be a unit vector so that $\mathbf{u}_1^{\mathrm{T}}\mathbf{u}_1 = 1$ (note that we are only interested in the direction defined by $\mathbf{u}_1$, not in the magnitude of $\mathbf{u}_1$ itself). Each data point $\mathbf{x}_n$ is then projected onto a scalar value $\mathbf{u}_1^{\mathrm{T}}\mathbf{x}_n$. The mean of the projected data is $\mathbf{u}_1^{\mathrm{T}}\overline{\mathbf{x}}$ where $\overline{\mathbf{x}}$ is the sample set mean given by

$$\overline{\mathbf{x}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n \tag{12.1}$$

and the variance of the projected data is given by

$$\frac{1}{N}\sum_{n=1}^{N}\left\{\mathbf{u}_1^{\mathrm{T}}\mathbf{x}_n - \mathbf{u}_1^{\mathrm{T}}\overline{\mathbf{x}}\right\}^2 = \mathbf{u}_1^{\mathrm{T}}\mathbf{S}\mathbf{u}_1 \tag{12.2}$$

where $\mathbf{S}$ is the data covariance matrix defined by

$$\mathbf{S} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n - \overline{\mathbf{x}})(\mathbf{x}_n - \overline{\mathbf{x}})^{\mathrm{T}}. \tag{12.3}$$

We now maximize the projected variance $\mathbf{u}_1^{\mathrm{T}}\mathbf{S}\mathbf{u}_1$ with respect to $\mathbf{u}_1$. Clearly, this has to be a constrained maximization to prevent $\|\mathbf{u}_1\| \to \infty$. The appropriate constraint comes from the normalization condition $\mathbf{u}_1^{\mathrm{T}}\mathbf{u}_1 = 1$. To enforce this constraint, we introduce a Lagrange multiplier that we shall denote by λ_1, and then make an unconstrained maximization of

Appendix E

$$\mathbf{u}_1^{\mathrm{T}}\mathbf{S}\mathbf{u}_1 + \lambda_1\left(1 - \mathbf{u}_1^{\mathrm{T}}\mathbf{u}_1\right). \tag{12.4}$$

By setting the derivative with respect to $\mathbf{u}_1$ equal to zero, we see that this quantity will have a stationary point when

$$\mathbf{S}\mathbf{u}_1 = \lambda_1\mathbf{u}_1 \tag{12.5}$$

which says that $\mathbf{u}_1$ must be an eigenvector of $\mathbf{S}$. If we left-multiply by $\mathbf{u}_1^{\mathrm{T}}$ and make use of $\mathbf{u}_1^{\mathrm{T}}\mathbf{u}_1 = 1$, we see that the variance is given by

$$\mathbf{u}_1^{\mathrm{T}}\mathbf{S}\mathbf{u}_1 = \lambda_1 \tag{12.6}$$

and so the variance will be a maximum when we set $\mathbf{u}_1$ equal to the eigenvector having the largest eigenvalue λ_1. This eigenvector is known as the first principal component.

We can define additional principal components in an incremental fashion by choosing each new direction to be that which maximizes the projected variance

amongst all possible directions orthogonal to those already considered. If we consider the general case of an M-dimensional projection space, the optimal linear projection for which the variance of the projected data is maximized is now defined by the M eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_M$ of the data covariance matrix $\mathbf{S}$ corresponding to the M largest eigenvalues $\lambda_1, \ldots, \lambda_M$. This is easily shown using proof by induction.

Exercise 12.1

To summarize, principal component analysis involves evaluating the mean $\overline{\mathbf{x}}$ and the covariance matrix $\mathbf{S}$ of the data set and then finding the M eigenvectors of $\mathbf{S}$ corresponding to the M largest eigenvalues. Algorithms for finding eigenvectors and eigenvalues, as well as additional theorems related to eigenvector decomposition, can be found in Golub and Van Loan (1996). Note that the computational cost of computing the full eigenvector decomposition for a matrix of size $D \times D$ is $O(D^3)$. If we plan to project our data onto the first M principal components, then we only need to find the first M eigenvalues and eigenvectors. This can be done with more efficient techniques, such as the *power method* (Golub and Van Loan, 1996), that scale like $O(MD^2)$, or alternatively we can make use of the EM algorithm.

Section 12.2.2

12.1.2 Minimum-error formulation

We now discuss an alternative formulation of PCA based on projection error minimization. To do this, we introduce a complete orthonormal set of D-dimensional basis vectors $\{\mathbf{u}_i\}$ where $i = 1, \ldots, D$ that satisfy

Appendix C

$$\mathbf{u}_i^{\mathrm{T}} \mathbf{u}_j = \delta_{ij}. \tag{12.7}$$

Because this basis is complete, each data point can be represented exactly by a linear combination of the basis vectors

$$\mathbf{x}_n = \sum_{i=1}^{D} \alpha_{ni} \mathbf{u}_i \tag{12.8}$$

where the coefficients α_{ni} will be different for different data points. This simply corresponds to a rotation of the coordinate system to a new system defined by the $\{\mathbf{u}_i\}$, and the original D components $\{x_{n1}, \ldots, x_{nD}\}$ are replaced by an equivalent set $\{\alpha_{n1}, \ldots, \alpha_{nD}\}$. Taking the inner product with $\mathbf{u}_j$, and making use of the orthonormality property, we obtain $\alpha_{nj} = \mathbf{x}_n^{\mathrm{T}} \mathbf{u}_j$, and so without loss of generality we can write

$$\mathbf{x}_n = \sum_{i=1}^{D} \left(\mathbf{x}_n^{\mathrm{T}} \mathbf{u}_i\right) \mathbf{u}_i. \tag{12.9}$$

Our goal, however, is to approximate this data point using a representation involving a restricted number $M < D$ of variables corresponding to a projection onto a lower-dimensional subspace. The M-dimensional linear subspace can be represented, without loss of generality, by the first M of the basis vectors, and so we approximate each data point $\mathbf{x}_n$ by

$$\widetilde{\mathbf{x}}_n = \sum_{i=1}^{M} z_{ni} \mathbf{u}_i + \sum_{i=M+1}^{D} b_i \mathbf{u}_i \tag{12.10}$$

where the $\{z_{ni}\}$ depend on the particular data point, whereas the $\{b_i\}$ are constants that are the same for all data points. We are free to choose the $\{\mathbf{u}_i\}$, the $\{z_{ni}\}$, and the $\{b_i\}$ so as to minimize the distortion introduced by the reduction in dimensionality. As our distortion measure, we shall use the squared distance between the original data point $\mathbf{x}_n$ and its approximation $\widetilde{\mathbf{x}}_n$, averaged over the data set, so that our goal is to minimize

$$J = \frac{1}{N}\sum_{n=1}^{N} \|\mathbf{x}_n - \widetilde{\mathbf{x}}_n\|^2. \tag{12.11}$$

Consider first of all the minimization with respect to the quantities $\{z_{ni}\}$. Substituting for $\widetilde{\mathbf{x}}_n$, setting the derivative with respect to z_{nj} to zero, and making use of the orthonormality conditions, we obtain

$$z_{nj} = \mathbf{x}_n^{\mathrm{T}}\mathbf{u}_j \tag{12.12}$$

where $j = 1, \ldots, M$. Similarly, setting the derivative of J with respect to b_i to zero, and again making use of the orthonormality relations, gives

$$b_j = \overline{\mathbf{x}}^{\mathrm{T}}\mathbf{u}_j \tag{12.13}$$

where $j = M+1, \ldots, D$. If we substitute for z_{ni} and b_i, and make use of the general expansion (12.9), we obtain

$$\mathbf{x}_n - \widetilde{\mathbf{x}}_n = \sum_{i=M+1}^{D} \left\{(\mathbf{x}_n - \overline{\mathbf{x}})^{\mathrm{T}}\mathbf{u}_i\right\}\mathbf{u}_i \tag{12.14}$$

from which we see that the displacement vector from $\mathbf{x}_n$ to $\widetilde{\mathbf{x}}_n$ lies in the space orthogonal to the principal subspace, because it is a linear combination of $\{\mathbf{u}_i\}$ for $i = M+1, \ldots, D$, as illustrated in Figure 12.2. This is to be expected because the projected points $\widetilde{\mathbf{x}}_n$ must lie within the principal subspace, but we can move them freely within that subspace, and so the minimum error is given by the orthogonal projection.

We therefore obtain an expression for the distortion measure J as a function purely of the $\{\mathbf{u}_i\}$ in the form

$$J = \frac{1}{N}\sum_{n=1}^{N}\sum_{i=M+1}^{D}\left(\mathbf{x}_n^{\mathrm{T}}\mathbf{u}_i - \overline{\mathbf{x}}^{\mathrm{T}}\mathbf{u}_i\right)^2 = \sum_{i=M+1}^{D}\mathbf{u}_i^{\mathrm{T}}\mathbf{S}\mathbf{u}_i. \tag{12.15}$$

There remains the task of minimizing J with respect to the $\{\mathbf{u}_i\}$, which must be a constrained minimization otherwise we will obtain the vacuous result $\mathbf{u}_i = 0$. The constraints arise from the orthonormality conditions and, as we shall see, the solution will be expressed in terms of the eigenvector expansion of the covariance matrix. Before considering a formal solution, let us try to obtain some intuition about the result by considering the case of a two-dimensional data space $D = 2$ and a one-dimensional principal subspace $M = 1$. We have to choose a direction $\mathbf{u}_2$ so as to

minimize $J = \mathbf{u}_2^{\mathrm{T}}\mathbf{S}\mathbf{u}_2$, subject to the normalization constraint $\mathbf{u}_2^{\mathrm{T}}\mathbf{u}_2 = 1$. Using a Lagrange multiplier λ_2 to enforce the constraint, we consider the minimization of

$$\widetilde{J} = \mathbf{u}_2^{\mathrm{T}}\mathbf{S}\mathbf{u}_2 + \lambda_2\left(1 - \mathbf{u}_2^{\mathrm{T}}\mathbf{u}_2\right). \tag{12.16}$$

Setting the derivative with respect to $\mathbf{u}_2$ to zero, we obtain $\mathbf{S}\mathbf{u}_2 = \lambda_2\mathbf{u}_2$ so that $\mathbf{u}_2$ is an eigenvector of $\mathbf{S}$ with eigenvalue λ_2. Thus any eigenvector will define a stationary point of the distortion measure. To find the value of J at the minimum, we back-substitute the solution for $\mathbf{u}_2$ into the distortion measure to give $J = \lambda_2$. We therefore obtain the minimum value of J by choosing $\mathbf{u}_2$ to be the eigenvector corresponding to the smaller of the two eigenvalues. Thus we should choose the principal subspace to be aligned with the eigenvector having the *larger* eigenvalue. This result accords with our intuition that, in order to minimize the average squared projection distance, we should choose the principal component subspace to pass through the mean of the data points and to be aligned with the directions of maximum variance. For the case when the eigenvalues are equal, any choice of principal direction will give rise to the same value of J.

Exercise 12.2 The general solution to the minimization of J for arbitrary D and arbitrary $M < D$ is obtained by choosing the $\{\mathbf{u}_i\}$ to be eigenvectors of the covariance matrix given by

$$\mathbf{S}\mathbf{u}_i = \lambda_i\mathbf{u}_i \tag{12.17}$$

where $i = 1, \ldots, D$, and as usual the eigenvectors $\{\mathbf{u}_i\}$ are chosen to be orthonormal. The corresponding value of the distortion measure is then given by

$$J = \sum_{i=M+1}^{D} \lambda_i \tag{12.18}$$

which is simply the sum of the eigenvalues of those eigenvectors that are orthogonal to the principal subspace. We therefore obtain the minimum value of J by selecting these eigenvectors to be those having the $D - M$ smallest eigenvalues, and hence the eigenvectors defining the principal subspace are those corresponding to the M largest eigenvalues.

Although we have considered $M < D$, the PCA analysis still holds if $M = D$, in which case there is no dimensionality reduction but simply a rotation of the coordinate axes to align with principal components.

Finally, it is worth noting that there exists a closely related linear dimensionality reduction technique called *canonical correlation analysis*, or *CCA* (Hotelling, 1936; Bach and Jordan, 2002). Whereas PCA works with a single random variable, CCA considers two (or more) variables and tries to find a corresponding pair of linear subspaces that have high cross-correlation, so that each component within one of the subspaces is correlated with a single component from the other subspace. Its solution can be expressed in terms of a generalized eigenvector problem.

12.1.3 Applications of PCA

We can illustrate the use of PCA for data compression by considering the off-line digits data set. Because each eigenvector of the covariance matrix is a vector

Appendix A

Figure 12.3 The mean vector $\overline{\mathbf{x}}$ along with the first four PCA eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_4$ for the off-line digits data set, together with the corresponding eigenvalues.

in the original D-dimensional space, we can represent the eigenvectors as images of the same size as the data points. The first five eigenvectors, along with the corresponding eigenvalues, are shown in Figure 12.3. A plot of the complete spectrum of eigenvalues, sorted into decreasing order, is shown in Figure 12.4(a). The distortion measure J associated with choosing a particular value of M is given by the sum of the eigenvalues from $M+1$ up to D and is plotted for different values of M in Figure 12.4(b).

If we substitute (12.12) and (12.13) into (12.10), we can write the PCA approximation to a data vector $\mathbf{x}_n$ in the form

$$\widetilde{\mathbf{x}}_n = \sum_{i=1}^{M} (\mathbf{x}_n^{\mathrm{T}} \mathbf{u}_i) \mathbf{u}_i + \sum_{i=M+1}^{D} (\overline{\mathbf{x}}^{\mathrm{T}} \mathbf{u}_i) \mathbf{u}_i \tag{12.19}$$

$$= \overline{\mathbf{x}} + \sum_{i=1}^{M} \left(\mathbf{x}_n^{\mathrm{T}} \mathbf{u}_i - \overline{\mathbf{x}}^{\mathrm{T}} \mathbf{u}_i \right) \mathbf{u}_i \tag{12.20}$$

Figure 12.4 (a) Plot of the eigenvalue spectrum for the off-line digits data set. (b) Plot of the sum of the discarded eigenvalues, which represents the sum-of-squares distortion J introduced by projecting the data onto a principal component subspace of dimensionality M.

Figure 12.5 An original example from the off-line digits data set together with its PCA reconstructions obtained by retaining M principal components for various values of M. As M increases the reconstruction becomes more accurate and would become perfect when $M = D = 28 \times 28 = 784$.

where we have made use of the relation

$$\overline{\mathbf{x}} = \sum_{i=1}^{D} \left(\overline{\mathbf{x}}^{\mathrm{T}} \mathbf{u}_i\right) \mathbf{u}_i \tag{12.21}$$

which follows from the completeness of the $\{\mathbf{u}_i\}$. This represents a compression of the data set, because for each data point we have replaced the D-dimensional vector $\mathbf{x}_n$ with an M-dimensional vector having components $\left(\mathbf{x}_n^{\mathrm{T}} \mathbf{u}_i - \overline{\mathbf{x}}^{\mathrm{T}} \mathbf{u}_i\right)$. The smaller the value of M, the greater the degree of compression. Examples of PCA reconstructions of data points for the digits data set are shown in Figure 12.5.

Another application of principal component analysis is to data pre-processing. In this case, the goal is not dimensionality reduction but rather the transformation of a data set in order to standardize certain of its properties. This can be important in allowing subsequent pattern recognition algorithms to be applied successfully to the data set. Typically, it is done when the original variables are measured in various different units or have significantly different variability. For instance in the Old Faithful data set, the time between eruptions is typically an order of magnitude greater than the duration of an eruption. When we applied the K-means algorithm to this data set, we first made a separate linear re-scaling of the individual variables such that each variable had zero mean and unit variance. This is known as *standardizing* the data, and the covariance matrix for the standardized data has components

Appendix A

Section 9.1

$$\rho_{ij} = \frac{1}{N} \sum_{n=1}^{N} \frac{(x_{ni} - \overline{x}_i)}{\sigma_i} \frac{(x_{nj} - \overline{x}_j)}{\sigma_j} \tag{12.22}$$

where σ_i is the variance of x_i. This is known as the *correlation* matrix of the original data and has the property that if two components x_i and x_j of the data are perfectly correlated, then $\rho_{ij} = 1$, and if they are uncorrelated, then $\rho_{ij} = 0$.

However, using PCA we can make a more substantial normalization of the data to give it zero mean and unit covariance, so that different variables become decorrelated. To do this, we first write the eigenvector equation (12.17) in the form

$$\mathbf{SU} = \mathbf{UL} \tag{12.23}$$

Figure 12.6 Illustration of the effects of linear pre-processing applied to the Old Faithful data set. The plot on the left shows the original data. The centre plot shows the result of standardizing the individual variables to zero mean and unit variance. Also shown are the principal axes of this normalized data set, plotted over the range $\pm\lambda_i^{1/2}$. The plot on the right shows the result of whitening of the data to give it zero mean and unit covariance.

where $\mathbf{L}$ is a $D \times D$ diagonal matrix with elements λ_i, and $\mathbf{U}$ is a $D \times D$ orthogonal matrix with columns given by $\mathbf{u}_i$. Then we define, for each data point $\mathbf{x}_n$, a transformed value given by

$$\mathbf{y}_n = \mathbf{L}^{-1/2}\mathbf{U}^{\mathrm{T}}(\mathbf{x}_n - \overline{\mathbf{x}}) \tag{12.24}$$

where $\overline{\mathbf{x}}$ is the sample mean defined by (12.1). Clearly, the set $\{\mathbf{y}_n\}$ has zero mean, and its covariance is given by the identity matrix because

$$\begin{aligned}\frac{1}{N}\sum_{n=1}^{N}\mathbf{y}_n\mathbf{y}_n^{\mathrm{T}} &= \frac{1}{N}\sum_{n=1}^{N}\mathbf{L}^{-1/2}\mathbf{U}^{\mathrm{T}}(\mathbf{x}_n - \overline{\mathbf{x}})(\mathbf{x}_n - \overline{\mathbf{x}})^{\mathrm{T}}\mathbf{U}\mathbf{L}^{-1/2} \\ &= \mathbf{L}^{-1/2}\mathbf{U}^{\mathrm{T}}\mathbf{S}\mathbf{U}\mathbf{L}^{-1/2} = \mathbf{L}^{-1/2}\mathbf{L}\mathbf{L}^{-1/2} = \mathbf{I}. \end{aligned} \tag{12.25}$$

This operation is known as *whitening* or *sphereing* the data and is illustrated for the Old Faithful data set in Figure 12.6.

Appendix A

It is interesting to compare PCA with the Fisher linear discriminant which was discussed in Section 4.1.4. Both methods can be viewed as techniques for linear dimensionality reduction. However, PCA is unsupervised and depends only on the values $\mathbf{x}_n$ whereas Fisher linear discriminant also uses class-label information. This difference is highlighted by the example in Figure 12.7.

Another common application of principal component analysis is to data visualization. Here each data point is projected onto a two-dimensional ($M = 2$) principal subspace, so that a data point $\mathbf{x}_n$ is plotted at Cartesian coordinates given by $\mathbf{x}_n^{\mathrm{T}}\mathbf{u}_1$ and $\mathbf{x}_n^{\mathrm{T}}\mathbf{u}_2$, where $\mathbf{u}_1$ and $\mathbf{u}_2$ are the eigenvectors corresponding to the largest and second largest eigenvalues. An example of such a plot, for the oil flow data set, is shown in Figure 12.8.

Appendix A

Figure 12.7 A comparison of principal component analysis with Fisher's linear discriminant for linear dimensionality reduction. Here the data in two dimensions, belonging to two classes shown in red and blue, is to be projected onto a single dimension. PCA chooses the direction of maximum variance, shown by the magenta curve, which leads to strong class overlap, whereas the Fisher linear discriminant takes account of the class labels and leads to a projection onto the green curve giving much better class separation.

Figure 12.8 Visualization of the oil flow data set obtained by projecting the data onto the first two principal components. The red, blue, and green points correspond to the 'laminar', 'homogeneous', and 'annular' flow configurations respectively.

12.1.4 PCA for high-dimensional data

In some applications of principal component analysis, the number of data points is smaller than the dimensionality of the data space. For example, we might want to apply PCA to a data set of a few hundred images, each of which corresponds to a vector in a space of potentially several million dimensions (corresponding to three colour values for each of the pixels in the image). Note that in a D-dimensional space a set of N points, where $N < D$, defines a linear subspace whose dimensionality is at most $N - 1$, and so there is little point in applying PCA for values of M that are greater than $N - 1$. Indeed, if we perform PCA we will find that at least $D - N + 1$ of the eigenvalues are zero, corresponding to eigenvectors along whose directions the data set has zero variance. Furthermore, typical algorithms for finding the eigenvectors of a $D \times D$ matrix have a computational cost that scales like $O(D^3)$, and so for applications such as the image example, a direct application of PCA will be computationally infeasible.

We can resolve this problem as follows. First, let us define $\mathbf{X}$ to be the $(N \times D)$-

dimensional centred data matrix, whose n^{th} row is given by $(\mathbf{x}_n - \overline{\mathbf{x}})^{\mathrm{T}}$. The covariance matrix (12.3) can then be written as $\mathbf{S} = N^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{X}$, and the corresponding eigenvector equation becomes

$$\frac{1}{N}\mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{u}_i = \lambda_i \mathbf{u}_i. \tag{12.26}$$

Now pre-multiply both sides by $\mathbf{X}$ to give

$$\frac{1}{N}\mathbf{X}\mathbf{X}^{\mathrm{T}}(\mathbf{X}\mathbf{u}_i) = \lambda_i(\mathbf{X}\mathbf{u}_i). \tag{12.27}$$

If we now define $\mathbf{v}_i = \mathbf{X}\mathbf{u}_i$, we obtain

$$\frac{1}{N}\mathbf{X}\mathbf{X}^{\mathrm{T}}\mathbf{v}_i = \lambda_i \mathbf{v}_i \tag{12.28}$$

which is an eigenvector equation for the $N \times N$ matrix $N^{-1}\mathbf{X}\mathbf{X}^{\mathrm{T}}$. We see that this has the same $N-1$ eigenvalues as the original covariance matrix (which itself has an additional $D - N + 1$ eigenvalues of value zero). Thus we can solve the eigenvector problem in spaces of lower dimensionality with computational cost $O(N^3)$ instead of $O(D^3)$. In order to determine the eigenvectors, we multiply both sides of (12.28) by $\mathbf{X}^{\mathrm{T}}$ to give

$$\left(\frac{1}{N}\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)(\mathbf{X}^{\mathrm{T}}\mathbf{v}_i) = \lambda_i(\mathbf{X}^{\mathrm{T}}\mathbf{v}_i) \tag{12.29}$$

from which we see that $(\mathbf{X}^{\mathrm{T}}\mathbf{v}_i)$ is an eigenvector of $\mathbf{S}$ with eigenvalue λ_i. Note, however, that these eigenvectors need not be normalized. To determine the appropriate normalization, we re-scale $\mathbf{u}_i \propto \mathbf{X}^{\mathrm{T}}\mathbf{v}_i$ by a constant such that $\|\mathbf{u}_i\| = 1$, which, assuming $\mathbf{v}_i$ has been normalized to unit length, gives

$$\mathbf{u}_i = \frac{1}{(N\lambda_i)^{1/2}}\mathbf{X}^{\mathrm{T}}\mathbf{v}_i. \tag{12.30}$$

In summary, to apply this approach we first evaluate $\mathbf{X}\mathbf{X}^{\mathrm{T}}$ and then find its eigenvectors and eigenvalues and then compute the eigenvectors in the original data space using (12.30).

12.2. Probabilistic PCA

The formulation of PCA discussed in the previous section was based on a linear projection of the data onto a subspace of lower dimensionality than the original data space. We now show that PCA can also be expressed as the maximum likelihood solution of a probabilistic latent variable model. This reformulation of PCA, known as *probabilistic PCA*, brings several advantages compared with conventional PCA:

- Probabilistic PCA represents a constrained form of the Gaussian distribution in which the number of free parameters can be restricted while still allowing the model to capture the dominant correlations in a data set.

- We can derive an EM algorithm for PCA that is computationally efficient in situations where only a few leading eigenvectors are required and that avoids having to evaluate the data covariance matrix as an intermediate step.

Section 12.2.2

- The combination of a probabilistic model and EM allows us to deal with missing values in the data set.
- Mixtures of probabilistic PCA models can be formulated in a principled way and trained using the EM algorithm.
- Probabilistic PCA forms the basis for a Bayesian treatment of PCA in which the dimensionality of the principal subspace can be found automatically from the data.

Section 12.2.3

- The existence of a likelihood function allows direct comparison with other probabilistic density models. By contrast, conventional PCA will assign a low reconstruction cost to data points that are close to the principal subspace even if they lie arbitrarily far from the training data.
- Probabilistic PCA can be used to model class-conditional densities and hence be applied to classification problems.
- The probabilistic PCA model can be run generatively to provide samples from the distribution.

This formulation of PCA as a probabilistic model was proposed independently by Tipping and Bishop (1997, 1999b) and by Roweis (1998). As we shall see later, it is closely related to *factor analysis* (Basilevsky, 1994).

Section 8.1.4

Probabilistic PCA is a simple example of the linear-Gaussian framework, in which all of the marginal and conditional distributions are Gaussian. We can formulate probabilistic PCA by first introducing an explicit latent variable $\mathbf{z}$ corresponding to the principal-component subspace. Next we define a Gaussian prior distribution $p(\mathbf{z})$ over the latent variable, together with a Gaussian conditional distribution $p(\mathbf{x}|\mathbf{z})$ for the observed variable $\mathbf{x}$ conditioned on the value of the latent variable. Specifically, the prior distribution over $\mathbf{z}$ is given by a zero-mean unit-covariance Gaussian

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbf{I}). \tag{12.31}$$

Similarly, the conditional distribution of the observed variable $\mathbf{x}$, conditioned on the value of the latent variable $\mathbf{z}$, is again Gaussian, of the form

$$p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|\mathbf{W}\mathbf{z} + \boldsymbol{\mu}, \sigma^2\mathbf{I}) \tag{12.32}$$

in which the mean of $\mathbf{x}$ is a general linear function of $\mathbf{z}$ governed by the $D \times M$ matrix $\mathbf{W}$ and the D-dimensional vector $\boldsymbol{\mu}$. Note that this factorizes with respect to the elements of $\mathbf{x}$, in other words this is an example of the naive Bayes model. As we shall see shortly, the columns of $\mathbf{W}$ span a linear subspace within the data space that corresponds to the principal subspace. The other parameter in this model is the scalar σ^2 governing the variance of the conditional distribution. Note that there is no

Section 8.2.2

Figure 12.9 An illustration of the generative view of the probabilistic PCA model for a two-dimensional data space and a one-dimensional latent space. An observed data point $\mathbf{x}$ is generated by first drawing a value $\hat{z}$ for the latent variable from its prior distribution $p(z)$ and then drawing a value for $\mathbf{x}$ from an isotropic Gaussian distribution (illustrated by the red circles) having mean $\mathbf{w}\hat{z} + \boldsymbol{\mu}$ and covariance $\sigma^2\mathbf{I}$. The green ellipses show the density contours for the marginal distribution $p(\mathbf{x})$.

loss of generality in assuming a zero mean, unit covariance Gaussian for the latent distribution $p(\mathbf{z})$ because a more general Gaussian distribution would give rise to an equivalent probabilistic model.

Exercise 12.4

We can view the probabilistic PCA model from a generative viewpoint in which a sampled value of the observed variable is obtained by first choosing a value for the latent variable and then sampling the observed variable conditioned on this latent value. Specifically, the D-dimensional observed variable $\mathbf{x}$ is defined by a linear transformation of the M-dimensional latent variable $\mathbf{z}$ plus additive Gaussian 'noise', so that

$$\mathbf{x} = \mathbf{W}\mathbf{z} + \boldsymbol{\mu} + \boldsymbol{\epsilon} \tag{12.33}$$

where $\mathbf{z}$ is an M-dimensional Gaussian latent variable, and $\boldsymbol{\epsilon}$ is a D-dimensional zero-mean Gaussian-distributed noise variable with covariance $\sigma^2\mathbf{I}$. This generative process is illustrated in Figure 12.9. Note that this framework is based on a mapping from latent space to data space, in contrast to the more conventional view of PCA discussed above. The reverse mapping, from data space to the latent space, will be obtained shortly using Bayes' theorem.

Suppose we wish to determine the values of the parameters $\mathbf{W}$, $\boldsymbol{\mu}$ and σ^2 using maximum likelihood. To write down the likelihood function, we need an expression for the marginal distribution $p(\mathbf{x})$ of the observed variable. This is expressed, from the sum and product rules of probability, in the form

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})\,\mathrm{d}\mathbf{z}. \tag{12.34}$$

Because this corresponds to a linear-Gaussian model, this marginal distribution is again Gaussian, and is given by

Exercise 12.7

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \mathbf{C}) \tag{12.35}$$

where the $D \times D$ covariance matrix $\mathbf{C}$ is defined by

$$\mathbf{C} = \mathbf{W}\mathbf{W}^{\mathrm{T}} + \sigma^2\mathbf{I}. \tag{12.36}$$

This result can also be derived more directly by noting that the predictive distribution will be Gaussian and then evaluating its mean and covariance using (12.33). This gives

$$\begin{aligned} \mathbb{E}[\mathbf{x}] &= \mathbb{E}[\mathbf{W}\mathbf{z} + \boldsymbol{\mu} + \boldsymbol{\epsilon}] = \boldsymbol{\mu} && (12.37) \\ \mathrm{cov}[\mathbf{x}] &= \mathbb{E}\left[(\mathbf{W}\mathbf{z} + \boldsymbol{\epsilon})(\mathbf{W}\mathbf{z} + \boldsymbol{\epsilon})^{\mathrm{T}}\right] \\ &= \mathbb{E}\left[\mathbf{W}\mathbf{z}\mathbf{z}^{\mathrm{T}}\mathbf{W}^{\mathrm{T}}\right] + \mathbb{E}[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}] = \mathbf{W}\mathbf{W}^{\mathrm{T}} + \sigma^2\mathbf{I} && (12.38) \end{aligned}$$

where we have used the fact that $\mathbf{z}$ and $\boldsymbol{\epsilon}$ are independent random variables and hence are uncorrelated.

Intuitively, we can think of the distribution $p(\mathbf{x})$ as being defined by taking an isotropic Gaussian 'spray can' and moving it across the principal subspace spraying Gaussian ink with density determined by σ^2 and weighted by the prior distribution. The accumulated ink density gives rise to a 'pancake' shaped distribution representing the marginal density $p(\mathbf{x})$.

The predictive distribution $p(\mathbf{x})$ is governed by the parameters $\boldsymbol{\mu}$, $\mathbf{W}$, and σ^2. However, there is redundancy in this parameterization corresponding to rotations of the latent space coordinates. To see this, consider a matrix $\widetilde{\mathbf{W}} = \mathbf{W}\mathbf{R}$ where $\mathbf{R}$ is an orthogonal matrix. Using the orthogonality property $\mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{I}$, we see that the quantity $\widetilde{\mathbf{W}}\widetilde{\mathbf{W}}^{\mathrm{T}}$ that appears in the covariance matrix $\mathbf{C}$ takes the form

$$\widetilde{\mathbf{W}}\widetilde{\mathbf{W}}^{\mathrm{T}} = \mathbf{W}\mathbf{R}\mathbf{R}^{\mathrm{T}}\mathbf{W}^{\mathrm{T}} = \mathbf{W}\mathbf{W}^{\mathrm{T}} \tag{12.39}$$

and hence is independent of $\mathbf{R}$. Thus there is a whole family of matrices $\widetilde{\mathbf{W}}$ all of which give rise to the same predictive distribution. This invariance can be understood in terms of rotations within the latent space. We shall return to a discussion of the number of independent parameters in this model later.

When we evaluate the predictive distribution, we require $\mathbf{C}^{-1}$, which involves the inversion of a $D \times D$ matrix. The computation required to do this can be reduced by making use of the matrix inversion identity (C.7) to give

$$\mathbf{C}^{-1} = \sigma^{-1}\mathbf{I} - \sigma^{-2}\mathbf{W}\mathbf{M}^{-1}\mathbf{W}^{\mathrm{T}} \tag{12.40}$$

where the $M \times M$ matrix $\mathbf{M}$ is defined by

$$\mathbf{M} = \mathbf{W}^{\mathrm{T}}\mathbf{W} + \sigma^2\mathbf{I}. \tag{12.41}$$

Because we invert $\mathbf{M}$ rather than inverting $\mathbf{C}$ directly, the cost of evaluating $\mathbf{C}^{-1}$ is reduced from $O(D^3)$ to $O(M^3)$.

As well as the predictive distribution $p(\mathbf{x})$, we will also require the posterior distribution $p(\mathbf{z}|\mathbf{x})$, which can again be written down directly using the result (2.116) for linear-Gaussian models to give

Exercise 12.8

$$p(\mathbf{z}|\mathbf{x}) = \mathcal{N}\left(\mathbf{z}|\mathbf{M}^{-1}\mathbf{W}^{\mathrm{T}}(\mathbf{x} - \boldsymbol{\mu}), \sigma^{-2}\mathbf{M}\right). \tag{12.42}$$

Note that the posterior mean depends on $\mathbf{x}$, whereas the posterior covariance is independent of $\mathbf{x}$.

Figure 12.10 The probabilistic PCA model for a data set of N observations of $\mathbf{x}$ can be expressed as a directed graph in which each observation $\mathbf{x}_n$ is associated with a value $\mathbf{z}_n$ of the latent variable.

12.2.1 Maximum likelihood PCA

We next consider the determination of the model parameters using maximum likelihood. Given a data set $\mathbf{X} = \{\mathbf{x}_n\}$ of observed data points, the probabilistic PCA model can be expressed as a directed graph, as shown in Figure 12.10. The corresponding log likelihood function is given, from (12.35), by

$$\begin{aligned}\ln p(\mathbf{X}|\boldsymbol{\mu}, \mathbf{W}, \sigma^2) &= \sum_{n=1}^{N} \ln p(\mathbf{x}_n|\mathbf{W}, \boldsymbol{\mu}, \sigma^2) \\ &= -\frac{ND}{2}\ln(2\pi) - \frac{N}{2}\ln|\mathbf{C}| - \frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n - \boldsymbol{\mu})^{\mathrm{T}}\mathbf{C}^{-1}(\mathbf{x}_n - \boldsymbol{\mu}). \quad (12.43)\end{aligned}$$

Setting the derivative of the log likelihood with respect to $\boldsymbol{\mu}$ equal to zero gives the expected result $\boldsymbol{\mu} = \overline{\mathbf{x}}$ where $\overline{\mathbf{x}}$ is the data mean defined by (12.1). Back-substituting we can then write the log likelihood function in the form

$$\ln p(\mathbf{X}|\mathbf{W}, \boldsymbol{\mu}, \sigma^2) = -\frac{N}{2}\left\{D\ln(2\pi) + \ln|\mathbf{C}| + \mathrm{Tr}\left(\mathbf{C}^{-1}\mathbf{S}\right)\right\} \quad (12.44)$$

where $\mathbf{S}$ is the data covariance matrix defined by (12.3). Because the log likelihood is a quadratic function of $\boldsymbol{\mu}$, this solution represents the unique maximum, as can be confirmed by computing second derivatives.

Maximization with respect to $\mathbf{W}$ and σ^2 is more complex but nonetheless has an exact closed-form solution. It was shown by Tipping and Bishop (1999b) that all of the stationary points of the log likelihood function can be written as

$$\mathbf{W}_{\mathrm{ML}} = \mathbf{U}_M(\mathbf{L}_M - \sigma^2\mathbf{I})^{1/2}\mathbf{R} \quad (12.45)$$

where $\mathbf{U}_M$ is a $D \times M$ matrix whose columns are given by any subset (of size M) of the eigenvectors of the data covariance matrix $\mathbf{S}$, the $M \times M$ diagonal matrix $\mathbf{L}_M$ has elements given by the corresponding eigenvalues λ_i, and $\mathbf{R}$ is an arbitrary $M \times M$ orthogonal matrix.

Furthermore, Tipping and Bishop (1999b) showed that the *maximum* of the likelihood function is obtained when the M eigenvectors are chosen to be those whose eigenvalues are the M largest (all other solutions being saddle points). A similar result was conjectured independently by Roweis (1998), although no proof was given.

Again, we shall assume that the eigenvectors have been arranged in order of decreasing values of the corresponding eigenvalues, so that the M principal eigenvectors are $\mathbf{u}_1, \ldots, \mathbf{u}_M$. In this case, the columns of $\mathbf{W}$ define the principal subspace of standard PCA. The corresponding maximum likelihood solution for σ^2 is then given by

$$\sigma^2_{\mathrm{ML}} = \frac{1}{D-M} \sum_{i=M+1}^{D} \lambda_i \tag{12.46}$$

so that σ^2_{ML} is the average variance associated with the discarded dimensions.

Because $\mathbf{R}$ is orthogonal, it can be interpreted as a rotation matrix in the $M \times M$ latent space. If we substitute the solution for $\mathbf{W}$ into the expression for $\mathbf{C}$, and make use of the orthogonality property $\mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{I}$, we see that $\mathbf{C}$ is independent of $\mathbf{R}$. This simply says that the predictive density is unchanged by rotations in the latent space as discussed earlier. For the particular case of $\mathbf{R} = \mathbf{I}$, we see that the columns of $\mathbf{W}$ are the principal component eigenvectors scaled by the variance parameters $\lambda_i - \sigma^2$. The interpretation of these scaling factors is clear once we recognize that for a convolution of independent Gaussian distributions (in this case the latent space distribution and the noise model) the variances are additive. Thus the variance λ_i in the direction of an eigenvector $\mathbf{u}_i$ is composed of the sum of a contribution $\lambda_i - \sigma^2$ from the projection of the unit-variance latent space distribution into data space through the corresponding column of $\mathbf{W}$, plus an isotropic contribution of variance σ^2 which is added in all directions by the noise model.

It is worth taking a moment to study the form of the covariance matrix given by (12.36). Consider the variance of the predictive distribution along some direction specified by the unit vector $\mathbf{v}$, where $\mathbf{v}^{\mathrm{T}}\mathbf{v} = 1$, which is given by $\mathbf{v}^{\mathrm{T}}\mathbf{C}\mathbf{v}$. First suppose that $\mathbf{v}$ is orthogonal to the principal subspace, in other words it is given by some linear combination of the discarded eigenvectors. Then $\mathbf{v}^{\mathrm{T}}\mathbf{U} = \mathbf{0}$ and hence $\mathbf{v}^{\mathrm{T}}\mathbf{C}\mathbf{v} = \sigma^2$. Thus the model predicts a noise variance orthogonal to the principal subspace, which, from (12.46), is just the average of the discarded eigenvalues. Now suppose that $\mathbf{v} = \mathbf{u}_i$ where $\mathbf{u}_i$ is one of the retained eigenvectors defining the principal subspace. Then $\mathbf{v}^{\mathrm{T}}\mathbf{C}\mathbf{v} = (\lambda_i - \sigma^2) + \sigma^2 = \lambda_i$. In other words, this model correctly captures the variance of the data along the principal axes, and approximates the variance in all remaining directions with a single average value σ^2.

One way to construct the maximum likelihood density model would simply be to find the eigenvectors and eigenvalues of the data covariance matrix and then to evaluate $\mathbf{W}$ and σ^2 using the results given above. In this case, we would choose $\mathbf{R} = \mathbf{I}$ for convenience. However, if the maximum likelihood solution is found by numerical optimization of the likelihood function, for instance using an algorithm such as conjugate gradients (Fletcher, 1987; Nocedal and Wright, 1999; Bishop and Nabney, 2008) or through the EM algorithm, then the resulting value of $\mathbf{R}$ is essentially arbitrary. This implies that the columns of $\mathbf{W}$ need not be orthogonal. If an orthogonal basis is required, the matrix $\mathbf{W}$ can be post-processed appropriately (Golub and Van Loan, 1996). Alternatively, the EM algorithm can be modified in such a way as to yield orthonormal principal directions, sorted in descending order of the corresponding eigenvalues, directly (Ahn and Oh, 2003).

Section 12.2.2

The rotational invariance in latent space represents a form of statistical nonidentifiability, analogous to that encountered for mixture models in the case of discrete latent variables. Here there is a continuum of parameters all of which lead to the same predictive density, in contrast to the discrete nonidentifiability associated with component re-labelling in the mixture setting.

If we consider the case of $M = D$, so that there is no reduction of dimensionality, then $\mathbf{U}_M = \mathbf{U}$ and $\mathbf{L}_M = \mathbf{L}$. Making use of the orthogonality properties $\mathbf{U}\mathbf{U}^{\mathrm{T}} = \mathbf{I}$ and $\mathbf{R}\mathbf{R}^{\mathrm{T}} = \mathbf{I}$, we see that the covariance $\mathbf{C}$ of the marginal distribution for $\mathbf{x}$ becomes

$$\mathbf{C} = \mathbf{U}(\mathbf{L} - \sigma^2\mathbf{I})^{1/2}\mathbf{R}\mathbf{R}^{\mathrm{T}}(\mathbf{L} - \sigma^2\mathbf{I})^{1/2}\mathbf{U}^{\mathrm{T}} + \sigma^2\mathbf{I} = \mathbf{U}\mathbf{L}\mathbf{U}^{\mathrm{T}} = \mathbf{S} \tag{12.47}$$

and so we obtain the standard maximum likelihood solution for an unconstrained Gaussian distribution in which the covariance matrix is given by the sample covariance.

Conventional PCA is generally formulated as a projection of points from the D-dimensional data space onto an M-dimensional linear subspace. Probabilistic PCA, however, is most naturally expressed as a mapping from the latent space into the data space via (12.33). For applications such as visualization and data compression, we can reverse this mapping using Bayes' theorem. Any point $\mathbf{x}$ in data space can then be summarized by its posterior mean and covariance in latent space. From (12.42) the mean is given by

$$\mathbb{E}[\mathbf{z}|\mathbf{x}] = \mathbf{M}^{-1}\mathbf{W}_{\mathrm{ML}}^{\mathrm{T}}(\mathbf{x} - \overline{\mathbf{x}}) \tag{12.48}$$

where $\mathbf{M}$ is given by (12.41). This projects to a point in data space given by

$$\mathbf{W}\mathbb{E}[\mathbf{z}|\mathbf{x}] + \boldsymbol{\mu}. \tag{12.49}$$

Section 3.3.1

Note that this takes the same form as the equations for regularized linear regression and is a consequence of maximizing the likelihood function for a linear Gaussian model. Similarly, the posterior covariance is given from (12.42) by $\sigma^2\mathbf{M}^{-1}$ and is independent of $\mathbf{x}$.

If we take the limit $\sigma^2 \to 0$, then the posterior mean reduces to

$$(\mathbf{W}_{\mathrm{ML}}^{\mathrm{T}}\mathbf{W}_{\mathrm{ML}})^{-1}\mathbf{W}_{\mathrm{ML}}^{\mathrm{T}}(\mathbf{x} - \overline{\mathbf{x}}) \tag{12.50}$$

which represents an orthogonal projection of the data point onto the latent space, and so we recover the standard PCA model. The posterior covariance in this limit is zero, however, and the density becomes singular. For $\sigma^2 > 0$, the latent projection is shifted towards the origin, relative to the orthogonal projection.

Exercise 12.11

Exercise 12.12

Finally, we note that an important role for the probabilistic PCA model is in defining a multivariate Gaussian distribution in which the number of degrees of freedom, in other words the number of independent parameters, can be controlled whilst still allowing the model to capture the dominant correlations in the data. Recall that a general Gaussian distribution has $D(D+1)/2$ independent parameters in its covariance matrix (plus another D parameters in its mean). Thus the number of parameters scales quadratically with D and can become excessive in spaces of high

Section 2.3

dimensionality. If we restrict the covariance matrix to be diagonal, then it has only D independent parameters, and so the number of parameters now grows linearly with dimensionality. However, it now treats the variables as if they were independent and hence can no longer express any correlations between them. Probabilistic PCA provides an elegant compromise in which the M most significant correlations can be captured while still ensuring that the total number of parameters grows only linearly with D. We can see this by evaluating the number of degrees of freedom in the PPCA model as follows. The covariance matrix $\mathbf{C}$ depends on the parameters $\mathbf{W}$, which has size $D \times M$, and σ^2, giving a total parameter count of $DM + 1$. However, we have seen that there is some redundancy in this parameterization associated with rotations of the coordinate system in the latent space. The orthogonal matrix $\mathbf{R}$ that expresses these rotations has size $M \times M$. In the first column of this matrix there are $M - 1$ independent parameters, because the column vector must be normalized to unit length. In the second column there are $M - 2$ independent parameters, because the column must be normalized and also must be orthogonal to the previous column, and so on. Summing this arithmetic series, we see that $\mathbf{R}$ has a total of $M(M-1)/2$ independent parameters. Thus the number of degrees of freedom in the covariance matrix $\mathbf{C}$ is given by

$$DM + 1 - M(M-1)/2. \tag{12.51}$$

The number of independent parameters in this model therefore only grows linearly with D, for fixed M. If we take $M = D - 1$, then we recover the standard result for a full covariance Gaussian. In this case, the variance along $D - 1$ linearly independent directions is controlled by the columns of $\mathbf{W}$, and the variance along the remaining direction is given by σ^2. If $M = 0$, the model is equivalent to the isotropic covariance case.

Exercise 12.14

12.2.2 EM algorithm for PCA

As we have seen, the probabilistic PCA model can be expressed in terms of a marginalization over a continuous latent space $\mathbf{z}$ in which for each data point $\mathbf{x}_n$, there is a corresponding latent variable $\mathbf{z}_n$. We can therefore make use of the EM algorithm to find maximum likelihood estimates of the model parameters. This may seem rather pointless because we have already obtained an exact closed-form solution for the maximum likelihood parameter values. However, in spaces of high dimensionality, there may be computational advantages in using an iterative EM procedure rather than working directly with the sample covariance matrix. This EM procedure can also be extended to the factor analysis model, for which there is no closed-form solution. Finally, it allows missing data to be handled in a principled way.

Section 12.2.4

We can derive the EM algorithm for probabilistic PCA by following the general framework for EM. Thus we write down the complete-data log likelihood and take its expectation with respect to the posterior distribution of the latent distribution evaluated using 'old' parameter values. Maximization of this expected complete-data log likelihood then yields the 'new' parameter values. Because the data points

Section 9.4

are assumed independent, the complete-data log likelihood function takes the form

$$\ln p\left(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \mathbf{W}, \sigma^2\right) = \sum_{n=1}^{N} \{\ln p(\mathbf{x}_n|\mathbf{z}_n) + \ln p(\mathbf{z}_n)\} \tag{12.52}$$

where the nth row of the matrix $\mathbf{Z}$ is given by $\mathbf{z}_n$. We already know that the exact maximum likelihood solution for $\boldsymbol{\mu}$ is given by the sample mean $\overline{\mathbf{x}}$ defined by (12.1), and it is convenient to substitute for $\boldsymbol{\mu}$ at this stage. Making use of the expressions (12.31) and (12.32) for the latent and conditional distributions, respectively, and taking the expectation with respect to the posterior distribution over the latent variables, we obtain

$$\begin{aligned}\mathbb{E}[\ln p\left(\mathbf{X}, \mathbf{Z}|\boldsymbol{\mu}, \mathbf{W}, \sigma^2\right)] = -\sum_{n=1}^{N} \Bigg\{ & \frac{D}{2}\ln(2\pi\sigma^2) + \frac{1}{2}\mathrm{Tr}\left(\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}]\right) \\ & + \frac{1}{2\sigma^2}\|\mathbf{x}_n - \boldsymbol{\mu}\|^2 - \frac{1}{\sigma^2}\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}}\mathbf{W}^{\mathrm{T}}(\mathbf{x}_n - \boldsymbol{\mu}) \\ & + \frac{1}{2\sigma^2}\mathrm{Tr}\left(\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}]\mathbf{W}^{\mathrm{T}}\mathbf{W}\right)\Bigg\}. \end{aligned} \tag{12.53}$$

Note that this depends on the posterior distribution only through the sufficient statistics of the Gaussian. Thus in the E step, we use the old parameter values to evaluate

$$\mathbb{E}[\mathbf{z}_n] = \mathbf{M}^{-1}\mathbf{W}^{\mathrm{T}}(\mathbf{x}_n - \overline{\mathbf{x}}) \tag{12.54}$$

$$\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}] = \sigma^2\mathbf{M}^{-1} + \mathbb{E}[\mathbf{z}_n]\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}} \tag{12.55}$$

which follow directly from the posterior distribution (12.42) together with the standard result $\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}] = \mathrm{cov}[\mathbf{z}_n] + \mathbb{E}[\mathbf{z}_n]\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}}$. Here $\mathbf{M}$ is defined by (12.41).

In the M step, we maximize with respect to $\mathbf{W}$ and σ^2, keeping the posterior statistics fixed. Maximization with respect to σ^2 is straightforward. For the maximization with respect to $\mathbf{W}$ we make use of (C.24), and obtain the M-step equations

Exercise 12.15

$$\mathbf{W}_{\text{new}} = \left[\sum_{n=1}^{N}(\mathbf{x}_n - \overline{\mathbf{x}})\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}}\right]\left[\sum_{n=1}^{N}\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}]\right]^{-1} \tag{12.56}$$

$$\begin{aligned}\sigma^2_{\text{new}} = \frac{1}{ND}\sum_{n=1}^{N}\Big\{ & \|\mathbf{x}_n - \overline{\mathbf{x}}\|^2 - 2\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}}\mathbf{W}_{\text{new}}^{\mathrm{T}}(\mathbf{x}_n - \overline{\mathbf{x}}) \\ & + \mathrm{Tr}\left(\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}]\mathbf{W}_{\text{new}}^{\mathrm{T}}\mathbf{W}_{\text{new}}\right)\Big\}. \end{aligned} \tag{12.57}$$

The EM algorithm for probabilistic PCA proceeds by initializing the parameters and then alternately computing the sufficient statistics of the latent space posterior distribution using (12.54) and (12.55) in the E step and revising the parameter values using (12.56) and (12.57) in the M step.

One of the benefits of the EM algorithm for PCA is computational efficiency for large-scale applications (Roweis, 1998). Unlike conventional PCA based on an

eigenvector decomposition of the sample covariance matrix, the EM approach is iterative and so might appear to be less attractive. However, each cycle of the EM algorithm can be computationally much more efficient than conventional PCA in spaces of high dimensionality. To see this, we note that the eigendecomposition of the covariance matrix requires $O(D^3)$ computation. Often we are interested only in the first M eigenvectors and their corresponding eigenvalues, in which case we can use algorithms that are $O(MD^2)$. However, the evaluation of the covariance matrix itself takes $O(ND^2)$ computations, where N is the number of data points. Algorithms such as the snapshot method (Sirovich, 1987), which assume that the eigenvectors are linear combinations of the data vectors, avoid direct evaluation of the covariance matrix but are $O(N^3)$ and hence unsuited to large data sets. The EM algorithm described here also does not construct the covariance matrix explicitly. Instead, the most computationally demanding steps are those involving sums over the data set that are $O(NDM)$. For large D, and $M \ll D$, this can be a significant saving compared to $O(ND^2)$ and can offset the iterative nature of the EM algorithm.

Note that this EM algorithm can be implemented in an on-line form in which each D-dimensional data point is read in and processed and then discarded before the next data point is considered. To see this, note that the quantities evaluated in the E step (an M-dimensional vector and an $M \times M$ matrix) can be computed for each data point separately, and in the M step we need to accumulate sums over data points, which we can do incrementally. This approach can be advantageous if both N and D are large.

Because we now have a fully probabilistic model for PCA, we can deal with missing data, provided that it is missing at random, by marginalizing over the distribution of the unobserved variables. Again these missing values can be treated using the EM algorithm. We give an example of the use of this approach for data visualization in Figure 12.11.

Another elegant feature of the EM approach is that we can take the limit $\sigma^2 \to 0$, corresponding to standard PCA, and still obtain a valid EM-like algorithm (Roweis, 1998). From (12.55), we see that the only quantity we need to compute in the E step is $\mathbb{E}[\mathbf{z}_n]$. Furthermore, the M step is simplified because $\mathbf{M} = \mathbf{W}^{\mathrm{T}}\mathbf{W}$. To emphasize the simplicity of the algorithm, let us define $\widetilde{\mathbf{X}}$ to be a matrix of size $N \times D$ whose n^{th} row is given by the vector $\mathbf{x}_n - \overline{\mathbf{x}}$ and similarly define $\mathbf{\Omega}$ to be a matrix of size $D \times M$ whose n^{th} row is given by the vector $\mathbb{E}[\mathbf{z}_n]$. The E step (12.54) of the EM algorithm for PCA then becomes

$$\mathbf{\Omega} = (\mathbf{W}_{\mathrm{old}}^{\mathrm{T}}\mathbf{W}_{\mathrm{old}})^{-1}\mathbf{W}_{\mathrm{old}}^{\mathrm{T}}\widetilde{\mathbf{X}} \tag{12.58}$$

and the M step (12.56) takes the form

$$\mathbf{W}_{\mathrm{new}} = \widetilde{\mathbf{X}}^{\mathrm{T}}\mathbf{\Omega}^{\mathrm{T}}(\mathbf{\Omega}\mathbf{\Omega}^{\mathrm{T}})^{-1}. \tag{12.59}$$

Again these can be implemented in an on-line form. These equations have a simple interpretation as follows. From our earlier discussion, we see that the E step involves an orthogonal projection of the data points onto the current estimate for the principal subspace. Correspondingly, the M step represents a re-estimation of the principal

Figure 12.11 Probabilistic PCA visualization of a portion of the oil flow data set for the first 100 data points. The left-hand plot shows the posterior mean projections of the data points on the principal subspace. The right-hand plot is obtained by first randomly omitting 30% of the variable values and then using EM to handle the missing values. Note that each data point then has at least one missing measurement but that the plot is very similar to the one obtained without missing values.

Exercise 12.17

subspace to minimize the squared reconstruction error in which the projections are fixed.

We can give a simple physical analogy for this EM algorithm, which is easily visualized for $D = 2$ and $M = 1$. Consider a collection of data points in two dimensions, and let the one-dimensional principal subspace be represented by a solid rod. Now attach each data point to the rod via a spring obeying Hooke's law (stored energy is proportional to the square of the spring's length). In the E step, we keep the rod fixed and allow the attachment points to slide up and down the rod so as to minimize the energy. This causes each attachment point (independently) to position itself at the orthogonal projection of the corresponding data point onto the rod. In the M step, we keep the attachment points fixed and then release the rod and allow it to move to the minimum energy position. The E and M steps are then repeated until a suitable convergence criterion is satisfied, as is illustrated in Figure 12.12.

12.2.3 Bayesian PCA

So far in our discussion of PCA, we have assumed that the value M for the dimensionality of the principal subspace is given. In practice, we must choose a suitable value according to the application. For visualization, we generally choose $M = 2$, whereas for other applications the appropriate choice for M may be less clear. One approach is to plot the eigenvalue spectrum for the data set, analogous to the example in Figure 12.4 for the off-line digits data set, and look to see if the eigenvalues naturally form two groups comprising a set of small values separated by a significant gap from a set of relatively large values, indicating a natural choice for M. In practice, such a gap is often not seen.

Figure 12.12 Synthetic data illustrating the EM algorithm for PCA defined by (12.58) and (12.59). (a) A data set $\mathbf{X}$ with the data points shown in green, together with the true principal components (shown as eigenvectors scaled by the square roots of the eigenvalues). (b) Initial configuration of the principal subspace defined by $\mathbf{W}$, shown in red, together with the projections of the latent points $\mathbf{Z}$ into the data space, given by $\mathbf{Z}\mathbf{W}^{\mathrm{T}}$, shown in cyan. (c) After one M step, the latent space has been updated with $\mathbf{Z}$ held fixed. (d) After the successive E step, the values of $\mathbf{Z}$ have been updated, giving orthogonal projections, with $\mathbf{W}$ held fixed. (e) After the second M step. (f) After the second E step.

Section 1.3

Because the probabilistic PCA model has a well-defined likelihood function, we could employ cross-validation to determine the value of dimensionality by selecting the largest log likelihood on a validation data set. Such an approach, however, can become computationally costly, particularly if we consider a probabilistic mixture of PCA models (Tipping and Bishop, 1999a) in which we seek to determine the appropriate dimensionality separately for each component in the mixture.

Given that we have a probabilistic formulation of PCA, it seems natural to seek a Bayesian approach to model selection. To do this, we need to marginalize out the model parameters $\boldsymbol{\mu}$, $\mathbf{W}$, and σ^2 with respect to appropriate prior distributions. This can be done by using a variational framework to approximate the analytically intractable marginalizations (Bishop, 1999b). The marginal likelihood values, given by the variational lower bound, can then be compared for a range of different values of M and the value giving the largest marginal likelihood selected.

Here we consider a simpler approach introduced by based on the *evidence ap-*

Figure 12.13 Probabilistic graphical model for Bayesian PCA in which the distribution over the parameter matrix $\mathbf{W}$ is governed by a vector $\boldsymbol{\alpha}$ of hyperparameters.

proximation, which is appropriate when the number of data points is relatively large and the corresponding posterior distribution is tightly peaked (Bishop, 1999a). It involves a specific choice of prior over $\mathbf{W}$ that allows surplus dimensions in the principal subspace to be pruned out of the model. This corresponds to an example of *automatic relevance determination*, or *ARD*, discussed in Section 7.2.2. Specifically, we define an independent Gaussian prior over each column of $\mathbf{W}$, which represent the vectors defining the principal subspace. Each such Gaussian has an independent variance governed by a precision hyperparameter α_i so that

$$p(\mathbf{W}|\boldsymbol{\alpha}) = \prod_{i=1}^{M} \left(\frac{\alpha_i}{2\pi}\right)^{D/2} \exp\left\{-\frac{1}{2}\alpha_i \mathbf{w}_i^{\mathrm{T}}\mathbf{w}_i\right\} \tag{12.60}$$

where $\mathbf{w}_i$ is the ith column of $\mathbf{W}$. The resulting model can be represented using the directed graph shown in Figure 12.13.

The values for α_i will be found iteratively by maximizing the marginal likelihood function in which $\mathbf{W}$ has been integrated out. As a result of this optimization, some of the α_i may be driven to infinity, with the corresponding parameters vector $\mathbf{w}_i$ being driven to zero (the posterior distribution becomes a delta function at the origin) giving a sparse solution. The effective dimensionality of the principal subspace is then determined by the number of finite α_i values, and the corresponding vectors $\mathbf{w}_i$ can be thought of as 'relevant' for modelling the data distribution. In this way, the Bayesian approach is automatically making the trade-off between improving the fit to the data, by using a larger number of vectors $\mathbf{w}_i$ with their corresponding eigenvalues λ_i each tuned to the data, and reducing the complexity of the model by suppressing some of the $\mathbf{w}_i$ vectors. The origins of this sparsity were discussed earlier in the context of relevance vector machines.

Section 7.2

The values of α_i are re-estimated during training by maximizing the log marginal likelihood given by

$$p(\mathbf{X}|\boldsymbol{\alpha}, \boldsymbol{\mu}, \sigma^2) = \int p(\mathbf{X}|\mathbf{W}, \boldsymbol{\mu}, \sigma^2)p(\mathbf{W}|\boldsymbol{\alpha})\,\mathrm{d}\mathbf{W} \tag{12.61}$$

where the log of $p(\mathbf{X}|\mathbf{W}, \boldsymbol{\mu}, \sigma^2)$ is given by (12.43). Note that for simplicity we also treat $\boldsymbol{\mu}$ and σ^2 as parameters to be estimated, rather than defining priors over these parameters.

Section 4.4

Section 3.5.3

Because this integration is intractable, we make use of the Laplace approximation. If we assume that the posterior distribution is sharply peaked, as will occur for sufficiently large data sets, then the re-estimation equations obtained by maximizing the marginal likelihood with respect to α_i take the simple form

$$\alpha_i^{\text{new}} = \frac{D}{\mathbf{w}_i^{\mathrm{T}}\mathbf{w}_i} \tag{12.62}$$

which follows from (3.98), noting that the dimensionality of $\mathbf{w}_i$ is D. These re-estimations are interleaved with the EM algorithm updates for determining $\mathbf{W}$ and σ^2. The E-step equations are again given by (12.54) and (12.55). Similarly, the M-step equation for σ^2 is again given by (12.57). The only change is to the M-step equation for $\mathbf{W}$, which is modified to give

$$\mathbf{W}_{\text{new}} = \left[\sum_{n=1}^{N}(\mathbf{x}_n - \overline{\mathbf{x}})\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}}\right]\left[\sum_{n=1}^{N}\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}] + \sigma^2\mathbf{A}\right]^{-1} \tag{12.63}$$

where $\mathbf{A} = \mathrm{diag}(\alpha_i)$. The value of $\boldsymbol{\mu}$ is given by the sample mean, as before.

If we choose $M = D - 1$ then, if all α_i values are finite, the model represents a full-covariance Gaussian, while if all the α_i go to infinity the model is equivalent to an isotropic Gaussian, and so the model can encompass all permissible values for the effective dimensionality of the principal subspace. It is also possible to consider smaller values of M, which will save on computational cost but which will limit the maximum dimensionality of the subspace. A comparison of the results of this algorithm with standard probabilistic PCA is shown in Figure 12.14.

Bayesian PCA provides an opportunity to illustrate the Gibbs sampling algorithm discussed in Section 11.3. Figure 12.15 shows an example of the samples from the hyperparameters $\ln \alpha_i$ for a data set in $D = 4$ dimensions in which the dimensionality of the latent space is $M = 3$ but in which the data set is generated from a probabilistic PCA model having one direction of high variance, with the remaining directions comprising low variance noise. This result shows clearly the presence of three distinct modes in the posterior distribution. At each step of the iteration, one of the hyperparameters has a small value and the remaining two have large values, so that two of the three latent variables are suppressed. During the course of the Gibbs sampling, the solution makes sharp transitions between the three modes.

The model described here involves a prior only over the matrix $\mathbf{W}$. A fully Bayesian treatment of PCA, including priors over $\boldsymbol{\mu}$, σ^2, and $\boldsymbol{\alpha}$, and solved using variational methods, is described in Bishop (1999b). For a discussion of various Bayesian approaches to determining the appropriate dimensionality for a PCA model, see Minka (2001c).

12.2.4 Factor analysis

Factor analysis is a linear-Gaussian latent variable model that is closely related to probabilistic PCA. Its definition differs from that of probabilistic PCA only in that the conditional distribution of the observed variable $\mathbf{x}$ given the latent variable $\mathbf{z}$ is

Figure 12.14 'Hinton' diagrams of the matrix $\mathbf{W}$ in which each element of the matrix is depicted as a square (white for positive and black for negative values) whose area is proportional to the magnitude of that element. The synthetic data set comprises 300 data points in $D = 10$ dimensions sampled from a Gaussian distribution having standard deviation 1.0 in 3 directions and standard deviation 0.5 in the remaining 7 directions for a data set in $D = 10$ dimensions having $M = 3$ directions with larger variance than the remaining 7 directions. The left-hand plot shows the result from maximum likelihood probabilistic PCA, and the left-hand plot shows the corresponding result from Bayesian PCA. We see how the Bayesian model is able to discover the appropriate dimensionality by suppressing the 6 surplus degrees of freedom.

taken to have a diagonal rather than an isotropic covariance so that

$$p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|\mathbf{W}\mathbf{z} + \boldsymbol{\mu}, \boldsymbol{\Psi}) \tag{12.64}$$

where $\boldsymbol{\Psi}$ is a $D \times D$ diagonal matrix. Note that the factor analysis model, in common with probabilistic PCA, assumes that the observed variables $x_1, \ldots, x_D$ are independent, given the latent variable $\mathbf{z}$. In essence, the factor analysis model is explaining the observed covariance structure of the data by representing the independent variance associated with each coordinate in the matrix $\boldsymbol{\Psi}$ and capturing the covariance between variables in the matrix $\mathbf{W}$. In the factor analysis literature, the columns of $\mathbf{W}$, which capture the correlations between observed variables, are called *factor loadings*, and the diagonal elements of $\boldsymbol{\Psi}$, which represent the independent noise variances for each of the variables, are called *uniquenesses*.

The origins of factor analysis are as old as those of PCA, and discussions of factor analysis can be found in the books by Everitt (1984), Bartholomew (1987), and Basilevsky (1994). Links between factor analysis and PCA were investigated by Lawley (1953) and Anderson (1963) who showed that at stationary points of the likelihood function, for a factor analysis model with $\boldsymbol{\Psi} = \sigma^2\mathbf{I}$, the columns of $\mathbf{W}$ are scaled eigenvectors of the sample covariance matrix, and σ^2 is the average of the discarded eigenvalues. Later, Tipping and Bishop (1999b) showed that the maximum of the log likelihood function occurs when the eigenvectors comprising $\mathbf{W}$ are chosen to be the principal eigenvectors.

Making use of (2.115), we see that the marginal distribution for the observed

Figure 12.15 Gibbs sampling for Bayesian PCA showing plots of $\ln \alpha_i$ versus iteration number for three α values, showing transitions between the three modes of the posterior distribution.

variable is given by $p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \mathbf{C})$ where now

$$\mathbf{C} = \mathbf{W}\mathbf{W}^{\mathrm{T}} + \boldsymbol{\Psi}. \tag{12.65}$$

Exercise 12.19 As with probabilistic PCA, this model is invariant to rotations in the latent space.

Historically, factor analysis has been the subject of controversy when attempts have been made to place an interpretation on the individual factors (the coordinates in $\mathbf{z}$-space), which has proven problematic due to the nonidentifiability of factor analysis associated with rotations in this space. From our perspective, however, we shall view factor analysis as a form of latent variable density model, in which the form of the latent space is of interest but not the particular choice of coordinates used to describe it. If we wish to remove the degeneracy associated with latent space rotations, we must consider non-Gaussian latent variable distributions, giving rise to independent component analysis (ICA) models. *Section 12.4*

We can determine the parameters $\boldsymbol{\mu}$, $\mathbf{W}$, and $\boldsymbol{\Psi}$ in the factor analysis model by maximum likelihood. The solution for $\boldsymbol{\mu}$ is again given by the sample mean. However, unlike probabilistic PCA, there is no longer a closed-form maximum likelihood solution for $\mathbf{W}$, which must therefore be found iteratively. Because factor analysis is a latent variable model, this can be done using an EM algorithm (Rubin and Thayer, 1982) that is analogous to the one used for probabilistic PCA. Specifically, the E-step equations are given by *Exercise 12.21*

$$\mathbb{E}[\mathbf{z}_n] = \mathbf{G}\mathbf{W}^{\mathrm{T}}\boldsymbol{\Psi}^{-1}(\mathbf{x}_n - \overline{\mathbf{x}}) \tag{12.66}$$

$$\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}] = \mathbf{G} + \mathbb{E}[\mathbf{z}_n]\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}} \tag{12.67}$$

where we have defined

$$\mathbf{G} = (\mathbf{I} + \mathbf{W}^{\mathrm{T}}\boldsymbol{\Psi}^{-1}\mathbf{W})^{-1}. \tag{12.68}$$

Note that this is expressed in a form that involves inversion of matrices of size $M \times M$ rather than $D \times D$ (except for the $D \times D$ diagonal matrix $\boldsymbol{\Psi}$ whose inverse is trivial

to compute in $O(D)$ steps), which is convenient because often $M \ll D$. Similarly, the M-step equations take the form

Exercise 12.22

$$\mathbf{W}^{\text{new}} = \left[\sum_{n=1}^{N}(\mathbf{x}_n - \overline{\mathbf{x}})\mathbb{E}[\mathbf{z}_n]^{\mathrm{T}}\right]\left[\sum_{n=1}^{N}\mathbb{E}[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}]\right]^{-1} \tag{12.69}$$

$$\mathbf{\Psi}^{\text{new}} = \operatorname{diag}\left\{\mathbf{S} - \mathbf{W}_{\text{new}}\frac{1}{N}\sum_{n=1}^{N}\mathbb{E}[\mathbf{z}_n](\mathbf{x}_n - \overline{\mathbf{x}})^{\mathrm{T}}\right\} \tag{12.70}$$

where the 'diag' operator sets all of the nondiagonal elements of a matrix to zero. A Bayesian treatment of the factor analysis model can be obtained by a straightforward application of the techniques discussed in this book.

Another difference between probabilistic PCA and factor analysis concerns their different behaviour under transformations of the data set. For PCA and probabilistic PCA, if we rotate the coordinate system in data space, then we obtain exactly the same fit to the data but with the $\mathbf{W}$ matrix transformed by the corresponding rotation matrix. However, for factor analysis, the analogous property is that if we make a component-wise re-scaling of the data vectors, then this is absorbed into a corresponding re-scaling of the elements of $\mathbf{\Psi}$.

Exercise 12.25

12.3. Kernel PCA

In Chapter 6, we saw how the technique of kernel substitution allows us to take an algorithm expressed in terms of scalar products of the form $\mathbf{x}^{\mathrm{T}}\mathbf{x}'$ and generalize that algorithm by replacing the scalar products with a nonlinear kernel. Here we apply this technique of kernel substitution to principal component analysis, thereby obtaining a nonlinear generalization called *kernel PCA* (Schölkopf *et al.*, 1998).

Consider a data set $\{\mathbf{x}_n\}$ of observations, where $n = 1, \ldots, N$, in a space of dimensionality D. In order to keep the notation uncluttered, we shall assume that we have already subtracted the sample mean from each of the vectors $\mathbf{x}_n$, so that $\sum_n \mathbf{x}_n = \mathbf{0}$. The first step is to express conventional PCA in such a form that the data vectors $\{\mathbf{x}_n\}$ appear only in the form of the scalar products $\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_m$. Recall that the principal components are defined by the eigenvectors $\mathbf{u}_i$ of the covariance matrix

$$\mathbf{S}\mathbf{u}_i = \lambda_i\mathbf{u}_i \tag{12.71}$$

where $i = 1, \ldots, D$. Here the $D \times D$ sample covariance matrix $\mathbf{S}$ is defined by

$$\mathbf{S} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^{\mathrm{T}}, \tag{12.72}$$

and the eigenvectors are normalized such that $\mathbf{u}_i^{\mathrm{T}}\mathbf{u}_i = 1$.

Now consider a nonlinear transformation $\phi(\mathbf{x})$ into an M-dimensional feature space, so that each data point $\mathbf{x}_n$ is thereby projected onto a point $\phi(\mathbf{x}_n)$. We can

Figure 12.16 Schematic illustration of kernel PCA. A data set in the original data space (left-hand plot) is projected by a nonlinear transformation $\phi(\mathbf{x})$ into a feature space (right-hand plot). By performing PCA in the feature space, we obtain the principal components, of which the first is shown in blue and is denoted by the vector $\mathbf{v}_1$. The green lines in feature space indicate the linear projections onto the first principal component, which correspond to nonlinear projections in the original data space. Note that in general it is not possible to represent the nonlinear principal component by a vector in $\mathbf{x}$ space.

now perform standard PCA in the feature space, which implicitly defines a nonlinear principal component model in the original data space, as illustrated in Figure 12.16.

For the moment, let us assume that the projected data set also has zero mean, so that $\sum_n \phi(\mathbf{x}_n) = \mathbf{0}$. We shall return to this point shortly. The $M \times M$ sample covariance matrix in feature space is given by

$$\mathbf{C} = \frac{1}{N} \sum_{n=1}^{N} \phi(\mathbf{x}_n)\phi(\mathbf{x}_n)^{\mathrm{T}} \tag{12.73}$$

and its eigenvector expansion is defined by

$$\mathbf{C}\mathbf{v}_i = \lambda_i \mathbf{v}_i \tag{12.74}$$

$i = 1, \ldots, M$. Our goal is to solve this eigenvalue problem without having to work explicitly in the feature space. From the definition of $\mathbf{C}$, the eigenvector equations tells us that $\mathbf{v}_i$ satisfies

$$\frac{1}{N} \sum_{n=1}^{N} \phi(\mathbf{x}_n) \left\{ \phi(\mathbf{x}_n)^{\mathrm{T}} \mathbf{v}_i \right\} = \lambda_i \mathbf{v}_i \tag{12.75}$$

and so we see that (provided $\lambda_i > 0$) the vector $\mathbf{v}_i$ is given by a linear combination of the $\phi(\mathbf{x}_n)$ and so can be written in the form

$$\mathbf{v}_i = \sum_{n=1}^{N} a_{in} \phi(\mathbf{x}_n). \tag{12.76}$$

Substituting this expansion back into the eigenvector equation, we obtain

$$\frac{1}{N}\sum_{n=1}^{N}\boldsymbol{\phi}(\mathbf{x}_n)\boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}}\sum_{m=1}^{N}a_{im}\boldsymbol{\phi}(\mathbf{x}_m)=\lambda_i\sum_{n=1}^{N}a_{in}\boldsymbol{\phi}(\mathbf{x}_n). \tag{12.77}$$

The key step is now to express this in terms of the kernel function $k(\mathbf{x}_n,\mathbf{x}_m)=\boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_m)$, which we do by multiplying both sides by $\boldsymbol{\phi}(\mathbf{x}_l)^{\mathrm{T}}$ to give

$$\frac{1}{N}\sum_{n=1}^{N}k(\mathbf{x}_l,\mathbf{x}_n)\sum_{m=1}^{m}a_{im}k(\mathbf{x}_n,\mathbf{x}_m)=\lambda_i\sum_{n=1}^{N}a_{in}k(\mathbf{x}_l,\mathbf{x}_n). \tag{12.78}$$

This can be written in matrix notation as

$$\mathbf{K}^2\mathbf{a}_i=\lambda_i N\mathbf{K}\mathbf{a}_i \tag{12.79}$$

where $\mathbf{a}_i$ is an N-dimensional column vector with elements a_{ni} for $n=1,\ldots,N$. We can find solutions for $\mathbf{a}_i$ by solving the following eigenvalue problem

$$\mathbf{K}\mathbf{a}_i=\lambda_i N\mathbf{a}_i \tag{12.80}$$

in which we have removed a factor of $\mathbf{K}$ from both sides of (12.79). Note that the solutions of (12.79) and (12.80) differ only by eigenvectors of $\mathbf{K}$ having zero eigenvalues that do not affect the principal components projection.

Exercise 12.26

The normalization condition for the coefficients $\mathbf{a}_i$ is obtained by requiring that the eigenvectors in feature space be normalized. Using (12.76) and (12.80), we have

$$1=\mathbf{v}_i^{\mathrm{T}}\mathbf{v}_i=\sum_{n=1}^{N}\sum_{m=1}^{N}a_{in}a_{im}\boldsymbol{\phi}(\mathbf{x}_n)^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_m)=\mathbf{a}_i^{\mathrm{T}}\mathbf{K}\mathbf{a}_i=\lambda_i N\mathbf{a}_i^{\mathrm{T}}\mathbf{a}_i. \tag{12.81}$$

Having solved the eigenvector problem, the resulting principal component projections can then also be cast in terms of the kernel function so that, using (12.76), the projection of a point $\mathbf{x}$ onto eigenvector i is given by

$$y_i(\mathbf{x})=\boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}}\mathbf{v}_i=\sum_{n=1}^{N}a_{in}\boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}_n)=\sum_{n=1}^{N}a_{in}k(\mathbf{x},\mathbf{x}_n) \tag{12.82}$$

and so again is expressed in terms of the kernel function.

In the original D-dimensional $\mathbf{x}$ space there are D orthogonal eigenvectors and hence we can find at most D linear principal components. The dimensionality M of the feature space, however, can be much larger than D (even infinite), and thus we can find a number of nonlinear principal components that can exceed D. Note, however, that the number of nonzero eigenvalues cannot exceed the number N of data points, because (even if $M>N$) the covariance matrix in feature space has rank at most equal to N. This is reflected in the fact that kernel PCA involves the eigenvector expansion of the $N\times N$ matrix $\mathbf{K}$.

So far we have assumed that the projected data set given by $\phi(\mathbf{x}_n)$ has zero mean, which in general will not be the case. We cannot simply compute and then subtract off the mean, since we wish to avoid working directly in feature space, and so again, we formulate the algorithm purely in terms of the kernel function. The projected data points after centralizing, denoted $\widetilde{\phi}(\mathbf{x}_n)$, are given by

$$\widetilde{\phi}(\mathbf{x}_n) = \phi(\mathbf{x}_n) - \frac{1}{N}\sum_{l=1}^{N}\phi(\mathbf{x}_l) \tag{12.83}$$

and the corresponding elements of the Gram matrix are given by

$$\begin{aligned}
\widetilde{K}_{nm} &= \widetilde{\phi}(\mathbf{x}_n)^{\mathrm{T}}\widetilde{\phi}(\mathbf{x}_m) \\
&= \phi(\mathbf{x}_n)^{\mathrm{T}}\phi(\mathbf{x}_m) - \frac{1}{N}\sum_{l=1}^{N}\phi(\mathbf{x}_n)^{\mathrm{T}}\phi(\mathbf{x}_l) \\
&\quad -\frac{1}{N}\sum_{l=1}^{N}\phi(\mathbf{x}_l)^{\mathrm{T}}\phi(\mathbf{x}_m) + \frac{1}{N^2}\sum_{j=1}^{N}\sum_{l=1}^{N}\phi(\mathbf{x}_j)^{\mathrm{T}}\phi(\mathbf{x}_l) \\
&= k(\mathbf{x}_n, \mathbf{x}_m) - \frac{1}{N}\sum_{l=1}^{N}k(\mathbf{x}_l, \mathbf{x}_m) \\
&\quad -\frac{1}{N}\sum_{l=1}^{N}k(\mathbf{x}_n, \mathbf{x}_l) + \frac{1}{N^2}\sum_{j=1}^{N}\sum_{l=1}^{N}k(\mathbf{x}_j, \mathbf{x}_l).
\end{aligned} \tag{12.84}$$

This can be expressed in matrix notation as

$$\widetilde{\mathbf{K}} = \mathbf{K} - \mathbf{1}_N\mathbf{K} - \mathbf{K}\mathbf{1}_N + \mathbf{1}_N\mathbf{K}\mathbf{1}_N \tag{12.85}$$

where $\mathbf{1}_N$ denotes the $N \times N$ matrix in which every element takes the value $1/N$. Thus we can evaluate $\widetilde{\mathbf{K}}$ using only the kernel function and then use $\widetilde{\mathbf{K}}$ to determine the eigenvalues and eigenvectors. Note that the standard PCA algorithm is recovered as a special case if we use a linear kernel $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^{\mathrm{T}}\mathbf{x}'$. Figure 12.17 shows an example of kernel PCA applied to a synthetic data set (Schölkopf *et al.*, 1998). Here a 'Gaussian' kernel of the form

Exercise 12.27

$$k(\mathbf{x}, \mathbf{x}') = \exp(-\|\mathbf{x} - \mathbf{x}'\|^2/0.1) \tag{12.86}$$

is applied to a synthetic data set. The lines correspond to contours along which the projection onto the corresponding principal component, defined by

$$\phi(\mathbf{x})^{\mathrm{T}}\mathbf{v}_i = \sum_{n=1}^{N} a_{in}k(\mathbf{x}, \mathbf{x}_n) \tag{12.87}$$

is constant.

Figure 12.17 Example of kernel PCA, with a Gaussian kernel applied to a synthetic data set in two dimensions, showing the first eight eigenfunctions along with their eigenvalues. The contours are lines along which the projection onto the corresponding principal component is constant. Note how the first two eigenvectors separate the three clusters, the next three eigenvectors split each of the cluster into halves, and the following three eigenvectors again split the clusters into halves along directions orthogonal to the previous splits.

One obvious disadvantage of kernel PCA is that it involves finding the eigenvectors of the $N \times N$ matrix $\widetilde{\mathbf{K}}$ rather than the $D \times D$ matrix $\mathbf{S}$ of conventional linear PCA, and so in practice for large data sets approximations are often used.

Finally, we note that in standard linear PCA, we often retain some reduced number $L < D$ of eigenvectors and then approximate a data vector $\mathbf{x}_n$ by its projection $\widehat{\mathbf{x}}_n$ onto the L-dimensional principal subspace, defined by

$$\widehat{\mathbf{x}}_n = \sum_{i=1}^{L} \left(\mathbf{x}_n^{\mathrm{T}} \mathbf{u}_i\right) \mathbf{u}_i. \tag{12.88}$$

In kernel PCA, this will in general not be possible. To see this, note that the mapping $\boldsymbol{\phi}(\mathbf{x})$ maps the D-dimensional $\mathbf{x}$ space into a D-dimensional *manifold* in the M-dimensional feature space $\boldsymbol{\phi}$. The vector $\mathbf{x}$ is known as the *pre-image* of the corresponding point $\boldsymbol{\phi}(\mathbf{x})$. However, the projection of points in feature space onto the linear PCA subspace in that space will typically not lie on the nonlinear D-dimensional manifold and so will not have a corresponding pre-image in data space. Techniques have therefore been proposed for finding approximate pre-images (Bakir *et al.*, 2004).

12.4. Nonlinear Latent Variable Models

In this chapter, we have focussed on the simplest class of models having continuous latent variables, namely those based on linear-Gaussian distributions. As well as having great practical importance, these models are relatively easy to analyse and to fit to data and can also be used as components in more complex models. Here we consider briefly some generalizations of this framework to models that are either nonlinear or non-Gaussian, or both.

In fact, the issues of nonlinearity and non-Gaussianity are related because a general probability density can be obtained from a simple fixed reference density, such as a Gaussian, by making a nonlinear change of variables. This idea forms the basis of several practical latent variable models as we shall see shortly.

Exercise 12.28

12.4.1 Independent component analysis

We begin by considering models in which the observed variables are related linearly to the latent variables, but for which the latent distribution is non-Gaussian. An important class of such models, known as *independent component analysis*, or *ICA*, arises when we consider a distribution over the latent variables that factorizes, so that

$$p(\mathbf{z}) = \prod_{j=1}^{M} p(z_j). \tag{12.89}$$

To understand the role of such models, consider a situation in which two people are talking at the same time, and we record their voices using two microphones. If we ignore effects such as time delay and echoes, then the signals received by the microphones at any point in time will be given by linear combinations of the amplitudes of the two voices. The coefficients of this linear combination will be constant, and if we can infer their values from sample data, then we can invert the mixing process (assuming it is nonsingular) and thereby obtain two clean signals each of which contains the voice of just one person. This is an example of a problem called *blind source separation* in which 'blind' refers to the fact that we are given only the mixed data, and neither the original sources nor the mixing coefficients are observed (Cardoso, 1998).

This type of problem is sometimes addressed using the following approach (MacKay, 2003) in which we ignore the temporal nature of the signals and treat the successive samples as i.i.d. We consider a generative model in which there are two latent variables corresponding to the unobserved speech signal amplitudes, and there are two observed variables given by the signal values at the microphones. The latent variables have a joint distribution that factorizes as above, and the observed variables are given by a linear combination of the latent variables. There is no need to include a noise distribution because the number of latent variables equals the number of observed variables, and therefore the marginal distribution of the observed variables will not in general be singular, so the observed variables are simply deterministic linear combinations of the latent variables. Given a data set of observations, the

likelihood function for this model is a function of the coefficients in the linear combination. The log likelihood can be maximized using gradient-based optimization giving rise to a particular version of independent component analysis.

The success of this approach requires that the latent variables have non-Gaussian distributions. To see this, recall that in probabilistic PCA (and in factor analysis) the latent-space distribution is given by a zero-mean isotropic Gaussian. The model therefore cannot distinguish between two different choices for the latent variables where these differ simply by a rotation in latent space. This can be verified directly by noting that the marginal density (12.35), and hence the likelihood function, is unchanged if we make the transformation $\mathbf{W} \rightarrow \mathbf{WR}$ where $\mathbf{R}$ is an orthogonal matrix satisfying $\mathbf{RR}^{\mathrm{T}} = \mathbf{I}$, because the matrix $\mathbf{C}$ given by (12.36) is itself invariant. Extending the model to allow more general Gaussian latent distributions does not change this conclusion because, as we have seen, such a model is equivalent to the zero-mean isotropic Gaussian latent variable model.

Another way to see why a Gaussian latent variable distribution in a linear model is insufficient to find independent components is to note that the principal components represent a rotation of the coordinate system in data space such as to diagonalize the covariance matrix, so that the data distribution in the new coordinates is then uncorrelated. Although zero correlation is a necessary condition for independence it is not, however, sufficient. In practice, a common choice for the latent-variable distribution is given by

Exercise 12.29

$$p(z_j) = \frac{1}{\pi \cosh(z_j)} = \frac{1}{\pi(e^{z_j} + e^{-z_j})} \tag{12.90}$$

which has heavy tails compared to a Gaussian, reflecting the observation that many real-world distributions also exhibit this property.

The original ICA model (Bell and Sejnowski, 1995) was based on the optimization of an objective function defined by information maximization. One advantage of a probabilistic latent variable formulation is that it helps to motivate and formulate generalizations of basic ICA. For instance, *independent factor analysis* (Attias, 1999a) considers a model in which the number of latent and observed variables can differ, the observed variables are noisy, and the individual latent variables have flexible distributions modelled by mixtures of Gaussians. The log likelihood for this model is maximized using EM, and the reconstruction of the latent variables is approximated using a variational approach. Many other types of model have been considered, and there is now a huge literature on ICA and its applications (Jutten and Herault, 1991; Comon *et al.*, 1991; Amari *et al.*, 1996; Pearlmutter and Parra, 1997; Hyvärinen and Oja, 1997; Hinton *et al.*, 2001; Miskin and MacKay, 2001; Hojen-Sorensen *et al.*, 2002; Choudrey and Roberts, 2003; Chan *et al.*, 2003; Stone, 2004).

12.4.2 Autoassociative neural networks

In Chapter 5 we considered neural networks in the context of supervised learning, where the role of the network is to predict the output variables given values

Figure 12.18 An autoassociative multilayer perceptron having two layers of weights. Such a network is trained to map input vectors onto themselves by minimization of a sum-of-squares error. Even with nonlinear units in the hidden layer, such a network is equivalent to linear principal component analysis. Links representing bias parameters have been omitted for clarity.

for the input variables. However, neural networks have also been applied to unsupervised learning where they have been used for dimensionality reduction. This is achieved by using a network having the same number of outputs as inputs, and optimizing the weights so as to minimize some measure of the reconstruction error between inputs and outputs with respect to a set of training data.

Consider first a multilayer perceptron of the form shown in Figure 12.18, having D inputs, D output units and M hidden units, with $M < D$. The targets used to train the network are simply the input vectors themselves, so that the network is attempting to map each input vector onto itself. Such a network is said to form an *autoassociative* mapping. Since the number of hidden units is smaller than the number of inputs, a perfect reconstruction of all input vectors is not in general possible. We therefore determine the network parameters $\mathbf{w}$ by minimizing an error function which captures the degree of mismatch between the input vectors and their reconstructions. In particular, we shall choose a sum-of-squares error of the form

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{x}_n\|^2. \tag{12.91}$$

If the hidden units have linear activations functions, then it can be shown that the error function has a unique global minimum, and that at this minimum the network performs a projection onto the M-dimensional subspace which is spanned by the first M principal components of the data (Bourlard and Kamp, 1988; Baldi and Hornik, 1989). Thus, the vectors of weights which lead into the hidden units in Figure 12.18 form a basis set which spans the principal subspace. Note, however, that these vectors need not be orthogonal or normalized. This result is unsurprising, since both principal component analysis and the neural network are using linear dimensionality reduction and are minimizing the same sum-of-squares error function.

It might be thought that the limitations of a linear dimensionality reduction could be overcome by using nonlinear (sigmoidal) activation functions for the hidden units in the network in Figure 12.18. However, even with nonlinear hidden units, the minimum error solution is again given by the projection onto the principal component subspace (Bourlard and Kamp, 1988). There is therefore no advantage in using two-layer neural networks to perform dimensionality reduction. Standard techniques for principal component analysis (based on singular value decomposition) are guaranteed to give the correct solution in finite time, and they also generate an ordered set of eigenvalues with corresponding orthonormal eigenvectors.

Figure 12.19 Addition of extra hidden layers of nonlinear units gives an autoassociative network which can perform a nonlinear dimensionality reduction.

The situation is different, however, if additional hidden layers are permitted in the network. Consider the four-layer autoassociative network shown in Figure 12.19. Again the output units are linear, and the M units in the second hidden layer can also be linear, however, the first and third hidden layers have sigmoidal nonlinear activation functions. The network is again trained by minimization of the error function (12.91). We can view this network as two successive functional mappings $\mathbf{F}_1$ and $\mathbf{F}_2$, as indicated in Figure 12.19. The first mapping $\mathbf{F}_1$ projects the original D-dimensional data onto an M-dimensional subspace $\mathcal{S}$ defined by the activations of the units in the second hidden layer. Because of the presence of the first hidden layer of nonlinear units, this mapping is very general, and in particular is not restricted to being linear. Similarly, the second half of the network defines an arbitrary functional mapping from the M-dimensional space back into the original D-dimensional input space. This has a simple geometrical interpretation, as indicated for the case $D = 3$ and $M = 2$ in Figure 12.20.

Such a network effectively performs a nonlinear principal component analysis.

Figure 12.20 Geometrical interpretation of the mappings performed by the network in Figure 12.19 for the case of $D = 3$ inputs and $M = 2$ units in the middle hidden layer. The function $\mathbf{F}_2$ maps from an M-dimensional space $\mathcal{S}$ into a D-dimensional space and therefore defines the way in which the space $\mathcal{S}$ is embedded within the original $\mathbf{x}$-space. Since the mapping $\mathbf{F}_2$ can be nonlinear, the embedding of $\mathcal{S}$ can be nonplanar, as indicated in the figure. The mapping $\mathbf{F}_1$ then defines a projection of points in the original D-dimensional space into the M-dimensional subspace $\mathcal{S}$.

It has the advantage of not being limited to linear transformations, although it contains standard principal component analysis as a special case. However, training the network now involves a nonlinear optimization problem, since the error function (12.91) is no longer a quadratic function of the network parameters. Computationally intensive nonlinear optimization techniques must be used, and there is the risk of finding a suboptimal local minimum of the error function. Also, the dimensionality of the subspace must be specified before training the network.

12.4.3 Modelling nonlinear manifolds

As we have already noted, many natural sources of data correspond to low-dimensional, possibly noisy, nonlinear manifolds embedded within the higher dimensional observed data space. Capturing this property explicitly can lead to improved density modelling compared with more general methods. Here we consider briefly a range of techniques that attempt to do this.

One way to model the nonlinear structure is through a combination of linear models, so that we make a piece-wise linear approximation to the manifold. This can be obtained, for instance, by using a clustering technique such as K-means based on Euclidean distance to partition the data set into local groups with standard PCA applied to each group. A better approach is to use the reconstruction error for cluster assignment (Kambhatla and Leen, 1997; Hinton *et al.*, 1997) as then a common cost function is being optimized in each stage. However, these approaches still suffer from limitations due to the absence of an overall density model. By using probabilistic PCA it is straightforward to define a fully probabilistic model simply by considering a mixture distribution in which the components are probabilistic PCA models (Tipping and Bishop, 1999a). Such a model has both discrete latent variables, corresponding to the discrete mixture, as well as continuous latent variables, and the likelihood function can be maximized using the EM algorithm. A fully Bayesian treatment, based on variational inference (Bishop and Winn, 2000), allows the number of components in the mixture, as well as the effective dimensionalities of the individual models, to be inferred from the data. There are many variants of this model in which parameters such as the $\mathbf{W}$ matrix or the noise variances are tied across components in the mixture, or in which the isotropic noise distributions are replaced by diagonal ones, giving rise to a mixture of factor analysers (Ghahramani and Hinton, 1996a; Ghahramani and Beal, 2000). The mixture of probabilistic PCA models can also be extended hierarchically to produce an interactive data visualization algorithm (Bishop and Tipping, 1998).

An alternative to considering a mixture of linear models is to consider a single nonlinear model. Recall that conventional PCA finds a linear subspace that passes close to the data in a least-squares sense. This concept can be extended to one-dimensional nonlinear surfaces in the form of *principal curves* (Hastie and Stuetzle, 1989). We can describe a curve in a D-dimensional data space using a vector-valued function $\mathbf{f}(\lambda)$, which is a vector each of whose elements is a function of the scalar λ. There are many possible ways to parameterize the curve, of which a natural choice is the arc length along the curve. For any given point $\widehat{\mathbf{x}}$ in data space, we can find the point on the curve that is closest in Euclidean distance. We denote this point by

$\lambda = g_{\mathbf{f}}(\mathbf{x})$ because it depends on the particular curve $\mathbf{f}(\lambda)$. For a continuous data density $p(\mathbf{x})$, a principal curve is defined as one for which every point on the curve is the mean of all those points in data space that project to it, so that

$$\mathbb{E}\left[\mathbf{x}|g_{\mathbf{f}}(\mathbf{x}) = \lambda\right] = \mathbf{f}(\lambda). \tag{12.92}$$

For a given continuous density, there can be many principal curves. In practice, we are interested in finite data sets, and we also wish to restrict attention to smooth curves. Hastie and Stuetzle (1989) propose a two-stage iterative procedure for finding such principal curves, somewhat reminiscent of the EM algorithm for PCA. The curve is initialized using the first principal component, and then the algorithm alternates between a data projection step and curve re-estimation step. In the projection step, each data point is assigned to a value of λ corresponding to the closest point on the curve. Then in the re-estimation step, each point on the curve is given by a weighted average of those points that project to nearby points on the curve, with points closest on the curve given the greatest weight. In the case where the subspace is constrained to be linear, the procedure converges to the first principal component and is equivalent to the power method for finding the largest eigenvector of the covariance matrix. Principal curves can be generalized to multidimensional manifolds called *principal surfaces* although these have found limited use due to the difficulty of data smoothing in higher dimensions even for two-dimensional manifolds.

PCA is often used to project a data set onto a lower-dimensional space, for example two dimensional, for the purposes of visualization. Another linear technique with a similar aim is *multidimensional scaling*, or *MDS* (Cox and Cox, 2000). It finds a low-dimensional projection of the data such as to preserve, as closely as possible, the pairwise distances between data points, and involves finding the eigenvectors of the distance matrix. In the case where the distances are Euclidean, it gives equivalent results to PCA. The MDS concept can be extended to a wide variety of data types specified in terms of a similarity matrix, giving *nonmetric* MDS.

Two other nonprobabilistic methods for dimensionality reduction and data visualization are worthy of mention. *Locally linear embedding*, or *LLE* (Roweis and Saul, 2000) first computes the set of coefficients that best reconstructs each data point from its neighbours. These coefficients are arranged to be invariant to rotations, translations, and scalings of that data point and its neighbours, and hence they characterize the local geometrical properties of the neighbourhood. LLE then maps the high-dimensional data points down to a lower dimensional space while preserving these neighbourhood coefficients. If the local neighbourhood for a particular data point can be considered linear, then the transformation can be achieved using a combination of translation, rotation, and scaling, such as to preserve the angles formed between the data points and their neighbours. Because the weights are invariant to these transformations, we expect the same weight values to reconstruct the data points in the low-dimensional space as in the high-dimensional data space. In spite of the nonlinearity, the optimization for LLE does not exhibit local minima.

In *isometric feature mapping*, or *isomap* (Tenenbaum *et al.*, 2000), the goal is to project the data to a lower-dimensional space using MDS, but where the dissimilarities are defined in terms of the *geodesic distances* measured along the mani-

fold. For instance, if two points lie on a circle, then the geodesic is the arc-length distance measured around the circumference of the circle not the straight line distance measured along the chord connecting them. The algorithm first defines the neighbourhood for each data point, either by finding the K nearest neighbours or by finding all points within a sphere of radius ϵ. A graph is then constructed by linking all neighbouring points and labelling them with their Euclidean distance. The geodesic distance between any pair of points is then approximated by the sum of the arc lengths along the shortest path connecting them (which itself is found using standard algorithms). Finally, metric MDS is applied to the geodesic distance matrix to find the low-dimensional projection.

Our focus in this chapter has been on models for which the observed variables are continuous. We can also consider models having continuous latent variables together with discrete observed variables, giving rise to *latent trait* models (Bartholomew, 1987). In this case, the marginalization over the continuous latent variables, even for a linear relationship between latent and observed variables, cannot be performed analytically, and so more sophisticated techniques are required. Tipping (1999) uses variational inference in a model with a two-dimensional latent space, allowing a binary data set to be visualized analogously to the use of PCA to visualize continuous data. Note that this model is the dual of the Bayesian logistic regression problem discussed in Section 4.5. In the case of logistic regression we have N observations of the feature vector $\boldsymbol{\phi}_n$ which are parameterized by a single parameter vector $\mathbf{w}$, whereas in the latent space visualization model there is a single latent space variable $\mathbf{x}$ (analogous to $\boldsymbol{\phi}$) and N copies of the latent variable $\mathbf{w}_n$. A generalization of probabilistic latent variable models to general exponential family distributions is described in Collins *et al.* (2002).

We have already noted that an arbitrary distribution can be formed by taking a Gaussian random variable and transforming it through a suitable nonlinearity. This is exploited in a general latent variable model called a *density network* (MacKay, 1995; MacKay and Gibbs, 1999) in which the nonlinear function is governed by a multilayered neural network. If the network has enough hidden units, it can approximate a given nonlinear function to any desired accuracy. The downside of having such a flexible model is that the marginalization over the latent variables, required in order to obtain the likelihood function, is no longer analytically tractable. Instead, the likelihood is approximated using Monte Carlo techniques by drawing samples from the Gaussian prior. The marginalization over the latent variables then becomes a simple sum with one term for each sample. However, because a large number of sample points may be required in order to give an accurate representation of the marginal, this procedure can be computationally costly.

Chapter 5

Chapter 11

If we consider more restricted forms for the nonlinear function, and make an appropriate choice of the latent variable distribution, then we can construct a latent variable model that is both nonlinear and efficient to train. The *generative topographic mapping*, or *GTM* (Bishop *et al.*, 1996; Bishop *et al.*, 1997a; Bishop *et al.*, 1998b) uses a latent distribution that is defined by a finite regular grid of delta functions over the (typically two-dimensional) latent space. Marginalization over the latent space then simply involves summing over the contributions from each of the grid locations.

Figure 12.21 Plot of the oil flow data set visualized using PCA on the left and GTM on the right. For the GTM model, each data point is plotted at the mean of its posterior distribution in latent space. The nonlinearity of the GTM model allows the separation between the groups of data points to be seen more clearly.

Chapter 3

Section 1.4

The nonlinear mapping is given by a linear regression model that allows for general nonlinearity while being a linear function of the adaptive parameters. Note that the usual limitation of linear regression models arising from the curse of dimensionality does not arise in the context of the GTM since the manifold generally has two dimensions irrespective of the dimensionality of the data space. A consequence of these two choices is that the likelihood function can be expressed analytically in closed form and can be optimized efficiently using the EM algorithm. The resulting GTM model fits a two-dimensional nonlinear manifold to the data set, and by evaluating the posterior distribution over latent space for the data points, they can be projected back to the latent space for visualization purposes. Figure 12.21 shows a comparison of the oil data set visualized with linear PCA and with the nonlinear GTM.

The GTM can be seen as a probabilistic version of an earlier model called the *self organizing map*, or *SOM* (Kohonen, 1982; Kohonen, 1995), which also represents a two-dimensional nonlinear manifold as a regular array of discrete points. The SOM is somewhat reminiscent of the K-means algorithm in that data points are assigned to nearby prototype vectors that are then subsequently updated. Initially, the prototypes are distributed at random, and during the training process they 'self organize' so as to approximate a smooth manifold. Unlike K-means, however, the SOM is not optimizing any well-defined cost function (Erwin *et al.*, 1992) making it difficult to set the parameters of the model and to assess convergence. There is also no guarantee that the 'self-organization' will take place as this is dependent on the choice of appropriate parameter values for any particular data set.

By contrast, GTM optimizes the log likelihood function, and the resulting model defines a probability density in data space. In fact, it corresponds to a constrained mixture of Gaussians in which the components share a common variance, and the means are constrained to lie on a smooth two-dimensional manifold. This proba-

bilistic foundation also makes it very straightforward to define generalizations of GTM (Bishop *et al.*, 1998a) such as a Bayesian treatment, dealing with missing values, a principled extension to discrete variables, the use of Gaussian processes to define the manifold, or a hierarchical GTM model (Tino and Nabney, 2002).

Section 6.4

Because the manifold in GTM is defined as a continuous surface, not just at the prototype vectors as in the SOM, it is possible to compute the *magnification factors* corresponding to the local expansions and compressions of the manifold needed to fit the data set (Bishop *et al.*, 1997b) as well as the *directional curvatures* of the manifold (Tino *et al.*, 2001). These can be visualized along with the projected data and provide additional insight into the model.

Exercises

12.1 ($\star\star$) **WWW** In this exercise, we use proof by induction to show that the linear projection onto an M-dimensional subspace that maximizes the variance of the projected data is defined by the M eigenvectors of the data covariance matrix $\mathbf{S}$, given by (12.3), corresponding to the M largest eigenvalues. In Section 12.1, this result was proven for the case of $M = 1$. Now suppose the result holds for some general value of M and show that it consequently holds for dimensionality $M + 1$. To do this, first set the derivative of the variance of the projected data with respect to a vector $\mathbf{u}_{M+1}$ defining the new direction in data space equal to zero. This should be done subject to the constraints that $\mathbf{u}_{M+1}$ be orthogonal to the existing vectors $\mathbf{u}_1, \ldots, \mathbf{u}_M$, and also that it be normalized to unit length. Use Lagrange multipliers to enforce these constraints. Then make use of the orthonormality properties of the vectors $\mathbf{u}_1, \ldots, \mathbf{u}_M$ to show that the new vector $\mathbf{u}_{M+1}$ is an eigenvector of $\mathbf{S}$. Finally, show that the variance is maximized if the eigenvector is chosen to be the one corresponding to eigenvector λ_{M+1} where the eigenvalues have been ordered in decreasing value.

Appendix E

12.2 ($\star\star$) Show that the minimum value of the PCA distortion measure J given by (12.15) with respect to the $\mathbf{u}_i$, subject to the orthonormality constraints (12.7), is obtained when the $\mathbf{u}_i$ are eigenvectors of the data covariance matrix $\mathbf{S}$. To do this, introduce a matrix $\mathbf{H}$ of Lagrange multipliers, one for each constraint, so that the modified distortion measure, in matrix notation reads

$$\widetilde{J} = \mathrm{Tr}\left\{\widehat{\mathbf{U}}^{\mathrm{T}}\mathbf{S}\widehat{\mathbf{U}}\right\} + \mathrm{Tr}\left\{\mathbf{H}(\mathbf{I} - \widehat{\mathbf{U}}^{\mathrm{T}}\widehat{\mathbf{U}})\right\} \tag{12.93}$$

where $\widehat{\mathbf{U}}$ is a matrix of dimension $D \times (D - M)$ whose columns are given by $\mathbf{u}_i$. Now minimize $\widetilde{J}$ with respect to $\widehat{\mathbf{U}}$ and show that the solution satisfies $\mathbf{S}\widehat{\mathbf{U}} = \widehat{\mathbf{U}}\mathbf{H}$. Clearly, one possible solution is that the columns of $\widehat{\mathbf{U}}$ are eigenvectors of $\mathbf{S}$, in which case $\mathbf{H}$ is a diagonal matrix containing the corresponding eigenvalues. To obtain the general solution, show that $\mathbf{H}$ can be assumed to be a symmetric matrix, and by using its eigenvector expansion show that the general solution to $\mathbf{S}\widehat{\mathbf{U}} = \widehat{\mathbf{U}}\mathbf{H}$ gives the same value for $\widetilde{J}$ as the specific solution in which the columns of $\widehat{\mathbf{U}}$ are

the eigenvectors of $\mathbf{S}$. Because these solutions are all equivalent, it is convenient to choose the eigenvector solution.

12.3 ($\star$) Verify that the eigenvectors defined by (12.30) are normalized to unit length, assuming that the eigenvectors $\mathbf{v}_i$ have unit length.

12.4 ($\star$) **WWW** Suppose we replace the zero-mean, unit-covariance latent space distribution (12.31) in the probabilistic PCA model by a general Gaussian distribution of the form $\mathcal{N}(\mathbf{z}|\mathbf{m}, \boldsymbol{\Sigma})$. By redefining the parameters of the model, show that this leads to an identical model for the marginal distribution $p(\mathbf{x})$ over the observed variables for any valid choice of $\mathbf{m}$ and $\boldsymbol{\Sigma}$.

12.5 ($\star\star$) Let $\mathbf{x}$ be a D-dimensional random variable having a Gaussian distribution given by $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$, and consider the M-dimensional random variable given by $\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b}$ where $\mathbf{A}$ is an $M \times D$ matrix. Show that $\mathbf{y}$ also has a Gaussian distribution, and find expressions for its mean and covariance. Discuss the form of this Gaussian distribution for $M < D$, for $M = D$, and for $M > D$.

12.6 ($\star$) **WWW** Draw a directed probabilistic graph for the probabilistic PCA model described in Section 12.2 in which the components of the observed variable $\mathbf{x}$ are shown explicitly as separate nodes. Hence verify that the probabilistic PCA model has the same independence structure as the naive Bayes model discussed in Section 8.2.2.

12.7 ($\star\star$) By making use of the results (2.270) and (2.271) for the mean and covariance of a general distribution, derive the result (12.35) for the marginal distribution $p(\mathbf{x})$ in the probabilistic PCA model.

12.8 ($\star\star$) **WWW** By making use of the result (2.116), show that the posterior distribution $p(\mathbf{z}|\mathbf{x})$ for the probabilistic PCA model is given by (12.42).

12.9 ($\star$) Verify that maximizing the log likelihood (12.43) for the probabilistic PCA model with respect to the parameter $\boldsymbol{\mu}$ gives the result $\boldsymbol{\mu}_{\mathrm{ML}} = \overline{\mathbf{x}}$ where $\overline{\mathbf{x}}$ is the mean of the data vectors.

12.10 ($\star\star$) By evaluating the second derivatives of the log likelihood function (12.43) for the probabilistic PCA model with respect to the parameter $\boldsymbol{\mu}$, show that the stationary point $\boldsymbol{\mu}_{\mathrm{ML}} = \overline{\mathbf{x}}$ represents the unique maximum.

12.11 ($\star\star$) **WWW** Show that in the limit $\sigma^2 \to 0$, the posterior mean for the probabilistic PCA model becomes an orthogonal projection onto the principal subspace, as in conventional PCA.

12.12 ($\star\star$) For $\sigma^2 > 0$ show that the posterior mean in the probabilistic PCA model is shifted towards the origin relative to the orthogonal projection.

12.13 ($\star\star$) Show that the optimal reconstruction of a data point under probabilistic PCA, according to the least squares projection cost of conventional PCA, is given by

$$\widetilde{\mathbf{x}} = \mathbf{W}_{\mathrm{ML}}(\mathbf{W}_{\mathrm{ML}}^{\mathrm{T}}\mathbf{W}_{\mathrm{ML}})^{-1}\mathbf{M}\mathbb{E}[\mathbf{z}|\mathbf{x}]. \tag{12.94}$$

12.14 ($\star$) The number of independent parameters in the covariance matrix for the probabilistic PCA model with an M-dimensional latent space and a D-dimensional data space is given by (12.51). Verify that in the case of $M = D - 1$, the number of independent parameters is the same as in a general covariance Gaussian, whereas for $M = 0$ it is the same as for a Gaussian with an isotropic covariance.

12.15 ($\star\star$) **WWW** Derive the M-step equations (12.56) and (12.57) for the probabilistic PCA model by maximization of the expected complete-data log likelihood function given by (12.53).

12.16 ($\star\star\star$) In Figure 12.11, we showed an application of probabilistic PCA to a data set in which some of the data values were missing at random. Derive the EM algorithm for maximizing the likelihood function for the probabilistic PCA model in this situation. Note that the $\{\mathbf{z}_n\}$, as well as the missing data values that are components of the vectors $\{\mathbf{x}_n\}$, are now latent variables. Show that in the special case in which all of the data values are observed, this reduces to the EM algorithm for probabilistic PCA derived in Section 12.2.2.

12.17 ($\star\star$) **WWW** Let $\mathbf{W}$ be a $D \times M$ matrix whose columns define a linear subspace of dimensionality M embedded within a data space of dimensionality D, and let $\boldsymbol{\mu}$ be a D-dimensional vector. Given a data set $\{\mathbf{x}_n\}$ where $n = 1, \ldots, N$, we can approximate the data points using a linear mapping from a set of M-dimensional vectors $\{\mathbf{z}_n\}$, so that $\mathbf{x}_n$ is approximated by $\mathbf{W}\mathbf{z}_n + \boldsymbol{\mu}$. The associated sum-of-squares reconstruction cost is given by

$$J = \sum_{n=1}^{N} \|\mathbf{x}_n - \boldsymbol{\mu} - \mathbf{W}\mathbf{z}_n\|^2. \tag{12.95}$$

First show that minimizing J with respect to $\boldsymbol{\mu}$ leads to an analogous expression with $\mathbf{x}_n$ and $\mathbf{z}_n$ replaced by zero-mean variables $\mathbf{x}_n - \overline{\mathbf{x}}$ and $\mathbf{z}_n - \overline{\mathbf{z}}$, respectively, where $\overline{\mathbf{x}}$ and $\overline{\mathbf{z}}$ denote sample means. Then show that minimizing J with respect to $\mathbf{z}_n$, where $\mathbf{W}$ is kept fixed, gives rise to the PCA E step (12.58), and that minimizing J with respect to $\mathbf{W}$, where $\{\mathbf{z}_n\}$ is kept fixed, gives rise to the PCA M step (12.59).

12.18 ($\star$) Derive an expression for the number of independent parameters in the factor analysis model described in Section 12.2.4.

12.19 ($\star\star$) **WWW** Show that the factor analysis model described in Section 12.2.4 is invariant under rotations of the latent space coordinates.

12.20 ($\star\star$) By considering second derivatives, show that the only stationary point of the log likelihood function for the factor analysis model discussed in Section 12.2.4 with respect to the parameter $\boldsymbol{\mu}$ is given by the sample mean defined by (12.1). Furthermore, show that this stationary point is a maximum.

12.21 ($\star\star$) Derive the formulae (12.66) and (12.67) for the E step of the EM algorithm for factor analysis. Note that from the result of Exercise 12.20, the parameter $\boldsymbol{\mu}$ can be replaced by the sample mean $\overline{\mathbf{x}}$.

12.22 ($\star\star$) Write down an expression for the expected complete-data log likelihood function for the factor analysis model, and hence derive the corresponding M step equations (12.69) and (12.70).

12.23 ($\star$) **WWW** Draw a directed probabilistic graphical model representing a discrete mixture of probabilistic PCA models in which each PCA model has its own values of $\mathbf{W}$, $\boldsymbol{\mu}$, and σ^2. Now draw a modified graph in which these parameter values are shared between the components of the mixture.

12.24 ($\star\star\star$) We saw in Section 2.3.7 that Student's t-distribution can be viewed as an infinite mixture of Gaussians in which we marginalize with respect to a continuous latent variable. By exploiting this representation, formulate an EM algorithm for maximizing the log likelihood function for a multivariate Student's t-distribution given an observed set of data points, and derive the forms of the E and M step equations.

12.25 ($\star\star$) **WWW** Consider a linear-Gaussian latent-variable model having a latent space distribution $p(\mathbf{z}) = \mathcal{N}(\mathbf{x}|\mathbf{0}, \mathbf{I})$ and a conditional distribution for the observed variable $p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|\mathbf{W}\mathbf{z} + \boldsymbol{\mu}, \boldsymbol{\Phi})$ where $\boldsymbol{\Phi}$ is an arbitrary symmetric, positive-definite noise covariance matrix. Now suppose that we make a nonsingular linear transformation of the data variables $\mathbf{x} \rightarrow \mathbf{A}\mathbf{x}$, where $\mathbf{A}$ is a $D \times D$ matrix. If $\boldsymbol{\mu}_{\mathrm{ML}}$, $\mathbf{W}_{\mathrm{ML}}$ and $\boldsymbol{\Phi}_{\mathrm{ML}}$ represent the maximum likelihood solution corresponding to the original untransformed data, show that $\mathbf{A}\boldsymbol{\mu}_{\mathrm{ML}}$, $\mathbf{A}\mathbf{W}_{\mathrm{ML}}$, and $\mathbf{A}\boldsymbol{\Phi}_{\mathrm{ML}}\mathbf{A}^{\mathrm{T}}$ will represent the corresponding maximum likelihood solution for the transformed data set. Finally, show that the form of the model is preserved in two cases: (i) $\mathbf{A}$ is a diagonal matrix and $\boldsymbol{\Phi}$ is a diagonal matrix. This corresponds to the case of factor analysis. The transformed $\boldsymbol{\Phi}$ remains diagonal, and hence factor analysis is *covariant* under component-wise re-scaling of the data variables; (ii) $\mathbf{A}$ is orthogonal and $\boldsymbol{\Phi}$ is proportional to the unit matrix so that $\boldsymbol{\Phi} = \sigma^2\mathbf{I}$. This corresponds to probabilistic PCA. The transformed $\boldsymbol{\Phi}$ matrix remains proportional to the unit matrix, and hence probabilistic PCA is covariant under a rotation of the axes of data space, as is the case for conventional PCA.

12.26 ($\star\star$) Show that any vector $\mathbf{a}_i$ that satisfies (12.80) will also satisfy (12.79). Also, show that for any solution of (12.80) having eigenvalue λ, we can add any multiple of an eigenvector of $\mathbf{K}$ having zero eigenvalue, and obtain a solution to (12.79) that also has eigenvalue λ. Finally, show that such modifications do not affect the principal-component projection given by (12.82).

12.27 ($\star\star$) Show that the conventional linear PCA algorithm is recovered as a special case of kernel PCA if we choose the linear kernel function given by $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^{\mathrm{T}}\mathbf{x}'$.

12.28 ($\star\star$) **WWW** Use the transformation property (1.27) of a probability density under a change of variable to show that any density $p(y)$ can be obtained from a fixed density $q(x)$ that is everywhere nonzero by making a nonlinear change of variable $y = f(x)$ in which $f(x)$ is a monotonic function so that $0 \leqslant f'(x) < \infty$. Write down the differential equation satisfied by $f(x)$ and draw a diagram illustrating the transformation of the density.

12.29 ($\star\star$) **WWW** Suppose that two variables z_1 and z_2 are independent so that $p(z_1, z_2) = p(z_1)p(z_2)$. Show that the covariance matrix between these variables is diagonal. This shows that independence is a sufficient condition for two variables to be uncorrelated. Now consider two variables y_1 and y_2 in which $-1 \leqslant y_1 \leqslant 1$ and $y_2 = y_2^2$. Write down the conditional distribution $p(y_2|y_1)$ and observe that this is dependent on y_1, showing that the two variables are not independent. Now show that the covariance matrix between these two variables is again diagonal. To do this, use the relation $p(y_1, y_2) = p(y_1)p(y_2|y_1)$ to show that the off-diagonal terms are zero. This counter-example shows that zero correlation is not a sufficient condition for independence.

13

Sequential Data

So far in this book, we have focussed primarily on sets of data points that were assumed to be independent and identically distributed (i.i.d.). This assumption allowed us to express the likelihood function as the product over all data points of the probability distribution evaluated at each data point. For many applications, however, the i.i.d. assumption will be a poor one. Here we consider a particularly important class of such data sets, namely those that describe sequential data. These often arise through measurement of time series, for example the rainfall measurements on successive days at a particular location, or the daily values of a currency exchange rate, or the acoustic features at successive time frames used for speech recognition. An example involving speech data is shown in Figure 13.1. Sequential data can also arise in contexts other than time series, for example the sequence of nucleotide base pairs along a strand of DNA or the sequence of characters in an English sentence. For convenience, we shall sometimes refer to 'past' and 'future' observations in a sequence. However, the models explored in this chapter are equally applicable to all

Figure 13.1 Example of a spectrogram of the spoken words "Bayes' theorem" showing a plot of the intensity of the spectral coefficients versus time index.

forms of sequential data, not just temporal sequences.

It is useful to distinguish between stationary and nonstationary sequential distributions. In the stationary case, the data evolves in time, but the distribution from which it is generated remains the same. For the more complex nonstationary situation, the generative distribution itself is evolving with time. Here we shall focus on the stationary case.

For many applications, such as financial forecasting, we wish to be able to predict the next value in a time series given observations of the previous values. Intuitively, we expect that recent observations are likely to be more informative than more historical observations in predicting future values. The example in Figure 13.1 shows that successive observations of the speech spectrum are indeed highly correlated. Furthermore, it would be impractical to consider a general dependence of future observations on all previous observations because the complexity of such a model would grow without limit as the number of observations increases. This leads us to consider *Markov models* in which we assume that future predictions are inde-

Figure 13.2 The simplest approach to modelling a sequence of observations is to treat them as independent, corresponding to a graph without links.

pendent of all but the most recent observations.

Although such models are tractable, they are also severely limited. We can obtain a more general framework, while still retaining tractability, by the introduction of latent variables, leading to *state space models*. As in Chapters 9 and 12, we shall see that complex models can thereby be constructed from simpler components (in particular, from distributions belonging to the exponential family) and can be readily characterized using the framework of probabilistic graphical models. Here we focus on the two most important examples of state space models, namely the *hidden Markov model*, in which the latent variables are discrete, and *linear dynamical systems*, in which the latent variables are Gaussian. Both models are described by directed graphs having a tree structure (no loops) for which inference can be performed efficiently using the sum-product algorithm.

13.1. Markov Models

The easiest way to treat sequential data would be simply to ignore the sequential aspects and treat the observations as i.i.d., corresponding to the graph in Figure 13.2. Such an approach, however, would fail to exploit the sequential patterns in the data, such as correlations between observations that are close in the sequence. Suppose, for instance, that we observe a binary variable denoting whether on a particular day it rained or not. Given a time series of recent observations of this variable, we wish to predict whether it will rain on the next day. If we treat the data as i.i.d., then the only information we can glean from the data is the relative frequency of rainy days. However, we know in practice that the weather often exhibits trends that may last for several days. Observing whether or not it rains today is therefore of significant help in predicting if it will rain tomorrow.

To express such effects in a probabilistic model, we need to relax the i.i.d. assumption, and one of the simplest ways to do this is to consider a *Markov model*. First of all we note that, without loss of generality, we can use the product rule to express the joint distribution for a sequence of observations in the form

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n | \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}). \tag{13.1}$$

If we now assume that each of the conditional distributions on the right-hand side is independent of all previous observations except the most recent, we obtain the *first-order Markov chain*, which is depicted as a graphical model in Figure 13.3. The

Figure 13.3 A first-order Markov chain of observations $\{\mathbf{x}_n\}$ in which the distribution $p(\mathbf{x}_n|\mathbf{x}_{n-1})$ of a particular observation $\mathbf{x}_n$ is conditioned on the value of the previous observation $\mathbf{x}_{n-1}$.

joint distribution for a sequence of N observations under this model is given by

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = p(\mathbf{x}_1) \prod_{n=2}^{N} p(\mathbf{x}_n|\mathbf{x}_{n-1}). \tag{13.2}$$

Section 8.2

From the d-separation property, we see that the conditional distribution for observation $\mathbf{x}_n$, given all of the observations up to time n, is given by

$$p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}) = p(\mathbf{x}_n|\mathbf{x}_{n-1}) \tag{13.3}$$

which is easily verified by direct evaluation starting from (13.2) and using the product rule of probability. Thus if we use such a model to predict the next observation in a sequence, the distribution of predictions will depend only on the value of the immediately preceding observation and will be independent of all earlier observations.

Exercise 13.1

In most applications of such models, the conditional distributions $p(\mathbf{x}_n|\mathbf{x}_{n-1})$ that define the model will be constrained to be equal, corresponding to the assumption of a stationary time series. The model is then known as a *homogeneous* Markov chain. For instance, if the conditional distributions depend on adjustable parameters (whose values might be inferred from a set of training data), then all of the conditional distributions in the chain will share the same values of those parameters.

Although this is more general than the independence model, it is still very restrictive. For many sequential observations, we anticipate that the trends in the data over several successive observations will provide important information in predicting the next value. One way to allow earlier observations to have an influence is to move to higher-order Markov chains. If we allow the predictions to depend also on the previous-but-one value, we obtain a second-order Markov chain, represented by the graph in Figure 13.4. The joint distribution is now given by

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1) \prod_{n=3}^{N} p(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{x}_{n-2}). \tag{13.4}$$

Again, using d-separation or by direct evaluation, we see that the conditional distribution of $\mathbf{x}_n$ given $\mathbf{x}_{n-1}$ and $\mathbf{x}_{n-2}$ is independent of all observations $\mathbf{x}_1, \ldots \mathbf{x}_{n-3}$.

Figure 13.4 A second-order Markov chain, in which the conditional distribution of a particular observation $\mathbf{x}_n$ depends on the values of the two previous observations $\mathbf{x}_{n-1}$ and $\mathbf{x}_{n-2}$.

Figure 13.5 We can represent sequential data using a Markov chain of latent variables, with each observation conditioned on the state of the corresponding latent variable. This important graphical structure forms the foundation both for the hidden Markov model and for linear dynamical systems.

Each observation is now influenced by two previous observations. We can similarly consider extensions to an Mth order Markov chain in which the conditional distribution for a particular variable depends on the previous M variables. However, we have paid a price for this increased flexibility because the number of parameters in the model is now much larger. Suppose the observations are discrete variables having K states. Then the conditional distribution $p(\mathbf{x}_n|\mathbf{x}_{n-1})$ in a first-order Markov chain will be specified by a set of $K-1$ parameters for each of the K states of $\mathbf{x}_{n-1}$ giving a total of $K(K-1)$ parameters. Now suppose we extend the model to an Mth order Markov chain, so that the joint distribution is built up from conditionals $p(\mathbf{x}_n|\mathbf{x}_{n-M}, \ldots, \mathbf{x}_{n-1})$. If the variables are discrete, and if the conditional distributions are represented by general conditional probability tables, then the number of parameters in such a model will have $K^{M-1}(K-1)$ parameters. Because this grows exponentially with M, it will often render this approach impractical for larger values of M.

For continuous variables, we can use linear-Gaussian conditional distributions in which each node has a Gaussian distribution whose mean is a linear function of its parents. This is known as an *autoregressive* or *AR* model (Box *et al.*, 1994; Thiesson *et al.*, 2004). An alternative approach is to use a parametric model for $p(\mathbf{x}_n|\mathbf{x}_{n-M}, \ldots, \mathbf{x}_{n-1})$ such as a neural network. This technique is sometimes called a *tapped delay line* because it corresponds to storing (delaying) the previous M values of the observed variable in order to predict the next value. The number of parameters can then be much smaller than in a completely general model (for example it may grow linearly with M), although this is achieved at the expense of a restricted family of conditional distributions.

Suppose we wish to build a model for sequences that is not limited by the Markov assumption to any order and yet that can be specified using a limited number of free parameters. We can achieve this by introducing additional latent variables to permit a rich class of models to be constructed out of simple components, as we did with mixture distributions in Chapter 9 and with continuous latent variable models in Chapter 12. For each observation $\mathbf{x}_n$, we introduce a corresponding latent variable $\mathbf{z}_n$ (which may be of different type or dimensionality to the observed variable). We now assume that it is the latent variables that form a Markov chain, giving rise to the graphical structure known as a *state space model*, which is shown in Figure 13.5. It satisfies the key conditional independence property that $\mathbf{z}_{n-1}$ and $\mathbf{z}_{n+1}$ are independent given $\mathbf{z}_n$, so that

$$\mathbf{z}_{n+1} \perp\!\!\!\perp \mathbf{z}_{n-1} \mid \mathbf{z}_n. \tag{13.5}$$

The joint distribution for this model is given by

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N, \mathbf{z}_1, \ldots, \mathbf{z}_N) = p(\mathbf{z}_1) \left[\prod_{n=2}^{N} p(\mathbf{z}_n|\mathbf{z}_{n-1}) \right] \prod_{n=1}^{N} p(\mathbf{x}_n|\mathbf{z}_n). \tag{13.6}$$

Using the d-separation criterion, we see that there is always a path connecting any two observed variables $\mathbf{x}_n$ and $\mathbf{x}_m$ via the latent variables, and that this path is never blocked. Thus the predictive distribution $p(\mathbf{x}_{n+1}|\mathbf{x}_1, \ldots, \mathbf{x}_n)$ for observation $\mathbf{x}_{n+1}$ given all previous observations does not exhibit any conditional independence properties, and so our predictions for $\mathbf{x}_{n+1}$ depends on all previous observations. The observed variables, however, do not satisfy the Markov property at any order. We shall discuss how to evaluate the predictive distribution in later sections of this chapter.

There are two important models for sequential data that are described by this graph. If the latent variables are discrete, then we obtain the *hidden Markov model*, or *HMM* (Elliott *et al.*, 1995). Note that the observed variables in an HMM may be discrete or continuous, and a variety of different conditional distributions can be used to model them. If both the latent and the observed variables are Gaussian (with a linear-Gaussian dependence of the conditional distributions on their parents), then we obtain the *linear dynamical system*.

Section 13.2

Section 13.3

13.2. Hidden Markov Models

The hidden Markov model can be viewed as a specific instance of the state space model of Figure 13.5 in which the latent variables are discrete. However, if we examine a single time slice of the model, we see that it corresponds to a mixture distribution, with component densities given by $p(\mathbf{x}|\mathbf{z})$. It can therefore also be interpreted as an extension of a mixture model in which the choice of mixture component for each observation is not selected independently but depends on the choice of component for the previous observation. The HMM is widely used in speech recognition (Jelinek, 1997; Rabiner and Juang, 1993), natural language modelling (Manning and Schütze, 1999), on-line handwriting recognition (Nag *et al.*, 1986), and for the analysis of biological sequences such as proteins and DNA (Krogh *et al.*, 1994; Durbin *et al.*, 1998; Baldi and Brunak, 2001).

As in the case of a standard mixture model, the latent variables are the discrete multinomial variables $\mathbf{z}_n$ describing which component of the mixture is responsible for generating the corresponding observation $\mathbf{x}_n$. Again, it is convenient to use a 1-of-K coding scheme, as used for mixture models in Chapter 9. We now allow the probability distribution of $\mathbf{z}_n$ to depend on the state of the previous latent variable $\mathbf{z}_{n-1}$ through a conditional distribution $p(\mathbf{z}_n|\mathbf{z}_{n-1})$. Because the latent variables are K-dimensional binary variables, this conditional distribution corresponds to a table of numbers that we denote by $\mathbf{A}$, the elements of which are known as *transition probabilities*. They are given by $A_{jk} \equiv p(z_{nk} = 1|z_{n-1,j} = 1)$, and because they are probabilities, they satisfy $0 \leqslant A_{jk} \leqslant 1$ with $\sum_k A_{jk} = 1$, so that the matrix $\mathbf{A}$

Figure 13.6 Transition diagram showing a model whose latent variables have three possible states corresponding to the three boxes. The black lines denote the elements of the transition matrix A_{jk}.

has $K(K-1)$ independent parameters. We can then write the conditional distribution explicitly in the form

$$p(\mathbf{z}_n|\mathbf{z}_{n-1,\mathbf{A}}) = \prod_{k=1}^{K}\prod_{j=1}^{K} A_{jk}^{z_{n-1,j}z_{nk}}. \tag{13.7}$$

The initial latent node $\mathbf{z}_1$ is special in that it does not have a parent node, and so it has a marginal distribution $p(\mathbf{z}_1)$ represented by a vector of probabilities $\boldsymbol{\pi}$ with elements $\pi_k \equiv p(z_{1k} = 1)$, so that

$$p(\mathbf{z}_1|\boldsymbol{\pi}) = \prod_{k=1}^{K} \pi_k^{z_{1k}} \tag{13.8}$$

where $\sum_k \pi_k = 1$.

The transition matrix is sometimes illustrated diagrammatically by drawing the states as nodes in a state transition diagram as shown in Figure 13.6 for the case of $K = 3$. Note that this does not represent a probabilistic graphical model, because the nodes are not separate variables but rather states of a single variable, and so we have shown the states as boxes rather than circles.

It is sometimes useful to take a state transition diagram, of the kind shown in Figure 13.6, and unfold it over time. This gives an alternative representation of the transitions between latent states, known as a *lattice* or *trellis* diagram, and which is shown for the case of the hidden Markov model in Figure 13.7.

Section 8.4.5

The specification of the probabilistic model is completed by defining the conditional distributions of the observed variables $p(\mathbf{x}_n|\mathbf{z}_n, \boldsymbol{\phi})$, where $\boldsymbol{\phi}$ is a set of parameters governing the distribution. These are known as *emission probabilities*, and might for example be given by Gaussians of the form (9.11) if the elements of $\mathbf{x}$ are continuous variables, or by conditional probability tables if $\mathbf{x}$ is discrete. Because $\mathbf{x}_n$ is observed, the distribution $p(\mathbf{x}_n|\mathbf{z}_n, \boldsymbol{\phi})$ consists, for a given value of $\boldsymbol{\phi}$, of a vector of K numbers corresponding to the K possible states of the binary vector $\mathbf{z}_n$.

Figure 13.7 If we unfold the state transition diagram of Figure 13.6 over time, we obtain a lattice, or trellis, representation of the latent states. Each column of this diagram corresponds to one of the latent variables $\mathbf{z}_n$.

We can represent the emission probabilities in the form

$$p(\mathbf{x}_n|\mathbf{z}_n, \boldsymbol{\phi}) = \prod_{k=1}^{K} p(\mathbf{x}_n|\boldsymbol{\phi}_k)^{z_{nk}}. \tag{13.9}$$

We shall focuss attention on *homogeneous* models for which all of the conditional distributions governing the latent variables share the same parameters $\mathbf{A}$, and similarly all of the emission distributions share the same parameters $\boldsymbol{\phi}$ (the extension to more general cases is straightforward). Note that a mixture model for an i.i.d. data set corresponds to the special case in which the parameters A_{jk} are the same for all values of j, so that the conditional distribution $p(\mathbf{z}_n|\mathbf{z}_{n-1})$ is independent of $\mathbf{z}_{n-1}$. This corresponds to deleting the horizontal links in the graphical model shown in Figure 13.5.

The joint probability distribution over both latent and observed variables is then given by

$$p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) = p(\mathbf{z}_1|\boldsymbol{\pi}) \left[\prod_{n=2}^{N} p(\mathbf{z}_n|\mathbf{z}_{n-1}, \mathbf{A})\right] \prod_{m=1}^{N} p(\mathbf{x}_m|\mathbf{z}_m, \boldsymbol{\phi}) \tag{13.10}$$

where $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, $\mathbf{Z} = \{\mathbf{z}_1, \ldots, \mathbf{z}_N\}$, and $\boldsymbol{\theta} = \{\boldsymbol{\pi}, \mathbf{A}, \boldsymbol{\phi}\}$ denotes the set of parameters governing the model. Most of our discussion of the hidden Markov model will be independent of the particular choice of the emission probabilities. Indeed, the model is tractable for a wide range of emission distributions including discrete tables, Gaussians, and mixtures of Gaussians. It is also possible to exploit discriminative models such as neural networks. These can be used to model the emission density $p(\mathbf{x}|\mathbf{z})$ directly, or to provide a representation for $p(\mathbf{z}|\mathbf{x})$ that can be converted into the required emission density $p(\mathbf{x}|\mathbf{z})$ using Bayes' theorem (Bishop *et al.*, 2004).

Exercise 13.4

We can gain a better understanding of the hidden Markov model by considering it from a generative point of view. Recall that to generate samples from a mixture of

Figure 13.8 Illustration of sampling from a hidden Markov model having a 3-state latent variable $\mathbf{z}$ and a Gaussian emission model $p(\mathbf{x}|\mathbf{z})$ where $\mathbf{x}$ is 2-dimensional. (a) Contours of constant probability density for the emission distributions corresponding to each of the three states of the latent variable. (b) A sample of 50 points drawn from the hidden Markov model, colour coded according to the component that generated them and with lines connecting the successive observations. Here the transition matrix was fixed so that in any state there is a 5% probability of making a transition to each of the other states, and consequently a 90% probability of remaining in the same state.

Gaussians, we first chose one of the components at random with probability given by the mixing coefficients π_k and then generate a sample vector $\mathbf{x}$ from the corresponding Gaussian component. This process is repeated N times to generate a data set of N independent samples. In the case of the hidden Markov model, this procedure is modified as follows. We first choose the initial latent variable $\mathbf{z}_1$ with probabilities governed by the parameters π_k and then sample the corresponding observation $\mathbf{x}_1$. Now we choose the state of the variable $\mathbf{z}_2$ according to the transition probabilities $p(\mathbf{z}_2|\mathbf{z}_1)$ using the already instantiated value of $\mathbf{z}_1$. Thus suppose that the sample for $\mathbf{z}_1$ corresponds to state j. Then we choose the state k of $\mathbf{z}_2$ with probabilities A_{jk} for $k = 1, \ldots, K$. Once we know $\mathbf{z}_2$ we can draw a sample for $\mathbf{x}_2$ and also sample the next latent variable $\mathbf{z}_3$ and so on. This is an example of ancestral sampling for a directed graphical model. If, for instance, we have a model in which the diagonal transition elements A_{kk} are much larger than the off-diagonal elements, then a typical data sequence will have long runs of points generated from a single component, with infrequent transitions from one component to another. The generation of samples from a hidden Markov model is illustrated in Figure 13.8.

Section 8.1.2

There are many variants of the standard HMM model, obtained for instance by imposing constraints on the form of the transition matrix $\mathbf{A}$ (Rabiner, 1989). Here we mention one of particular practical importance called the *left-to-right* HMM, which is obtained by setting the elements A_{jk} of $\mathbf{A}$ to zero if $k < j$, as illustrated in the

Figure 13.9 Example of the state transition diagram for a 3-state left-to-right hidden Markov model. Note that once a state has been vacated, it cannot later be re-entered.

state transition diagram for a 3-state HMM in Figure 13.9. Typically for such models the initial state probabilities for $p(\mathbf{z}_1)$ are modified so that $p(z_{11}) = 1$ and $p(z_{1j}) = 0$ for $j \neq 1$, in other words every sequence is constrained to start in state $j = 1$. The transition matrix may be further constrained to ensure that large changes in the state index do not occur, so that $A_{jk} = 0$ if $k > j + \Delta$. This type of model is illustrated using a lattice diagram in Figure 13.10.

Many applications of hidden Markov models, for example speech recognition, or on-line character recognition, make use of left-to-right architectures. As an illustration of the left-to-right hidden Markov model, we consider an example involving handwritten digits. This uses on-line data, meaning that each digit is represented by the trajectory of the pen as a function of time in the form of a sequence of pen coordinates, in contrast to the off-line digits data, discussed in Appendix A, which comprises static two-dimensional pixellated images of the ink. Examples of the on-line digits are shown in Figure 13.11. Here we train a hidden Markov model on a subset of data comprising 45 examples of the digit '2'. There are $K = 16$ states, each of which can generate a line segment of fixed length having one of 16 possible angles, and so the emission distribution is simply a 16×16 table of probabilities associated with the allowed angle values for each state index value. Transition probabilities are all set to zero except for those that keep the state index k the same or that increment it by 1, and the model parameters are optimized using 25 iterations of EM. We can gain some insight into the resulting model by running it generatively, as shown in Figure 13.11.

Figure 13.10 Lattice diagram for a 3-state left-to-right HMM in which the state index k is allowed to increase by at most 1 at each transition.

Figure 13.11 Top row: examples of on-line handwritten digits. Bottom row: synthetic digits sampled generatively from a left-to-right hidden Markov model that has been trained on a data set of 45 handwritten digits.

One of the most powerful properties of hidden Markov models is their ability to exhibit some degree of invariance to local warping (compression and stretching) of the time axis. To understand this, consider the way in which the digit '2' is written in the on-line handwritten digits example. A typical digit comprises two distinct sections joined at a cusp. The first part of the digit, which starts at the top left, has a sweeping arc down to the cusp or loop at the bottom left, followed by a second more-or-less straight sweep ending at the bottom right. Natural variations in writing style will cause the relative sizes of the two sections to vary, and hence the location of the cusp or loop within the temporal sequence will vary. From a generative perspective such variations can be accommodated by the hidden Markov model through changes in the number of transitions to the same state versus the number of transitions to the successive state. Note, however, that if a digit '2' is written in the reverse order, that is, starting at the bottom right and ending at the top left, then even though the pen tip coordinates may be identical to an example from the training set, the probability of the observations under the model will be extremely small. In the speech recognition context, warping of the time axis is associated with natural variations in the speed of speech, and again the hidden Markov model can accommodate such a distortion and not penalize it too heavily.

13.2.1 Maximum likelihood for the HMM

If we have observed a data set $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, we can determine the parameters of an HMM using maximum likelihood. The likelihood function is obtained from the joint distribution (13.10) by marginalizing over the latent variables

$$p(\mathbf{X}|\boldsymbol{\theta}) = \sum_{\mathbf{Z}} p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}). \tag{13.11}$$

Because the joint distribution $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ does not factorize over n (in contrast to the mixture distribution considered in Chapter 9), we cannot simply treat each of the summations over $\mathbf{z}_n$ independently. Nor can we perform the summations explicitly because there are N variables to be summed over, each of which has K states, resulting in a total of K^N terms. Thus the number of terms in the summation grows

exponentially with the length of the chain. In fact, the summation in (13.11) corresponds to summing over exponentially many paths through the lattice diagram in Figure 13.7.

We have already encountered a similar difficulty when we considered the inference problem for the simple chain of variables in Figure 8.32. There we were able to make use of the conditional independence properties of the graph to re-order the summations in order to obtain an algorithm whose cost scales linearly, instead of exponentially, with the length of the chain. We shall apply a similar technique to the hidden Markov model.

A further difficulty with the expression (13.11) for the likelihood function is that, because it corresponds to a generalization of a mixture distribution, it represents a summation over the emission models for different settings of the latent variables. Direct maximization of the likelihood function will therefore lead to complex expressions with no closed-form solutions, as was the case for simple mixture models (recall that a mixture model for i.i.d. data is a special case of the HMM).

Section 9.2

We therefore turn to the expectation maximization algorithm to find an efficient framework for maximizing the likelihood function in hidden Markov models. The EM algorithm starts with some initial selection for the model parameters, which we denote by $\boldsymbol{\theta}^{\text{old}}$. In the E step, we take these parameter values and find the posterior distribution of the latent variables $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}})$. We then use this posterior distribution to evaluate the expectation of the logarithm of the complete-data likelihood function, as a function of the parameters $\boldsymbol{\theta}$, to give the function $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$ defined by

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) = \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}) \ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}). \tag{13.12}$$

At this point, it is convenient to introduce some notation. We shall use $\gamma(\mathbf{z}_n)$ to denote the marginal posterior distribution of a latent variable $\mathbf{z}_n$, and $\xi(\mathbf{z}_{n-1}, \mathbf{z}_n)$ to denote the joint posterior distribution of two successive latent variables, so that

$$\gamma(\mathbf{z}_n) = p(\mathbf{z}_n|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}) \tag{13.13}$$

$$\xi(\mathbf{z}_{n-1}, \mathbf{z}_n) = p(\mathbf{z}_{n-1}, \mathbf{z}_n|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}). \tag{13.14}$$

For each value of n, we can store $\gamma(\mathbf{z}_n)$ using a set of K nonnegative numbers that sum to unity, and similarly we can store $\xi(\mathbf{z}_{n-1}, \mathbf{z}_n)$ using a $K \times K$ matrix of nonnegative numbers that again sum to unity. We shall also use $\gamma(z_{nk})$ to denote the conditional probability of $z_{nk} = 1$, with a similar use of notation for $\xi(z_{n-1,j}, z_{nk})$ and for other probabilistic variables introduced later. Because the expectation of a binary random variable is just the probability that it takes the value 1, we have

$$\gamma(z_{nk}) = \mathbb{E}[z_{nk}] = \sum_{\mathbf{z}} \gamma(\mathbf{z}) z_{nk} \tag{13.15}$$

$$\xi(z_{n-1,j}, z_{nk}) = \mathbb{E}[z_{n-1,j} z_{nk}] = \sum_{\mathbf{z}} \gamma(\mathbf{z}) z_{n-1,j} z_{nk}. \tag{13.16}$$

If we substitute the joint distribution $p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})$ given by (13.10) into (13.12),

and make use of the definitions of γ and ξ , we obtain

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) = \sum_{k=1}^{K} \gamma(z_{1k}) \ln \pi_k + \sum_{n=2}^{N} \sum_{j=1}^{K} \sum_{k=1}^{K} \xi(z_{n-1,j}, z_{nk}) \ln A_{jk} + \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma(z_{nk}) \ln p(\mathbf{x}_n|\boldsymbol{\phi}_k). \tag{13.17}$$

The goal of the E step will be to evaluate the quantities $\gamma(\mathbf{z}_n)$ and $\xi(\mathbf{z}_{n-1}, \mathbf{z}_n)$ efficiently, and we shall discuss this in detail shortly.

In the M step, we maximize $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$ with respect to the parameters $\boldsymbol{\theta} = \{\boldsymbol{\pi}, \mathbf{A}, \boldsymbol{\phi}\}$ in which we treat $\gamma(\mathbf{z}_n)$ and $\xi(\mathbf{z}_{n-1}, \mathbf{z}_n)$ as constant. Maximization with respect to $\boldsymbol{\pi}$ and $\mathbf{A}$ is easily achieved using appropriate Lagrange multipliers with the results

Exercise 13.5

$$\pi_k = \frac{\gamma(z_{1k})}{\sum_{j=1}^{K} \gamma(z_{1j})} \tag{13.18}$$

$$A_{jk} = \frac{\sum_{n=2}^{N} \xi(z_{n-1,j}, z_{nk})}{\sum_{l=1}^{K} \sum_{n=2}^{N} \xi(z_{n-1,j}, z_{nl})}. \tag{13.19}$$

The EM algorithm must be initialized by choosing starting values for $\boldsymbol{\pi}$ and $\mathbf{A}$, which should of course respect the summation constraints associated with their probabilistic interpretation. Note that any elements of $\boldsymbol{\pi}$ or $\mathbf{A}$ that are set to zero initially will remain zero in subsequent EM updates. A typical initialization procedure would involve selecting random starting values for these parameters subject to the summation and non-negativity constraints. Note that no particular modification to the EM results are required for the case of left-to-right models beyond choosing initial values for the elements A_{jk} in which the appropriate elements are set to zero, because these will remain zero throughout.

Exercise 13.6

To maximize $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$ with respect to $\boldsymbol{\phi}_k$, we notice that only the final term in (13.17) depends on $\boldsymbol{\phi}_k$, and furthermore this term has exactly the same form as the data-dependent term in the corresponding function for a standard mixture distribution for i.i.d. data, as can be seen by comparison with (9.40) for the case of a Gaussian mixture. Here the quantities $\gamma(z_{nk})$ are playing the role of the responsibilities. If the parameters $\boldsymbol{\phi}_k$ are independent for the different components, then this term decouples into a sum of terms one for each value of k, each of which can be maximized independently. We are then simply maximizing the weighted log likelihood function for the emission density $p(\mathbf{x}|\boldsymbol{\phi}_k)$ with weights $\gamma(z_{nk})$. Here we shall suppose that this maximization can be done efficiently. For instance, in the case of

Gaussian emission densities we have $p(\mathbf{x}|\boldsymbol{\phi}_k) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$, and maximization of the function $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$ then gives

$$\boldsymbol{\mu}_k = \frac{\sum_{n=1}^{N} \gamma(z_{nk})\mathbf{x}_n}{\sum_{n=1}^{N} \gamma(z_{nk})} \tag{13.20}$$

$$\boldsymbol{\Sigma}_k = \frac{\sum_{n=1}^{N} \gamma(z_{nk})(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}}}{\sum_{n=1}^{N} \gamma(z_{nk})}. \tag{13.21}$$

For the case of discrete multinomial observed variables, the conditional distribution of the observations takes the form

$$p(\mathbf{x}|\mathbf{z}) = \prod_{i=1}^{D} \prod_{k=1}^{K} \mu_{ik}^{x_i z_k} \tag{13.22}$$

Exercise 13.8 and the corresponding M-step equations are given by

$$\mu_{ik} = \frac{\sum_{n=1}^{N} \gamma(z_{nk}) x_{ni}}{\sum_{n=1}^{N} \gamma(z_{nk})}. \tag{13.23}$$

An analogous result holds for Bernoulli observed variables.

The EM algorithm requires initial values for the parameters of the emission distribution. One way to set these is first to treat the data initially as i.i.d. and fit the emission density by maximum likelihood, and then use the resulting values to initialize the parameters for EM.

13.2.2 The forward-backward algorithm

Next we seek an efficient procedure for evaluating the quantities $\gamma(z_{nk})$ and $\xi(z_{n-1,j}, z_{nk})$, corresponding to the E step of the EM algorithm. The graph for the hidden Markov model, shown in Figure 13.5, is a tree, and so we know that the posterior distribution of the latent variables can be obtained efficiently using a two-stage message passing algorithm. In the particular context of the hidden Markov model, this is known as the *forward-backward* algorithm (Rabiner, 1989), or the *Baum-Welch* algorithm (Baum, 1972). There are in fact several variants of the basic algorithm, all of which lead to the exact marginals, according to the precise form of

Section 8.4

the messages that are propagated along the chain (Jordan, 2007). We shall focus on the most widely used of these, known as the alpha-beta algorithm.

As well as being of great practical importance in its own right, the forward-backward algorithm provides us with a nice illustration of many of the concepts introduced in earlier chapters. We shall therefore begin in this section with a 'conventional' derivation of the forward-backward equations, making use of the sum and product rules of probability, and exploiting conditional independence properties which we shall obtain from the corresponding graphical model using d-separation. Then in Section 13.2.3, we shall see how the forward-backward algorithm can be obtained very simply as a specific example of the sum-product algorithm introduced in Section 8.4.4.

It is worth emphasizing that evaluation of the posterior distributions of the latent variables is independent of the form of the emission density $p(\mathbf{x}|\mathbf{z})$ or indeed of whether the observed variables are continuous or discrete. All we require is the values of the quantities $p(\mathbf{x}_n|\mathbf{z}_n)$ for each value of $\mathbf{z}_n$ for every n. Also, in this section and the next we shall omit the explicit dependence on the model parameters $\boldsymbol{\theta}^{\text{old}}$ because these fixed throughout.

We therefore begin by writing down the following conditional independence properties (Jordan, 2007)

$$
\begin{aligned}
p(\mathbf{X}|\mathbf{z}_n) &= p(\mathbf{x}_1,\ldots,\mathbf{x}_n|\mathbf{z}_n) \\
&\quad\; p(\mathbf{x}_{n+1},\ldots,\mathbf{x}_N|\mathbf{z}_n) && (13.24)\\
p(\mathbf{x}_1,\ldots,\mathbf{x}_{n-1}|\mathbf{x}_n,\mathbf{z}_n) &= p(\mathbf{x}_1,\ldots,\mathbf{x}_{n-1}|\mathbf{z}_n) && (13.25)\\
p(\mathbf{x}_1,\ldots,\mathbf{x}_{n-1}|\mathbf{z}_{n-1},\mathbf{z}_n) &= p(\mathbf{x}_1,\ldots,\mathbf{x}_{n-1}|\mathbf{z}_{n-1}) && (13.26)\\
p(\mathbf{x}_{n+1},\ldots,\mathbf{x}_N|\mathbf{z}_n,\mathbf{z}_{n+1}) &= p(\mathbf{x}_{n+1},\ldots,\mathbf{x}_N|\mathbf{z}_{n+1}) && (13.27)\\
p(\mathbf{x}_{n+2},\ldots,\mathbf{x}_N|\mathbf{z}_{n+1},\mathbf{x}_{n+1}) &= p(\mathbf{x}_{n+2},\ldots,\mathbf{x}_N|\mathbf{z}_{n+1}) && (13.28)\\
p(\mathbf{X}|\mathbf{z}_{n-1},\mathbf{z}_n) &= p(\mathbf{x}_1,\ldots,\mathbf{x}_{n-1}|\mathbf{z}_{n-1}) \\
&\quad\; p(\mathbf{x}_n|\mathbf{z}_n)p(\mathbf{x}_{n+1},\ldots,\mathbf{x}_N|\mathbf{z}_n) && (13.29)\\
p(\mathbf{x}_{N+1}|\mathbf{X},\mathbf{z}_{N+1}) &= p(\mathbf{x}_{N+1}|\mathbf{z}_{N+1}) && (13.30)\\
p(\mathbf{z}_{N+1}|\mathbf{z}_N,\mathbf{X}) &= p(\mathbf{z}_{N+1}|\mathbf{z}_N) && (13.31)
\end{aligned}
$$

where $\mathbf{X} = \{\mathbf{x}_1,\ldots,\mathbf{x}_N\}$. These relations are most easily proved using d-separation. For instance in the first of these results, we note that every path from any one of the nodes $\mathbf{x}_1,\ldots,\mathbf{x}_{n-1}$ to the node $\mathbf{x}_n$ passes through the node $\mathbf{z}_n$, which is observed. Because all such paths are head-to-tail, it follows that the conditional independence property must hold. The reader should take a few moments to verify each of these properties in turn, as an exercise in the application of d-separation. These relations can also be proved directly, though with significantly greater effort, from the joint distribution for the hidden Markov model using the sum and product rules of probability.

Exercise 13.10

Let us begin by evaluating $\gamma(z_{nk})$. Recall that for a discrete multinomial random variable the expected value of one of its components is just the probability of that component having the value 1. Thus we are interested in finding the posterior distribution $p(\mathbf{z}_n|\mathbf{x}_1,\ldots,\mathbf{x}_N)$ of $\mathbf{z}_n$ given the observed data set $\mathbf{x}_1,\ldots,\mathbf{x}_N$. This

represents a vector of length K whose entries correspond to the expected values of z_{nk}. Using Bayes' theorem, we have

$$\gamma(\mathbf{z}_n) = p(\mathbf{z}_n|\mathbf{X}) = \frac{p(\mathbf{X}|\mathbf{z}_n)p(\mathbf{z}_n)}{p(\mathbf{X})}. \tag{13.32}$$

Note that the denominator $p(\mathbf{X})$ is implicitly conditioned on the parameters $\boldsymbol{\theta}^{\text{old}}$ of the HMM and hence represents the likelihood function. Using the conditional independence property (13.24), together with the product rule of probability, we obtain

$$\gamma(\mathbf{z}_n) = \frac{p(\mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{z}_n)p(\mathbf{x}_{n+1}, \ldots, \mathbf{x}_N|\mathbf{z}_n)}{p(\mathbf{X})} = \frac{\alpha(\mathbf{z}_n)\beta(\mathbf{z}_n)}{p(\mathbf{X})} \tag{13.33}$$

where we have defined

$$\alpha(\mathbf{z}_n) \equiv p(\mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{z}_n) \tag{13.34}$$

$$\beta(\mathbf{z}_n) \equiv p(\mathbf{x}_{n+1}, \ldots, \mathbf{x}_N|\mathbf{z}_n). \tag{13.35}$$

The quantity $\alpha(\mathbf{z}_n)$ represents the joint probability of observing all of the given data up to time n and the value of $\mathbf{z}_n$, whereas $\beta(\mathbf{z}_n)$ represents the conditional probability of all future data from time $n+1$ up to N given the value of $\mathbf{z}_n$. Again, $\alpha(\mathbf{z}_n)$ and $\beta(\mathbf{z}_n)$ each represent set of K numbers, one for each of the possible settings of the 1-of-K coded binary vector $\mathbf{z}_n$. We shall use the notation $\alpha(z_{nk})$ to denote the value of $\alpha(\mathbf{z}_n)$ when $z_{nk} = 1$, with an analogous interpretation of $\beta(z_{nk})$.

We now derive recursion relations that allow $\alpha(\mathbf{z}_n)$ and $\beta(\mathbf{z}_n)$ to be evaluated efficiently. Again, we shall make use of conditional independence properties, in particular (13.25) and (13.26), together with the sum and product rules, allowing us to express $\alpha(\mathbf{z}_n)$ in terms of $\alpha(\mathbf{z}_{n-1})$ as follows

$$\begin{aligned}
\alpha(\mathbf{z}_n) &= p(\mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{z}_n) \\
&= p(\mathbf{x}_1, \ldots, \mathbf{x}_n|\mathbf{z}_n)p(\mathbf{z}_n) \\
&= p(\mathbf{x}_n|\mathbf{z}_n)p(\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}|\mathbf{z}_n)p(\mathbf{z}_n) \\
&= p(\mathbf{x}_n|\mathbf{z}_n)p(\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}, \mathbf{z}_n) \\
&= p(\mathbf{x}_n|\mathbf{z}_n)\sum_{\mathbf{z}_{n-1}} p(\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}, \mathbf{z}_{n-1}, \mathbf{z}_n) \\
&= p(\mathbf{x}_n|\mathbf{z}_n)\sum_{\mathbf{z}_{n-1}} p(\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}, \mathbf{z}_n|\mathbf{z}_{n-1})p(\mathbf{z}_{n-1}) \\
&= p(\mathbf{x}_n|\mathbf{z}_n)\sum_{\mathbf{z}_{n-1}} p(\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}|\mathbf{z}_{n-1})p(\mathbf{z}_n|\mathbf{z}_{n-1})p(\mathbf{z}_{n-1}) \\
&= p(\mathbf{x}_n|\mathbf{z}_n)\sum_{\mathbf{z}_{n-1}} p(\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}, \mathbf{z}_{n-1})p(\mathbf{z}_n|\mathbf{z}_{n-1})
\end{aligned}$$

Making use of the definition (13.34) for $\alpha(\mathbf{z}_n)$, we then obtain

$$\alpha(\mathbf{z}_n) = p(\mathbf{x}_n|\mathbf{z}_n)\sum_{\mathbf{z}_{n-1}} \alpha(\mathbf{z}_{n-1})p(\mathbf{z}_n|\mathbf{z}_{n-1}). \tag{13.36}$$

Figure 13.12 Illustration of the forward recursion (13.36) for evaluation of the α variables. In this fragment of the lattice, we see that the quantity $\alpha(z_{n1})$ is obtained by taking the elements $\alpha(z_{n-1,j})$ of $\alpha(\mathbf{z}_{n-1})$ at step $n-1$ and summing them up with weights given by A_{j1}, corresponding to the values of $p(\mathbf{z}_n|\mathbf{z}_{n-1})$, and then multiplying by the data contribution $p(\mathbf{x}_n|z_{n1})$.

It is worth taking a moment to study this recursion relation in some detail. Note that there are K terms in the summation, and the right-hand side has to be evaluated for each of the K values of $\mathbf{z}_n$ so each step of the α recursion has computational cost that scaled like $O(K^2)$. The forward recursion equation for $\alpha(\mathbf{z}_n)$ is illustrated using a lattice diagram in Figure 13.12.

In order to start this recursion, we need an initial condition that is given by

$$\alpha(\mathbf{z}_1) = p(\mathbf{x}_1, \mathbf{z}_1) = p(\mathbf{z}_1)p(\mathbf{x}_1|\mathbf{z}_1) = \prod_{k=1}^{K} \{\pi_k p(\mathbf{x}_1|\boldsymbol{\phi}_k)\}^{z_{1k}} \tag{13.37}$$

which tells us that $\alpha(z_{1k})$, for $k = 1, \ldots, K$, takes the value $\pi_k p(\mathbf{x}_1|\boldsymbol{\phi}_k)$. Starting at the first node of the chain, we can then work along the chain and evaluate $\alpha(\mathbf{z}_n)$ for every latent node. Because each step of the recursion involves multiplying by a $K \times K$ matrix, the overall cost of evaluating these quantities for the whole chain is of $O(K^2N)$.

We can similarly find a recursion relation for the quantities $\beta(\mathbf{z}_n)$ by making use of the conditional independence properties (13.27) and (13.28) giving

$$\begin{aligned}
\beta(\mathbf{z}_n) &= p(\mathbf{x}_{n+1}, \ldots, \mathbf{x}_N|\mathbf{z}_n) \\
&= \sum_{\mathbf{z}_{n+1}} p(\mathbf{x}_{n+1}, \ldots, \mathbf{x}_N, \mathbf{z}_{n+1}|\mathbf{z}_n) \\
&= \sum_{\mathbf{z}_{n+1}} p(\mathbf{x}_{n+1}, \ldots, \mathbf{x}_N|\mathbf{z}_n, \mathbf{z}_{n+1})p(\mathbf{z}_{n+1}|\mathbf{z}_n) \\
&= \sum_{\mathbf{z}_{n+1}} p(\mathbf{x}_{n+1}, \ldots, \mathbf{x}_N|\mathbf{z}_{n+1})p(\mathbf{z}_{n+1}|\mathbf{z}_n) \\
&= \sum_{\mathbf{z}_{n+1}} p(\mathbf{x}_{n+2}, \ldots, \mathbf{x}_N|\mathbf{z}_{n+1})p(\mathbf{x}_{n+1}|\mathbf{z}_{n+1})p(\mathbf{z}_{n+1}|\mathbf{z}_n).
\end{aligned}$$

Figure 13.13 Illustration of the backward recursion (13.38) for evaluation of the β variables. In this fragment of the lattice, we see that the quantity $\beta(z_{n1})$ is obtained by taking the components $\beta(z_{n+1,k})$ of $\beta(\mathbf{z}_{n+1})$ at step $n+1$ and summing them up with weights given by the products of A_{1k}, corresponding to the values of $p(\mathbf{z}_{n+1}|\mathbf{z}_n)$ and the corresponding values of the emission density $p(\mathbf{x}_n|z_{n+1,k})$.

Making use of the definition (13.35) for $\beta(\mathbf{z}_n)$, we then obtain

$$\beta(\mathbf{z}_n) = \sum_{\mathbf{z}_{n+1}} \beta(\mathbf{z}_{n+1}) p(\mathbf{x}_{n+1}|\mathbf{z}_{n+1}) p(\mathbf{z}_{n+1}|\mathbf{z}_n). \tag{13.38}$$

Note that in this case we have a backward message passing algorithm that evaluates $\beta(\mathbf{z}_n)$ in terms of $\beta(\mathbf{z}_{n+1})$. At each step, we absorb the effect of observation $\mathbf{x}_{n+1}$ through the emission probability $p(\mathbf{x}_{n+1}|\mathbf{z}_{n+1})$, multiply by the transition matrix $p(\mathbf{z}_{n+1}|\mathbf{z}_n)$, and then marginalize out $\mathbf{z}_{n+1}$. This is illustrated in Figure 13.13.

Again we need a starting condition for the recursion, namely a value for $\beta(\mathbf{z}_N)$. This can be obtained by setting $n = N$ in (13.33) and replacing $\alpha(\mathbf{z}_N)$ with its definition (13.34) to give

$$p(\mathbf{z}_N|\mathbf{X}) = \frac{p(\mathbf{X}, \mathbf{z}_N)\beta(\mathbf{z}_N)}{p(\mathbf{X})} \tag{13.39}$$

which we see will be correct provided we take $\beta(\mathbf{z}_N) = 1$ for all settings of $\mathbf{z}_N$.

In the M step equations, the quantity $p(\mathbf{X})$ will cancel out, as can be seen, for instance, in the M-step equation for $\boldsymbol{\mu}_k$ given by (13.20), which takes the form

$$\boldsymbol{\mu}_k = \frac{\sum_{n=1}^{n} \gamma(z_{nk})\mathbf{x}_n}{\sum_{n=1}^{n} \gamma(z_{nk})} = \frac{\sum_{n=1}^{n} \alpha(z_{nk})\beta(z_{nk})\mathbf{x}_n}{\sum_{n=1}^{n} \alpha(z_{nk})\beta(z_{nk})}. \tag{13.40}$$

However, the quantity $p(\mathbf{X})$ represents the likelihood function whose value we typically wish to monitor during the EM optimization, and so it is useful to be able to evaluate it. If we sum both sides of (13.33) over $\mathbf{z}_n$, and use the fact that the left-hand side is a normalized distribution, we obtain

$$p(\mathbf{X}) = \sum_{\mathbf{z}_n} \alpha(\mathbf{z}_n)\beta(\mathbf{z}_n). \tag{13.41}$$

Thus we can evaluate the likelihood function by computing this sum, for any convenient choice of n. For instance, if we only want to evaluate the likelihood function, then we can do this by running the α recursion from the start to the end of the chain, and then use this result for $n = N$, making use of the fact that $\beta(\mathbf{z}_N)$ is a vector of 1s. In this case no β recursion is required, and we simply have

$$p(\mathbf{X}) = \sum_{\mathbf{z}_N} \alpha(\mathbf{z}_N). \tag{13.42}$$

Let us take a moment to interpret this result for $p(\mathbf{X})$. Recall that to compute the likelihood we should take the joint distribution $p(\mathbf{X}, \mathbf{Z})$ and sum over all possible values of $\mathbf{Z}$. Each such value represents a particular choice of hidden state for every time step, in other words every term in the summation is a path through the lattice diagram, and recall that there are exponentially many such paths. By expressing the likelihood function in the form (13.42), we have reduced the computational cost from being exponential in the length of the chain to being linear by swapping the order of the summation and multiplications, so that at each time step n we sum the contributions from all paths passing through each of the states z_{nk} to give the intermediate quantities $\alpha(\mathbf{z}_n)$.

Next we consider the evaluation of the quantities $\xi(\mathbf{z}_{n-1}, \mathbf{z}_n)$, which correspond to the values of the conditional probabilities $p(\mathbf{z}_{n-1}, \mathbf{z}_n|\mathbf{X})$ for each of the $K \times K$ settings for $(\mathbf{z}_{n-1}, \mathbf{z}_n)$. Using the definition of $\xi(\mathbf{z}_{n-1}, \mathbf{z}_n)$, and applying Bayes' theorem, we have

$$\begin{aligned}
\xi(\mathbf{z}_{n-1}, \mathbf{z}_n) &= p(\mathbf{z}_{n-1}, \mathbf{z}_n|\mathbf{X}) \\
&= \frac{p(\mathbf{X}|\mathbf{z}_{n-1}, \mathbf{z}_n)p(\mathbf{z}_{n-1}, \mathbf{z}_n)}{p(\mathbf{X})} \\
&= \frac{p(\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}|\mathbf{z}_{n-1})p(\mathbf{x}_n|\mathbf{z}_n)p(\mathbf{x}_{n+1}, \ldots, \mathbf{x}_N|\mathbf{z}_n)p(\mathbf{z}_n|\mathbf{z}_{n-1})p(\mathbf{z}_{n-1})}{p(\mathbf{X})} \\
&= \frac{\alpha(\mathbf{z}_{n-1})p(\mathbf{x}_n|\mathbf{z}_n)p(\mathbf{z}_n|\mathbf{z}_{n-1})\beta(\mathbf{z}_n)}{p(\mathbf{X})}
\end{aligned} \tag{13.43}$$

where we have made use of the conditional independence property (13.29) together with the definitions of $\alpha(\mathbf{z}_n)$ and $\beta(\mathbf{z}_n)$ given by (13.34) and (13.35). Thus we can calculate the $\xi(\mathbf{z}_{n-1}, \mathbf{z}_n)$ directly by using the results of the α and β recursions.

Let us summarize the steps required to train a hidden Markov model using the EM algorithm. We first make an initial selection of the parameters $\boldsymbol{\theta}^{\text{old}}$ where $\boldsymbol{\theta} \equiv (\boldsymbol{\pi}, \mathbf{A}, \boldsymbol{\phi})$. The $\mathbf{A}$ and $\boldsymbol{\pi}$ parameters are often initialized either uniformly or randomly from a uniform distribution (respecting their non-negativity and summation constraints). Initialization of the parameters $\boldsymbol{\phi}$ will depend on the form of the distribution. For instance in the case of Gaussians, the parameters $\boldsymbol{\mu}_k$ might be initialized by applying the K-means algorithm to the data, and $\boldsymbol{\Sigma}_k$ might be initialized to the covariance matrix of the corresponding K means cluster. Then we run both the forward α recursion and the backward β recursion and use the results to evaluate $\gamma(\mathbf{z}_n)$ and $\xi(\mathbf{z}_{n-1}, \mathbf{z}_n)$. At this stage, we can also evaluate the likelihood function.

This completes the E step, and we use the results to find a revised set of parameters $\boldsymbol{\theta}^{\text{new}}$ using the M-step equations from Section 13.2.1. We then continue to alternate between E and M steps until some convergence criterion is satisfied, for instance when the change in the likelihood function is below some threshold.

Note that in these recursion relations the observations enter through conditional distributions of the form $p(\mathbf{x}_n|\mathbf{z}_n)$. The recursions are therefore independent of the type or dimensionality of the observed variables or the form of this conditional distribution, so long as its value can be computed for each of the K possible states of $\mathbf{z}_n$. Since the observed variables $\{\mathbf{x}_n\}$ are fixed, the quantities $p(\mathbf{x}_n|\mathbf{z}_n)$ can be pre-computed as functions of $\mathbf{z}_n$ at the start of the EM algorithm, and remain fixed throughout.

We have seen in earlier chapters that the maximum likelihood approach is most effective when the number of data points is large in relation to the number of parameters. Here we note that a hidden Markov model can be trained effectively, using maximum likelihood, provided the training sequence is sufficiently long. Alternatively, we can make use of multiple shorter sequences, which requires a straightforward modification of the hidden Markov model EM algorithm. In the case of left-to-right models, this is particularly important because, in a given observation sequence, a given state transition corresponding to a nondiagonal element of $\mathbf{A}$ will seen at most once.

Exercise 13.12

Another quantity of interest is the predictive distribution, in which the observed data is $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ and we wish to predict $\mathbf{x}_{N+1}$, which would be important for real-time applications such as financial forecasting. Again we make use of the sum and product rules together with the conditional independence properties (13.29) and (13.31) giving

$$
\begin{aligned}
p(\mathbf{x}_{N+1}|\mathbf{X}) &= \sum_{\mathbf{z}_{N+1}} p(\mathbf{x}_{N+1}, \mathbf{z}_{N+1}|\mathbf{X}) \\
&= \sum_{\mathbf{z}_{N+1}} p(\mathbf{x}_{N+1}|\mathbf{z}_{N+1}) p(\mathbf{z}_{N+1}|\mathbf{X}) \\
&= \sum_{\mathbf{z}_{N+1}} p(\mathbf{x}_{N+1}|\mathbf{z}_{N+1}) \sum_{\mathbf{z}_N} p(\mathbf{z}_{N+1}, \mathbf{z}_N|\mathbf{X}) \\
&= \sum_{\mathbf{z}_{N+1}} p(\mathbf{x}_{N+1}|\mathbf{z}_{N+1}) \sum_{\mathbf{z}_N} p(\mathbf{z}_{N+1}|\mathbf{z}_N) p(\mathbf{z}_N|\mathbf{X}) \\
&= \sum_{\mathbf{z}_{N+1}} p(\mathbf{x}_{N+1}|\mathbf{z}_{N+1}) \sum_{\mathbf{z}_N} p(\mathbf{z}_{N+1}|\mathbf{z}_N) \frac{p(\mathbf{z}_N, \mathbf{X})}{p(\mathbf{X})} \\
&= \frac{1}{p(\mathbf{X})} \sum_{\mathbf{z}_{N+1}} p(\mathbf{x}_{N+1}|\mathbf{z}_{N+1}) \sum_{\mathbf{z}_N} p(\mathbf{z}_{N+1}|\mathbf{z}_N) \alpha(\mathbf{z}_N) \qquad (13.44)
\end{aligned}
$$

which can be evaluated by first running a forward α recursion and then computing the final summations over $\mathbf{z}_N$ and $\mathbf{z}_{N+1}$. The result of the first summation over $\mathbf{z}_N$ can be stored and used once the value of $\mathbf{x}_{N+1}$ is observed in order to run the α recursion forward to the next step in order to predict the subsequent value $\mathbf{x}_{N+2}$.

Figure 13.14 A fragment of the factor graph representation for the hidden Markov model.

Note that in (13.44), the influence of all data from $\mathbf{x}_1$ to $\mathbf{x}_N$ is summarized in the K values of $\alpha(\mathbf{z}_N)$. Thus the predictive distribution can be carried forward indefinitely using a fixed amount of storage, as may be required for real-time applications.

Here we have discussed the estimation of the parameters of an HMM using maximum likelihood. This framework is easily extended to regularized maximum likelihood by introducing priors over the model parameters $\boldsymbol{\pi}$, $\mathbf{A}$ and $\boldsymbol{\phi}$ whose values are then estimated by maximizing their posterior probability. This can again be done using the EM algorithm in which the E step is the same as discussed above, and the M step involves adding the log of the prior distribution $p(\boldsymbol{\theta})$ to the function $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$ before maximization and represents a straightforward application of the techniques developed at various points in this book. Furthermore, we can use variational methods to give a fully Bayesian treatment of the HMM in which we marginalize over the parameter distributions (MacKay, 1997). As with maximum likelihood, this leads to a two-pass forward-backward recursion to compute posterior probabilities.

Section 10.1

13.2.3 The sum-product algorithm for the HMM

The directed graph that represents the hidden Markov model, shown in Figure 13.5, is a tree and so we can solve the problem of finding local marginals for the hidden variables using the sum-product algorithm. Not surprisingly, this turns out to be equivalent to the forward-backward algorithm considered in the previous section, and so the sum-product algorithm therefore provides us with a simple way to derive the alpha-beta recursion formulae.

Section 8.4.4

We begin by transforming the directed graph of Figure 13.5 into a factor graph, of which a representative fragment is shown in Figure 13.14. This form of the factor graph shows all variables, both latent and observed, explicitly. However, for the purpose of solving the inference problem, we shall always be conditioning on the variables $\mathbf{x}_1, \ldots, \mathbf{x}_N$, and so we can simplify the factor graph by absorbing the emission probabilities into the transition probability factors. This leads to the simplified factor graph representation in Figure 13.15, in which the factors are given by

$$h(\mathbf{z}_1) = p(\mathbf{z}_1)p(\mathbf{x}_1|\mathbf{z}_1) \tag{13.45}$$

$$f_n(\mathbf{z}_{n-1}, \mathbf{z}_n) = p(\mathbf{z}_n|\mathbf{z}_{n-1})p(\mathbf{x}_n|\mathbf{z}_n). \tag{13.46}$$

Figure 13.15 A simplified form of factor graph to describe the hidden Markov model.

To derive the alpha-beta algorithm, we denote the final hidden variable $\mathbf{z}_N$ as the root node, and first pass messages from the leaf node h to the root. From the general results (8.66) and (8.69) for message propagation, we see that the messages which are propagated in the hidden Markov model take the form

$$\mu_{\mathbf{z}_{n-1}\to f_n}(\mathbf{z}_{n-1}) = \mu_{f_{n-1}\to\mathbf{z}_{n-1}}(\mathbf{z}_{n-1}) \tag{13.47}$$

$$\mu_{f_n\to\mathbf{z}_n}(\mathbf{z}_n) = \sum_{\mathbf{z}_{n-1}} f_n(\mathbf{z}_{n-1},\mathbf{z}_n)\mu_{\mathbf{z}_{n-1}\to f_n}(\mathbf{z}_{n-1}) \tag{13.48}$$

These equations represent the propagation of messages forward along the chain and are equivalent to the alpha recursions derived in the previous section, as we shall now show. Note that because the variable nodes $\mathbf{z}_n$ have only two neighbours, they perform no computation.

We can eliminate $\mu_{\mathbf{z}_{n-1}\to f_n}(\mathbf{z}_{n-1})$ from (13.48) using (13.47) to give a recursion for the $f \to \mathbf{z}$ messages of the form

$$\mu_{f_n\to\mathbf{z}_n}(\mathbf{z}_n) = \sum_{\mathbf{z}_{n-1}} f_n(\mathbf{z}_{n-1},\mathbf{z}_n)\mu_{f_{n-1}\to\mathbf{z}_{n-1}}(\mathbf{z}_{n-1}). \tag{13.49}$$

If we now recall the definition (13.46), and if we define

$$\alpha(\mathbf{z}_n) = \mu_{f_n\to\mathbf{z}_n}(\mathbf{z}_n) \tag{13.50}$$

then we obtain the alpha recursion given by (13.36). We also need to verify that the quantities $\alpha(\mathbf{z}_n)$ are themselves equivalent to those defined previously. This is easily done by using the initial condition (8.71) and noting that $\alpha(\mathbf{z}_1)$ is given by $h(\mathbf{z}_1) = p(\mathbf{z}_1)p(\mathbf{x}_1|\mathbf{z}_1)$ which is identical to (13.37). Because the initial α is the same, and because they are iteratively computed using the same equation, all subsequent α quantities must be the same.

Next we consider the messages that are propagated from the root node back to the leaf node. These take the form

$$\mu_{f_{n+1}\to f_n}(\mathbf{z}_n) = \sum_{\mathbf{z}_{n+1}} f_{n+1}(\mathbf{z}_n,\mathbf{z}_{n+1})\mu_{f_{n+2}\to f_{n+1}}(\mathbf{z}_{n+1}) \tag{13.51}$$

where, as before, we have eliminated the messages of the type $\mathbf{z} \to f$ since the variable nodes perform no computation. Using the definition (13.46) to substitute for $f_{n+1}(\mathbf{z}_n,\mathbf{z}_{n+1})$, and defining

$$\beta(\mathbf{z}_n) = \mu_{f_{n+1}\to\mathbf{z}_n}(\mathbf{z}_n) \tag{13.52}$$

we obtain the beta recursion given by (13.38). Again, we can verify that the beta variables themselves are equivalent by noting that (8.70) implies that the initial message send by the root variable node is $\mu_{\mathbf{z}_N \to f_N}(\mathbf{z}_N) = 1$, which is identical to the initialization of $\beta(\mathbf{z}_N)$ given in Section 13.2.2.

The sum-product algorithm also specifies how to evaluate the marginals once all the messages have been evaluated. In particular, the result (8.63) shows that the local marginal at the node $\mathbf{z}_n$ is given by the product of the incoming messages. Because we have conditioned on the variables $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, we are computing the joint distribution

$$p(\mathbf{z}_n, \mathbf{X}) = \mu_{f_n \to \mathbf{z}_n}(\mathbf{z}_n)\mu_{f_{n+1} \to \mathbf{z}_n}(\mathbf{z}_n) = \alpha(\mathbf{z}_n)\beta(\mathbf{z}_n). \tag{13.53}$$

Dividing both sides by $p(\mathbf{X})$, we then obtain

$$\gamma(\mathbf{z}_n) = \frac{p(\mathbf{z}_n, \mathbf{X})}{p(\mathbf{X})} = \frac{\alpha(\mathbf{z}_n)\beta(\mathbf{z}_n)}{p(\mathbf{X})} \tag{13.54}$$

Exercise 13.11 in agreement with (13.33). The result (13.43) can similarly be derived from (8.72).

13.2.4 Scaling factors

There is an important issue that must be addressed before we can make use of the forward backward algorithm in practice. From the recursion relation (13.36), we note that at each step the new value $\alpha(\mathbf{z}_n)$ is obtained from the previous value $\alpha(\mathbf{z}_{n-1})$ by multiplying by quantities $p(\mathbf{z}_n|\mathbf{z}_{n-1})$ and $p(\mathbf{x}_n|\mathbf{z}_n)$. Because these probabilities are often significantly less than unity, as we work our way forward along the chain, the values of $\alpha(\mathbf{z}_n)$ can go to zero exponentially quickly. For moderate lengths of chain (say 100 or so), the calculation of the $\alpha(\mathbf{z}_n)$ will soon exceed the dynamic range of the computer, even if double precision floating point is used.

In the case of i.i.d. data, we implicitly circumvented this problem with the evaluation of likelihood functions by taking logarithms. Unfortunately, this will not help here because we are forming sums of products of small numbers (we are in fact implicitly summing over all possible paths through the lattice diagram of Figure 13.7). We therefore work with re-scaled versions of $\alpha(\mathbf{z}_n)$ and $\beta(\mathbf{z}_n)$ whose values remain of order unity. As we shall see, the corresponding scaling factors cancel out when we use these re-scaled quantities in the EM algorithm.

In (13.34), we defined $\alpha(\mathbf{z}_n) = p(\mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{z}_n)$ representing the joint distribution of all the observations up to $\mathbf{x}_n$ and the latent variable $\mathbf{z}_n$. Now we define a normalized version of α given by

$$\widehat{\alpha}(\mathbf{z}_n) = p(\mathbf{z}_n|\mathbf{x}_1, \ldots, \mathbf{x}_n) = \frac{\alpha(\mathbf{z}_n)}{p(\mathbf{x}_1, \ldots, \mathbf{x}_n)} \tag{13.55}$$

which we expect to be well behaved numerically because it is a probability distribution over K variables for any value of n. In order to relate the scaled and original alpha variables, we introduce scaling factors defined by conditional distributions over the observed variables

$$c_n = p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}). \tag{13.56}$$

From the product rule, we then have

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_n) = \prod_{m=1}^{n} c_m \tag{13.57}$$

and so

$$\alpha(\mathbf{z}_n) = p(\mathbf{z}_n|\mathbf{x}_1, \ldots, \mathbf{x}_n)p(\mathbf{x}_1, \ldots, \mathbf{x}_n) = \left(\prod_{m=1}^{n} c_m\right) \widehat{\alpha}(\mathbf{z}_n). \tag{13.58}$$

We can then turn the recursion equation (13.36) for α into one for $\widehat{\alpha}$ given by

$$c_n \widehat{\alpha}(\mathbf{z}_n) = p(\mathbf{x}_n|\mathbf{z}_n) \sum_{\mathbf{z}_{n-1}} \widehat{\alpha}(\mathbf{z}_{n-1}) p(\mathbf{z}_n|\mathbf{z}_{n-1}). \tag{13.59}$$

Note that at each stage of the forward message passing phase, used to evaluate $\widehat{\alpha}(\mathbf{z}_n)$, we have to evaluate and store c_n, which is easily done because it is the coefficient that normalizes the right-hand side of (13.59) to give $\widehat{\alpha}(\mathbf{z}_n)$.

We can similarly define re-scaled variables $\widehat{\beta}(\mathbf{z}_n)$ using

$$\beta(\mathbf{z}_n) = \left(\prod_{m=n+1}^{N} c_m\right) \widehat{\beta}(\mathbf{z}_n) \tag{13.60}$$

which will again remain within machine precision because, from (13.35), the quantities $\widehat{\beta}(\mathbf{z}_n)$ are simply the ratio of two conditional probabilities

$$\widehat{\beta}(\mathbf{z}_n) = \frac{p(\mathbf{x}_{n+1}, \ldots, \mathbf{x}_N|\mathbf{z}_n)}{p(\mathbf{x}_{n+1}, \ldots, \mathbf{x}_N|\mathbf{x}_1, \ldots, \mathbf{x}_n)}. \tag{13.61}$$

The recursion result (13.38) for β then gives the following recursion for the re-scaled variables

$$c_{n+1}\widehat{\beta}(\mathbf{z}_n) = \sum_{\mathbf{z}_{n+1}} \widehat{\beta}(\mathbf{z}_{n+1}) p(\mathbf{x}_{n+1}|\mathbf{z}_{n+1}) p(\mathbf{z}_{n+1}|\mathbf{z}_n). \tag{13.62}$$

In applying this recursion relation, we make use of the scaling factors c_n that were previously computed in the α phase.

From (13.57), we see that the likelihood function can be found using

$$p(\mathbf{X}) = \prod_{n=1}^{N} c_n. \tag{13.63}$$

Similarly, using (13.33) and (13.43), together with (13.63), we see that the required marginals are given by

Exercise 13.15

$$\gamma(\mathbf{z}_n) = \widehat{\alpha}(\mathbf{z}_n)\widehat{\beta}(\mathbf{z}_n) \tag{13.64}$$

$$\xi(\mathbf{z}_{n-1}, \mathbf{z}_n) = c_n \widehat{\alpha}(\mathbf{z}_{n-1}) p(\mathbf{x}_n|\mathbf{z}_n) p(\mathbf{z}_n|\mathbf{z}_{-1}) \widehat{\beta}(\mathbf{z}_n). \tag{13.65}$$

Finally, we note that there is an alternative formulation of the forward-backward algorithm (Jordan, 2007) in which the backward pass is defined by a recursion based the quantities $\gamma(\mathbf{z}_n) = \widehat{\alpha}(\mathbf{z}_n)\widehat{\beta}(\mathbf{z}_n)$ instead of using $\widehat{\beta}(\mathbf{z}_n)$. This α–γ recursion requires that the forward pass be completed first so that all the quantities $\widehat{\alpha}(\mathbf{z}_n)$ are available for the backward pass, whereas the forward and backward passes of the α–β algorithm can be done independently. Although these two algorithms have comparable computational cost, the α–β version is the most commonly encountered one in the case of hidden Markov models, whereas for linear dynamical systems a recursion analogous to the α–γ form is more usual.

Section 13.3

13.2.5 The Viterbi algorithm

In many applications of hidden Markov models, the latent variables have some meaningful interpretation, and so it is often of interest to find the most probable sequence of hidden states for a given observation sequence. For instance in speech recognition, we might wish to find the most probable phoneme sequence for a given series of acoustic observations. Because the graph for the hidden Markov model is a directed tree, this problem can be solved exactly using the max-sum algorithm. We recall from our discussion in Section 8.4.5 that the problem of finding the most probable sequence of latent states is not the same as that of finding the set of states that are individually the most probable. The latter problem can be solved by first running the forward-backward (sum-product) algorithm to find the latent variable marginals $\gamma(\mathbf{z}_n)$ and then maximizing each of these individually (Duda *et al.*, 2001). However, the set of such states will not, in general, correspond to the most probable sequence of states. In fact, this set of states might even represent a sequence having zero probability, if it so happens that two successive states, which in isolation are individually the most probable, are such that the transition matrix element connecting them is zero.

In practice, we are usually interested in finding the most probable *sequence* of states, and this can be solved efficiently using the max-sum algorithm, which in the context of hidden Markov models is known as the *Viterbi* algorithm (Viterbi, 1967). Note that the max-sum algorithm works with log probabilities and so there is no need to use re-scaled variables as was done with the forward-backward algorithm. Figure 13.16 shows a fragment of the hidden Markov model expanded as lattice diagram. As we have already noted, the number of possible paths through the lattice grows exponentially with the length of the chain. The Viterbi algorithm searches this space of paths efficiently to find the most probable path with a computational cost that grows only linearly with the length of the chain.

As with the sum-product algorithm, we first represent the hidden Markov model as a factor graph, as shown in Figure 13.15. Again, we treat the variable node $\mathbf{z}_N$ as the root, and pass messages to the root starting with the leaf nodes. Using the results (8.93) and (8.94), we see that the messages passed in the max-sum algorithm are given by

$$\mu_{\mathbf{z}_n \to f_{n+1}}(\mathbf{z}_n) = \mu_{f_n \to \mathbf{z}_n}(\mathbf{z}_n) \tag{13.66}$$

$$\mu_{f_{n+1} \to \mathbf{z}_{n+1}}(\mathbf{z}_{n+1}) = \max_{\mathbf{z}_n} \left\{ \ln f_{n+1}(\mathbf{z}_n, \mathbf{z}_{n+1}) + \mu_{\mathbf{z}_n \to f_{n+1}}(\mathbf{z}_n) \right\}. \tag{13.67}$$

Figure 13.16 A fragment of the HMM lattice showing two possible paths. The Viterbi algorithm efficiently determines the most probable path from amongst the exponentially many possibilities. For any given path, the corresponding probability is given by the product of the elements of the transition matrix A_{jk}, corresponding to the probabilities $p(\mathbf{z}_{n+1}|\mathbf{z}_n)$ for each segment of the path, along with the emission densities $p(\mathbf{x}_n|k)$ associated with each node on the path.

If we eliminate $\mu_{\mathbf{z}_n \to f_{n+1}}(\mathbf{z}_n)$ between these two equations, and make use of (13.46), we obtain a recursion for the $f \to \mathbf{z}$ messages of the form

$$\omega(\mathbf{z}_{n+1}) = \ln p(\mathbf{x}_{n+1}|\mathbf{z}_{n+1}) + \max_{\mathbf{z}_n} \{\ln p(\mathbf{x}_{+1}|\mathbf{z}_n) + \omega(\mathbf{z}_n)\} \tag{13.68}$$

where we have introduced the notation $\omega(\mathbf{z}_n) \equiv \mu_{f_n \to \mathbf{z}_n}(\mathbf{z}_n)$.

From (8.95) and (8.96), these messages are initialized using

$$\omega(\mathbf{z}_1) = \ln p(\mathbf{z}_1) + \ln p(\mathbf{x}_1|\mathbf{z}_1). \tag{13.69}$$

where we have used (13.45). Note that to keep the notation uncluttered, we omit the dependence on the model parameters $\boldsymbol{\theta}$ that are held fixed when finding the most probable sequence.

The Viterbi algorithm can also be derived directly from the definition (13.6) of the joint distribution by taking the logarithm and then exchanging maximizations and summations. It is easily seen that the quantities $\omega(\mathbf{z}_n)$ have the probabilistic interpretation

Exercise 13.16

$$\omega(\mathbf{z}_n) = \max_{\mathbf{z}_1,\ldots,\mathbf{z}_{n-1}} p(\mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{z}_1, \ldots, \mathbf{z}_n). \tag{13.70}$$

Once we have completed the final maximization over $\mathbf{z}_N$, we will obtain the value of the joint distribution $p(\mathbf{X}, \mathbf{Z})$ corresponding to the most probable path. We also wish to find the sequence of latent variable values that corresponds to this path. To do this, we simply make use of the back-tracking procedure discussed in Section 8.4.5. Specifically, we note that the maximization over $\mathbf{z}_n$ must be performed for each of the K possible values of $\mathbf{z}_{n+1}$. Suppose we keep a record of the values of $\mathbf{z}_n$ that correspond to the maxima for each value of the K values of $\mathbf{z}_{n+1}$. Let us denote this function by $\psi(k_n)$ where $k \in \{1, \ldots, K\}$. Once we have passed messages to the end of the chain and found the most probable state of $\mathbf{z}_N$, we can then use this function to backtrack along the chain by applying it recursively

$$k_n^{\max} = \psi(k_{n+1}^{\max}). \tag{13.71}$$

Intuitively, we can understand the Viterbi algorithm as follows. Naively, we could consider explicitly all of the exponentially many paths through the lattice, evaluate the probability for each, and then select the path having the highest probability. However, we notice that we can make a dramatic saving in computational cost as follows. Suppose that for each path we evaluate its probability by summing up products of transition and emission probabilities as we work our way forward along each path through the lattice. Consider a particular time step n and a particular state k at that time step. There will be many possible paths converging on the corresponding node in the lattice diagram. However, we need only retain that particular path that so far has the highest probability. Because there are K states at time step n, we need to keep track of K such paths. At time step $n+1$, there will be K^2 possible paths to consider, comprising K possible paths leading out of each of the K current states, but again we need only retain K of these corresponding to the best path for each state at time $n+1$. When we reach the final time step N we will discover which state corresponds to the overall most probable path. Because there is a unique path coming into that state we can trace the path back to step $N-1$ to see what state it occupied at that time, and so on back through the lattice to the state $n=1$.

13.2.6 Extensions of the hidden Markov model

The basic hidden Markov model, along with the standard training algorithm based on maximum likelihood, has been extended in numerous ways to meet the requirements of particular applications. Here we discuss a few of the more important examples.

We see from the digits example in Figure 13.11 that hidden Markov models can be quite poor generative models for the data, because many of the synthetic digits look quite unrepresentative of the training data. If the goal is sequence classification, there can be significant benefit in determining the parameters of hidden Markov models using discriminative rather than maximum likelihood techniques. Suppose we have a training set of R observation sequences $\mathbf{X}_r$, where $r = 1, \ldots, R$, each of which is labelled according to its class m, where $m = 1, \ldots, M$. For each class, we have a separate hidden Markov model with its own parameters $\boldsymbol{\theta}_m$, and we treat the problem of determining the parameter values as a standard classification problem in which we optimize the cross-entropy

$$\sum_{r=1}^{R} \ln p(m_r|\mathbf{X}_r). \tag{13.72}$$

Using Bayes' theorem this can be expressed in terms of the sequence probabilities associated with the hidden Markov models

$$\sum_{r=1}^{R} \ln \left\{ \frac{p(\mathbf{X}_r|\boldsymbol{\theta}_r)p(m_r)}{\sum_{l=1}^{M} p(\mathbf{X}_r|\boldsymbol{\theta}_l)p(l_r)} \right\} \tag{13.73}$$

where $p(m)$ is the prior probability of class m. Optimization of this cost function is more complex than for maximum likelihood (Kapadia, 1998), and in particular

Figure 13.17 Section of an autoregressive hidden Markov model, in which the distribution of the observation $\mathbf{x}_n$ depends on a subset of the previous observations as well as on the hidden state $\mathbf{z}_n$. In this example, the distribution of $\mathbf{x}_n$ depends on the two previous observations $\mathbf{x}_{n-1}$ and $\mathbf{x}_{n-2}$.

requires that every training sequence be evaluated under each of the models in order to compute the denominator in (13.73). Hidden Markov models, coupled with discriminative training methods, are widely used in speech recognition (Kapadia, 1998).

A significant weakness of the hidden Markov model is the way in which it represents the distribution of times for which the system remains in a given state. To see the problem, note that the probability that a sequence sampled from a given hidden Markov model will spend precisely T steps in state k and then make a transition to a different state is given by

$$p(T) = (A_{kk})^T (1 - A_{kk}) \propto \exp\left(-T \ln A_{kk}\right) \tag{13.74}$$

and so is an exponentially decaying function of T. For many applications, this will be a very unrealistic model of state duration. The problem can be resolved by modelling state duration directly in which the diagonal coefficients A_{kk} are all set to zero, and each state k is explicitly associated with a probability distribution $p(T|k)$ of possible duration times. From a generative point of view, when a state k is entered, a value T representing the number of time steps that the system will remain in state k is then drawn from $p(T|k)$. The model then emits T values of the observed variable $\mathbf{x}_t$, which are generally assumed to be independent so that the corresponding emission density is simply $\prod_{t=1}^{T} p(\mathbf{x}_t|k)$. This approach requires some straightforward modifications to the EM optimization procedure (Rabiner, 1989).

Another limitation of the standard HMM is that it is poor at capturing long-range correlations between the observed variables (i.e., between variables that are separated by many time steps) because these must be mediated via the first-order Markov chain of hidden states. Longer-range effects could in principle be included by adding extra links to the graphical model of Figure 13.5. One way to address this is to generalize the HMM to give the *autoregressive hidden Markov model* (Ephraim *et al.*, 1989), an example of which is shown in Figure 13.17. For discrete observations, this corresponds to expanded tables of conditional probabilities for the emission distributions. In the case of a Gaussian emission density, we can use the linear-Gaussian framework in which the conditional distribution for $\mathbf{x}_n$ given the values of the previous observations, and the value of $\mathbf{z}_n$, is a Gaussian whose mean is a linear combination of the values of the conditioning variables. Clearly the number of additional links in the graph must be limited to avoid an excessive the number of free parameters. In the example shown in Figure 13.17, each observation depends on

Figure 13.18 Example of an input-output hidden Markov model. In this case, both the emission probabilities and the transition probabilities depend on the values of a sequence of observations $\mathbf{u}_1, \ldots, \mathbf{u}_N$.

the two preceding observed variables as well as on the hidden state. Although this graph looks messy, we can again appeal to d-separation to see that in fact it still has a simple probabilistic structure. In particular, if we imagine conditioning on $\mathbf{z}_n$ we see that, as with the standard HMM, the values of $\mathbf{z}_{n-1}$ and $\mathbf{z}_{n+1}$ are independent, corresponding to the conditional independence property (13.5). This is easily verified by noting that every path from node $\mathbf{z}_{n-1}$ to node $\mathbf{z}_{n+1}$ passes through at least one observed node that is head-to-tail with respect to that path. As a consequence, we can again use a forward-backward recursion in the E step of the EM algorithm to determine the posterior distributions of the latent variables in a computational time that is linear in the length of the chain. Similarly, the M step involves only a minor modification of the standard M-step equations. In the case of Gaussian emission densities this involves estimating the parameters using the standard linear regression equations, discussed in Chapter 3.

We have seen that the autoregressive HMM appears as a natural extension of the standard HMM when viewed as a graphical model. In fact the probabilistic graphical modelling viewpoint motivates a plethora of different graphical structures based on the HMM. Another example is the *input-output* hidden Markov model (Bengio and Frasconi, 1995), in which we have a sequence of observed variables $\mathbf{u}_1, \ldots, \mathbf{u}_N$, in addition to the output variables $\mathbf{x}_1, \ldots, \mathbf{x}_N$, whose values influence either the distribution of latent variables or output variables, or both. An example is shown in Figure 13.18. This extends the HMM framework to the domain of supervised learning for sequential data. It is again easy to show, through the use of the d-separation criterion, that the Markov property (13.5) for the chain of latent variables still holds. To verify this, simply note that there is only one path from node $\mathbf{z}_{n-1}$ to node $\mathbf{z}_{n+1}$ and this is head-to-tail with respect to the observed node $\mathbf{z}_n$. This conditional independence property again allows the formulation of a computationally efficient learning algorithm. In particular, we can determine the parameters $\boldsymbol{\theta}$ of the model by maximizing the likelihood function $L(\boldsymbol{\theta}) = p(\mathbf{X}|\mathbf{U}, \boldsymbol{\theta})$ where $\mathbf{U}$ is a matrix whose rows are given by $\mathbf{u}_n^{\mathrm{T}}$. As a consequence of the conditional independence property (13.5) this likelihood function can be maximized efficiently using an EM algorithm in which the E step involves forward and backward recursions.

Exercise 13.18

Another variant of the HMM worthy of mention is the *factorial hidden Markov model* (Ghahramani and Jordan, 1997), in which there are multiple independent

Figure 13.19 A factorial hidden Markov model comprising two Markov chains of latent variables. For continuous observed variables $\mathbf{x}$, one possible choice of emission model is a linear-Gaussian density in which the mean of the Gaussian is a linear combination of the states of the corresponding latent variables.

Markov chains of latent variables, and the distribution of the observed variable at a given time step is conditional on the states of all of the corresponding latent variables at that same time step. Figure 13.19 shows the corresponding graphical model. The motivation for considering factorial HMM can be seen by noting that in order to represent, say, 10 bits of information at a given time step, a standard HMM would need $K = 2^{10} = 1024$ latent states, whereas a factorial HMM could make use of 10 binary latent chains. The primary disadvantage of factorial HMMs, however, lies in the additional complexity of training them. The M step for the factorial HMM model is straightforward. However, observation of the $\mathbf{x}$ variables introduces dependencies between the latent chains, leading to difficulties with the E step. This can be seen by noting that in Figure 13.19, the variables $\mathbf{z}_n^{(1)}$ and $\mathbf{z}_n^{(2)}$ are connected by a path which is head-to-head at node $\mathbf{x}_n$ and hence they are not d-separated. The exact E step for this model does *not* correspond to running forward and backward recursions along the M Markov chains independently. This is confirmed by noting that the key conditional independence property (13.5) is not satisfied for the individual Markov chains in the factorial HMM model, as is shown using d-separation in Figure 13.20. Now suppose that there are M chains of hidden nodes and for simplicity suppose that all latent variables have the same number K of states. Then one approach would be to note that there are K^M combinations of latent variables at a given time step

Figure 13.20 Example of a path, highlighted in green, which is head-to-head at the observed nodes $\mathbf{x}_{n-1}$ and $\mathbf{x}_{n+1}$, and head-to-tail at the unobserved nodes $\mathbf{z}_{n-1}^{(2)}$, $\mathbf{z}_n^{(2)}$ and $\mathbf{z}_{n+1}^{(2)}$. Thus the path is not blocked and so the conditional independence property (13.5) does not hold for the individual latent chains of the factorial HMM model. As a consequence, there is no efficient exact E step for this model.

and so we can transform the model into an equivalent standard HMM having a single chain of latent variables each of which has K^M latent states. We can then run the standard forward-backward recursions in the E step. This has computational complexity $O(NK^{2M})$ that is exponential in the number M of latent chains and so will be intractable for anything other than small values of M. One solution would be to use sampling methods (discussed in Chapter 11). As an elegant deterministic alternative, Ghahramani and Jordan (1997) exploited variational inference techniques to obtain a tractable algorithm for approximate inference. This can be done using a simple variational posterior distribution that is fully factorized with respect to the latent variables, or alternatively by using a more powerful approach in which the variational distribution is described by independent Markov chains corresponding to the chains of latent variables in the original model. In the latter case, the variational inference algorithms involves running independent forward and backward recursions along each chain, which is computationally efficient and yet is also able to capture correlations between variables within the same chain.

Section 10.1

Clearly, there are many possible probabilistic structures that can be constructed according to the needs of particular applications. Graphical models provide a general technique for motivating, describing, and analysing such structures, and variational methods provide a powerful framework for performing inference in those models for which exact solution is intractable.

13.3. Linear Dynamical Systems

In order to motivate the concept of linear dynamical systems, let us consider the following simple problem, which often arises in practical settings. Suppose we wish to measure the value of an unknown quantity $\mathbf{z}$ using a noisy sensor that returns a observation $\mathbf{x}$ representing the value of $\mathbf{z}$ plus zero-mean Gaussian noise. Given a single measurement, our best guess for $\mathbf{z}$ is to assume that $\mathbf{z} = \mathbf{x}$. However, we can improve our estimate for $\mathbf{z}$ by taking lots of measurements and averaging them, because the random noise terms will tend to cancel each other. Now let's make the situation more complicated by assuming that we wish to measure a quantity $\mathbf{z}$ that is changing over time. We can take regular measurements of $\mathbf{x}$ so that at some point in time we have obtained $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and we wish to find the corresponding values $\mathbf{z}_1, \ldots, \mathbf{x}_N$. If we simply average the measurements, the error due to random noise will be reduced, but unfortunately we will just obtain a single averaged estimate, in which we have averaged over the changing value of $\mathbf{z}$, thereby introducing a new source of error.

Intuitively, we could imagine doing a bit better as follows. To estimate the value of $\mathbf{z}_N$, we take only the most recent few measurements, say $\mathbf{x}_{N-L}, \ldots, \mathbf{x}_N$ and just average these. If $\mathbf{z}$ is changing slowly, and the random noise level in the sensor is high, it would make sense to choose a relatively long window of observations to average. Conversely, if the signal is changing quickly, and the noise levels are small, we might be better just to use $\mathbf{x}_N$ directly as our estimate of $\mathbf{z}_N$. Perhaps we could do even better if we take a weighted average, in which more recent measurements

make a greater contribution than less recent ones.

Although this sort of intuitive argument seems plausible, it does not tell us how to form a weighted average, and any sort of hand-crafted weighing is hardly likely to be optimal. Fortunately, we can address problems such as this much more systematically by defining a probabilistic model that captures the time evolution and measurement processes and then applying the inference and learning methods developed in earlier chapters. Here we shall focus on a widely used model known as a *linear dynamical system*.

As we have seen, the HMM corresponds to the state space model shown in Figure 13.5 in which the latent variables are discrete but with arbitrary emission probability distributions. This graph of course describes a much broader class of probability distributions, all of which factorize according to (13.6). We now consider extensions to other distributions for the latent variables. In particular, we consider continuous latent variables in which the summations of the sum-product algorithm become integrals. The general form of the inference algorithms will, however, be the same as for the hidden Markov model. It is interesting to note that, historically, hidden Markov models and linear dynamical systems were developed independently. Once they are both expressed as graphical models, however, the deep relationship between them immediately becomes apparent.

One key requirement is that we retain an efficient algorithm for inference which is linear in the length of the chain. This requires that, for instance, when we take a quantity $\widehat{\alpha}(\mathbf{z}_{n-1})$, representing the posterior probability of $\mathbf{z}_n$ given observations $\mathbf{x}_1, \ldots, \mathbf{x}_n$, and multiply by the transition probability $p(\mathbf{z}_n|\mathbf{z}_{n-1})$ and the emission probability $p(\mathbf{x}_n|\mathbf{z}_n)$ and then marginalize over $\mathbf{z}_{n-1}$, we obtain a distribution over $\mathbf{z}_n$ that is of the same functional form as that over $\widehat{\alpha}(\mathbf{z}_{n-1})$. That is to say, the distribution must not become more complex at each stage, but must only change in its parameter values. Not surprisingly, the only distributions that have this property of being closed under multiplication are those belonging to the exponential family.

Here we consider the most important example from a practical perspective, which is the Gaussian. In particular, we consider a linear-Gaussian state space model so that the latent variables $\{\mathbf{z}_n\}$, as well as the observed variables $\{\mathbf{x}_n\}$, are multivariate Gaussian distributions whose means are linear functions of the states of their parents in the graph. We have seen that a directed graph of linear-Gaussian units is equivalent to a joint Gaussian distribution over all of the variables. Furthermore, marginals such as $\widehat{\alpha}(\mathbf{z}_n)$ are also Gaussian, so that the functional form of the messages is preserved and we will obtain an efficient inference algorithm. By contrast, suppose that the emission densities $p(\mathbf{x}_n|\mathbf{z}_n)$ comprise a mixture of K Gaussians each of which has a mean that is linear in $\mathbf{z}_n$. Then even if $\widehat{\alpha}(\mathbf{z}_1)$ is Gaussian, the quantity $\widehat{\alpha}(\mathbf{z}_2)$ will be a mixture of K Gaussians, $\widehat{\alpha}(\mathbf{z}_3)$ will be a mixture of K^2 Gaussians, and so on, and exact inference will not be of practical value.

We have seen that the hidden Markov model can be viewed as an extension of the mixture models of Chapter 9 to allow for sequential correlations in the data. In a similar way, we can view the linear dynamical system as a generalization of the continuous latent variable models of Chapter 12 such as probabilistic PCA and factor analysis. Each pair of nodes $\{\mathbf{z}_n, \mathbf{x}_n\}$ represents a linear-Gaussian latent variable

model for that particular observation. However, the latent variables $\{\mathbf{z}_n\}$ are no longer treated as independent but now form a Markov chain.

Because the model is represented by a tree-structured directed graph, inference problems can be solved efficiently using the sum-product algorithm. The forward recursions, analogous to the α messages of the hidden Markov model, are known as the *Kalman filter* equations (Kalman, 1960; Zarchan and Musoff, 2005), and the backward recursions, analogous to the β messages, are known as the *Kalman smoother* equations, or the *Rauch-Tung-Striebel* (RTS) equations (Rauch *et al.*, 1965). The Kalman filter is widely used in many real-time tracking applications.

Because the linear dynamical system is a linear-Gaussian model, the joint distribution over all variables, as well as all marginals and conditionals, will be Gaussian. It follows that the sequence of individually most probable latent variable values is the same as the most probable latent sequence. There is thus no need to consider the analogue of the Viterbi algorithm for the linear dynamical system.

Exercise 13.19

Because the model has linear-Gaussian conditional distributions, we can write the transition and emission distributions in the general form

$$p(\mathbf{z}_n|\mathbf{z}_{n-1}) = \mathcal{N}(\mathbf{z}_n|\mathbf{A}\mathbf{z}_{n-1}, \boldsymbol{\Gamma}) \tag{13.75}$$

$$p(\mathbf{x}_n|\mathbf{z}_n) = \mathcal{N}(\mathbf{x}_n|\mathbf{C}\mathbf{z}_n, \boldsymbol{\Sigma}). \tag{13.76}$$

The initial latent variable also has a Gaussian distribution which we write as

$$p(\mathbf{z}_1) = \mathcal{N}(\mathbf{z}_1|\boldsymbol{\mu}_0, \mathbf{V}_0). \tag{13.77}$$

Note that in order to simplify the notation, we have omitted additive constant terms from the means of the Gaussians. In fact, it is straightforward to include them if desired. Traditionally, these distributions are more commonly expressed in an equivalent form in terms of noisy linear equations given by

Exercise 13.24

$$\mathbf{z}_n = \mathbf{A}\mathbf{z}_{n-1} + \mathbf{w}_n \tag{13.78}$$

$$\mathbf{x}_n = \mathbf{C}\mathbf{z}_n + \mathbf{v}_n \tag{13.79}$$

$$\mathbf{z}_1 = \boldsymbol{\mu}_0 + \mathbf{u} \tag{13.80}$$

where the noise terms have the distributions

$$\mathbf{w} \sim \mathcal{N}(\mathbf{w}|\mathbf{0}, \boldsymbol{\Gamma}) \tag{13.81}$$

$$\mathbf{v} \sim \mathcal{N}(\mathbf{v}|\mathbf{0}, \boldsymbol{\Sigma}) \tag{13.82}$$

$$\mathbf{u} \sim \mathcal{N}(\mathbf{u}|\mathbf{0}, \mathbf{V}_0). \tag{13.83}$$

The parameters of the model, denoted by $\boldsymbol{\theta} = \{\mathbf{A}, \boldsymbol{\Gamma}, \mathbf{C}, \boldsymbol{\Sigma}, \boldsymbol{\mu}_0, \mathbf{V}_0\}$, can be determined using maximum likelihood through the EM algorithm. In the E step, we need to solve the inference problem of determining the local posterior marginals for the latent variables, which can be solved efficiently using the sum-product algorithm, as we discuss in the next section.

13.3.1 Inference in LDS

We now turn to the problem of finding the marginal distributions for the latent variables conditional on the observation sequence. For given parameter settings, we also wish to make predictions of the next latent state $\mathbf{z}_n$ and of the next observation $\mathbf{x}_n$ conditioned on the observed data $\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}$ for use in real-time applications. These inference problems can be solved efficiently using the sum-product algorithm, which in the context of the linear dynamical system gives rise to the Kalman filter and Kalman smoother equations.

It is worth emphasizing that because the linear dynamical system is a linear-Gaussian model, the joint distribution over all latent and observed variables is simply a Gaussian, and so in principle we could solve inference problems by using the standard results derived in previous chapters for the marginals and conditionals of a multivariate Gaussian. The role of the sum-product algorithm is to provide a more efficient way to perform such computations.

Linear dynamical systems have the identical factorization, given by (13.6), to hidden Markov models, and are again described by the factor graphs in Figures 13.14 and 13.15. Inference algorithms therefore take precisely the same form except that summations over latent variables are replaced by integrations. We begin by considering the forward equations in which we treat $\mathbf{z}_N$ as the root node, and propagate messages from the leaf node $h(\mathbf{z}_1)$ to the root. From (13.77), the initial message will be Gaussian, and because each of the factors is Gaussian, all subsequent messages will also be Gaussian. By convention, we shall propagate messages that are normalized marginal distributions corresponding to $p(\mathbf{z}_n|\mathbf{x}_1, \ldots, \mathbf{x}_n)$, which we denote by

$$\widehat{\alpha}(\mathbf{z}_n) = \mathcal{N}(\mathbf{z}_n|\boldsymbol{\mu}_n, \mathbf{V}_n). \tag{13.84}$$

This is precisely analogous to the propagation of scaled variables $\widehat{\alpha}(\mathbf{z}_n)$ given by (13.59) in the discrete case of the hidden Markov model, and so the recursion equation now takes the form

$$c_n\widehat{\alpha}(\mathbf{z}_n) = p(\mathbf{x}_n|\mathbf{z}_n) \int \widehat{\alpha}(\mathbf{z}_{n-1})p(\mathbf{z}_n|\mathbf{z}_{n-1})\,\mathrm{d}\mathbf{z}_{n-1}. \tag{13.85}$$

Substituting for the conditionals $p(\mathbf{z}_n|\mathbf{z}_{n-1})$ and $p(\mathbf{x}_n|\mathbf{z}_n)$, using (13.75) and (13.76), respectively, and making use of (13.84), we see that (13.85) becomes

$$\begin{aligned} c_n\mathcal{N}(\mathbf{z}_n|\boldsymbol{\mu}_n, \mathbf{V}_n) &= \mathcal{N}(\mathbf{x}_n|\mathbf{C}\mathbf{z}_n, \boldsymbol{\Sigma}) \\ &\int \mathcal{N}(\mathbf{z}_n|\mathbf{A}\mathbf{z}_{n-1}, \boldsymbol{\Gamma})\mathcal{N}(\mathbf{z}_{n-1}|\boldsymbol{\mu}_{n-1}, \mathbf{V}_{n-1})\,\mathrm{d}\mathbf{z}_{n-1}. \end{aligned} \tag{13.86}$$

Here we are supposing that $\boldsymbol{\mu}_{n-1}$ and $\mathbf{V}_{n-1}$ are known, and by evaluating the integral in (13.86), we wish to determine values for $\boldsymbol{\mu}_n$ and $\mathbf{V}_n$. The integral is easily evaluated by making use of the result (2.115), from which it follows that

$$\begin{aligned} &\int \mathcal{N}(\mathbf{z}_n|\mathbf{A}\mathbf{z}_{n-1}, \boldsymbol{\Gamma})\mathcal{N}(\mathbf{z}_{n-1}|\boldsymbol{\mu}_{n-1}, \mathbf{V}_{n-1})\,\mathrm{d}\mathbf{z}_{n-1} \\ &\qquad = \mathcal{N}(\mathbf{z}_n|\mathbf{A}\boldsymbol{\mu}_{n-1}, \mathbf{P}_{n-1}) \end{aligned} \tag{13.87}$$

where we have defined

$$\mathbf{P}_{n-1} = \mathbf{A}\mathbf{V}_{n-1}\mathbf{A}^{\mathrm{T}} + \boldsymbol{\Gamma}. \tag{13.88}$$

We can now combine this result with the first factor on the right-hand side of (13.86) by making use of (2.115) and (2.116) to give

$$\begin{aligned}
\boldsymbol{\mu}_n &= \mathbf{A}\boldsymbol{\mu}_{n-1} + \mathbf{K}_n(\mathbf{x}_n - \mathbf{C}\mathbf{A}\boldsymbol{\mu}_{n-1}) && (13.89)\\
\mathbf{V}_n &= (\mathbf{I} - \mathbf{K}_n\mathbf{C})\mathbf{P}_{n-1} && (13.90)\\
c_n &= \mathcal{N}(\mathbf{x}_n|\mathbf{C}\mathbf{A}\boldsymbol{\mu}_{n-1}, \mathbf{C}\mathbf{P}_{n-1}\mathbf{C}^{\mathrm{T}} + \boldsymbol{\Sigma}). && (13.91)
\end{aligned}$$

Here we have made use of the matrix inverse identities (C.5) and (C.7) and also defined the *Kalman gain matrix*

$$\mathbf{K}_n = \mathbf{P}_{n-1}\mathbf{C}^{\mathrm{T}}\left(\mathbf{C}\mathbf{P}_{n-1}\mathbf{C}^{\mathrm{T}} + \boldsymbol{\Sigma}\right)^{-1}. \tag{13.92}$$

Thus, given the values of $\boldsymbol{\mu}_{n-1}$ and $\mathbf{V}_{n-1}$, together with the new observation $\mathbf{x}_n$, we can evaluate the Gaussian marginal for $\mathbf{z}_n$ having mean $\boldsymbol{\mu}_n$ and covariance $\mathbf{V}_n$, as well as the normalization coefficient c_n.

The initial conditions for these recursion equations are obtained from

$$c_1\widehat{\alpha}(\mathbf{z}_1) = p(\mathbf{z}_1)p(\mathbf{x}_1|\mathbf{z}_1). \tag{13.93}$$

Because $p(\mathbf{z}_1)$ is given by (13.77), and $p(\mathbf{x}_1|\mathbf{z}_1)$ is given by (13.76), we can again make use of (2.115) to calculate c_1 and (2.116) to calculate $\boldsymbol{\mu}_1$ and $\mathbf{V}_1$ giving

$$\begin{aligned}
\boldsymbol{\mu}_1 &= \boldsymbol{\mu}_0 + \mathbf{K}_1(\mathbf{x}_1 - \mathbf{C}\boldsymbol{\mu}_0) && (13.94)\\
\mathbf{V}_1 &= (\mathbf{I} - \mathbf{K}_1\mathbf{C})\mathbf{V}_0 && (13.95)\\
c_1 &= \mathcal{N}(\mathbf{x}_1|\mathbf{C}\boldsymbol{\mu}_0, \mathbf{C}\mathbf{V}_0\mathbf{C}^{\mathrm{T}} + \boldsymbol{\Sigma}) && (13.96)
\end{aligned}$$

where

$$\mathbf{K}_1 = \mathbf{V}_0\mathbf{C}^{\mathrm{T}}\left(\mathbf{C}\mathbf{V}_0\mathbf{C}^{\mathrm{T}} + \boldsymbol{\Sigma}\right)^{-1}. \tag{13.97}$$

Similarly, the likelihood function for the linear dynamical system is given by (13.63) in which the factors c_n are found using the Kalman filtering equations.

We can interpret the steps involved in going from the posterior marginal over $\mathbf{z}_{n-1}$ to the posterior marginal over $\mathbf{z}_n$ as follows. In (13.89), we can view the quantity $\mathbf{A}\boldsymbol{\mu}_{n-1}$ as the prediction of the mean over $\mathbf{z}_n$ obtained by simply taking the mean over $\mathbf{z}_{n-1}$ and projecting it forward one step using the transition probability matrix $\mathbf{A}$. This predicted mean would give a predicted observation for $\mathbf{x}_n$ given by $\mathbf{C}\mathbf{A}\mathbf{z}_{n-1}$ obtained by applying the emission probability matrix $\mathbf{C}$ to the predicted hidden state mean. We can view the update equation (13.89) for the mean of the hidden variable distribution as taking the predicted mean $\mathbf{A}\boldsymbol{\mu}_{n-1}$ and then adding a correction that is proportional to the error $\mathbf{x}_n - \mathbf{C}\mathbf{A}\mathbf{z}_{n-1}$ between the predicted observation and the actual observation. The coefficient of this correction is given by the Kalman gain matrix. Thus we can view the Kalman filter as a process of making successive predictions and then correcting these predictions in the light of the new observations. This is illustrated graphically in Figure 13.21.

Figure 13.21 The linear dynamical system can be viewed as a sequence of steps in which increasing uncertainty in the state variable due to diffusion is compensated by the arrival of new data. In the left-hand plot, the blue curve shows the distribution $p(\mathbf{z}_{n-1}|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$, which incorporates all the data up to step $n-1$. The diffusion arising from the nonzero variance of the transition probability $p(\mathbf{z}_n|\mathbf{z}_{n-1})$ gives the distribution $p(\mathbf{z}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$, shown in red in the centre plot. Note that this is broader and shifted relative to the blue curve (which is shown dashed in the centre plot for comparison). The next data observation $\mathbf{x}_n$ contributes through the emission density $p(\mathbf{x}_n|\mathbf{z}_n)$, which is shown as a function of $\mathbf{z}_n$ in green on the right-hand plot. Note that this is not a density with respect to $\mathbf{z}_n$ and so is not normalized to one. Inclusion of this new data point leads to a revised distribution $p(\mathbf{z}_n|\mathbf{x}_1, \ldots, \mathbf{x}_n)$ for the state density shown in blue. We see that observation of the data has shifted and narrowed the distribution compared to $p(\mathbf{z}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$ (which is shown in dashed in the right-hand plot for comparison).

If we consider a situation in which the measurement noise is small compared to the rate at which the latent variable is evolving, then we find that the posterior distribution for $\mathbf{z}_n$ depends only on the current measurement $\mathbf{x}_n$, in accordance with the intuition from our simple example at the start of the section. Similarly, if the latent variable is evolving slowly relative to the observation noise level, we find that the posterior mean for $\mathbf{z}_n$ is obtained by averaging all of the measurements obtained up to that time.

Exercise 13.27

Exercise 13.28

One of the most important applications of the Kalman filter is to tracking, and this is illustrated using a simple example of an object moving in two dimensions in Figure 13.22.

So far, we have solved the inference problem of finding the posterior marginal for a node $\mathbf{z}_n$ given observations from $\mathbf{x}_1$ up to $\mathbf{x}_n$. Next we turn to the problem of finding the marginal for a node $\mathbf{z}_n$ given all observations $\mathbf{x}_1$ to $\mathbf{x}_N$. For temporal data, this corresponds to the inclusion of future as well as past observations. Although this cannot be used for real-time prediction, it plays a key role in learning the parameters of the model. By analogy with the hidden Markov model, this problem can be solved by propagating messages from node $\mathbf{x}_N$ back to node $\mathbf{x}_1$ and combining this information with that obtained during the forward message passing stage used to compute the $\widehat{\alpha}(\mathbf{z}_n)$.

In the LDS literature, it is usual to formulate this backward recursion in terms of $\gamma(\mathbf{z}_n) = \widehat{\alpha}(\mathbf{z}_n)\widehat{\beta}(\mathbf{z}_n)$ rather than in terms of $\widehat{\beta}(\mathbf{z}_n)$. Because $\gamma(\mathbf{z}_n)$ must also be Gaussian, we write it in the form

$$\gamma(\mathbf{z}_n) = \widehat{\alpha}(\mathbf{z}_n)\widehat{\beta}(\mathbf{z}_n) = \mathcal{N}(\mathbf{z}_n|\widehat{\boldsymbol{\mu}}_n, \widehat{\mathbf{V}}_n). \tag{13.98}$$

To derive the required recursion, we start from the backward recursion (13.62) for

Figure 13.22 An illustration of a linear dynamical system being used to track a moving object. The blue points indicate the true positions of the object in a two-dimensional space at successive time steps, the green points denote noisy measurements of the positions, and the red crosses indicate the means of the inferred posterior distributions of the positions obtained by running the Kalman filtering equations. The covariances of the inferred positions are indicated by the red ellipses, which correspond to contours having one standard deviation.

$\widehat{\beta}(\mathbf{z}_n)$, which, for continuous latent variables, can be written in the form

$$c_{n+1}\widehat{\beta}(\mathbf{z}_n) = \int \widehat{\beta}(\mathbf{z}_{n+1})p(\mathbf{x}_{n+1}|\mathbf{z}_{n+1})p(\mathbf{z}_{n+1}|\mathbf{z}_n)\,\mathrm{d}\mathbf{z}_{n+1}. \tag{13.99}$$

We now multiply both sides of (13.99) by $\widehat{\alpha}(\mathbf{z}_n)$ and substitute for $p(\mathbf{x}_{n+1}|\mathbf{z}_{n+1})$ and $p(\mathbf{z}_{n+1}|\mathbf{z}_n)$ using (13.75) and (13.76). Then we make use of (13.89), (13.90) and (13.91), together with (13.98), and after some manipulation we obtain

Exercise 13.29

$$\widehat{\boldsymbol{\mu}}_n = \boldsymbol{\mu}_n + \mathbf{J}_n\left(\widehat{\boldsymbol{\mu}}_{n+1} - \mathbf{A}\boldsymbol{\mu}_N\right) \tag{13.100}$$

$$\widehat{\mathbf{V}}_n = \mathbf{V}_n + \mathbf{J}_n\left(\widehat{\mathbf{V}}_{n+1} - \mathbf{P}_n\right)\mathbf{J}_n^{\mathrm{T}} \tag{13.101}$$

where we have defined

$$\mathbf{J}_n = \mathbf{V}_n\mathbf{A}^{\mathrm{T}}\left(\mathbf{P}_n\right)^{-1} \tag{13.102}$$

and we have made use of $\mathbf{A}\mathbf{V}_n = \mathbf{P}_n\mathbf{J}_n^{\mathrm{T}}$. Note that these recursions require that the forward pass be completed first so that the quantities $\boldsymbol{\mu}_n$ and $\mathbf{V}_n$ will be available for the backward pass.

For the EM algorithm, we also require the pairwise posterior marginals, which can be obtained from (13.65) in the form

$$\begin{aligned}\xi(\mathbf{z}_{n-1},\mathbf{z}_n) &= (c_n)^{-1}\,\widehat{\alpha}(\mathbf{z}_{n-1})p(\mathbf{x}_n|\mathbf{z}_n)p(\mathbf{z}_n|\mathbf{z}_{-1})\widehat{\beta}(\mathbf{z}_n)\\ &= \frac{\mathcal{N}(\mathbf{z}_{n-1}|\boldsymbol{\mu}_{n-1},\mathbf{V}_{n-1})\mathcal{N}(\mathbf{z}_n|\mathbf{A}\mathbf{z}_{n-1},\boldsymbol{\Gamma})\mathcal{N}(\mathbf{x}_n|\mathbf{C}\mathbf{z}_n,\boldsymbol{\Sigma})\mathcal{N}(\mathbf{z}_n|\widehat{\boldsymbol{\mu}}_n,\widehat{\mathbf{V}}_n)}{c_n\widehat{\alpha}(\mathbf{z}_n)}.\end{aligned} \tag{13.103}$$

Substituting for $\widehat{\alpha}(\mathbf{z}_n)$ using (13.84) and rearranging, we see that $\xi(\mathbf{z}_{n-1},\mathbf{z}_n)$ is a Gaussian with mean given with components $\gamma(\mathbf{z}_{n-1})$ and $\gamma(\mathbf{z}_n)$, and a covariance between $\mathbf{z}_n$ and $\mathbf{z}_{n-1}$ given by

Exercise 13.31

$$\mathrm{cov}[\mathbf{z}_n,\mathbf{z}_{n-1}] = \mathbf{J}_{n-1}\widehat{\mathbf{V}}_n. \tag{13.104}$$

13.3.2 Learning in LDS

So far, we have considered the inference problem for linear dynamical systems, assuming that the model parameters $\boldsymbol{\theta} = \{\mathbf{A}, \boldsymbol{\Gamma}, \mathbf{C}, \boldsymbol{\Sigma}, \boldsymbol{\mu}_0, \mathbf{V}_0\}$ are known. Next, we consider the determination of these parameters using maximum likelihood (Ghahramani and Hinton, 1996b). Because the model has latent variables, this can be addressed using the EM algorithm, which was discussed in general terms in Chapter 9.

We can derive the EM algorithm for the linear dynamical system as follows. Let us denote the estimated parameter values at some particular cycle of the algorithm by $\boldsymbol{\theta}^{\text{old}}$. For these parameter values, we can run the inference algorithm to determine the posterior distribution of the latent variables $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}})$, or more precisely those local posterior marginals that are required in the M step. In particular, we shall require the following expectations

$$\mathbb{E}\left[\mathbf{z}_n\right] = \widehat{\boldsymbol{\mu}}_n \tag{13.105}$$

$$\mathbb{E}\left[\mathbf{z}_n \mathbf{z}_{n-1}^{\mathrm{T}}\right] = \mathbf{J}_{n-1}\widehat{\mathbf{V}}_n + \widehat{\boldsymbol{\mu}}_n \widehat{\boldsymbol{\mu}}_{n-1}^{\mathrm{T}} \tag{13.106}$$

$$\mathbb{E}\left[\mathbf{z}_n \mathbf{z}_n^{\mathrm{T}}\right] = \widehat{\mathbf{V}}_n + \widehat{\boldsymbol{\mu}}_n \widehat{\boldsymbol{\mu}}_n^{\mathrm{T}} \tag{13.107}$$

where we have used (13.104).

Now we consider the complete-data log likelihood function, which is obtained by taking the logarithm of (13.6) and is therefore given by

$$\begin{aligned} \ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) &= \ln p(\mathbf{z}_1|\boldsymbol{\mu}_0, \mathbf{V}_0) + \sum_{n=2}^{N} \ln p(\mathbf{z}_n|\mathbf{z}_{n-1}, \mathbf{A}, \boldsymbol{\Gamma}) \\ &\quad + \sum_{n=1}^{N} \ln p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{C}, \boldsymbol{\Sigma}) \end{aligned} \tag{13.108}$$

in which we have made the dependence on the parameters explicit. We now take the expectation of the complete-data log likelihood with respect to the posterior distribution $p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}})$ which defines the function

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) = \mathbb{E}_{\mathbf{Z}|\boldsymbol{\theta}^{\text{old}}}\left[\ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})\right]. \tag{13.109}$$

In the M step, this function is maximized with respect to the components of $\boldsymbol{\theta}$.

Consider first the parameters $\boldsymbol{\mu}_0$ and $\mathbf{V}_0$. If we substitute for $p(\mathbf{z}_1|\boldsymbol{\mu}_0, \mathbf{V}_0)$ in (13.108) using (13.77), and then take the expectation with respect to $\mathbf{Z}$, we obtain

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) = -\frac{1}{2}\ln|\mathbf{V}_0| - \mathbb{E}_{\mathbf{Z}|\boldsymbol{\theta}^{\text{old}}}\left[\frac{1}{2}(\mathbf{z}_1 - \boldsymbol{\mu}_0)^{\mathrm{T}}\mathbf{V}_0^{-1}(\mathbf{z}_1 - \boldsymbol{\mu}_0)\right] + \text{const}$$

where all terms not dependent on $\boldsymbol{\mu}_0$ or $\mathbf{V}_0$ have been absorbed into the additive constant. Maximization with respect to $\boldsymbol{\mu}_0$ and $\mathbf{V}_0$ is easily performed by making use of the maximum likelihood solution for a Gaussian distribution discussed in Section 2.3.4, giving

Exercise 13.32

$$\boldsymbol{\mu}_0^{\text{new}} = \mathbb{E}[\mathbf{z}_1] \tag{13.110}$$
$$\mathbf{V}_0^{\text{new}} = \mathbb{E}[\mathbf{z}_1\mathbf{z}_1^{\mathrm{T}}] - \mathbb{E}[\mathbf{z}_1]\mathbb{E}[\mathbf{z}_1^{\mathrm{T}}]. \tag{13.111}$$

Similarly, to optimize $\mathbf{A}$ and $\boldsymbol{\Gamma}$, we substitute for $p(\mathbf{z}_n|\mathbf{z}_{n-1}, \mathbf{A}, \boldsymbol{\Gamma})$ in (13.108) using (13.75) giving

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) = -\frac{N-1}{2}\ln|\boldsymbol{\Gamma}| - \mathbb{E}_{\mathbf{Z}|\boldsymbol{\theta}^{\text{old}}}\left[\frac{1}{2}\sum_{n=2}^{N}(\mathbf{z}_n - \mathbf{A}\mathbf{z}_{n-1})^{\mathrm{T}}\boldsymbol{\Gamma}^{-1}(\mathbf{z}_n - \mathbf{A}\mathbf{z}_{n-1})\right] + \text{const} \tag{13.112}$$

in which the constant comprises terms that are independent of $\mathbf{A}$ and $\boldsymbol{\Gamma}$. Maximizing with respect to these parameters then gives

Exercise 13.33

$$\mathbf{A}^{\text{new}} = \left(\sum_{n=2}^{N}\mathbb{E}\left[\mathbf{z}_n\mathbf{z}_{n-1}^{\mathrm{T}}\right]\right)\left(\sum_{n=2}^{N}\mathbb{E}\left[\mathbf{z}_{n-1}\mathbf{z}_{n-1}^{\mathrm{T}}\right]\right)^{-1} \tag{13.113}$$

$$\boldsymbol{\Gamma}^{\text{new}} = \frac{1}{N-1}\sum_{n=2}^{N}\left\{\mathbb{E}\left[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}\right] - \mathbf{A}^{\text{new}}\mathbb{E}\left[\mathbf{z}_{n-1}\mathbf{z}_n^{\mathrm{T}}\right] - \mathbb{E}\left[\mathbf{z}_n\mathbf{z}_{n-1}^{\mathrm{T}}\right]\mathbf{A}^{\text{new}} + \mathbf{A}^{\text{new}}\mathbb{E}\left[\mathbf{z}_{n-1}\mathbf{z}_{n-1}^{\mathrm{T}}\right](\mathbf{A}^{\text{new}})^{\mathrm{T}}\right\}. \tag{13.114}$$

Note that $\mathbf{A}^{\text{new}}$ must be evaluated first, and the result can then be used to determine $\boldsymbol{\Gamma}^{\text{new}}$.

Finally, in order to determine the new values of $\mathbf{C}$ and $\boldsymbol{\Sigma}$, we substitute for $p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{C}, \boldsymbol{\Sigma})$ in (13.108) using (13.76) giving

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) = -\frac{N}{2}\ln|\boldsymbol{\Sigma}| - \mathbb{E}_{\mathbf{Z}|\boldsymbol{\theta}^{\text{old}}}\left[\frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n - \mathbf{C}\mathbf{z}_n)^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}_n - \mathbf{C}\mathbf{z}_n)\right] + \text{const}.$$

Exercise 13.34

Maximizing with respect to $\mathbf{C}$ and $\boldsymbol{\Sigma}$ then gives

$$\mathbf{C}^{\text{new}} = \left(\sum_{n=1}^{N}\mathbf{x}_n\mathbb{E}\left[\mathbf{z}_n^{\mathrm{T}}\right]\right)\left(\sum_{n=1}^{N}\mathbb{E}\left[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}\right]\right)^{-1} \tag{13.115}$$

$$\boldsymbol{\Sigma}^{\text{new}} = \frac{1}{N}\sum_{n=1}^{N}\left\{\mathbf{x}_n\mathbf{x}_n^{\mathrm{T}} - \mathbf{C}^{\text{new}}\mathbb{E}\left[\mathbf{z}_n\right]\mathbf{x}_n^{\mathrm{T}} - \mathbf{x}_n\mathbb{E}\left[\mathbf{z}_n^{\mathrm{T}}\right]\mathbf{C}^{\text{new}} + \mathbf{C}^{\text{new}}\mathbb{E}\left[\mathbf{z}_n\mathbf{z}_n^{\mathrm{T}}\right]\mathbf{C}^{\text{new}}\right\}. \tag{13.116}$$

We have approached parameter learning in the linear dynamical system using maximum likelihood. Inclusion of priors to give a MAP estimate is straightforward, and a fully Bayesian treatment can be found by applying the analytical approximation techniques discussed in Chapter 10, though a detailed treatment is precluded here due to lack of space.

13.3.3 Extensions of LDS

As with the hidden Markov model, there is considerable interest in extending the basic linear dynamical system in order to increase its capabilities. Although the assumption of a linear-Gaussian model leads to efficient algorithms for inference and learning, it also implies that the marginal distribution of the observed variables is simply a Gaussian, which represents a significant limitation. One simple extension of the linear dynamical system is to use a Gaussian mixture as the initial distribution for $\mathbf{z}_1$. If this mixture has K components, then the forward recursion equations (13.85) will lead to a mixture of K Gaussians over each hidden variable $\mathbf{z}_n$, and so the model is again tractable.

For many applications, the Gaussian emission density is a poor approximation. If instead we try to use a mixture of K Gaussians as the emission density, then the posterior $\widehat{\alpha}(\mathbf{z}_1)$ will also be a mixture of K Gaussians. However, from (13.85) the posterior $\widehat{\alpha}(\mathbf{z}_2)$ will comprise a mixture of K^2 Gaussians, and so on, with $\widehat{\alpha}(\mathbf{z}_n)$ being given by a mixture of K^n Gaussians. Thus the number of components grows exponentially with the length of the chain, and so this model is impractical.

More generally, introducing transition or emission models that depart from the linear-Gaussian (or other exponential family) model leads to an intractable inference problem. We can make deterministic approximations such as assumed density filtering or expectation propagation, or we can make use of sampling methods, as discussed in Section 13.3.4. One widely used approach is to make a Gaussian approximation by linearizing around the mean of the predicted distribution, which gives rise to the *extended Kalman filter* (Zarchan and Musoff, 2005).

Chapter 10

As with hidden Markov models, we can develop interesting extensions of the basic linear dynamical system by expanding its graphical representation. For example, the *switching state space model* (Ghahramani and Hinton, 1998) can be viewed as a combination of the hidden Markov model with a set of linear dynamical systems. The model has multiple Markov chains of continuous linear-Gaussian latent variables, each of which is analogous to the latent chain of the linear dynamical system discussed earlier, together with a Markov chain of discrete variables of the form used in a hidden Markov model. The output at each time step is determined by stochastically choosing one of the continuous latent chains, using the state of the discrete latent variable as a switch, and then emitting an observation from the corresponding conditional output distribution. Exact inference in this model is intractable, but variational methods lead to an efficient inference scheme involving forward-backward recursions along each of the continuous and discrete Markov chains independently. Note that, if we consider multiple chains of discrete latent variables, and use one as the switch to select from the remainder, we obtain an analogous model having only discrete latent variables known as the *switching hidden Markov model*.

13.3.4 Particle filters

Chapter 11

For dynamical systems which do not have a linear-Gaussian, for example, if they use a non-Gaussian emission density, we can turn to sampling methods in order to find a tractable inference algorithm. In particular, we can apply the sampling-importance-resampling formalism of Section 11.1.5 to obtain a sequential Monte Carlo algorithm known as the particle filter.

Consider the class of distributions represented by the graphical model in Figure 13.5, and suppose we are given the observed values $\mathbf{X}_n = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and we wish to draw L samples from the posterior distribution $p(\mathbf{z}_n|\mathbf{X}_n)$. Using Bayes' theorem, we have

$$\begin{aligned}
\mathbb{E}[f(\mathbf{z}_n)] &= \int f(\mathbf{z}_n)p(\mathbf{z}_n|\mathbf{X}_n)\,\mathrm{d}\mathbf{z}_n \\
&= \int f(\mathbf{z}_n)p(\mathbf{z}_n|\mathbf{x}_n, \mathbf{X}_{n-1})\,\mathrm{d}\mathbf{z}_n \\
&= \frac{\displaystyle\int f(\mathbf{z}_n)p(\mathbf{x}_n|\mathbf{z}_n)p(\mathbf{z}_n|\mathbf{X}_{n-1})\,\mathrm{d}\mathbf{z}_n}{\displaystyle\int p(\mathbf{x}_n|\mathbf{z}_n)p(\mathbf{z}_n|\mathbf{X}_{n-1})\,\mathrm{d}\mathbf{z}_n} \\
&\simeq \sum_{l=1}^{L} w_n^{(l)} f(\mathbf{z}_n^{(l)}) \qquad (13.117)
\end{aligned}$$

where $\{\mathbf{z}_n^{(l)}\}$ is a set of samples drawn from $p(\mathbf{z}_n|\mathbf{X}_{n-1})$ and we have made use of the conditional independence property $p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{X}_{n-1}) = p(\mathbf{x}_n|\mathbf{z}_n)$, which follows from the graph in Figure 13.5. The sampling weights $\{w_n^{(l)}\}$ are defined by

$$w_n^{(l)} = \frac{p(\mathbf{x}_n|\mathbf{z}_n^{(l)})}{\sum_{m=1}^{L} p(\mathbf{x}_n|\mathbf{z}_n^{(m)})} \qquad (13.118)$$

where the same samples are used in the numerator as in the denominator. Thus the posterior distribution $p(\mathbf{z}_n|\mathbf{x}_n)$ is represented by the set of samples $\{\mathbf{z}_n^{(l)}\}$ together with the corresponding weights $\{w_n^{(l)}\}$. Note that these weights satisfy $0 \leqslant w_n^{(l)} 1$ and $\sum_l w_n^{(l)} = 1$.

Because we wish to find a sequential sampling scheme, we shall suppose that a set of samples and weights have been obtained at time step n, and that we have subsequently observed the value of $\mathbf{x}_{n+1}$, and we wish to find the weights and samples at time step $n+1$. We first sample from the distribution $p(\mathbf{z}_{n+1}|\mathbf{X}_n)$. This is

straightforward since, again using Bayes' theorem

$$
\begin{aligned}
p(\mathbf{z}_{n+1}|\mathbf{X}_n) &= \int p(\mathbf{z}_{n+1}|\mathbf{z}_n, \mathbf{X}_n) p(\mathbf{z}_n|\mathbf{X}_n)\, \mathrm{d}\mathbf{z}_n \\
&= \int p(\mathbf{z}_{n+1}|\mathbf{z}_n) p(\mathbf{z}_n|\mathbf{X}_n)\, \mathrm{d}\mathbf{z}_n \\
&= \int p(\mathbf{z}_{n+1}|\mathbf{z}_n) p(\mathbf{z}_n|\mathbf{x}_n, \mathbf{X}_{n-1})\, \mathrm{d}\mathbf{z}_n \\
&= \frac{\displaystyle\int p(\mathbf{z}_{n+1}|\mathbf{z}_n) p(\mathbf{x}_n|\mathbf{z}_n) p(\mathbf{z}_n|\mathbf{X}_{n-1})\, \mathrm{d}\mathbf{z}_n}{\displaystyle\int p(\mathbf{x}_n|\mathbf{z}_n) p(\mathbf{z}_n|\mathbf{X}_{n-1})\, \mathrm{d}\mathbf{z}_n} \\
&= \sum_l w_n^{(l)} p(\mathbf{z}_{n+1}|\mathbf{z}_n^{(l)})
\end{aligned}
\tag{13.119}
$$

where we have made use of the conditional independence properties

$$
p(\mathbf{z}_{n+1}|\mathbf{z}_n, \mathbf{X}_n) = p(\mathbf{z}_{n+1}|\mathbf{z}_n) \tag{13.120}
$$

$$
p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{X}_{n-1}) = p(\mathbf{x}_n|\mathbf{z}_n) \tag{13.121}
$$

which follow from the application of the d-separation criterion to the graph in Figure 13.5. The distribution given by (13.119) is a mixture distribution, and samples can be drawn by choosing a component l with probability given by the mixing coefficients $w^{(l)}$ and then drawing a sample from the corresponding component.

In summary, we can view each step of the particle filter algorithm as comprising two stages. At time step n, we have a sample representation of the posterior distribution $p(\mathbf{z}_n|\mathbf{X}_n)$ expressed as samples $\{\mathbf{z}_n^{(l)}\}$ with corresponding weights $\{w_n^{(l)}\}$. This can be viewed as a mixture representation of the form (13.119). To obtain the corresponding representation for the next time step, we first draw L samples from the mixture distribution (13.119), and then for each sample we use the new observation $\mathbf{x}_{n+1}$ to evaluate the corresponding weights $w_{n+1}^{(l)} \propto p(\mathbf{x}_{n+1}|\mathbf{z}_{n+1}^{(l)})$. This is illustrated, for the case of a single variable z, in Figure 13.23.

The particle filtering, or sequential Monte Carlo, approach has appeared in the literature under various names including the *bootstrap filter* (Gordon *et al.*, 1993), *survival of the fittest* (Kanazawa *et al.*, 1995), and the *condensation* algorithm (Isard and Blake, 1998).

Exercises

13.1 ($\star$) **WWW** Use the technique of d-separation, discussed in Section 8.2, to verify that the Markov model shown in Figure 13.3 having N nodes in total satisfies the conditional independence properties (13.3) for $n = 2, \ldots, N$. Similarly, show that a model described by the graph in Figure 13.4 in which there are N nodes in total

Figure 13.23 Schematic illustration of the operation of the particle filter for a one-dimensional latent space. At time step n, the posterior $p(z_n|\mathbf{x}_n)$ is represented as a mixture distribution, shown schematically as circles whose sizes are proportional to the weights $w_n^{(l)}$. A set of L samples is then drawn from this distribution and the new weights $w_{n+1}^{(l)}$ evaluated using $p(\mathbf{x}_{n+1}|\mathbf{z}_{n+1}^{(l)})$.

satisfies the conditional independence properties

$$p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}) = p(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{x}_{n-2}) \tag{13.122}$$

for $n = 3, \ldots, N$.

13.2 ($\star\star$) Consider the joint probability distribution (13.2) corresponding to the directed graph of Figure 13.3. Using the sum and product rules of probability, verify that this joint distribution satisfies the conditional independence property (13.3) for $n = 2, \ldots, N$. Similarly, show that the second-order Markov model described by the joint distribution (13.4) satisfies the conditional independence property

$$p(\mathbf{x}_n|\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}) = p(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{x}_{n-2}) \tag{13.123}$$

for $n = 3, \ldots, N$.

13.3 ($\star$) By using d-separation, show that the distribution $p(\mathbf{x}_1, \ldots, \mathbf{x}_N)$ of the observed data for the state space model represented by the directed graph in Figure 13.5 does not satisfy any conditional independence properties and hence does not exhibit the Markov property at any finite order.

13.4 ($\star\star$) **www** Consider a hidden Markov model in which the emission densities are represented by a parametric model $p(\mathbf{x}|\mathbf{z}, \mathbf{w})$, such as a linear regression model or a neural network, in which $\mathbf{w}$ is a vector of adaptive parameters. Describe how the parameters $\mathbf{w}$ can be learned from data using maximum likelihood.

13.5 ($\star\star$) Verify the M-step equations (13.18) and (13.19) for the initial state probabilities and transition probability parameters of the hidden Markov model by maximization of the expected complete-data log likelihood function (13.17), using appropriate Lagrange multipliers to enforce the summation constraints on the components of $\boldsymbol{\pi}$ and $\mathbf{A}$.

13.6 ($\star$) Show that if any elements of the parameters $\boldsymbol{\pi}$ or $\mathbf{A}$ for a hidden Markov model are initially set to zero, then those elements will remain zero in all subsequent updates of the EM algorithm.

13.7 ($\star$) Consider a hidden Markov model with Gaussian emission densities. Show that maximization of the function $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$ with respect to the mean and covariance parameters of the Gaussians gives rise to the M-step equations (13.20) and (13.21).

13.8 ($\star\star$) **www** For a hidden Markov model having discrete observations governed by a multinomial distribution, show that the conditional distribution of the observations given the hidden variables is given by (13.22) and the corresponding M step equations are given by (13.23). Write down the analogous equations for the conditional distribution and the M step equations for the case of a hidden Markov with multiple binary output variables each of which is governed by a Bernoulli conditional distribution. Hint: refer to Sections 2.1 and 2.2 for a discussion of the corresponding maximum likelihood solutions for i.i.d. data if required.

13.9 ($\star\star$) **www** Use the d-separation criterion to verify that the conditional independence properties (13.24)–(13.31) are satisfied by the joint distribution for the hidden Markov model defined by (13.6).

13.10 ($\star\star\star$) By applying the sum and product rules of probability, verify that the conditional independence properties (13.24)–(13.31) are satisfied by the joint distribution for the hidden Markov model defined by (13.6).

13.11 ($\star\star$) Starting from the expression (8.72) for the marginal distribution over the variables of a factor in a factor graph, together with the results for the messages in the sum-product algorithm obtained in Section 13.2.3, derive the result (13.43) for the joint posterior distribution over two successive latent variables in a hidden Markov model.

13.12 ($\star\star$) Suppose we wish to train a hidden Markov model by maximum likelihood using data that comprises R independent sequences of observations, which we denote by $\mathbf{X}^{(r)}$ where $r = 1, \ldots, R$. Show that in the E step of the EM algorithm, we simply evaluate posterior probabilities for the latent variables by running the α and β recursions independently for each of the sequences. Also show that in the M step, the initial probability and transition probability parameters are re-estimated

using modified forms of (13.18) and (13.19) given by

$$\pi_k = \frac{\sum_{r=1}^{R} \gamma(z_{1k}^{(r)})}{\sum_{r=1}^{R}\sum_{j=1}^{K} \gamma(z_{1j}^{(r)})} \tag{13.124}$$

$$A_{jk} = \frac{\sum_{r=1}^{R}\sum_{n=2}^{N} \xi(z_{n-1,j}^{(r)}, z_{n,k}^{(r)})}{\sum_{r=1}^{R}\sum_{l=1}^{K}\sum_{n=2}^{N} \xi(z_{n-1,j}^{(r)}, z_{n,l}^{(r)})} \tag{13.125}$$

where, for notational convenience, we have assumed that the sequences are of the same length (the generalization to sequences of different lengths is straightforward). Similarly, show that the M-step equation for re-estimation of the means of Gaussian emission models is given by

$$\boldsymbol{\mu}_k = \frac{\sum_{r=1}^{R}\sum_{n=1}^{N} \gamma(z_{nk}^{(r)})\mathbf{x}_n^{(r)}}{\sum_{r=1}^{R}\sum_{n=1}^{N} \gamma(z_{nk}^{(r)})}. \tag{13.126}$$

Note that the M-step equations for other emission model parameters and distributions take an analogous form.

13.13 ($\star\star$) **WWW** Use the definition (8.64) of the messages passed from a factor node to a variable node in a factor graph, together with the expression (13.6) for the joint distribution in a hidden Markov model, to show that the definition (13.50) of the alpha message is the same as the definition (13.34).

13.14 ($\star\star$) Use the definition (8.67) of the messages passed from a factor node to a variable node in a factor graph, together with the expression (13.6) for the joint distribution in a hidden Markov model, to show that the definition (13.52) of the beta message is the same as the definition (13.35).

13.15 ($\star\star$) Use the expressions (13.33) and (13.43) for the marginals in a hidden Markov model to derive the corresponding results (13.64) and (13.65) expressed in terms of re-scaled variables.

13.16 ($\star\star\star$) In this exercise, we derive the forward message passing equation for the Viterbi algorithm directly from the expression (13.6) for the joint distribution. This involves maximizing over all of the hidden variables $\mathbf{z}_1, \ldots, \mathbf{z}_N$. By taking the logarithm and then exchanging maximizations and summations, derive the recursion

(13.68) where the quantities $\omega(\mathbf{z}_n)$ are defined by (13.70). Show that the initial condition for this recursion is given by (13.69).

13.17 ($\star$) **www** Show that the directed graph for the input-output hidden Markov model, given in Figure 13.18, can be expressed as a tree-structured factor graph of the form shown in Figure 13.15 and write down expressions for the initial factor $h(\mathbf{z}_1)$ and for the general factor $f_n(\mathbf{z}_{n-1}, \mathbf{z}_n)$ where $2 \leqslant n \leqslant N$.

13.18 ($\star\star\star$) Using the result of Exercise 13.17, derive the recursion equations, including the initial conditions, for the forward-backward algorithm for the input-output hidden Markov model shown in Figure 13.18.

13.19 ($\star$) **www** The Kalman filter and smoother equations allow the posterior distributions over individual latent variables, conditioned on all of the observed variables, to be found efficiently for linear dynamical systems. Show that the sequence of latent variable values obtained by maximizing each of these posterior distributions individually is the same as the most probable sequence of latent values. To do this, simply note that the joint distribution of all latent and observed variables in a linear dynamical system is Gaussian, and hence all conditionals and marginals will also be Gaussian, and then make use of the result (2.98).

13.20 ($\star\star$) **www** Use the result (2.115) to prove (13.87).

13.21 ($\star\star$) Use the results (2.115) and (2.116), together with the matrix identities (C.5) and (C.7), to derive the results (13.89), (13.90), and (13.91), where the Kalman gain matrix $\mathbf{K}_n$ is defined by (13.92).

13.22 ($\star\star$) **www** Using (13.93), together with the definitions (13.76) and (13.77) and the result (2.115), derive (13.96).

13.23 ($\star\star$) Using (13.93), together with the definitions (13.76) and (13.77) and the result (2.116), derive (13.94), (13.95) and (13.97).

13.24 ($\star\star$) **www** Consider a generalization of (13.75) and (13.76) in which we include constant terms $\mathbf{a}$ and $\mathbf{c}$ in the Gaussian means, so that

$$p(\mathbf{z}_n|\mathbf{z}_{n-1}) = \mathcal{N}(\mathbf{z}_n|\mathbf{A}\mathbf{z}_{n-1} + \mathbf{a}, \mathbf{\Gamma}) \tag{13.127}$$

$$p(\mathbf{x}_n|\mathbf{z}_n) = \mathcal{N}(\mathbf{x}_n|\mathbf{C}\mathbf{z}_n + \mathbf{c}, \mathbf{\Sigma}). \tag{13.128}$$

Show that this extension can be re-case in the framework discussed in this chapter by defining a state vector $\mathbf{z}$ with an additional component fixed at unity, and then augmenting the matrices $\mathbf{A}$ and $\mathbf{C}$ using extra columns corresponding to the parameters $\mathbf{a}$ and $\mathbf{c}$.

13.25 ($\star\star$) In this exercise, we show that when the Kalman filter equations are applied to independent observations, they reduce to the results given in Section 2.3 for the maximum likelihood solution for a single Gaussian distribution. Consider the problem of finding the mean μ of a single Gaussian random variable x, in which we are given a set of independent observations $\{x_1, \ldots, x_N\}$. To model this we can use

a linear dynamical system governed by (13.75) and (13.76), with latent variables $\{z_1, \ldots, z_N\}$ in which $\mathbf{C}$ becomes the identity matrix and where the transition probability $\mathbf{A} = \mathbf{0}$ because the observations are independent. Let the parameters $\mathbf{m}_0$ and $\mathbf{V}_0$ of the initial state be denoted by μ_0 and σ_0^2, respectively, and suppose that $\boldsymbol{\Sigma}$ becomes σ^2. Write down the corresponding Kalman filter equations starting from the general results (13.89) and (13.90), together with (13.94) and (13.95). Show that these are equivalent to the results (2.141) and (2.142) obtained directly by considering independent data.

13.26 ($\star\star\star$) Consider a special case of the linear dynamical system of Section 13.3 that is equivalent to probabilistic PCA, so that the transition matrix $\mathbf{A} = \mathbf{0}$, the covariance $\boldsymbol{\Gamma} = \mathbf{I}$, and the noise covariance $\boldsymbol{\Sigma} = \sigma^2\mathbf{I}$. By making use of the matrix inversion identity (C.7) show that, if the emission density matrix $\mathbf{C}$ is denoted $\mathbf{W}$, then the posterior distribution over the hidden states defined by (13.89) and (13.90) reduces to the result (12.42) for probabilistic PCA.

13.27 ($\star$) **www** Consider a linear dynamical system of the form discussed in Section 13.3 in which the amplitude of the observation noise goes to zero, so that $\boldsymbol{\Sigma} = \mathbf{0}$. Show that the posterior distribution for $\mathbf{z}_n$ has mean $\mathbf{x}_n$ and zero variance. This accords with our intuition that if there is no noise, we should just use the current observation $\mathbf{x}_n$ to estimate the state variable $\mathbf{z}_n$ and ignore all previous observations.

13.28 ($\star\star\star$) Consider a special case of the linear dynamical system of Section 13.3 in which the state variable $\mathbf{z}_n$ is constrained to be equal to the previous state variable, which corresponds to $\mathbf{A} = \mathbf{I}$ and $\boldsymbol{\Gamma} = \mathbf{0}$. For simplicity, assume also that $\mathbf{V}_0 \to \infty$ so that the initial conditions for $\mathbf{z}$ are unimportant, and the predictions are determined purely by the data. Use proof by induction to show that the posterior mean for state $\mathbf{z}_n$ is determined by the average of $\mathbf{x}_1, \ldots, \mathbf{x}_n$. This corresponds to the intuitive result that if the state variable is constant, our best estimate is obtained by averaging the observations.

13.29 ($\star\star\star$) Starting from the backwards recursion equation (13.99), derive the RTS smoothing equations (13.100) and (13.101) for the Gaussian linear dynamical system.

13.30 ($\star\star$) Starting from the result (13.65) for the pairwise posterior marginal in a state space model, derive the specific form (13.103) for the case of the Gaussian linear dynamical system.

13.31 ($\star\star$) Starting from the result (13.103) and by substituting for $\widehat{\alpha}(\mathbf{z}_n)$ using (13.84), verify the result (13.104) for the covariance between $\mathbf{z}_n$ and $\mathbf{z}_{n-1}$.

13.32 ($\star\star$) **www** Verify the results (13.110) and (13.111) for the M-step equations for $\boldsymbol{\mu}_0$ and $\mathbf{V}_0$ in the linear dynamical system.

13.33 ($\star\star$) Verify the results (13.113) and (13.114) for the M-step equations for $\mathbf{A}$ and $\boldsymbol{\Gamma}$ in the linear dynamical system.

13.34 ($\star\star$) Verify the results (13.115) and (13.116) for the M-step equations for $\mathbf{C}$ and $\boldsymbol{\Sigma}$ in the linear dynamical system.

14

Combining Models

In earlier chapters, we have explored a range of different models for solving classification and regression problems. It is often found that improved performance can be obtained by combining multiple models together in some way, instead of just using a single model in isolation. For instance, we might train L different models and then make predictions using the average of the predictions made by each model. Such combinations of models are sometimes called *committees*. In Section 14.2, we discuss ways to apply the committee concept in practice, and we also give some insight into why it can sometimes be an effective procedure.

One important variant of the committee method, known as *boosting*, involves training multiple models in sequence in which the error function used to train a particular model depends on the performance of the previous models. This can produce substantial improvements in performance compared to the use of a single model and is discussed in Section 14.3.

Instead of averaging the predictions of a set of models, an alternative form of

model combination is to select one of the models to make the prediction, in which the choice of model is a function of the input variables. Thus different models become responsible for making predictions in different regions of input space. One widely used framework of this kind is known as a *decision tree* in which the selection process can be described as a sequence of binary selections corresponding to the traversal of a tree structure and is discussed in Section 14.4. In this case, the individual models are generally chosen to be very simple, and the overall flexibility of the model arises from the input-dependent selection process. Decision trees can be applied to both classification and regression problems.

One limitation of decision trees is that the division of input space is based on hard splits in which only one model is responsible for making predictions for any given value of the input variables. The decision process can be softened by moving to a probabilistic framework for combining models, as discussed in Section 14.5. For example, if we have a set of K models for a conditional distribution $p(t|\mathbf{x}, k)$ where $\mathbf{x}$ is the input variable, t is the target variable, and $k = 1, \ldots, K$ indexes the model, then we can form a probabilistic mixture of the form

$$p(t|\mathbf{x}) = \sum_{k=1}^{K} \pi_k(\mathbf{x}) p(t|\mathbf{x}, k) \tag{14.1}$$

in which $\pi_k(\mathbf{x}) = p(k|\mathbf{x})$ represent the input-dependent mixing coefficients. Such models can be viewed as mixture distributions in which the component densities, as well as the mixing coefficients, are conditioned on the input variables and are known as *mixtures of experts*. They are closely related to the mixture density network model discussed in Section 5.6.

14.1. Bayesian Model Averaging

It is important to distinguish between model combination methods and Bayesian model averaging, as the two are often confused. To understand the difference, consider the example of density estimation using a mixture of Gaussians in which several Gaussian components are combined probabilistically. The model contains a binary latent variable $\mathbf{z}$ that indicates which component of the mixture is responsible for generating the corresponding data point. Thus the model is specified in terms of a joint distribution

Section 9.2

$$p(\mathbf{x}, \mathbf{z}) \tag{14.2}$$

and the corresponding density over the observed variable $\mathbf{x}$ is obtained by marginalizing over the latent variable

$$p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{x}, \mathbf{z}). \tag{14.3}$$

In the case of our Gaussian mixture example, this leads to a distribution of the form

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{14.4}$$

with the usual interpretation of the symbols. This is an example of model combination. For independent, identically distributed data, we can use (14.3) to write the marginal probability of a data set $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ in the form

$$p(\mathbf{X}) = \prod_{n=1}^{N} p(\mathbf{x}_n) = \prod_{n=1}^{N} \left[\sum_{\mathbf{z}_n} p(\mathbf{x}_n, \mathbf{z}_n) \right]. \tag{14.5}$$

Thus we see that each observed data point $\mathbf{x}_n$ has a corresponding latent variable $\mathbf{z}_n$.

Now suppose we have several different models indexed by $h = 1, \ldots, H$ with prior probabilities $p(h)$. For instance one model might be a mixture of Gaussians and another model might be a mixture of Cauchy distributions. The marginal distribution over the data set is given by

$$p(\mathbf{X}) = \sum_{h=1}^{H} p(\mathbf{X}|h)p(h). \tag{14.6}$$

This is an example of Bayesian model averaging. The interpretation of this summation over h is that just one model is responsible for generating the whole data set, and the probability distribution over h simply reflects our uncertainty as to which model that is. As the size of the data set increases, this uncertainty reduces, and the posterior probabilities $p(h|\mathbf{X})$ become increasingly focussed on just one of the models.

This highlights the key difference between Bayesian model averaging and model combination, because in Bayesian model averaging the whole data set is generated by a single model. By contrast, when we combine multiple models, as in (14.5), we see that different data points within the data set can potentially be generated from different values of the latent variable $\mathbf{z}$ and hence by different components.

Although we have considered the marginal probability $p(\mathbf{X})$, the same considerations apply for the predictive density $p(\mathbf{x}|\mathbf{X})$ or for conditional distributions such as $p(\mathbf{t}|\mathbf{x}, \mathbf{X}, \mathbf{T})$.

Exercise 14.1

14.2. Committees

The simplest way to construct a committee is to average the predictions of a set of individual models. Such a procedure can be motivated from a frequentist perspective by considering the trade-off between bias and variance, which decomposes the error due to a model into the bias component that arises from differences between the model and the true function to be predicted, and the variance component that represents the sensitivity of the model to the individual data points. Recall from Figure 3.5

Section 3.2

that when we trained multiple polynomials using the sinusoidal data, and then averaged the resulting functions, the contribution arising from the variance term tended to cancel, leading to improved predictions. When we averaged a set of low-bias models (corresponding to higher order polynomials), we obtained accurate predictions for the underlying sinusoidal function from which the data were generated.

In practice, of course, we have only a single data set, and so we have to find a way to introduce variability between the different models within the committee. One approach is to use *bootstrap* data sets, discussed in Section 1.2.3. Consider a regression problem in which we are trying to predict the value of a single continuous variable, and suppose we generate M bootstrap data sets and then use each to train a separate copy $y_m(\mathbf{x})$ of a predictive model where $m = 1, \ldots, M$. The committee prediction is given by

$$y_{\mathrm{COM}}(\mathbf{x}) = \frac{1}{M} \sum_{m=1}^{M} y_m(\mathbf{x}). \tag{14.7}$$

This procedure is known as bootstrap aggregation or *bagging* (Breiman, 1996).

Suppose the true regression function that we are trying to predict is given by $h(\mathbf{x})$, so that the output of each of the models can be written as the true value plus an error in the form

$$y_m(\mathbf{x}) = h(\mathbf{x}) + \epsilon_m(\mathbf{x}). \tag{14.8}$$

The average sum-of-squares error then takes the form

$$\mathbb{E}_{\mathbf{x}}\left[\{y_m(\mathbf{x}) - h(\mathbf{x})\}^2\right] = \mathbb{E}_{\mathbf{x}}\left[\epsilon_m(\mathbf{x})^2\right] \tag{14.9}$$

where $\mathbb{E}_{\mathbf{x}}[\cdot]$ denotes a frequentist expectation with respect to the distribution of the input vector $\mathbf{x}$. The average error made by the models acting individually is therefore

$$E_{\mathrm{AV}} = \frac{1}{M} \sum_{m=1}^{M} \mathbb{E}_{\mathbf{x}}\left[\epsilon_m(\mathbf{x})^2\right]. \tag{14.10}$$

Similarly, the expected error from the committee (14.7) is given by

$$\begin{aligned} E_{\mathrm{COM}} &= \mathbb{E}_{\mathbf{x}}\left[\left\{\frac{1}{M}\sum_{m=1}^{M} y_m(\mathbf{x}) - h(\mathbf{x})\right\}^2\right] \\ &= \mathbb{E}_{\mathbf{x}}\left[\left\{\frac{1}{M}\sum_{m=1}^{M} \epsilon_m(\mathbf{x})\right\}^2\right] \end{aligned} \tag{14.11}$$

If we assume that the errors have zero mean and are uncorrelated, so that

$$\mathbb{E}_{\mathbf{x}}\left[\epsilon_m(\mathbf{x})\right] = 0 \tag{14.12}$$

$$\mathbb{E}_{\mathbf{x}}\left[\epsilon_m(\mathbf{x})\epsilon_l(\mathbf{x})\right] = 0, \qquad m \neq l \tag{14.13}$$

Exercise 14.2 then we obtain

$$E_{\mathrm{COM}} = \frac{1}{M} E_{\mathrm{AV}}. \tag{14.14}$$

This apparently dramatic result suggests that the average error of a model can be reduced by a factor of M simply by averaging M versions of the model. Unfortunately, it depends on the key assumption that the errors due to the individual models are uncorrelated. In practice, the errors are typically highly correlated, and the reduction in overall error is generally small. It can, however, be shown that the expected committee error will not exceed the expected error of the constituent models, so that *Exercise 14.3* $E_{\mathrm{COM}} \leqslant E_{\mathrm{AV}}$. In order to achieve more significant improvements, we turn to a more sophisticated technique for building committees, known as boosting.

14.3. Boosting

Boosting is a powerful technique for combining multiple 'base' classifiers to produce a form of committee whose performance can be significantly better than that of any of the base classifiers. Here we describe the most widely used form of boosting algorithm called *AdaBoost*, short for 'adaptive boosting', developed by Freund and Schapire (1996). Boosting can give good results even if the base classifiers have a performance that is only slightly better than random, and hence sometimes the base classifiers are known as *weak learners*. Originally designed for solving classification problems, boosting can also be extended to regression (Friedman, 2001).

The principal difference between boosting and the committee methods such as bagging discussed above, is that the base classifiers are trained in sequence, and each base classifier is trained using a weighted form of the data set in which the weighting coefficient associated with each data point depends on the performance of the previous classifiers. In particular, points that are misclassified by one of the base classifiers are given greater weight when used to train the next classifier in the sequence. Once all the classifiers have been trained, their predictions are then combined through a weighted majority voting scheme, as illustrated schematically in Figure 14.1.

Consider a two-class classification problem, in which the training data comprises input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ along with corresponding binary target variables $t_1, \ldots, t_N$ where $t_n \in \{-1, 1\}$. Each data point is given an associated weighting parameter w_n, which is initially set $1/N$ for all data points. We shall suppose that we have a procedure available for training a base classifier using weighted data to give a function $y(\mathbf{x}) \in \{-1, 1\}$. At each stage of the algorithm, AdaBoost trains a new classifier using a data set in which the weighting coefficients are adjusted according to the performance of the previously trained classifier so as to give greater weight to the misclassified data points. Finally, when the desired number of base classifiers have been trained, they are combined to form a committee using coefficients that give different weight to different base classifiers. The precise form of the AdaBoost algorithm is given below.

Figure 14.1 Schematic illustration of the boosting framework. Each base classifier $y_m(\mathbf{x})$ is trained on a weighted form of the training set (blue arrows) in which the weights $w_n^{(m)}$ depend on the performance of the previous base classifier $y_{m-1}(\mathbf{x})$ (green arrows). Once all base classifiers have been trained, they are combined to give the final classifier $Y_M(\mathbf{x})$ (red arrows).

AdaBoost

1. Initialize the data weighting coefficients $\{w_n\}$ by setting $w_n^{(1)} = 1/N$ for $n = 1, \ldots, N$.
2. For $m = 1, \ldots, M$:
   (a) Fit a classifier $y_m(\mathbf{x})$ to the training data by minimizing the weighted error function

$$J_m = \sum_{n=1}^{N} w_n^{(m)} I(y_m(\mathbf{x}_n) \neq t_n) \tag{14.15}$$

   where $I(y_m(\mathbf{x}_n) \neq t_n)$ is the indicator function and equals 1 when $y_m(\mathbf{x}_n) \neq t_n$ and 0 otherwise.

   (b) Evaluate the quantities

$$\epsilon_m = \frac{\sum_{n=1}^{N} w_n^{(m)} I(y_m(\mathbf{x}_n) \neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}} \tag{14.16}$$

   and then use these to evaluate

$$\alpha_m = \ln\left\{\frac{1-\epsilon_m}{\epsilon_m}\right\}. \tag{14.17}$$

   (c) Update the data weighting coefficients

$$w_n^{(m+1)} = w_n^{(m)} \exp\left\{\alpha_m I(y_m(\mathbf{x}_n) \neq t_n)\right\} \tag{14.18}$$

3. Make predictions using the final model, which is given by

$$Y_M(\mathbf{x}) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m y_m(\mathbf{x})\right). \tag{14.19}$$

We see that the first base classifier $y_1(\mathbf{x})$ is trained using weighting coefficients $w_n^{(1)}$ that are all equal, which therefore corresponds to the usual procedure for training a single classifier. From (14.18), we see that in subsequent iterations the weighting coefficients $w_n^{(m)}$ are increased for data points that are misclassified and decreased for data points that are correctly classified. Successive classifiers are therefore forced to place greater emphasis on points that have been misclassified by previous classifiers, and data points that continue to be misclassified by successive classifiers receive ever greater weight. The quantities ϵ_m represent weighted measures of the error rates of each of the base classifiers on the data set. We therefore see that the weighting coefficients α_m defined by (14.17) give greater weight to the more accurate classifiers when computing the overall output given by (14.19).

The AdaBoost algorithm is illustrated in Figure 14.2, using a subset of 30 data points taken from the toy classification data set shown in Figure A.7. Here each base learners consists of a threshold on one of the input variables. This simple classifier corresponds to a form of decision tree known as a 'decision stumps', i.e., a decision tree with a single node. Thus each base learner classifies an input according to whether one of the input features exceeds some threshold and therefore simply partitions the space into two regions separated by a linear decision surface that is parallel to one of the axes.

Section 14.4

14.3.1 Minimizing exponential error

Boosting was originally motivated using statistical learning theory, leading to upper bounds on the generalization error. However, these bounds turn out to be too loose to have practical value, and the actual performance of boosting is much better than the bounds alone would suggest. Friedman *et al.* (2000) gave a different and very simple interpretation of boosting in terms of the sequential minimization of an exponential error function.

Consider the exponential error function defined by

$$E = \sum_{n=1}^{N} \exp\{-t_n f_m(\mathbf{x}_n)\} \tag{14.20}$$

where $f_m(\mathbf{x})$ is a classifier defined in terms of a linear combination of base classifiers $y_l(\mathbf{x})$ of the form

$$f_m(\mathbf{x}) = \frac{1}{2}\sum_{l=1}^{m} \alpha_l y_l(\mathbf{x}) \tag{14.21}$$

and $t_n \in \{-1, 1\}$ are the training set target values. Our goal is to minimize E with respect to both the weighting coefficients α_l and the parameters of the base classifiers $y_l(\mathbf{x})$.

Figure 14.2 Illustration of boosting in which the base learners consist of simple thresholds applied to one or other of the axes. Each figure shows the number m of base learners trained so far, along with the decision boundary of the most recent base learner (dashed black line) and the combined decision boundary of the ensemble (solid green line). Each data point is depicted by a circle whose radius indicates the weight assigned to that data point when training the most recently added base learner. Thus, for instance, we see that points that are misclassified by the $m = 1$ base learner are given greater weight when training the $m = 2$ base learner.

Instead of doing a global error function minimization, however, we shall suppose that the base classifiers $y_1(\mathbf{x}), \ldots, y_{m-1}(\mathbf{x})$ are fixed, as are their coefficients $\alpha_1, \ldots, \alpha_{m-1}$, and so we are minimizing only with respect to α_m and $y_m(\mathbf{x})$. Separating off the contribution from base classifier $y_m(\mathbf{x})$, we can then write the error function in the form

$$\begin{aligned} E &= \sum_{n=1}^{N} \exp\left\{-t_n f_{m-1}(\mathbf{x}_n) - \frac{1}{2} t_n \alpha_m y_m(\mathbf{x}_n)\right\} \\ &= \sum_{n=1}^{N} w_n^{(m)} \exp\left\{-\frac{1}{2} t_n \alpha_m y_m(\mathbf{x}_n)\right\} \end{aligned} \tag{14.22}$$

where the coefficients $w_n^{(m)} = \exp\{-t_n f_{m-1}(\mathbf{x}_n)\}$ can be viewed as constants because we are optimizing only α_m and $y_m(\mathbf{x})$. If we denote by $\mathcal{T}_m$ the set of data points that are correctly classified by $y_m(\mathbf{x})$, and if we denote the remaining misclassified points by $\mathcal{M}_m$, then we can in turn rewrite the error function in the

form

$$\begin{aligned} E &= e^{-\alpha_m/2} \sum_{n \in \mathcal{T}_m} w_n^{(m)} + e^{\alpha_m/2} \sum_{n \in \mathcal{M}_m} w_n^{(m)} \\ &= (e^{\alpha_m/2} - e^{-\alpha_m/2}) \sum_{n=1}^{N} w_n^{(m)} I(y_m(\mathbf{x}_n) \neq t_n) + e^{-\alpha_m/2} \sum_{n=1}^{N} w_n^{(m)}. \end{aligned} \tag{14.23}$$

When we minimize this with respect to $y_m(\mathbf{x})$, we see that the second term is constant, and so this is equivalent to minimizing (14.15) because the overall multiplicative factor in front of the summation does not affect the location of the minimum. Similarly, minimizing with respect to α_m, we obtain (14.17) in which ϵ_m is defined by (14.16).

Exercise 14.6

From (14.22) we see that, having found α_m and $y_m(\mathbf{x})$, the weights on the data points are updated using

$$w_n^{(m+1)} = w_n^{(m)} \exp\left\{-\frac{1}{2} t_n \alpha_m y_m(\mathbf{x}_n)\right\}. \tag{14.24}$$

Making use of the fact that

$$t_n y_m(\mathbf{x}_n) = 1 - 2I(y_m(\mathbf{x}_n) \neq t_n) \tag{14.25}$$

we see that the weights $w_n^{(m)}$ are updated at the next iteration using

$$w_n^{(m+1)} = w_n^{(m)} \exp(-\alpha_m/2) \exp\left\{\alpha_m I(y_m(\mathbf{x}_n) \neq t_n)\right\}. \tag{14.26}$$

Because the term $\exp(-\alpha_m/2)$ is independent of n, we see that it weights all data points by the same factor and so can be discarded. Thus we obtain (14.18).

Finally, once all the base classifiers are trained, new data points are classified by evaluating the sign of the combined function defined according to (14.21). Because the factor of $1/2$ does not affect the sign it can be omitted, giving (14.19).

14.3.2 Error functions for boosting

The exponential error function that is minimized by the AdaBoost algorithm differs from those considered in previous chapters. To gain some insight into the nature of the exponential error function, we first consider the expected error given by

$$\mathbb{E}_{\mathbf{x},t}\left[\exp\{-ty(\mathbf{x})\}\right] = \sum_t \int \exp\{-ty(\mathbf{x})\} p(t|\mathbf{x}) p(\mathbf{x}) \, \mathrm{d}\mathbf{x}. \tag{14.27}$$

If we perform a variational minimization with respect to all possible functions $y(\mathbf{x})$, we obtain

Exercise 14.7

$$y(\mathbf{x}) = \frac{1}{2} \ln\left\{\frac{p(t=1|\mathbf{x})}{p(t=-1|\mathbf{x})}\right\} \tag{14.28}$$

Figure 14.3 Plot of the exponential (green) and rescaled cross-entropy (red) error functions along with the hinge error (blue) used in support vector machines, and the misclassification error (black). Note that for large negative values of $z = ty(\mathbf{x})$, the cross-entropy gives a linearly increasing penalty, whereas the exponential loss gives an exponentially increasing penalty.

which is half the log-odds. Thus the AdaBoost algorithm is seeking the best approximation to the log odds ratio, within the space of functions represented by the linear combination of base classifiers, subject to the constrained minimization resulting from the sequential optimization strategy. This result motivates the use of the sign function in (14.19) to arrive at the final classification decision.

We have already seen that the minimizer $y(\mathbf{x})$ of the cross-entropy error (4.90) for two-class classification is given by the posterior class probability. In the case of a target variable $t \in \{-1, 1\}$, we have seen that the error function is given by $\ln(1 + \exp(-yt))$. This is compared with the exponential error function in Figure 14.3, where we have divided the cross-entropy error by a constant factor $\ln(2)$ so that it passes through the point $(0, 1)$ for ease of comparison. We see that both can be seen as continuous approximations to the ideal misclassification error function. An advantage of the exponential error is that its sequential minimization leads to the simple AdaBoost scheme. One drawback, however, is that it penalizes large negative values of $ty(\mathbf{x})$ much more strongly than cross-entropy. In particular, we see that for large negative values of ty, the cross-entropy grows linearly with $|ty|$, whereas the exponential error function grows exponentially with $|ty|$. Thus the exponential error function will be much less robust to outliers or misclassified data points. Another important difference between cross-entropy and the exponential error function is that the latter cannot be interpreted as the log likelihood function of any well-defined probabilistic model. Furthermore, the exponential error does not generalize to classification problems having $K > 2$ classes, again in contrast to the cross-entropy for a probabilistic model, which is easily generalized to give (4.108).

Section 7.1.2

Exercise 14.8

Section 4.3.4

The interpretation of boosting as the sequential optimization of an additive model under an exponential error (Friedman *et al.*, 2000) opens the door to a wide range of boosting-like algorithms, including multiclass extensions, by altering the choice of error function. It also motivates the extension to regression problems (Friedman, 2001). If we consider a sum-of-squares error function for regression, then sequential minimization of an additive model of the form (14.21) simply involves fitting each new base classifier to the residual errors $t_n - f_{m-1}(\mathbf{x}_n)$ from the previous model. As we have noted, however, the sum-of-squares error is not robust to outliers, and this

Exercise 14.9

Figure 14.4 Comparison of the squared error (green) with the absolute error (red) showing how the latter places much less emphasis on large errors and hence is more robust to outliers and mislabelled data points.

can be addressed by basing the boosting algorithm on the absolute deviation $|y - t|$ instead. These two error functions are compared in Figure 14.4.

14.4. Tree-based Models

There are various simple, but widely used, models that work by partitioning the input space into cuboid regions, whose edges are aligned with the axes, and then assigning a simple model (for example, a constant) to each region. They can be viewed as a model combination method in which only one model is responsible for making predictions at any given point in input space. The process of selecting a specific model, given a new input $\mathbf{x}$, can be described by a sequential decision making process corresponding to the traversal of a binary tree (one that splits into two branches at each node). Here we focus on a particular tree-based framework called *classification and regression trees*, or *CART* (Breiman *et al.*, 1984), although there are many other variants going by such names as ID3 and C4.5 (Quinlan, 1986; Quinlan, 1993).

Figure 14.5 shows an illustration of a recursive binary partitioning of the input space, along with the corresponding tree structure. In this example, the first step

Figure 14.5 Illustration of a two-dimensional input space that has been partitioned into five regions using axis-aligned boundaries.

Figure 14.6 Binary tree corresponding to the partitioning of input space shown in Figure 14.5.

divides the whole of the input space into two regions according to whether $x_1 \leqslant \theta_1$ or $x_1 > \theta_1$ where θ_1 is a parameter of the model. This creates two subregions, each of which can then be subdivided independently. For instance, the region $x_1 \leqslant \theta_1$ is further subdivided according to whether $x_2 \leqslant \theta_2$ or $x_2 > \theta_2$, giving rise to the regions denoted A and B. The recursive subdivision can be described by the traversal of the binary tree shown in Figure 14.6. For any new input $\mathbf{x}$, we determine which region it falls into by starting at the top of the tree at the root node and following a path down to a specific leaf node according to the decision criteria at each node. Note that such decision trees are not probabilistic graphical models.

Within each region, there is a separate model to predict the target variable. For instance, in regression we might simply predict a constant over each region, or in classification we might assign each region to a specific class. A key property of tree-based models, which makes them popular in fields such as medical diagnosis, for example, is that they are readily interpretable by humans because they correspond to a sequence of binary decisions applied to the individual input variables. For instance, to predict a patient's disease, we might first ask "is their temperature greater than some threshold?". If the answer is yes, then we might next ask "is their blood pressure less than some threshold?". Each leaf of the tree is then associated with a specific diagnosis.

In order to learn such a model from a training set, we have to determine the structure of the tree, including which input variable is chosen at each node to form the split criterion as well as the value of the threshold parameter θ_i for the split. We also have to determine the values of the predictive variable within each region.

Consider first a regression problem in which the goal is to predict a single target variable t from a D-dimensional vector $\mathbf{x} = (x_1, \ldots, x_D)^{\mathrm{T}}$ of input variables. The training data consists of input vectors $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ along with the corresponding continuous labels $\{t_1, \ldots, t_N\}$. If the partitioning of the input space is given, and we minimize the sum-of-squares error function, then the optimal value of the predictive variable within any given region is just given by the average of the values of t_n for those data points that fall in that region.

Exercise 14.10

Now consider how to determine the structure of the decision tree. Even for a fixed number of nodes in the tree, the problem of determining the optimal structure (including choice of input variable for each split as well as the corresponding thresh-

olds) to minimize the sum-of-squares error is usually computationally infeasible due to the combinatorially large number of possible solutions. Instead, a greedy optimization is generally done by starting with a single root node, corresponding to the whole input space, and then growing the tree by adding nodes one at a time. At each step there will be some number of candidate regions in input space that can be split, corresponding to the addition of a pair of leaf nodes to the existing tree. For each of these, there is a choice of which of the D input variables to split, as well as the value of the threshold. The joint optimization of the choice of region to split, and the choice of input variable and threshold, can be done efficiently by exhaustive search noting that, for a given choice of split variable and threshold, the optimal choice of predictive variable is given by the local average of the data, as noted earlier. This is repeated for all possible choices of variable to be split, and the one that gives the smallest residual sum-of-squares error is retained.

Given a greedy strategy for growing the tree, there remains the issue of when to stop adding nodes. A simple approach would be to stop when the reduction in residual error falls below some threshold. However, it is found empirically that often none of the available splits produces a significant reduction in error, and yet after several more splits a substantial error reduction is found. For this reason, it is common practice to grow a large tree, using a stopping criterion based on the number of data points associated with the leaf nodes, and then prune back the resulting tree. The pruning is based on a criterion that balances residual error against a measure of model complexity. If we denote the starting tree for pruning by T_0, then we define $T \subset T_0$ to be a subtree of T_0 if it can be obtained by pruning nodes from T_0 (in other words, by collapsing internal nodes by combining the corresponding regions). Suppose the leaf nodes are indexed by $\tau = 1, \ldots, |T|$, with leaf node τ representing a region $\mathcal{R}_\tau$ of input space having N_τ data points, and $|T|$ denoting the total number of leaf nodes. The optimal prediction for region $\mathcal{R}_\tau$ is then given by

$$y_\tau = \frac{1}{N_\tau} \sum_{\mathbf{x}_n \in \mathcal{R}_\tau} t_n \tag{14.29}$$

and the corresponding contribution to the residual sum-of-squares is then

$$Q_\tau(T) = \sum_{\mathbf{x}_n \in \mathcal{R}_\tau} \{t_n - y_\tau\}^2 . \tag{14.30}$$

The pruning criterion is then given by

$$C(T) = \sum_{\tau=1}^{|T|} Q_\tau(T) + \lambda |T| \tag{14.31}$$

The regularization parameter λ determines the trade-off between the overall residual sum-of-squares error and the complexity of the model as measured by the number $|T|$ of leaf nodes, and its value is chosen by cross-validation.

For classification problems, the process of growing and pruning the tree is similar, except that the sum-of-squares error is replaced by a more appropriate measure

of performance. If we define $p_{\tau k}$ to be the proportion of data points in region $\mathcal{R}_\tau$ assigned to class k, where $k = 1, \ldots, K$, then two commonly used choices are the cross-entropy

$$Q_\tau(T) = \sum_{k=1}^{K} p_{\tau k} \ln p_{\tau k} \tag{14.32}$$

and the *Gini index*

$$Q_\tau(T) = \sum_{k=1}^{K} p_{\tau k} \left(1 - p_{\tau k}\right). \tag{14.33}$$

These both vanish for $p_{\tau k} = 0$ and $p_{\tau k} = 1$ and have a maximum at $p_{\tau k} = 0.5$. They encourage the formation of regions in which a high proportion of the data points are assigned to one class. The cross entropy and the Gini index are better measures than the misclassification rate for growing the tree because they are more sensitive to the node probabilities. Also, unlike misclassification rate, they are differentiable and hence better suited to gradient based optimization methods. For subsequent pruning of the tree, the misclassification rate is generally used.

Exercise 14.11

The human interpretability of a tree model such as CART is often seen as its major strength. However, in practice it is found that the particular tree structure that is learned is very sensitive to the details of the data set, so that a small change to the training data can result in a very different set of splits (Hastie *et al.*, 2001).

There are other problems with tree-based methods of the kind considered in this section. One is that the splits are aligned with the axes of the feature space, which may be very suboptimal. For instance, to separate two classes whose optimal decision boundary runs at 45 degrees to the axes would need a large number of axis-parallel splits of the input space as compared to a single non-axis-aligned split. Furthermore, the splits in a decision tree are hard, so that each region of input space is associated with one, and only one, leaf node model. The last issue is particularly problematic in regression where we are typically aiming to model smooth functions, and yet the tree model produces piecewise-constant predictions with discontinuities at the split boundaries.

14.5. Conditional Mixture Models

We have seen that standard decision trees are restricted by hard, axis-aligned splits of the input space. These constraints can be relaxed, at the expense of interpretability, by allowing soft, probabilistic splits that can be functions of all of the input variables, not just one of them at a time. If we also give the leaf models a probabilistic interpretation, we arrive at a fully probabilistic tree-based model called the *hierarchical mixture of experts*, which we consider in Section 14.5.3.

An alternative way to motivate the hierarchical mixture of experts model is to start with a standard probabilistic mixtures of unconditional density models such as Gaussians and replace the component densities with conditional distributions. Here we consider mixtures of linear regression models (Section 14.5.1) and mixtures of

Chapter 9

logistic regression models (Section 14.5.2). In the simplest case, the mixing coefficients are independent of the input variables. If we make a further generalization to allow the mixing coefficients also to depend on the inputs then we obtain a *mixture of experts* model. Finally, if we allow each component in the mixture model to be itself a mixture of experts model, then we obtain a hierarchical mixture of experts.

14.5.1 Mixtures of linear regression models

One of the many advantages of giving a probabilistic interpretation to the linear regression model is that it can then be used as a component in more complex probabilistic models. This can be done, for instance, by viewing the conditional distribution representing the linear regression model as a node in a directed probabilistic graph. Here we consider a simple example corresponding to a mixture of linear regression models, which represents a straightforward extension of the Gaussian mixture model discussed in Section 9.2 to the case of conditional Gaussian distributions.

We therefore consider K linear regression models, each governed by its own weight parameter $\mathbf{w}_k$. In many applications, it will be appropriate to use a common noise variance, governed by a precision parameter β, for all K components, and this is the case we consider here. We will once again restrict attention to a single target variable t, though the extension to multiple outputs is straightforward. If we denote the mixing coefficients by π_k, then the mixture distribution can be written

Exercise 14.12

$$p(t|\boldsymbol{\theta}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(t|\mathbf{w}_k^{\mathrm{T}}\boldsymbol{\phi}, \beta^{-1}) \tag{14.34}$$

where $\boldsymbol{\theta}$ denotes the set of all adaptive parameters in the model, namely $\mathbf{W} = \{\mathbf{w}_k\}$, $\boldsymbol{\pi} = \{\pi_k\}$, and β. The log likelihood function for this model, given a data set of observations $\{\boldsymbol{\phi}_n, t_n\}$, then takes the form

$$\ln p(\mathbf{t}|\boldsymbol{\theta}) = \sum_{n=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}(t_n|\mathbf{w}_k^{\mathrm{T}}\boldsymbol{\phi}_n, \beta^{-1}) \right) \tag{14.35}$$

where $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$ denotes the vector of target variables.

In order to maximize this likelihood function, we can once again appeal to the EM algorithm, which will turn out to be a simple extension of the EM algorithm for unconditional Gaussian mixtures of Section 9.2. We can therefore build on our experience with the unconditional mixture and introduce a set $\mathbf{Z} = \{\mathbf{z}_n\}$ of binary latent variables where $z_{nk} \in \{0, 1\}$ in which, for each data point n, all of the elements $k = 1, \ldots, K$ are zero except for a single value of 1 indicating which component of the mixture was responsible for generating that data point. The joint distribution over latent and observed variables can be represented by the graphical model shown in Figure 14.7.

Exercise 14.13

The complete-data log likelihood function then takes the form

$$\ln p(\mathbf{t}, \mathbf{Z}|\boldsymbol{\theta}) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \ln \left\{ \pi_k \mathcal{N}(t_n|\mathbf{w}_k^{\mathrm{T}}\boldsymbol{\phi}_n, \beta^{-1}) \right\}. \tag{14.36}$$

Figure 14.7 Probabilistic directed graph representing a mixture of linear regression models, defined by (14.35).

The EM algorithm begins by first choosing an initial value $\boldsymbol{\theta}^{\text{old}}$ for the model parameters. In the E step, these parameter values are then used to evaluate the posterior probabilities, or responsibilities, of each component k for every data point n given by

$$\gamma_{nk} = \mathbb{E}[z_{nk}] = p(k|\boldsymbol{\phi}_n, \boldsymbol{\theta}^{\text{old}}) = \frac{\pi_k \mathcal{N}(t_n|\mathbf{w}_k^{\text{T}}\boldsymbol{\phi}_n, \beta^{-1})}{\sum_j \pi_j \mathcal{N}(t_n|\mathbf{w}_j^{\text{T}}\boldsymbol{\phi}_n, \beta^{-1})}. \tag{14.37}$$

The responsibilities are then used to determine the expectation, with respect to the posterior distribution $p(\mathbf{Z}|\mathbf{t}, \boldsymbol{\theta}^{\text{old}})$, of the complete-data log likelihood, which takes the form

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) = \mathbb{E}_{\mathbf{Z}}\left[\ln p(\mathbf{t}, \mathbf{Z}|\boldsymbol{\theta})\right] = \sum_{n=1}^{N}\sum_{k=1}^{K} \gamma_{nk}\left\{\ln \pi_k + \ln \mathcal{N}(t_n|\mathbf{w}_k^{\text{T}}\boldsymbol{\phi}_n, \beta^{-1})\right\}.$$

In the M step, we maximize the function $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$ with respect to $\boldsymbol{\theta}$, keeping the γ_{nk} fixed. For the optimization with respect to the mixing coefficients π_k we need to take account of the constraint $\sum_k \pi_k = 1$, which can be done with the aid of a Lagrange multiplier, leading to an M-step re-estimation equation for π_k in the form

Exercise 14.14

$$\pi_k = \frac{1}{N}\sum_{n=1}^{N} \gamma_{nk}. \tag{14.38}$$

Note that this has exactly the same form as the corresponding result for a simple mixture of unconditional Gaussians given by (9.22).

Next consider the maximization with respect to the parameter vector $\mathbf{w}_k$ of the kth linear regression model. Substituting for the Gaussian distribution, we see that the function $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$, as a function of the parameter vector $\mathbf{w}_k$, takes the form

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}}) = \sum_{n=1}^{N} \gamma_{nk}\left\{-\frac{\beta}{2}\left(t_n - \mathbf{w}_k^{\text{T}}\boldsymbol{\phi}_n\right)^2\right\} + \text{const} \tag{14.39}$$

where the constant term includes the contributions from other weight vectors $\mathbf{w}_j$ for $j \neq k$. Note that the quantity we are maximizing is similar to the (negative of the) standard sum-of-squares error (3.12) for a single linear regression model, but with the inclusion of the responsibilities γ_{nk}. This represents a *weighted least squares*

problem, in which the term corresponding to the nth data point carries a weighting coefficient given by $\beta\gamma_{nk}$, which could be interpreted as an effective precision for each data point. We see that each component linear regression model in the mixture, governed by its own parameter vector $\mathbf{w}_k$, is fitted separately to the whole data set in the M step, but with each data point n weighted by the responsibility γ_{nk} that model k takes for that data point. Setting the derivative of (14.39) with respect to $\mathbf{w}_k$ equal to zero gives

$$0 = \sum_{n=1}^{N} \gamma_{nk} \left(t_n - \mathbf{w}_k^{\mathrm{T}} \boldsymbol{\phi}_n\right) \boldsymbol{\phi}_n \tag{14.40}$$

which we can write in matrix notation as

$$0 = \boldsymbol{\Phi}^{\mathrm{T}} \mathbf{R}_k (\mathbf{t} - \boldsymbol{\Phi} \mathbf{w}_k) \tag{14.41}$$

where $\mathbf{R}_k = \mathrm{diag}(\gamma_{nk})$ is a diagonal matrix of size $N \times N$. Solving for $\mathbf{w}_k$, we obtain

$$\mathbf{w}_k = \left(\boldsymbol{\Phi}^{\mathrm{T}} \mathbf{R}_k \boldsymbol{\Phi}\right)^{-1} \boldsymbol{\Phi}^{\mathrm{T}} \mathbf{R}_k \mathbf{t}. \tag{14.42}$$

This represents a set of modified normal equations corresponding to the weighted least squares problem, of the same form as (4.99) found in the context of logistic regression. Note that after each E step, the matrix $\mathbf{R}_k$ will change and so we will have to solve the normal equations afresh in the subsequent M step.

Finally, we maximize $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\mathrm{old}})$ with respect to β. Keeping only terms that depend on β, the function $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\mathrm{old}})$ can be written

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\mathrm{old}}) = \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma_{nk} \left\{ \frac{1}{2} \ln \beta - \frac{\beta}{2} \left(t_n - \mathbf{w}_k^{\mathrm{T}} \boldsymbol{\phi}_n\right)^2 \right\}. \tag{14.43}$$

Setting the derivative with respect to β equal to zero, and rearranging, we obtain the M-step equation for β in the form

$$\frac{1}{\beta} = \frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma_{nk} \left(t_n - \mathbf{w}_k^{\mathrm{T}} \boldsymbol{\phi}_n\right)^2. \tag{14.44}$$

In Figure 14.8, we illustrate this EM algorithm using the simple example of fitting a mixture of two straight lines to a data set having one input variable x and one target variable t. The predictive density (14.34) is plotted in Figure 14.9 using the converged parameter values obtained from the EM algorithm, corresponding to the right-hand plot in Figure 14.8. Also shown in this figure is the result of fitting a single linear regression model, which gives a unimodal predictive density. We see that the mixture model gives a much better representation of the data distribution, and this is reflected in the higher likelihood value. However, the mixture model also assigns significant probability mass to regions where there is no data because its predictive distribution is bimodal for all values of x. This problem can be resolved by extending the model to allow the mixture coefficients themselves to be functions of x, leading to models such as the mixture density networks discussed in Section 5.6, and hierarchical mixture of experts discussed in Section 14.5.3.

Figure 14.8 Example of a synthetic data set, shown by the green points, having one input variable x and one target variable t, together with a mixture of two linear regression models whose mean functions $y(x, \mathbf{w}_k)$, where $k \in \{1, 2\}$, are shown by the blue and red lines. The upper three plots show the initial configuration (left), the result of running 30 iterations of EM (centre), and the result after 50 iterations of EM (right). Here β was initialized to the reciprocal of the true variance of the set of target values. The lower three plots show the corresponding responsibilities plotted as a vertical line for each data point in which the length of the blue segment gives the posterior probability of the blue line for that data point (and similarly for the red segment).

14.5.2 Mixtures of logistic models

Because the logistic regression model defines a conditional distribution for the target variable, given the input vector, it is straightforward to use it as the component distribution in a mixture model, thereby giving rise to a richer family of conditional distributions compared to a single logistic regression model. This example involves a straightforward combination of ideas encountered in earlier sections of the book and will help consolidate these for the reader.

The conditional distribution of the target variable, for a probabilistic mixture of K logistic regression models, is given by

$$p(t|\boldsymbol{\phi}, \boldsymbol{\theta}) = \sum_{k=1}^{K} \pi_k y_k^t \left[1 - y_k\right]^{1-t} \tag{14.45}$$

where $\boldsymbol{\phi}$ is the feature vector, $y_k = \sigma\left(\mathbf{w}_k^{\mathrm{T}} \boldsymbol{\phi}\right)$ is the output of component k, and $\boldsymbol{\theta}$ denotes the adjustable parameters namely $\{\pi_k\}$ and $\{\mathbf{w}_k\}$.

Now suppose we are given a data set $\{\boldsymbol{\phi}_n, t_n\}$. The corresponding likelihood

Figure 14.9 The left plot shows the predictive conditional density corresponding to the converged solution in Figure 14.8. This gives a log likelihood value of -3.0. A vertical slice through one of these plots at a particular value of x represents the corresponding conditional distribution $p(t|x)$, which we see is bimodal. The plot on the right shows the predictive density for a single linear regression model fitted to the same data set using maximum likelihood. This model has a smaller log likelihood of -27.6.

function is then given by

$$p(\mathbf{t}|\boldsymbol{\theta}) = \prod_{n=1}^{N} \left(\sum_{k=1}^{K} \pi_k y_{nk}^{t_n} \left[1 - y_{nk}\right]^{1-t_n} \right) \tag{14.46}$$

where $y_{nk} = \sigma(\mathbf{w}_k^{\mathrm{T}} \boldsymbol{\phi}_n)$ and $\mathbf{t} = (t_1, \ldots, t_N)^{\mathrm{T}}$. We can maximize this likelihood function iteratively by making use of the EM algorithm. This involves introducing latent variables z_{nk} that correspond to a 1-of-K coded binary indicator variable for each data point n. The complete-data likelihood function is then given by

$$p(\mathbf{t}, \mathbf{Z}|\boldsymbol{\theta}) = \prod_{n=1}^{N} \prod_{k=1}^{K} \left\{ \pi_k y_{nk}^{t_n} \left[1 - y_{nk}\right]^{1-t_n} \right\}^{z_{nk}} \tag{14.47}$$

where $\mathbf{Z}$ is the matrix of latent variables with elements z_{nk}. We initialize the EM algorithm by choosing an initial value $\boldsymbol{\theta}^{\mathrm{old}}$ for the model parameters. In the E step, we then use these parameter values to evaluate the posterior probabilities of the components k for each data point n, which are given by

$$\gamma_{nk} = \mathbb{E}[z_{nk}] = p(k|\boldsymbol{\phi}_n, \boldsymbol{\theta}^{\mathrm{old}}) = \frac{\pi_k y_{nk}^{t_n} \left[1 - y_{nk}\right]^{1-t_n}}{\sum_j \pi_j y_{nj}^{t_n} \left[1 - y_{nj}\right]^{1-t_n}}. \tag{14.48}$$

These responsibilities are then used to find the expected complete-data log likelihood as a function of $\boldsymbol{\theta}$, given by

$$\begin{aligned} Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\mathrm{old}}) &= \mathbb{E}_{\mathbf{Z}} \left[\ln p(\mathbf{t}, \mathbf{Z}|\boldsymbol{\theta})\right] \\ &= \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma_{nk} \left\{ \ln \pi_k + t_n \ln y_{nk} + (1 - t_n) \ln (1 - y_{nk}) \right\}. \end{aligned} \tag{14.49}$$

The M step involves maximization of this function with respect to $\boldsymbol{\theta}$, keeping $\boldsymbol{\theta}^{\text{old}}$, and hence γ_{nk}, fixed. Maximization with respect to π_k can be done in the usual way, with a Lagrange multiplier to enforce the summation constraint $\sum_k \pi_k = 1$, giving the familiar result

$$\pi_k = \frac{1}{N}\sum_{n=1}^{N}\gamma_{nk}. \tag{14.50}$$

To determine the $\{\mathbf{w}_k\}$, we note that the $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$ function comprises a sum over terms indexed by k each of which depends only on one of the vectors $\mathbf{w}_k$, so that the different vectors are decoupled in the M step of the EM algorithm. In other words, the different components interact only via the responsibilities, which are fixed during the M step. Note that the M step does not have a closed-form solution and must be solved iteratively using, for instance, the iterative reweighted least squares (IRLS) algorithm. The gradient and the Hessian for the vector $\mathbf{w}_k$ are given by

Section 4.3.3

$$\nabla_k Q = \sum_{n=1}^{N}\gamma_{nk}(t_n - y_{nk})\boldsymbol{\phi}_n \tag{14.51}$$

$$\mathbf{H}_k = -\nabla_k\nabla_k Q = \sum_{n=1}^{N}\gamma_{nk}y_{nk}(1-y_{nk})\boldsymbol{\phi}_n\boldsymbol{\phi}_n^{\mathrm{T}} \tag{14.52}$$

where ∇_k denotes the gradient with respect to $\mathbf{w}_k$. For fixed γ_{nk}, these are independent of $\{\mathbf{w}_j\}$ for $j \neq k$ and so we can solve for each $\mathbf{w}_k$ separately using the IRLS algorithm. Thus the M-step equations for component k correspond simply to fitting a single logistic regression model to a weighted data set in which data point n carries a weight γ_{nk}. Figure 14.10 shows an example of the mixture of logistic regression models applied to a simple classification problem. The extension of this model to a mixture of softmax models for more than two classes is straightforward.

Section 4.3.3

Exercise 14.16

14.5.3 Mixtures of experts

In Section 14.5.1, we considered a mixture of linear regression models, and in Section 14.5.2 we discussed the analogous mixture of linear classifiers. Although these simple mixtures extend the flexibility of linear models to include more complex (e.g., multimodal) predictive distributions, they are still very limited. We can further increase the capability of such models by allowing the mixing coefficients themselves to be functions of the input variable, so that

$$p(\mathbf{t}|\mathbf{x}) = \sum_{k=1}^{K}\pi_k(\mathbf{x})p_k(\mathbf{t}|\mathbf{x}). \tag{14.53}$$

This is known as a *mixture of experts* model (Jacobs *et al.*, 1991) in which the mixing coefficients $\pi_k(\mathbf{x})$ are known as *gating* functions and the individual component densities $p_k(\mathbf{t}|\mathbf{x})$ are called *experts*. The notion behind the terminology is that different components can model the distribution in different regions of input space (they

Figure 14.10 Illustration of a mixture of logistic regression models. The left plot shows data points drawn from two classes denoted red and blue, in which the background colour (which varies from pure red to pure blue) denotes the true probability of the class label. The centre plot shows the result of fitting a single logistic regression model using maximum likelihood, in which the background colour denotes the corresponding probability of the class label. Because the colour is a near-uniform purple, we see that the model assigns a probability of around 0.5 to each of the classes over most of input space. The right plot shows the result of fitting a mixture of two logistic regression models, which now gives much higher probability to the correct labels for many of the points in the blue class.

are 'experts' at making predictions in their own regions), and the gating functions determine which components are dominant in which region.

The gating functions $\pi_k(\mathbf{x})$ must satisfy the usual constraints for mixing coefficients, namely $0 \leqslant \pi_k(\mathbf{x}) \leqslant 1$ and $\sum_k \pi_k(\mathbf{x}) = 1$. They can therefore be represented, for example, by linear softmax models of the form (4.104) and (4.105). If the experts are also linear (regression or classification) models, then the whole model can be fitted efficiently using the EM algorithm, with iterative reweighted least squares being employed in the M step (Jordan and Jacobs, 1994).

Such a model still has significant limitations due to the use of linear models for the gating and expert functions. A much more flexible model is obtained by using a multilevel gating function to give the *hierarchical mixture of experts*, or *HME* model (Jordan and Jacobs, 1994). To understand the structure of this model, imagine a mixture distribution in which each component in the mixture is itself a mixture distribution. For simple unconditional mixtures, this hierarchical mixture is trivially equivalent to a single flat mixture distribution. However, when the mixing coefficients are input dependent, this hierarchical model becomes nontrivial. The HME model can also be viewed as a probabilistic version of *decision trees* discussed in Section 14.4 and can again be trained efficiently by maximum likelihood using an EM algorithm with IRLS in the M step. A Bayesian treatment of the HME has been given by Bishop and Svensén (2003) based on variational inference.

Exercise 14.17

Section 4.3.3

We shall not discuss the HME in detail here. However, it is worth pointing out the close connection with the *mixture density network* discussed in Section 5.6. The principal advantage of the mixtures of experts model is that it can be optimized by EM in which the M step for each mixture component and gating model involves a convex optimization (although the overall optimization is nonconvex). By contrast, the advantage of the mixture density network approach is that the component

densities and the mixing coefficients share the hidden units of the neural network. Furthermore, in the mixture density network, the splits of the input space are further relaxed compared to the hierarchical mixture of experts in that they are not only soft, and not constrained to be axis aligned, but they can also be nonlinear.

Exercises

14.1 ($\star\star$) **www** Consider a set models of the form $p(\mathbf{t}|\mathbf{x}, \mathbf{z}_h, \boldsymbol{\theta}_h, h)$ in which $\mathbf{x}$ is the input vector, $\mathbf{t}$ is the target vector, h indexes the different models, $\mathbf{z}_h$ is a latent variable for model h, and $\boldsymbol{\theta}_h$ is the set of parameters for model h. Suppose the models have prior probabilities $p(h)$ and that we are given a training set $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ and $\mathbf{T} = \{\mathbf{t}_1, \ldots, \mathbf{t}_N\}$. Write down the formulae needed to evaluate the predictive distribution $p(\mathbf{t}|\mathbf{x}, \mathbf{X}, \mathbf{T})$ in which the latent variables and the model index are marginalized out. Use these formulae to highlight the difference between Bayesian averaging of different models and the use of latent variables within a single model.

14.2 ($\star$) The expected sum-of-squares error E_{AV} for a simple committee model can be defined by (14.10), and the expected error of the committee itself is given by (14.11). Assuming that the individual errors satisfy (14.12) and (14.13), derive the result (14.14).

14.3 ($\star$) **www** By making use of Jensen's inequality (1.115), for the special case of the convex function $f(x) = x^2$, show that the average expected sum-of-squares error E_{AV} of the members of a simple committee model, given by (14.10), and the expected error E_{COM} of the committee itself, given by (14.11), satisfy

$$E_{\mathrm{COM}} \leqslant E_{\mathrm{AV}}. \tag{14.54}$$

14.4 ($\star\star$) By making use of Jensen's in equality (1.115), show that the result (14.54) derived in the previous exercise hods for any error function $E(y)$, not just sum-of-squares, provided it is a convex function of y.

14.5 ($\star\star$) **www** Consider a committee in which we allow unequal weighting of the constituent models, so that

$$y_{\mathrm{COM}}(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m y_m(\mathbf{x}). \tag{14.55}$$

In order to ensure that the predictions $y_{\mathrm{COM}}(\mathbf{x})$ remain within sensible limits, suppose that we require that they be bounded at each value of $\mathbf{x}$ by the minimum and maximum values given by any of the members of the committee, so that

$$y_{\min}(\mathbf{x}) \leqslant y_{\mathrm{COM}}(\mathbf{x}) \leqslant y_{\max}(\mathbf{x}). \tag{14.56}$$

Show that a necessary and sufficient condition for this constraint is that the coefficients α_m satisfy

$$\alpha_m \geqslant 0, \qquad \sum_{m=1}^{M} \alpha_m = 1. \tag{14.57}$$

14.6 ($\star$) **WWW** By differentiating the error function (14.23) with respect to α_m, show that the parameters α_m in the AdaBoost algorithm are updated using (14.17) in which ϵ_m is defined by (14.16).

14.7 ($\star$) By making a variational minimization of the expected exponential error function given by (14.27) with respect to all possible functions $y(\mathbf{x})$, show that the minimizing function is given by (14.28).

14.8 ($\star$) Show that the exponential error function (14.20), which is minimized by the AdaBoost algorithm, does not correspond to the log likelihood of any well-behaved probabilistic model. This can be done by showing that the corresponding conditional distribution $p(t|\mathbf{x})$ cannot be correctly normalized.

14.9 ($\star$) **WWW** Show that the sequential minimization of the sum-of-squares error function for an additive model of the form (14.21) in the style of boosting simply involves fitting each new base classifier to the residual errors $t_n - f_{m-1}(\mathbf{x}_n)$ from the previous model.

14.10 ($\star$) Verify that if we minimize the sum-of-squares error between a set of training values $\{t_n\}$ and a single predictive value t, then the optimal solution for t is given by the mean of the $\{t_n\}$.

14.11 ($\star\star$) Consider a data set comprising 400 data points from class $\mathcal{C}_1$ and 400 data points from class $\mathcal{C}_2$. Suppose that a tree model A splits these into $(300, 100)$ at the first leaf node and $(100, 300)$ at the second leaf node, where (n, m) denotes that n points are assigned to $\mathcal{C}_1$ and m points are assigned to $\mathcal{C}_2$. Similarly, suppose that a second tree model B splits them into $(200, 400)$ and $(200, 0)$. Evaluate the misclassification rates for the two trees and hence show that they are equal. Similarly, evaluate the cross-entropy (14.32) and Gini index (14.33) for the two trees and show that they are both lower for tree B than for tree A.

14.12 ($\star\star$) Extend the results of Section 14.5.1 for a mixture of linear regression models to the case of multiple target values described by a vector $\mathbf{t}$. To do this, make use of the results of Section 3.1.5.

14.13 ($\star$) **WWW** Verify that the complete-data log likelihood function for the mixture of linear regression models is given by (14.36).

14.14 ($\star$) Use the technique of Lagrange multipliers (Appendix E) to show that the M-step re-estimation equation for the mixing coefficients in the mixture of linear regression models trained by maximum likelihood EM is given by (14.38).

14.15 ($\star$) **WWW** We have already noted that if we use a squared loss function in a regression problem, the corresponding optimal prediction of the target variable for a new input vector is given by the conditional mean of the predictive distribution. Show that the conditional mean for the mixture of linear regression models discussed in Section 14.5.1 is given by a linear combination of the means of each component distribution. Note that if the conditional distribution of the target data is multimodal, the conditional mean can give poor predictions.

14.16 ($\star\star\star$) Extend the logistic regression mixture model of Section 14.5.2 to a mixture of softmax classifiers representing $C \geqslant 2$ classes. Write down the EM algorithm for determining the parameters of this model through maximum likelihood.

14.17 ($\star\star$) **www** Consider a mixture model for a conditional distribution $p(t|\mathbf{x})$ of the form

$$p(t|\mathbf{x}) = \sum_{k=1}^{K} \pi_k \psi_k(t|\mathbf{x}) \tag{14.58}$$

in which each mixture component $\psi_k(t|\mathbf{x})$ is itself a mixture model. Show that this two-level hierarchical mixture is equivalent to a conventional single-level mixture model. Now suppose that the mixing coefficients in both levels of such a hierarchical model are arbitrary functions of $\mathbf{x}$. Again, show that this hierarchical model is again equivalent to a single-level model with $\mathbf{x}$-dependent mixing coefficients. Finally, consider the case in which the mixing coefficients at both levels of the hierarchical mixture are constrained to be linear classification (logistic or softmax) models. Show that the hierarchical mixture cannot in general be represented by a single-level mixture having linear classification models for the mixing coefficients. Hint: to do this it is sufficient to construct a single counter-example, so consider a mixture of two components in which one of those components is itself a mixture of two components, with mixing coefficients given by linear-logistic models. Show that this cannot be represented by a single-level mixture of 3 components having mixing coefficients determined by a linear-softmax model.

Appendix A. Data Sets

In this appendix, we give a brief introduction to the data sets used to illustrate some of the algorithms described in this book. Detailed information on file formats for these data sets, as well as the data files themselves, can be obtained from the book web site:

http://research.microsoft.com/∼cmbishop/PRML

Handwritten Digits

The digits data used in this book is taken from the MNIST data set (LeCun *et al.*, 1998), which itself was constructed by modifying a subset of the much larger data set produced by NIST (the National Institute of Standards and Technology). It comprises a training set of $60,000$ examples and a test set of $10,000$ examples. Some of the data was collected from Census Bureau employees and the rest was collected from high-school children, and care was taken to ensure that the test examples were written by different individuals to the training examples.

The original NIST data had binary (black or white) pixels. To create MNIST, these images were size normalized to fit in a 20×20 pixel box while preserving their aspect ratio. As a consequence of the anti-aliasing used to change the resolution of the images, the resulting MNIST digits are grey scale. These images were then centred in a 28×28 box. Examples of the MNIST digits are shown in Figure A.1.

Error rates for classifying the digits range from 12% for a simple linear classifier, through 0.56% for a carefully designed support vector machine, to 0.4% for a convolutional neural network (LeCun *et al.*, 1998).

Figure A.1 One hundred examples of the MNIST digits chosen at random from the training set.

Oil Flow

This is a synthetic data set that arose out of a project aimed at measuring noninvasively the proportions of oil, water, and gas in North Sea oil transfer pipelines (Bishop and James, 1993). It is based on the principle of *dual-energy gamma densitometry*. The ideas is that if a narrow beam of gamma rays is passed through the pipe, the attenuation in the intensity of the beam provides information about the density of material along its path. Thus, for instance, the beam will be attenuated more strongly by oil than by gas.

A single attenuation measurement alone is not sufficient because there are two degrees of freedom corresponding to the fraction of oil and the fraction of water (the fraction of gas is redundant because the three fractions must add to one). To address this, two gamma beams of different energies (in other words different frequencies or wavelengths) are passed through the pipe along the same path, and the attenuation of each is measured. Because the absorbtion properties of different materials vary differently as a function of energy, measurement of the attenuations at the two energies provides two independent pieces of information. Given the known absorbtion properties of oil, water, and gas at the two energies, it is then a simple matter to calculate the average fractions of oil and water (and hence of gas) measured *along the path* of the gamma beams.

There is a further complication, however, associated with the motion of the materials along the pipe. If the flow velocity is small, then the oil floats on top of the water with the gas sitting above the oil. This is known as a *laminar* or *stratified*

Figure A.2 The three geometrical configurations of the oil, water, and gas phases used to generate the oil-flow data set. For each configuration, the proportions of the three phases can vary.

flow configuration and is illustrated in Figure A.2. As the flow velocity is increased, more complex geometrical configurations of the oil, water, and gas can arise. For the purposes of this data set, two specific idealizations are considered. In the *annular* configuration the oil, water, and gas form concentric cylinders with the water around the outside and the gas in the centre, whereas in the *homogeneous* configuration the oil, water and gas are assumed to be intimately mixed as might occur at high flow velocities under turbulent conditions. These configurations are also illustrated in Figure A.2.

We have seen that a single dual-energy beam gives the oil and water fractions measured along the path length, whereas we are interested in the volume fractions of oil and water. This can be addressed by using multiple dual-energy gamma densitometers whose beams pass through different regions of the pipe. For this particular data set, there are six such beams, and their spatial arrangement is shown in Figure A.3. A single observation is therefore represented by a 12-dimensional vector comprising the fractions of oil and water measured along the paths of each of the beams. We are, however, interested in obtaining the overall volume fractions of the three phases in the pipe. This is much like the classical problem of tomographic reconstruction, used in medical imaging for example, in which a two-dimensional dis-

Figure A.3 Cross section of the pipe showing the arrangement of the six beam lines, each of which comprises a single dual-energy gamma densitometer. Note that the vertical beams are asymmetrically arranged relative to the central axis (shown by the dotted line).

tribution is to be reconstructed from an number of one-dimensional averages. Here there are far fewer line measurements than in a typical tomography application. On the other hand the range of geometrical configurations is much more limited, and so the configuration, as well as the phase fractions, can be predicted with reasonable accuracy from the densitometer data.

For safety reasons, the intensity of the gamma beams is kept relatively weak and so to obtain an accurate measurement of the attenuation, the measured beam intensity is integrated over a specific time interval. For a finite integration time, there are random fluctuations in the measured intensity due to the fact that the gamma beams comprise discrete packets of energy called photons. In practice, the integration time is chosen as a compromise between reducing the noise level (which requires a long integration time) and detecting temporal variations in the flow (which requires a short integration time). The oil flow data set is generated using realistic known values for the absorption properties of oil, water, and gas at the two gamma energies used, and with a specific choice of integration time (10 seconds) chosen as characteristic of a typical practical setup.

Each point in the data set is generated independently using the following steps:

1. Choose one of the three phase configurations at random with equal probability.

2. Choose three random numbers f_1, f_2 and f_3 from the uniform distribution over $(0, 1)$ and define

$$f_{\text{oil}} = \frac{f_1}{f_1 + f_2 + f_3}, \qquad f_{\text{water}} = \frac{f_2}{f_1 + f_2 + f_3}. \tag{A.1}$$

   This treats the three phases on an equal footing and ensures that the volume fractions add to one.

3. For each of the six beam lines, calculate the effective path lengths through oil and water for the given phase configuration.

4. Perturb the path lengths using the Poisson distribution based on the known beam intensities and integration time to allow for the effect of photon statistics.

Each point in the data set comprises the 12 path length measurements, together with the fractions of oil and water and a binary label describing the phase configuration. The data set is divided into training, validation, and test sets, each of which comprises $1,000$ independent data points. Details of the data format are available from the book web site.

In Bishop and James (1993), statistical machine learning techniques were used to predict the volume fractions and also the geometrical configuration of the phases shown in Figure A.2, from the 12-dimensional vector of measurements. The 12-dimensional observation vectors can also be used to test data visualization algorithms.

This data set has a rich and interesting structure, as follows. For any given configuration there are two degrees of freedom corresponding to the fractions of

oil and water, and so for infinite integration time the data will locally live on a two-dimensional manifold. For a finite integration time, the individual data points will be perturbed away from the manifold by the photon noise. In the homogeneous phase configuration, the path lengths in oil and water are linearly related to the fractions of oil and water, and so the data points lie close to a linear manifold. For the annular configuration, the relationship between phase fraction and path length is nonlinear and so the manifold will be nonlinear. In the case of the laminar configuration the situation is even more complex because small variations in the phase fractions can cause one of the horizontal phase boundaries to move across one of the horizontal beam lines leading to a discontinuous jump in the 12-dimensional observation space. In this way, the two-dimensional nonlinear manifold for the laminar configuration is broken into six distinct segments. Note also that some of the manifolds for different phase configurations meet at specific points, for example if the pipe is filled entirely with oil, it corresponds to specific instances of the laminar, annular, and homogeneous configurations.

Old Faithful

Old Faithful, shown in Figure A.4, is a hydrothermal geyser in Yellowstone National Park in the state of Wyoming, U.S.A., and is a popular tourist attraction. Its name stems from the supposed regularity of its eruptions.

The data set comprises 272 observations, each of which represents a single eruption and contains two variables corresponding to the duration in minutes of the eruption, and the time until the next eruption, also in minutes. Figure A.5 shows a plot of the time to the next eruption versus the duration of the eruptions. It can be seen that the time to the next eruption varies considerably, although knowledge of the duration of the current eruption allows it to be predicted more accurately. Note that there exist several other data sets relating to the eruptions of Old Faithful.

Figure A.4 The Old Faithful geyser in Yellowstone National Park. ©Bruce T. Gourley www.brucegourley.com.

Figure A.5 Plot of the time to the next eruption in minutes (vertical axis) versus the duration of the eruption in minutes (horizontal axis) for the Old Faithful data set.

Synthetic Data

Throughout the book, we use two simple synthetic data sets to illustrate many of the algorithms. The first of these is a regression problem, based on the sinusoidal function, shown in Figure A.6. The input values $\{x_n\}$ are generated uniformly in range $(0, 1)$, and the corresponding target values $\{t_n\}$ are obtained by first computing the corresponding values of the function $\sin(2\pi x)$, and then adding random noise with a Gaussian distribution having standard deviation 0.3. Various forms of this data set, having different numbers of data points, are used in the book.

The second data set is a classification problem having two classes, with equal prior probabilities, and is shown in Figure A.7. The blue class is generated from a single Gaussian while the red class comes from a mixture of two Gaussians. Because we know the class priors and the class-conditional densities, it is straightforward to evaluate and plot the true posterior probabilities as well as the minimum misclassification-rate decision boundary, as shown in Figure A.7.

Figure A.6 The left-hand plot shows the synthetic regression data set along with the underlying sinusoidal function from which the data points were generated. The right-hand plot shows the true conditional distribution $p(t|x)$ from which the labels are generated, in which the green curve denotes the mean, and the shaded region spans one standard deviation on each side of the mean.

Figure A.7 The left plot shows the synthetic classification data set with data from the two classes shown in red and blue. On the right is a plot of the true posterior probabilities, shown on a colour scale going from pure red denoting probability of the red class is 1 to pure blue denoting probability of the red class is 0. Because these probabilities are known, the optimal decision boundary for minimizing the misclassification rate (which corresponds to the contour along which the posterior probabilities for each class equal 0.5) can be evaluated and is shown by the green curve. This decision boundary is also plotted on the left-hand figure.

Appendix B. Probability Distributions

In this appendix, we summarize the main properties of some of the most widely used probability distributions, and for each distribution we list some key statistics such as the expectation $\mathbb{E}[\mathbf{x}]$, the variance (or covariance), the mode, and the entropy $\mathrm{H}[\mathbf{x}]$. All of these distributions are members of the exponential family and are widely used as building blocks for more sophisticated probabilistic models.

Bernoulli

This is the distribution for a single binary variable $x \in \{0, 1\}$ representing, for example, the result of flipping a coin. It is governed by a single continuous parameter $\mu \in [0, 1]$ that represents the probability of $x = 1$.

$$\mathrm{Bern}(x|\mu) = \mu^x (1-\mu)^{1-x} \tag{B.1}$$

$$\mathbb{E}[x] = \mu \tag{B.2}$$

$$\mathrm{var}[x] = \mu(1-\mu) \tag{B.3}$$

$$\mathrm{mode}[x] = \begin{cases} 1 & \text{if } \mu \geqslant 0.5, \\ 0 & \text{otherwise} \end{cases} \tag{B.4}$$

$$\mathrm{H}[x] = -\mu \ln \mu - (1-\mu)\ln(1-\mu). \tag{B.5}$$

The Bernoulli is a special case of the binomial distribution for the case of a single observation. Its conjugate prior for μ is the beta distribution.

Beta

This is a distribution over a continuous variable $\mu \in [0, 1]$, which is often used to represent the probability for some binary event. It is governed by two parameters a and b that are constrained by $a > 0$ and $b > 0$ to ensure that the distribution can be normalized.

$$\mathrm{Beta}(\mu|a, b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\mu^{a-1}(1-\mu)^{b-1} \tag{B.6}$$

$$\mathbb{E}[\mu] = \frac{a}{a+b} \tag{B.7}$$

$$\mathrm{var}[\mu] = \frac{ab}{(a+b)^2(a+b+1)} \tag{B.8}$$

$$\mathrm{mode}[\mu] = \frac{a-1}{a+b-2}. \tag{B.9}$$

The beta is the conjugate prior for the Bernoulli distribution, for which a and b can be interpreted as the effective prior number of observations of $x = 1$ and $x = 0$, respectively. Its density is finite if $a \geqslant 1$ and $b \geqslant 1$, otherwise there is a singularity at $\mu = 0$ and/or $\mu = 1$. For $a = b = 1$, it reduces to a uniform distribution. The beta distribution is a special case of the K-state Dirichlet distribution for $K = 2$.

Binomial

The binomial distribution gives the probability of observing m occurrences of $x = 1$ in a set of N samples from a Bernoulli distribution, where the probability of observing $x = 1$ is $\mu \in [0, 1]$.

$$\mathrm{Bin}(m|N, \mu) = \binom{N}{m}\mu^m(1-\mu)^{N-m} \tag{B.10}$$

$$\mathbb{E}[m] = N\mu \tag{B.11}$$

$$\mathrm{var}[m] = N\mu(1-\mu) \tag{B.12}$$

$$\mathrm{mode}[m] = \lfloor (N+1)\mu \rfloor \tag{B.13}$$

where $\lfloor (N+1)\mu \rfloor$ denotes the largest integer that is less than or equal to $(N+1)\mu$, and the quantity

$$\binom{N}{m} = \frac{N!}{m!(N-m)!} \tag{B.14}$$

denotes the number of ways of choosing m objects out of a total of N identical objects. Here $m!$, pronounced 'factorial m', denotes the product $m \times (m-1) \times \ldots, \times 2 \times 1$. The particular case of the binomial distribution for $N = 1$ is known as the Bernoulli distribution, and for large N the binomial distribution is approximately Gaussian. The conjugate prior for μ is the beta distribution.

Dirichlet

The Dirichlet is a multivariate distribution over K random variables $0 \leqslant \mu_k \leqslant 1$, where $k = 1, \ldots, K$, subject to the constraints

$$0 \leqslant \mu_k \leqslant 1, \qquad \sum_{k=1}^{K} \mu_k = 1. \tag{B.15}$$

Denoting $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_K)^{\mathrm{T}}$ and $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_K)^{\mathrm{T}}$, we have

$$\begin{aligned}
\mathrm{Dir}(\boldsymbol{\mu}|\boldsymbol{\alpha}) &= C(\boldsymbol{\alpha}) \prod_{k=1}^{K} \mu_k^{\alpha_k - 1} && \text{(B.16)}\\
\mathbb{E}[\mu_k] &= \frac{\alpha_k}{\widehat{\alpha}} && \text{(B.17)}\\
\mathrm{var}[\mu_k] &= \frac{\alpha_k(\widehat{\alpha} - \alpha_k)}{\widehat{\alpha}^2(\widehat{\alpha} + 1)} && \text{(B.18)}\\
\mathrm{cov}[\mu_j \mu_k] &= -\frac{\alpha_j \alpha_k}{\widehat{\alpha}^2(\widehat{\alpha} + 1)} && \text{(B.19)}\\
\mathrm{mode}[\mu_k] &= \frac{\alpha_k - 1}{\widehat{\alpha} - K} && \text{(B.20)}\\
\mathbb{E}[\ln \mu_k] &= \psi(\alpha_k) - \psi(\widehat{\alpha}) && \text{(B.21)}\\
\mathrm{H}[\boldsymbol{\mu}] &= -\sum_{k=1}^{K} (\alpha_k - 1) \left\{\psi(\alpha_k) - \psi(\widehat{\alpha})\right\} - \ln C(\boldsymbol{\alpha}) && \text{(B.22)}
\end{aligned}$$

where

$$C(\boldsymbol{\alpha}) = \frac{\Gamma(\widehat{\alpha})}{\Gamma(\alpha_1) \cdots \Gamma(\alpha_K)} \tag{B.23}$$

and

$$\widehat{\alpha} = \sum_{k=1}^{K} \alpha_k. \tag{B.24}$$

Here

$$\psi(a) \equiv \frac{d}{da} \ln \Gamma(a) \tag{B.25}$$

is known as the *digamma* function (Abramowitz and Stegun, 1965). The parameters α_k are subject to the constraint $\alpha_k > 0$ in order to ensure that the distribution can be normalized.

The Dirichlet forms the conjugate prior for the multinomial distribution and represents a generalization of the beta distribution. In this case, the parameters α_k can be interpreted as effective numbers of observations of the corresponding values of the K-dimensional binary observation vector $\mathbf{x}$. As with the beta distribution, the Dirichlet has finite density everywhere provided $\alpha_k \geqslant 1$ for all k.

Gamma

The Gamma is a probability distribution over a positive random variable $\tau > 0$ governed by parameters a and b that are subject to the constraints $a > 0$ and $b > 0$ to ensure that the distribution can be normalized.

$$\begin{aligned}
\mathrm{Gam}(\tau|a,b) &= \frac{1}{\Gamma(a)} b^a \tau^{a-1} e^{-b\tau} && \text{(B.26)}\\
\mathbb{E}[\tau] &= \frac{a}{b} && \text{(B.27)}\\
\mathrm{var}[\tau] &= \frac{a}{b^2} && \text{(B.28)}\\
\mathrm{mode}[\tau] &= \frac{a-1}{b} \quad \text{for } \alpha \geqslant 1 && \text{(B.29)}\\
\mathbb{E}[\ln \tau] &= \psi(a) - \ln b && \text{(B.30)}\\
\mathrm{H}[\tau] &= \ln \Gamma(a) - (a-1)\psi(a) - \ln b + a && \text{(B.31)}
\end{aligned}$$

where $\psi(\cdot)$ is the digamma function defined by (B.25). The gamma distribution is the conjugate prior for the precision (inverse variance) of a univariate Gaussian. For $a \geqslant 1$ the density is everywhere finite, and the special case of $a = 1$ is known as the *exponential* distribution.

Gaussian

The Gaussian is the most widely used distribution for continuous variables. It is also known as the *normal* distribution. In the case of a single variable $x \in (-\infty, \infty)$ it is governed by two parameters, the mean $\mu \in (-\infty, \infty)$ and the variance $\sigma^2 > 0$.

$$\begin{aligned}
\mathcal{N}(x|\mu,\sigma^2) &= \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\} && \text{(B.32)}\\
\mathbb{E}[x] &= \mu && \text{(B.33)}\\
\mathrm{var}[x] &= \sigma^2 && \text{(B.34)}\\
\mathrm{mode}[x] &= \mu && \text{(B.35)}\\
\mathrm{H}[x] &= \frac{1}{2}\ln\sigma^2 + \frac{1}{2}\left(1 + \ln(2\pi)\right). && \text{(B.36)}
\end{aligned}$$

The inverse of the variance $\tau = 1/\sigma^2$ is called the precision, and the square root of the variance σ is called the standard deviation. The conjugate prior for μ is the Gaussian, and the conjugate prior for τ is the gamma distribution. If both μ and τ are unknown, their joint conjugate prior is the Gaussian-gamma distribution.

For a D-dimensional vector $\mathbf{x}$, the Gaussian is governed by a D-dimensional mean vector $\boldsymbol{\mu}$ and a $D \times D$ covariance matrix $\boldsymbol{\Sigma}$ that must be symmetric and

positive-definite.

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}}\frac{1}{|\boldsymbol{\Sigma}|^{1/2}}\exp\left\{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right\} \quad \text{(B.37)}$$

$$\mathbb{E}[\mathbf{x}] = \boldsymbol{\mu} \quad \text{(B.38)}$$

$$\mathrm{cov}[\mathbf{x}] = \boldsymbol{\Sigma} \quad \text{(B.39)}$$

$$\mathrm{mode}[\mathbf{x}] = \boldsymbol{\mu} \quad \text{(B.40)}$$

$$\mathrm{H}[\mathbf{x}] = \frac{1}{2}\ln|\boldsymbol{\Sigma}| + \frac{D}{2}\left(1+\ln(2\pi)\right). \quad \text{(B.41)}$$

The inverse of the covariance matrix $\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1}$ is the precision matrix, which is also symmetric and positive definite. Averages of random variables tend to a Gaussian, by the central limit theorem, and the sum of two Gaussian variables is again Gaussian. The Gaussian is the distribution that maximizes the entropy for a given variance (or covariance). Any linear transformation of a Gaussian random variable is again Gaussian. The marginal distribution of a multivariate Gaussian with respect to a subset of the variables is itself Gaussian, and similarly the conditional distribution is also Gaussian. The conjugate prior for $\boldsymbol{\mu}$ is the Gaussian, the conjugate prior for $\boldsymbol{\Lambda}$ is the Wishart, and the conjugate prior for $(\boldsymbol{\mu},\boldsymbol{\Lambda})$ is the Gaussian-Wishart.

If we have a marginal Gaussian distribution for $\mathbf{x}$ and a conditional Gaussian distribution for $\mathbf{y}$ given $\mathbf{x}$ in the form

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\Lambda}^{-1}) \quad \text{(B.42)}$$

$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}|\mathbf{A}\mathbf{x}+\mathbf{b},\mathbf{L}^{-1}) \quad \text{(B.43)}$$

then the marginal distribution of $\mathbf{y}$, and the conditional distribution of $\mathbf{x}$ given $\mathbf{y}$, are given by

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{y}|\mathbf{A}\boldsymbol{\mu}+\mathbf{b},\mathbf{L}^{-1}+\mathbf{A}\boldsymbol{\Lambda}^{-1}\mathbf{A}^{\mathrm{T}}) \quad \text{(B.44)}$$

$$p(\mathbf{x}|\mathbf{y}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\Sigma}\{\mathbf{A}^{\mathrm{T}}\mathbf{L}(\mathbf{y}-\mathbf{b})+\boldsymbol{\Lambda}\boldsymbol{\mu}\},\boldsymbol{\Sigma}) \quad \text{(B.45)}$$

where

$$\boldsymbol{\Sigma} = (\boldsymbol{\Lambda}+\mathbf{A}^{\mathrm{T}}\mathbf{L}\mathbf{A})^{-1}. \quad \text{(B.46)}$$

If we have a joint Gaussian distribution $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\Sigma})$ with $\boldsymbol{\Lambda} \equiv \boldsymbol{\Sigma}^{-1}$ and we define the following partitions

$$\mathbf{x} = \begin{pmatrix}\mathbf{x}_a\\ \mathbf{x}_b\end{pmatrix}, \qquad \boldsymbol{\mu} = \begin{pmatrix}\boldsymbol{\mu}_a\\ \boldsymbol{\mu}_b\end{pmatrix} \quad \text{(B.47)}$$

$$\boldsymbol{\Sigma} = \begin{pmatrix}\boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab}\\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb}\end{pmatrix}, \qquad \boldsymbol{\Lambda} = \begin{pmatrix}\boldsymbol{\Lambda}_{aa} & \boldsymbol{\Lambda}_{ab}\\ \boldsymbol{\Lambda}_{ba} & \boldsymbol{\Lambda}_{bb}\end{pmatrix} \quad \text{(B.48)}$$

then the conditional distribution $p(\mathbf{x}_a|\mathbf{x}_b)$ is given by

$$p(\mathbf{x}_a|\mathbf{x}_b) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_{a|b},\boldsymbol{\Lambda}_{aa}^{-1}) \quad \text{(B.49)}$$

$$\boldsymbol{\mu}_{a|b} = \boldsymbol{\mu}_a - \boldsymbol{\Lambda}_{aa}^{-1}\boldsymbol{\Lambda}_{ab}(\mathbf{x}_b-\boldsymbol{\mu}_b) \quad \text{(B.50)}$$

and the marginal distribution $p(\mathbf{x}_a)$ is given by

$$p(\mathbf{x}_a) = \mathcal{N}(\mathbf{x}_a|\boldsymbol{\mu}_a, \boldsymbol{\Sigma}_{aa}). \tag{B.51}$$

Gaussian-Gamma

This is the conjugate prior distribution for a univariate Gaussian $\mathcal{N}(x|\mu, \lambda^{-1})$ in which the mean μ and the precision λ are both unknown and is also called the *normal-gamma* distribution. It comprises the product of a Gaussian distribution for μ, whose precision is proportional to λ, and a gamma distribution over λ.

$$p(\mu, \lambda|\mu_0, \beta, a, b) = \mathcal{N}\left(\mu|\mu_o, (\beta\lambda)^{-1}\right) \mathrm{Gam}(\lambda|a, b). \tag{B.52}$$

Gaussian-Wishart

This is the conjugate prior distribution for a multivariate Gaussian $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Lambda})$ in which both the mean $\boldsymbol{\mu}$ and the precision $\boldsymbol{\Lambda}$ are unknown, and is also called the normal-Wishart distribution. It comprises the product of a Gaussian distribution for $\boldsymbol{\mu}$, whose precision is proportional to $\boldsymbol{\Lambda}$, and a Wishart distribution over $\boldsymbol{\Lambda}$.

$$p(\boldsymbol{\mu}, \boldsymbol{\Lambda}|\boldsymbol{\mu}_0, \beta, \mathbf{W}, \nu) = \mathcal{N}\left(\boldsymbol{\mu}|\boldsymbol{\mu}_0, (\beta\boldsymbol{\Lambda})^{-1}\right) \mathcal{W}(\boldsymbol{\Lambda}|\mathbf{W}, \nu). \tag{B.53}$$

For the particular case of a scalar x, this is equivalent to the Gaussian-gamma distribution.

Multinomial

If we generalize the Bernoulli distribution to an K-dimensional binary variable $\mathbf{x}$ with components $x_k \in \{0, 1\}$ such that $\sum_k x_k = 1$, then we obtain the following discrete distribution

$$p(\mathbf{x}) = \prod_{k=1}^{K} \mu_k^{x_k} \tag{B.54}$$

$$\mathbb{E}[x_k] = \mu_k \tag{B.55}$$

$$\mathrm{var}[x_k] = \mu_k(1 - \mu_k) \tag{B.56}$$

$$\mathrm{cov}[x_j x_k] = I_{jk}\mu_k \tag{B.57}$$

$$\mathrm{H}[\mathbf{x}] = -\sum_{k=1}^{M} \mu_k \ln \mu_k \tag{B.58}$$

where I_{jk} is the j, k element of the identity matrix. Because $p(x_k = 1) = \mu_k$, the parameters must satisfy $0 \leqslant \mu_k \leqslant 1$ and $\sum_k \mu_k = 1$.

The multinomial distribution is a multivariate generalization of the binomial and gives the distribution over counts m_k for a K-state discrete variable to be in state k given a total number of observations N.

$$\mathrm{Mult}(m_1, m_2, \ldots, m_K | \boldsymbol{\mu}, N) = \binom{N}{m_1 m_2 \ldots m_M} \prod_{k=1}^{M} \mu_k^{m_k} \tag{B.59}$$

$$\mathbb{E}[m_k] = N\mu_k \tag{B.60}$$

$$\mathrm{var}[m_k] = N\mu_k(1 - \mu_k) \tag{B.61}$$

$$\mathrm{cov}[m_j m_k] = -N\mu_j\mu_k \tag{B.62}$$

where $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_K)^{\mathrm{T}}$, and the quantity

$$\binom{N}{m_1 m_2 \ldots m_K} = \frac{N!}{m_1! \ldots m_K!} \tag{B.63}$$

gives the number of ways of taking N identical objects and assigning m_k of them to bin k for $k = 1, \ldots, K$. The value of μ_k gives the probability of the random variable taking state k, and so these parameters are subject to the constraints $0 \leqslant \mu_k \leqslant 1$ and $\sum_k \mu_k = 1$. The conjugate prior distribution for the parameters $\{\mu_k\}$ is the Dirichlet.

Normal

The normal distribution is simply another name for the Gaussian. In this book, we use the term Gaussian throughout, although we retain the conventional use of the symbol $\mathcal{N}$ to denote this distribution. For consistency, we shall refer to the normal-gamma distribution as the Gaussian-gamma distribution, and similarly the normal-Wishart is called the Gaussian-Wishart.

Student's t

This distribution was published by William Gosset in 1908, but his employer, Guiness Breweries, required him to publish under a pseudonym, so he chose 'Student'. In the univariate form, Student's t-distribution is obtained by placing a conjugate gamma prior over the precision of a univariate Gaussian distribution and then integrating out the precision variable. It can therefore be viewed as an infinite mixture

of Gaussians having the same mean but different variances.

$$\mathrm{St}(x|\mu,\lambda,\nu) = \frac{\Gamma(\nu/2+1/2)}{\Gamma(\nu/2)}\left(\frac{\lambda}{\pi\nu}\right)^{1/2}\left[1+\frac{\lambda(x-\mu)^2}{\nu}\right]^{-\nu/2-1/2} \tag{B.64}$$

$$\mathbb{E}[x] = \mu \quad \text{for } \nu > 1 \tag{B.65}$$

$$\mathrm{var}[x] = \frac{1}{\lambda}\frac{\nu}{\nu-2} \quad \text{for } \nu > 2 \tag{B.66}$$

$$\mathrm{mode}[x] = \mu. \tag{B.67}$$

Here $\nu > 0$ is called the number of degrees of freedom of the distribution. The particular case of $\nu = 1$ is called the *Cauchy* distribution.

For a D-dimensional variable $\mathbf{x}$, Student's t-distribution corresponds to marginalizing the precision matrix of a multivariate Gaussian with respect to a conjugate Wishart prior and takes the form

$$\mathrm{St}(\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\Lambda},\nu) = \frac{\Gamma(\nu/2+D/2)}{\Gamma(\nu/2)}\frac{|\boldsymbol{\Lambda}|^{1/2}}{(\nu\pi)^{D/2}}\left[1+\frac{\Delta^2}{\nu}\right]^{-\nu/2-D/2} \tag{B.68}$$

$$\mathbb{E}[\mathbf{x}] = \boldsymbol{\mu} \quad \text{for } \nu > 1 \tag{B.69}$$

$$\mathrm{cov}[\mathbf{x}] = \frac{\nu}{\nu-2}\boldsymbol{\Lambda}^{-1} \quad \text{for } \nu > 2 \tag{B.70}$$

$$\mathrm{mode}[\mathbf{x}] = \boldsymbol{\mu} \tag{B.71}$$

where Δ^2 is the squared Mahalanobis distance defined by

$$\Delta^2 = (\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Lambda}(\mathbf{x}-\boldsymbol{\mu}). \tag{B.72}$$

In the limit $\nu \to \infty$, the t-distribution reduces to a Gaussian with mean μ and precision $\boldsymbol{\Lambda}$. Student's t-distribution provides a generalization of the Gaussian whose maximum likelihood parameter values are robust to outliers.

Uniform

This is a simple distribution for a continuous variable x defined over a finite interval $x \in [a, b]$ where $b > a$.

$$\mathrm{U}(x|a,b) = \frac{1}{b-a} \tag{B.73}$$

$$\mathbb{E}[x] = \frac{(b+a)}{2} \tag{B.74}$$

$$\mathrm{var}[x] = \frac{(b-a)^2}{12} \tag{B.75}$$

$$\mathrm{H}[x] = \ln(b-a). \tag{B.76}$$

If x has distribution $\mathrm{U}(x|0,1)$, then $a+(b-a)x$ will have distribution $\mathrm{U}(x|a,b)$.

Von Mises

The von Mises distribution, also known as the circular normal or the circular Gaussian, is a univariate Gaussian-like periodic distribution for a variable $\theta \in [0, 2\pi)$.

$$p(\theta|\theta_0, m) = \frac{1}{2\pi I_0(m)} \exp\{m\cos(\theta - \theta_0)\} \tag{B.77}$$

where $I_0(m)$ is the zeroth-order Bessel function of the first kind. The distribution has period 2π so that $p(\theta + 2\pi) = p(\theta)$ for all θ. Care must be taken in interpreting this distribution because simple expectations will be dependent on the (arbitrary) choice of origin for the variable θ. The parameter θ_0 is analogous to the mean of a univariate Gaussian, and the parameter $m > 0$, known as the *concentration* parameter, is analogous to the precision (inverse variance). For large m, the von Mises distribution is approximately a Gaussian centred on θ_0.

Wishart

The Wishart distribution is the conjugate prior for the precision matrix of a multivariate Gaussian.

$$\mathcal{W}(\mathbf{\Lambda}|\mathbf{W}, \nu) = B(\mathbf{W}, \nu)|\mathbf{\Lambda}|^{(\nu-D-1)/2} \exp\left(-\frac{1}{2}\mathrm{Tr}(\mathbf{W}^{-1}\mathbf{\Lambda})\right) \tag{B.78}$$

where

$$B(\mathbf{W}, \nu) \equiv |\mathbf{W}|^{-\nu/2} \left(2^{\nu D/2}\, \pi^{D(D-1)/4} \prod_{i=1}^{D} \Gamma\left(\frac{\nu+1-i}{2}\right)\right)^{-1} \tag{B.79}$$

$$\mathbb{E}[\mathbf{\Lambda}] = \nu\mathbf{W} \tag{B.80}$$

$$\mathbb{E}\left[\ln|\mathbf{\Lambda}|\right] = \sum_{i=1}^{D} \psi\left(\frac{\nu+1-i}{2}\right) + D\ln 2 + \ln|\mathbf{W}| \tag{B.81}$$

$$\mathrm{H}[\mathbf{\Lambda}] = -\ln B(\mathbf{W}, \nu) - \frac{(\nu-D-1)}{2}\mathbb{E}\left[\ln|\mathbf{\Lambda}|\right] + \frac{\nu D}{2} \tag{B.82}$$

where $\mathbf{W}$ is a $D \times D$ symmetric, positive definite matrix, and $\psi(\cdot)$ is the digamma function defined by (B.25). The parameter ν is called the *number of degrees of freedom* of the distribution and is restricted to $\nu > D - 1$ to ensure that the Gamma function in the normalization factor is well-defined. In one dimension, the Wishart reduces to the gamma distribution $\mathrm{Gam}(\lambda|a, b)$ given by (B.26) with parameters $a = \nu/2$ and $b = 1/2W$.

Appendix C. Properties of Matrices

In this appendix, we gather together some useful properties and identities involving matrices and determinants. This is not intended to be an introductory tutorial, and it is assumed that the reader is already familiar with basic linear algebra. For some results, we indicate how to prove them, whereas in more complex cases we leave the interested reader to refer to standard textbooks on the subject. In all cases, we assume that inverses exist and that matrix dimensions are such that the formulae are correctly defined. A comprehensive discussion of linear algebra can be found in Golub and Van Loan (1996), and an extensive collection of matrix properties is given by Lütkepohl (1996). Matrix derivatives are discussed in Magnus and Neudecker (1999).

Basic Matrix Identities

A matrix $\mathbf{A}$ has elements A_{ij} where i indexes the rows, and j indexes the columns. We use $\mathbf{I}_N$ to denote the $N \times N$ identity matrix (also called the unit matrix), and where there is no ambiguity over dimensionality we simply use $\mathbf{I}$. The transpose matrix $\mathbf{A}^{\mathrm{T}}$ has elements $(\mathbf{A}^{\mathrm{T}})_{ij} = A_{ji}$. From the definition of transpose, we have

$$(\mathbf{AB})^{\mathrm{T}} = \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}} \tag{C.1}$$

which can be verified by writing out the indices. The inverse of $\mathbf{A}$, denoted $\mathbf{A}^{-1}$, satisfies

$$\mathbf{AA}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}. \tag{C.2}$$

Because $\mathbf{ABB}^{-1}\mathbf{A}^{-1} = \mathbf{I}$, we have

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}. \tag{C.3}$$

Also we have

$$\left(\mathbf{A}^{\mathrm{T}}\right)^{-1} = \left(\mathbf{A}^{-1}\right)^{\mathrm{T}} \tag{C.4}$$

which is easily proven by taking the transpose of (C.2) and applying (C.1).

A useful identity involving matrix inverses is the following

$$(\mathbf{P}^{-1} + \mathbf{B}^{\mathrm{T}}\mathbf{R}^{-1}\mathbf{B})^{-1}\mathbf{B}^{\mathrm{T}}\mathbf{R}^{-1} = \mathbf{P}\mathbf{B}^{\mathrm{T}}(\mathbf{B}\mathbf{P}\mathbf{B}^{\mathrm{T}} + \mathbf{R})^{-1}. \tag{C.5}$$

which is easily verified by right multiplying both sides by $(\mathbf{B}\mathbf{P}\mathbf{B}^{\mathrm{T}} + \mathbf{R})$. Suppose that $\mathbf{P}$ has dimensionality $N \times N$ while $\mathbf{R}$ has dimensionality $M \times M$, so that $\mathbf{B}$ is $M \times N$. Then if $M \ll N$, it will be much cheaper to evaluate the right-hand side of (C.5) than the left-hand side. A special case that sometimes arises is

$$(\mathbf{I} + \mathbf{A}\mathbf{B})^{-1}\mathbf{A} = \mathbf{A}(\mathbf{I} + \mathbf{B}\mathbf{A})^{-1}. \tag{C.6}$$

Another useful identity involving inverses is the following:

$$(\mathbf{A} + \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{B}(\mathbf{D} + \mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1}\mathbf{C}\mathbf{A}^{-1} \tag{C.7}$$

which is known as the *Woodbury identity* and which can be verified by multiplying both sides by $(\mathbf{A} + \mathbf{B}\mathbf{D}^{-1}\mathbf{C})$. This is useful, for instance, when $\mathbf{A}$ is large and diagonal, and hence easy to invert, while $\mathbf{B}$ has many rows but few columns (and conversely for $\mathbf{C}$) so that the right-hand side is much cheaper to evaluate than the left-hand side.

A set of vectors $\{\mathbf{a}_1, \ldots, \mathbf{a}_N\}$ is said to be *linearly independent* if the relation $\sum_n \alpha_n \mathbf{a}_n = 0$ holds only if all $\alpha_n = 0$. This implies that none of the vectors can be expressed as a linear combination of the remainder. The rank of a matrix is the maximum number of linearly independent rows (or equivalently the maximum number of linearly independent columns).

Traces and Determinants

Trace and determinant apply to square matrices. The trace $\mathrm{Tr}(\mathbf{A})$ of a matrix $\mathbf{A}$ is defined as the sum of the elements on the leading diagonal. By writing out the indices, we see that

$$\mathrm{Tr}(\mathbf{A}\mathbf{B}) = \mathrm{Tr}(\mathbf{B}\mathbf{A}). \tag{C.8}$$

By applying this formula multiple times to the product of three matrices, we see that

$$\mathrm{Tr}(\mathbf{A}\mathbf{B}\mathbf{C}) = \mathrm{Tr}(\mathbf{C}\mathbf{A}\mathbf{B}) = \mathrm{Tr}(\mathbf{B}\mathbf{C}\mathbf{A}) \tag{C.9}$$

which is known as the *cyclic* property of the trace operator and which clearly extends to the product of any number of matrices. The determinant $|\mathbf{A}|$ of an $N \times N$ matrix $\mathbf{A}$ is defined by

$$|\mathbf{A}| = \sum (\pm 1) A_{1i_1} A_{2i_2} \cdots A_{Ni_N} \tag{C.10}$$

in which the sum is taken over all products consisting of precisely one element from each row and one element from each column, with a coefficient $+1$ or -1 according

to whether the permutation $i_1 i_2 \ldots i_N$ is even or odd, respectively. Note that $|\mathbf{I}| = 1$. Thus, for a 2×2 matrix, the determinant takes the form

$$|\mathbf{A}| = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}. \tag{C.11}$$

The determinant of a product of two matrices is given by

$$|\mathbf{AB}| = |\mathbf{A}||\mathbf{B}| \tag{C.12}$$

as can be shown from (C.10). Also, the determinant of an inverse matrix is given by

$$\left|\mathbf{A}^{-1}\right| = \frac{1}{|\mathbf{A}|} \tag{C.13}$$

which can be shown by taking the determinant of (C.2) and applying (C.12).

If $\mathbf{A}$ and $\mathbf{B}$ are matrices of size $N \times M$, then

$$\left|\mathbf{I}_N + \mathbf{AB}^{\mathrm{T}}\right| = \left|\mathbf{I}_M + \mathbf{A}^{\mathrm{T}}\mathbf{B}\right|. \tag{C.14}$$

A useful special case is

$$\left|\mathbf{I}_N + \mathbf{ab}^{\mathrm{T}}\right| = 1 + \mathbf{a}^{\mathrm{T}}\mathbf{b} \tag{C.15}$$

where $\mathbf{a}$ and $\mathbf{b}$ are N-dimensional column vectors.

Matrix Derivatives

Sometimes we need to consider derivatives of vectors and matrices with respect to scalars. The derivative of a vector $\mathbf{a}$ with respect to a scalar x is itself a vector whose components are given by

$$\left(\frac{\partial \mathbf{a}}{\partial x}\right)_i = \frac{\partial a_i}{\partial x} \tag{C.16}$$

with an analogous definition for the derivative of a matrix. Derivatives with respect to vectors and matrices can also be defined, for instance

$$\left(\frac{\partial x}{\partial \mathbf{a}}\right)_i = \frac{\partial x}{\partial a_i} \tag{C.17}$$

and similarly

$$\left(\frac{\partial \mathbf{a}}{\partial \mathbf{b}}\right)_{ij} = \frac{\partial a_i}{\partial b_j}. \tag{C.18}$$

The following is easily proven by writing out the components

$$\frac{\partial}{\partial \mathbf{x}}\left(\mathbf{x}^{\mathrm{T}}\mathbf{a}\right) = \frac{\partial}{\partial \mathbf{x}}\left(\mathbf{a}^{\mathrm{T}}\mathbf{x}\right) = \mathbf{a}. \tag{C.19}$$

Similarly

$$\frac{\partial}{\partial \mathbf{x}}(\mathbf{AB}) = \frac{\partial \mathbf{A}}{\partial \mathbf{x}}\mathbf{B} + \mathbf{A}\frac{\partial \mathbf{B}}{\partial \mathbf{x}}. \tag{C.20}$$

The derivative of the inverse of a matrix can be expressed as

$$\frac{\partial}{\partial x}\left(\mathbf{A}^{-1}\right) = -\mathbf{A}^{-1}\frac{\partial \mathbf{A}}{\partial x}\mathbf{A}^{-1} \tag{C.21}$$

as can be shown by differentiating the equation $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ using (C.20) and then right multiplying by $\mathbf{A}^{-1}$. Also

$$\frac{\partial}{\partial x}\ln|\mathbf{A}| = \mathrm{Tr}\left(\mathbf{A}^{-1}\frac{\partial \mathbf{A}}{\partial x}\right) \tag{C.22}$$

which we shall prove later. If we choose x to be one of the elements of $\mathbf{A}$, we have

$$\frac{\partial}{\partial A_{ij}}\mathrm{Tr}\,(\mathbf{AB}) = B_{ji} \tag{C.23}$$

as can be seen by writing out the matrices using index notation. We can write this result more compactly in the form

$$\frac{\partial}{\partial \mathbf{A}}\mathrm{Tr}\,(\mathbf{AB}) = \mathbf{B}^{\mathrm{T}}. \tag{C.24}$$

With this notation, we have the following properties

$$\frac{\partial}{\partial \mathbf{A}}\mathrm{Tr}\left(\mathbf{A}^{\mathrm{T}}\mathbf{B}\right) = \mathbf{B} \tag{C.25}$$

$$\frac{\partial}{\partial \mathbf{A}}\mathrm{Tr}(\mathbf{A}) = \mathbf{I} \tag{C.26}$$

$$\frac{\partial}{\partial \mathbf{A}}\mathrm{Tr}(\mathbf{ABA}^{\mathrm{T}}) = \mathbf{A}(\mathbf{B} + \mathbf{B}^{\mathrm{T}}) \tag{C.27}$$

which can again be proven by writing out the matrix indices. We also have

$$\frac{\partial}{\partial \mathbf{A}}\ln|\mathbf{A}| = \left(\mathbf{A}^{-1}\right)^{\mathrm{T}} \tag{C.28}$$

which follows from (C.22) and (C.26).

Eigenvector Equation

For a square matrix $\mathbf{A}$ of size $M \times M$, the eigenvector equation is defined by

$$\mathbf{A}\mathbf{u}_i = \lambda_i \mathbf{u}_i \tag{C.29}$$

for $i = 1, \ldots, M$, where $\mathbf{u}_i$ is an *eigenvector* and λ_i is the corresponding *eigenvalue*. This can be viewed as a set of M simultaneous homogeneous linear equations, and the condition for a solution is that

$$|\mathbf{A} - \lambda_i \mathbf{I}| = 0 \tag{C.30}$$

which is known as the *characteristic equation*. Because this is a polynomial of order M in λ_i, it must have M solutions (though these need not all be distinct). The rank of $\mathbf{A}$ is equal to the number of nonzero eigenvalues.

Of particular interest are symmetric matrices, which arise as covariance matrices, kernel matrices, and Hessians. Symmetric matrices have the property that $A_{ij} = A_{ji}$, or equivalently $\mathbf{A}^{\mathrm{T}} = \mathbf{A}$. The inverse of a symmetric matrix is also symmetric, as can be seen by taking the transpose of $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ and using $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ together with the symmetry of $\mathbf{I}$.

In general, the eigenvalues of a matrix are complex numbers, but for symmetric matrices the eigenvalues λ_i are real. This can be seen by first left multiplying (C.29) by $(\mathbf{u}_i^\star)^{\mathrm{T}}$, where $\star$ denotes the complex conjugate, to give

$$(\mathbf{u}_i^\star)^{\mathrm{T}} \mathbf{A}\mathbf{u}_i = \lambda_i \, (\mathbf{u}_i^\star)^{\mathrm{T}} \mathbf{u}_i. \tag{C.31}$$

Next we take the complex conjugate of (C.29) and left multiply by $\mathbf{u}_i^{\mathrm{T}}$ to give

$$\mathbf{u}_i^{\mathrm{T}} \mathbf{A}\mathbf{u}_i^\star = \lambda_i^\star \mathbf{u}_i^{\mathrm{T}} \mathbf{u}_i^\star. \tag{C.32}$$

where we have used $\mathbf{A}^\star = \mathbf{A}$ because we consider only real matrices $\mathbf{A}$. Taking the transpose of the second of these equations, and using $\mathbf{A}^{\mathrm{T}} = \mathbf{A}$, we see that the left-hand sides of the two equations are equal, and hence that $\lambda_i^\star = \lambda_i$ and so λ_i must be real.

The eigenvectors $\mathbf{u}_i$ of a real symmetric matrix can be chosen to be orthonormal (i.e., orthogonal and of unit length) so that

$$\mathbf{u}_i^{\mathrm{T}} \mathbf{u}_j = I_{ij} \tag{C.33}$$

where I_{ij} are the elements of the identity matrix $\mathbf{I}$. To show this, we first left multiply (C.29) by $\mathbf{u}_j^{\mathrm{T}}$ to give

$$\mathbf{u}_j^{\mathrm{T}} \mathbf{A}\mathbf{u}_i = \lambda_i \mathbf{u}_j^{\mathrm{T}} \mathbf{u}_i \tag{C.34}$$

and hence, by exchange of indices, we have

$$\mathbf{u}_i^{\mathrm{T}} \mathbf{A}\mathbf{u}_j = \lambda_j \mathbf{u}_i^{\mathrm{T}} \mathbf{u}_j. \tag{C.35}$$

We now take the transpose of the second equation and make use of the symmetry property $\mathbf{A}^{\mathrm{T}} = \mathbf{A}$, and then subtract the two equations to give

$$(\lambda_i - \lambda_j) \, \mathbf{u}_i^{\mathrm{T}} \mathbf{u}_j = 0. \tag{C.36}$$

Hence, for $\lambda_i \neq \lambda_j$, we have $\mathbf{u}_i^{\mathrm{T}} \mathbf{u}_j = 0$, and hence $\mathbf{u}_i$ and $\mathbf{u}_j$ are orthogonal. If the two eigenvalues are equal, then any linear combination $\alpha \mathbf{u}_i + \beta \mathbf{u}_j$ is also an eigenvector with the same eigenvalue, so we can select one linear combination arbitrarily,

and then choose the second to be orthogonal to the first (it can be shown that the degenerate eigenvectors are never linearly dependent). Hence the eigenvectors can be chosen to be orthogonal, and by normalizing can be set to unit length. Because there are M eigenvalues, the corresponding M orthogonal eigenvectors form a complete set and so any M-dimensional vector can be expressed as a linear combination of the eigenvectors.

We can take the eigenvectors $\mathbf{u}_i$ to be the columns of an $M \times M$ matrix $\mathbf{U}$, which from orthonormality satisfies

$$\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{I}. \tag{C.37}$$

Such a matrix is said to be *orthogonal*. Interestingly, the rows of this matrix are also orthogonal, so that $\mathbf{U}\mathbf{U}^{\mathrm{T}} = \mathbf{I}$. To show this, note that (C.37) implies $\mathbf{U}^{\mathrm{T}}\mathbf{U}\mathbf{U}^{-1} = \mathbf{U}^{-1} = \mathbf{U}^{\mathrm{T}}$ and so $\mathbf{U}\mathbf{U}^{-1} = \mathbf{U}\mathbf{U}^{\mathrm{T}} = \mathbf{I}$. Using (C.12), it also follows that $|\mathbf{U}| = 1$.

The eigenvector equation (C.29) can be expressed in terms of $\mathbf{U}$ in the form

$$\mathbf{A}\mathbf{U} = \mathbf{U}\boldsymbol{\Lambda} \tag{C.38}$$

where $\boldsymbol{\Lambda}$ is an $M \times M$ diagonal matrix whose diagonal elements are given by the eigenvalues λ_i.

If we consider a column vector $\mathbf{x}$ that is transformed by an orthogonal matrix $\mathbf{U}$ to give a new vector

$$\widetilde{\mathbf{x}} = \mathbf{U}\mathbf{x} \tag{C.39}$$

then the length of the vector is preserved because

$$\widetilde{\mathbf{x}}^{\mathrm{T}}\widetilde{\mathbf{x}} = \mathbf{x}^{\mathrm{T}}\mathbf{U}^{\mathrm{T}}\mathbf{U}\mathbf{x} = \mathbf{x}^{\mathrm{T}}\mathbf{x} \tag{C.40}$$

and similarly the angle between any two such vectors is preserved because

$$\widetilde{\mathbf{x}}^{\mathrm{T}}\widetilde{\mathbf{y}} = \mathbf{x}^{\mathrm{T}}\mathbf{U}^{\mathrm{T}}\mathbf{U}\mathbf{y} = \mathbf{x}^{\mathrm{T}}\mathbf{y}. \tag{C.41}$$

Thus, multiplication by $\mathbf{U}$ can be interpreted as a rigid rotation of the coordinate system.

From (C.38), it follows that

$$\mathbf{U}^{\mathrm{T}}\mathbf{A}\mathbf{U} = \boldsymbol{\Lambda} \tag{C.42}$$

and because $\boldsymbol{\Lambda}$ is a diagonal matrix, we say that the matrix $\mathbf{A}$ is *diagonalized* by the matrix $\mathbf{U}$. If we left multiply by $\mathbf{U}$ and right multiply by $\mathbf{U}^{\mathrm{T}}$, we obtain

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^{\mathrm{T}} \tag{C.43}$$

Taking the inverse of this equation, and using (C.3) together with $\mathbf{U}^{-1} = \mathbf{U}^{\mathrm{T}}$, we have

$$\mathbf{A}^{-1} = \mathbf{U}\boldsymbol{\Lambda}^{-1}\mathbf{U}^{\mathrm{T}}. \tag{C.44}$$

These last two equations can also be written in the form

$$\mathbf{A} = \sum_{i=1}^{M} \lambda_i \mathbf{u}_i \mathbf{u}_i^{\mathrm{T}} \tag{C.45}$$

$$\mathbf{A}^{-1} = \sum_{i=1}^{M} \frac{1}{\lambda_i} \mathbf{u}_i \mathbf{u}_i^{\mathrm{T}}. \tag{C.46}$$

If we take the determinant of (C.43), and use (C.12), we obtain

$$|\mathbf{A}| = \prod_{i=1}^{M} \lambda_i. \tag{C.47}$$

Similarly, taking the trace of (C.43), and using the cyclic property (C.8) of the trace operator together with $\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{I}$, we have

$$\mathrm{Tr}(\mathbf{A}) = \sum_{i=1}^{M} \lambda_i. \tag{C.48}$$

We leave it as an exercise for the reader to verify (C.22) by making use of the results (C.33), (C.45), (C.46), and (C.47).

A matrix $\mathbf{A}$ is said to be *positive definite*, denoted by $\mathbf{A} \succ 0$, if $\mathbf{w}^{\mathrm{T}}\mathbf{A}\mathbf{w} > 0$ for all values of the vector $\mathbf{w}$. Equivalently, a positive definite matrix has $\lambda_i > 0$ for all of its eigenvalues (as can be seen by setting $\mathbf{w}$ to each of the eigenvectors in turn, and by noting that an arbitrary vector can be expanded as a linear combination of the eigenvectors). Note that positive definite is not the same as all the elements being positive. For example, the matrix

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \tag{C.49}$$

has eigenvalues $\lambda_1 \simeq 5.37$ and $\lambda_2 \simeq -0.37$. A matrix is said to be *positive semidefinite* if $\mathbf{w}^{\mathrm{T}}\mathbf{A}\mathbf{w} \geqslant 0$ holds for all values of $\mathbf{w}$, which is denoted $\mathbf{A} \succeq 0$, and is equivalent to $\lambda_i \geqslant 0$.

Appendix D. Calculus of Variations

We can think of a function $y(x)$ as being an operator that, for any input value x, returns an output value y. In the same way, we can define a *functional* $F[y]$ to be an operator that takes a function $y(x)$ and returns an output value F. An example of a functional is the length of a curve drawn in a two-dimensional plane in which the path of the curve is defined in terms of a function. In the context of machine learning, a widely used functional is the entropy $\mathrm{H}[x]$ for a continuous variable x because, for any choice of probability density function $p(x)$, it returns a scalar value representing the entropy of x under that density. Thus the entropy of $p(x)$ could equally well have been written as $\mathrm{H}[p]$.

A common problem in conventional calculus is to find a value of x that maximizes (or minimizes) a function $y(x)$. Similarly, in the calculus of variations we seek a function $y(x)$ that maximizes (or minimizes) a functional $F[y]$. That is, of all possible functions $y(x)$, we wish to find the particular function for which the functional $F[y]$ is a maximum (or minimum). The calculus of variations can be used, for instance, to show that the shortest path between two points is a straight line or that the maximum entropy distribution is a Gaussian.

If we weren't familiar with the rules of ordinary calculus, we could evaluate a conventional derivative $\mathrm{d}y/\mathrm{d}x$ by making a small change ϵ to the variable x and then expanding in powers of ϵ, so that

$$y(x+\epsilon) = y(x) + \frac{\mathrm{d}y}{\mathrm{d}x}\epsilon + O(\epsilon^2) \tag{D.1}$$

and finally taking the limit $\epsilon \to 0$. Similarly, for a function of several variables $y(x_1, \ldots, x_D)$, the corresponding partial derivatives are defined by

$$y(x_1+\epsilon_1, \ldots, x_D+\epsilon_D) = y(x_1, \ldots, x_D) + \sum_{i=1}^{D} \frac{\partial y}{\partial x_i}\epsilon_i + O(\epsilon^2). \tag{D.2}$$

The analogous definition of a functional derivative arises when we consider how much a functional $F[y]$ changes when we make a small change $\epsilon\eta(x)$ to the function

Figure D.1 A functional derivative can be defined by considering how the value of a functional $F[y]$ changes when the function $y(x)$ is changed to $y(x) + \epsilon\eta(x)$ where $\eta(x)$ is an arbitrary function of x.

$y(x)$, where $\eta(x)$ is an arbitrary function of x, as illustrated in Figure D.1. We denote the functional derivative of $E[f]$ with respect to $f(x)$ by $\delta F/\delta f(x)$, and define it by the following relation:

$$F[y(x) + \epsilon\eta(x)] = F[y(x)] + \epsilon \int \frac{\delta F}{\delta y(x)} \eta(x)\, \mathrm{d}x + O(\epsilon^2). \tag{D.3}$$

This can be seen as a natural extension of (D.2) in which $F[y]$ now depends on a continuous set of variables, namely the values of y at all points x. Requiring that the functional be stationary with respect to small variations in the function $y(x)$ gives

$$\int \frac{\delta E}{\delta y(x)} \eta(x)\, \mathrm{d}x = 0. \tag{D.4}$$

Because this must hold for an arbitrary choice of $\eta(x)$, it follows that the functional derivative must vanish. To see this, imagine choosing a perturbation $\eta(x)$ that is zero everywhere except in the neighbourhood of a point $\widehat{x}$, in which case the functional derivative must be zero at $x = \widehat{x}$. However, because this must be true for every choice of $\widehat{x}$, the functional derivative must vanish for all values of x.

Consider a functional that is defined by an integral over a function $G(y, y', x)$ that depends on both $y(x)$ and its derivative $y'(x)$ as well as having a direct dependence on x

$$F[y] = \int G\left(y(x), y'(x), x\right)\, \mathrm{d}x \tag{D.5}$$

where the value of $y(x)$ is assumed to be fixed at the boundary of the region of integration (which might be at infinity). If we now consider variations in the function $y(x)$, we obtain

$$F[y(x) + \epsilon\eta(x)] = F[y(x)] + \epsilon \int \left\{ \frac{\partial G}{\partial y}\eta(x) + \frac{\partial G}{\partial y'}\eta'(x) \right\} \mathrm{d}x + O(\epsilon^2). \tag{D.6}$$

We now have to cast this in the form (D.3). To do so, we integrate the second term by parts and make use of the fact that $\eta(x)$ must vanish at the boundary of the integral (because $y(x)$ is fixed at the boundary). This gives

$$F[y(x) + \epsilon\eta(x)] = F[y(x)] + \epsilon \int \left\{ \frac{\partial G}{\partial y} - \frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{\partial G}{\partial y'}\right) \right\} \eta(x)\, \mathrm{d}x + O(\epsilon^2) \tag{D.7}$$

from which we can read off the functional derivative by comparison with (D.3). Requiring that the functional derivative vanishes then gives

$$\frac{\partial G}{\partial y} - \frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{\partial G}{\partial y'}\right) = 0 \tag{D.8}$$

which are known as the *Euler-Lagrange* equations. For example, if

$$G = y(x)^2 + \left(y'(x)\right)^2 \tag{D.9}$$

then the Euler-Lagrange equations take the form

$$y(x) - \frac{\mathrm{d}^2 y}{\mathrm{d}x^2} = 0. \tag{D.10}$$

This second order differential equation can be solved for $y(x)$ by making use of the boundary conditions on $y(x)$.

Often, we consider functionals defined by integrals whose integrands take the form $G(y, x)$ and that do not depend on the derivatives of $y(x)$. In this case, stationarity simply requires that $\partial G/\partial y(x) = 0$ for all values of x.

If we are optimizing a functional with respect to a probability distribution, then we need to maintain the normalization constraint on the probabilities. This is often most conveniently done using a Lagrange multiplier, which then allows an unconstrained optimization to be performed.

Appendix E

The extension of the above results to a multidimensional variable $\mathbf{x}$ is straightforward. For a more comprehensive discussion of the calculus of variations, see Sagan (1969).

Appendix E. Lagrange Multipliers

Lagrange multipliers, also sometimes called *undetermined multipliers*, are used to find the stationary points of a function of several variables subject to one or more constraints.

Consider the problem of finding the maximum of a function $f(x_1, x_2)$ subject to a constraint relating x_1 and x_2, which we write in the form

$$g(x_1, x_2) = 0. \tag{E.1}$$

One approach would be to solve the constraint equation (E.1) and thus express x_2 as a function of x_1 in the form $x_2 = h(x_1)$. This can then be substituted into $f(x_1, x_2)$ to give a function of x_1 alone of the form $f(x_1, h(x_1))$. The maximum with respect to x_1 could then be found by differentiation in the usual way, to give the stationary value $x_1^\star$, with the corresponding value of x_2 given by $x_2^\star = h(x_1^\star)$.

One problem with this approach is that it may be difficult to find an analytic solution of the constraint equation that allows x_2 to be expressed as an explicit function of x_1. Also, this approach treats x_1 and x_2 differently and so spoils the natural symmetry between these variables.

A more elegant, and often simpler, approach is based on the introduction of a parameter λ called a Lagrange multiplier. We shall motivate this technique from a geometrical perspective. Consider a D-dimensional variable $\mathbf{x}$ with components $x_1, \ldots, x_D$. The constraint equation $g(\mathbf{x}) = 0$ then represents a $(D-1)$-dimensional surface in $\mathbf{x}$-space as indicated in Figure E.1.

We first note that at any point on the constraint surface the gradient $\nabla g(\mathbf{x})$ of the constraint function will be orthogonal to the surface. To see this, consider a point $\mathbf{x}$ that lies on the constraint surface, and consider a nearby point $\mathbf{x} + \boldsymbol{\epsilon}$ that also lies on the surface. If we make a Taylor expansion around $\mathbf{x}$, we have

$$g(\mathbf{x} + \boldsymbol{\epsilon}) \simeq g(\mathbf{x}) + \boldsymbol{\epsilon}^{\mathrm{T}} \nabla g(\mathbf{x}). \tag{E.2}$$

Because both $\mathbf{x}$ and $\mathbf{x} + \boldsymbol{\epsilon}$ lie on the constraint surface, we have $g(\mathbf{x}) = g(\mathbf{x} + \boldsymbol{\epsilon})$ and hence $\boldsymbol{\epsilon}^{\mathrm{T}} \nabla g(\mathbf{x}) \simeq 0$. In the limit $\|\boldsymbol{\epsilon}\| \to 0$ we have $\boldsymbol{\epsilon}^{\mathrm{T}} \nabla g(\mathbf{x}) = 0$, and because $\boldsymbol{\epsilon}$ is

Figure E.1 A geometrical picture of the technique of Lagrange multipliers in which we seek to maximize a function $f(\mathbf{x})$, subject to the constraint $g(\mathbf{x}) = 0$. If $\mathbf{x}$ is D dimensional, the constraint $g(\mathbf{x}) = 0$ corresponds to a subspace of dimensionality $D - 1$, indicated by the red curve. The problem can be solved by optimizing the Lagrangian function $L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda g(\mathbf{x})$.

then parallel to the constraint surface $g(\mathbf{x}) = 0$, we see that the vector ∇g is normal to the surface.

Next we seek a point $\mathbf{x}^\star$ on the constraint surface such that $f(\mathbf{x})$ is maximized. Such a point must have the property that the vector $\nabla f(\mathbf{x})$ is also orthogonal to the constraint surface, as illustrated in Figure E.1, because otherwise we could increase the value of $f(\mathbf{x})$ by moving a short distance along the constraint surface. Thus ∇f and ∇g are parallel (or anti-parallel) vectors, and so there must exist a parameter λ such that

$$\nabla f + \lambda \nabla g = 0 \tag{E.3}$$

where $\lambda \neq 0$ is known as a *Lagrange multiplier*. Note that λ can have either sign.

At this point, it is convenient to introduce the *Lagrangian* function defined by

$$L(\mathbf{x}, \lambda) \equiv f(\mathbf{x}) + \lambda g(\mathbf{x}). \tag{E.4}$$

The constrained stationarity condition (E.3) is obtained by setting $\nabla_{\mathbf{x}} L = 0$. Furthermore, the condition $\partial L / \partial \lambda = 0$ leads to the constraint equation $g(\mathbf{x}) = 0$.

Thus to find the maximum of a function $f(\mathbf{x})$ subject to the constraint $g(\mathbf{x}) = 0$, we define the Lagrangian function given by (E.4) and we then find the stationary point of $L(\mathbf{x}, \lambda)$ with respect to both $\mathbf{x}$ and λ. For a D-dimensional vector $\mathbf{x}$, this gives $D + 1$ equations that determine both the stationary point $\mathbf{x}^\star$ and the value of λ. If we are only interested in $\mathbf{x}^\star$, then we can eliminate λ from the stationarity equations without needing to find its value (hence the term 'undetermined multiplier').

As a simple example, suppose we wish to find the stationary point of the function $f(x_1, x_2) = 1 - x_1^2 - x_2^2$ subject to the constraint $g(x_1, x_2) = x_1 + x_2 - 1 = 0$, as illustrated in Figure E.2. The corresponding Lagrangian function is given by

$$L(\mathbf{x}, \lambda) = 1 - x_1^2 - x_2^2 + \lambda(x_1 + x_2 - 1). \tag{E.5}$$

The conditions for this Lagrangian to be stationary with respect to x_1, x_2, and λ give the following coupled equations:

$$-2x_1 + \lambda = 0 \tag{E.6}$$

$$-2x_2 + \lambda = 0 \tag{E.7}$$

$$x_1 + x_2 - 1 = 0. \tag{E.8}$$

Figure E.2 A simple example of the use of Lagrange multipliers in which the aim is to maximize $f(x_1, x_2) = 1 - x_1^2 - x_2^2$ subject to the constraint $g(x_1, x_2) = 0$ where $g(x_1, x_2) = x_1 + x_2 - 1$. The circles show contours of the function $f(x_1, x_2)$, and the diagonal line shows the constraint surface $g(x_1, x_2) = 0$.

Solution of these equations then gives the stationary point as $(x_1^\star, x_2^\star) = (\frac{1}{2}, \frac{1}{2})$, and the corresponding value for the Lagrange multiplier is $\lambda = 1$.

So far, we have considered the problem of maximizing a function subject to an *equality constraint* of the form $g(\mathbf{x}) = 0$. We now consider the problem of maximizing $f(\mathbf{x})$ subject to an *inequality constraint* of the form $g(\mathbf{x}) \geqslant 0$, as illustrated in Figure E.3.

There are now two kinds of solution possible, according to whether the constrained stationary point lies in the region where $g(\mathbf{x}) > 0$, in which case the constraint is *inactive*, or whether it lies on the boundary $g(\mathbf{x}) = 0$, in which case the constraint is said to be *active*. In the former case, the function $g(\mathbf{x})$ plays no role and so the stationary condition is simply $\nabla f(\mathbf{x}) = 0$. This again corresponds to a stationary point of the Lagrange function (E.4) but this time with $\lambda = 0$. The latter case, where the solution lies on the boundary, is analogous to the equality constraint discussed previously and corresponds to a stationary point of the Lagrange function (E.4) with $\lambda \neq 0$. Now, however, the sign of the Lagrange multiplier is crucial, because the function $f(\mathbf{x})$ will only be at a maximum if its gradient is oriented away from the region $g(\mathbf{x}) > 0$, as illustrated in Figure E.3. We therefore have $\nabla f(\mathbf{x}) = -\lambda \nabla g(\mathbf{x})$ for some value of $\lambda > 0$.

For either of these two cases, the product $\lambda g(\mathbf{x}) = 0$. Thus the solution to the

Figure E.3 Illustration of the problem of maximizing $f(\mathbf{x})$ subject to the inequality constraint $g(\mathbf{x}) \geqslant 0$.

problem of maximizing $f(\mathbf{x})$ subject to $g(\mathbf{x}) \geqslant 0$ is obtained by optimizing the Lagrange function (E.4) with respect to $\mathbf{x}$ and λ subject to the conditions

$$g(\mathbf{x}) \geqslant 0 \tag{E.9}$$

$$\lambda \geqslant 0 \tag{E.10}$$

$$\lambda g(\mathbf{x}) = 0 \tag{E.11}$$

These are known as the *Karush-Kuhn-Tucker* (KKT) conditions (Karush, 1939; Kuhn and Tucker, 1951).

Note that if we wish to minimize (rather than maximize) the function $f(\mathbf{x})$ subject to an inequality constraint $g(\mathbf{x}) \geqslant 0$, then we minimize the Lagrangian function $L(\mathbf{x}, \lambda) = f(\mathbf{x}) - \lambda g(\mathbf{x})$ with respect to $\mathbf{x}$, again subject to $\lambda \geqslant 0$.

Finally, it is straightforward to extend the technique of Lagrange multipliers to the case of multiple equality and inequality constraints. Suppose we wish to maximize $f(\mathbf{x})$ subject to $g_j(\mathbf{x}) = 0$ for $j = 1, \ldots, J$, and $h_k(\mathbf{x}) \geqslant 0$ for $k = 1, \ldots, K$. We then introduce Lagrange multipliers $\{\lambda_j\}$ and $\{\mu_k\}$, and then optimize the Lagrangian function given by

$$L(\mathbf{x}, \{\lambda_j\}, \{\mu_k\}) = f(\mathbf{x}) + \sum_{j=1}^{J} \lambda_j g_j(\mathbf{x}) + \sum_{k=1}^{K} \mu_k h_k(\mathbf{x}) \tag{E.12}$$

subject to $\mu_k \geqslant 0$ and $\mu_k h_k(\mathbf{x}) = 0$ for $k = 1, \ldots, K$. Extensions to constrained functional derivatives are similarly straightforward. For a more detailed discussion of the technique of Lagrange multipliers, see Nocedal and Wright (1999).

Appendix D

References

Abramowitz, M. and I. A. Stegun (1965). *Handbook of Mathematical Functions*. Dover.

Adler, S. L. (1981). Over-relaxation method for the Monte Carlo evaluation of the partition function for multiquadratic actions. *Physical Review D* **23**, 2901–2904.

Ahn, J. H. and J. H. Oh (2003). A constrained EM algorithm for principal component analysis. *Neural Computation* **15**(1), 57–65.

Aizerman, M. A., E. M. Braverman, and L. I. Rozonoer (1964). The probability problem of pattern recognition learning and the method of potential functions. *Automation and Remote Control* **25**, 1175–1190.

Akaike, H. (1974). A new look at statistical model identification. *IEEE Transactions on Automatic Control* **19**, 716–723.

Ali, S. M. and S. D. Silvey (1966). A general class of coefficients of divergence of one distribution from another. *Journal of the Royal Statistical Society, B* **28**(1), 131–142.

Allwein, E. L., R. E. Schapire, and Y. Singer (2000). Reducing multiclass to binary: a unifying approach for margin classifiers. *Journal of Machine Learning Research* **1**, 113–141.

Amari, S. (1985). *Differential-Geometrical Methods in Statistics*. Springer.

Amari, S., A. Cichocki, and H. H. Yang (1996). A new learning algorithm for blind signal separation. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo (Eds.), *Advances in Neural Information Processing Systems*, Volume 8, pp. 757–763. MIT Press.

Amari, S. I. (1998). Natural gradient works efficiently in learning. *Neural Computation* **10**, 251–276.

Anderson, J. A. and E. Rosenfeld (Eds.) (1988). *Neurocomputing: Foundations of Research*. MIT Press.

Anderson, T. W. (1963). Asymptotic theory for principal component analysis. *Annals of Mathematical Statistics* **34**, 122–148.

Andrieu, C., N. de Freitas, A. Doucet, and M. I. Jordan (2003). An introduction to MCMC for machine learning. *Machine Learning* **50**, 5–43.

Anthony, M. and N. Biggs (1992). *An Introduction to Computational Learning Theory*. Cambridge University Press.

Attias, H. (1999a). Independent factor analysis. *Neural Computation* **11**(4), 803–851.

Attias, H. (1999b). Inferring parameters and structure of latent variable models by variational Bayes. In K. B. Laskey and H. Prade (Eds.),

Uncertainty in Artificial Intelligence: Proceedings of the Fifth Conference, pp. 21–30. Morgan Kaufmann.

Bach, F. R. and M. I. Jordan (2002). Kernel independent component analysis. *Journal of Machine Learning Research* **3**, 1–48.

Bakir, G. H., J. Weston, and B. Schölkopf (2004). Learning to find pre-images. In S. Thrun, L. K. Saul, and B. Schölkopf (Eds.), *Advances in Neural Information Processing Systems*, Volume 16, pp. 449–456. MIT Press.

Baldi, P. and S. Brunak (2001). *Bioinformatics: The Machine Learning Approach* (Second ed.). MIT Press.

Baldi, P. and K. Hornik (1989). Neural networks and principal component analysis: learning from examples without local minima. *Neural Networks* **2**(1), 53–58.

Barber, D. and C. M. Bishop (1997). Bayesian model comparison by Monte Carlo chaining. In M. Mozer, M. Jordan, and T. Petsche (Eds.), *Advances in Neural Information Processing Systems*, Volume 9, pp. 333–339. MIT Press.

Barber, D. and C. M. Bishop (1998a). Ensemble learning for multi-layer networks. In M. I. Jordan, K. J. Kearns, and S. A. Solla (Eds.), *Advances in Neural Information Processing Systems*, Volume 10, pp. 395–401.

Barber, D. and C. M. Bishop (1998b). Ensemble learning in Bayesian neural networks. In C. M. Bishop (Ed.), *Generalization in Neural Networks and Machine Learning*, pp. 215–237. Springer.

Bartholomew, D. J. (1987). *Latent Variable Models and Factor Analysis*. Charles Griffin.

Basilevsky, A. (1994). *Statistical Factor Analysis and Related Methods: Theory and Applications*. Wiley.

Bather, J. (2000). *Decision Theory: An Introduction to Dynamic Programming and Sequential Decisions*. Wiley.

Baudat, G. and F. Anouar (2000). Generalized discriminant analysis using a kernel approach. *Neural Computation* **12**(10), 2385–2404.

Baum, L. E. (1972). An inequality and associated maximization technique in statistical estimation of probabilistic functions of Markov processes. *Inequalities* **3**, 1–8.

Becker, S. and Y. Le Cun (1989). Improving the convergence of back-propagation learning with second order methods. In D. Touretzky, G. E. Hinton, and T. J. Sejnowski (Eds.), *Proceedings of the 1988 Connectionist Models Summer School*, pp. 29–37. Morgan Kaufmann.

Bell, A. J. and T. J. Sejnowski (1995). An information maximization approach to blind separation and blind deconvolution. *Neural Computation* **7**(6), 1129–1159.

Bellman, R. (1961). *Adaptive Control Processes: A Guided Tour*. Princeton University Press.

Bengio, Y. and P. Frasconi (1995). An input output HMM architecture. In G. Tesauro, D. S. Touretzky, and T. K. Leen (Eds.), *Advances in Neural Information Processing Systems*, Volume 7, pp. 427–434. MIT Press.

Bennett, K. P. (1992). Robust linear programming discrimination of two linearly separable sets. *Optimization Methods and Software* **1**, 23–34.

Berger, J. O. (1985). *Statistical Decision Theory and Bayesian Analysis* (Second ed.). Springer.

Bernardo, J. M. and A. F. M. Smith (1994). *Bayesian Theory*. Wiley.

Berrou, C., A. Glavieux, and P. Thitimajshima (1993). Near Shannon limit error-correcting coding and decoding: Turbo-codes (1). In *Proceedings ICC'93*, pp. 1064–1070.

Besag, J. (1974). On spatio-temporal models and Markov fields. In *Transactions of the 7th Prague Conference on Information Theory, Statistical Decision Functions and Random Processes*, pp. 47–75. Academia.

Besag, J. (1986). On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society* **B-48**, 259–302.

Besag, J., P. J. Green, D. Hidgon, and K. Megersen (1995). Bayesian computation and stochastic systems. *Statistical Science* **10**(1), 3–66.

Bishop, C. M. (1991). A fast procedure for retraining the multilayer perceptron. *International Journal of Neural Systems* **2**(3), 229–236.

Bishop, C. M. (1992). Exact calculation of the Hessian matrix for the multilayer perceptron. *Neural Computation* **4**(4), 494–501.

Bishop, C. M. (1993). Curvature-driven smoothing: a learning algorithm for feedforward networks. *IEEE Transactions on Neural Networks* **4**(5), 882–884.

Bishop, C. M. (1994). Novelty detection and neural network validation. *IEE Proceedings: Vision, Image and Signal Processing* **141**(4), 217–222. Special issue on applications of neural networks.

Bishop, C. M. (1995a). *Neural Networks for Pattern Recognition*. Oxford University Press.

Bishop, C. M. (1995b). Training with noise is equivalent to Tikhonov regularization. *Neural Computation* **7**(1), 108–116.

Bishop, C. M. (1999a). Bayesian PCA. In M. S. Kearns, S. A. Solla, and D. A. Cohn (Eds.), *Advances in Neural Information Processing Systems*, Volume 11, pp. 382–388. MIT Press.

Bishop, C. M. (1999b). Variational principal components. In *Proceedings Ninth International Conference on Artificial Neural Networks, ICANN'99*, Volume 1, pp. 509–514. IEE.

Bishop, C. M. and G. D. James (1993). Analysis of multiphase flows using dual-energy gamma densitometry and neural networks. *Nuclear Instruments and Methods in Physics Research* **A327**, 580–593.

Bishop, C. M. and I. T. Nabney (1996). Modelling conditional probability distributions for periodic variables. *Neural Computation* **8**(5), 1123–1133.

Bishop, C. M. and I. T. Nabney (2008). *Pattern Recognition and Machine Learning: A Matlab Companion*. Springer. In preparation.

Bishop, C. M., D. Spiegelhalter, and J. Winn (2003). VIBES: A variational inference engine for Bayesian networks. In S. Becker, S. Thrun, and K. Obermeyer (Eds.), *Advances in Neural Information Processing Systems*, Volume 15, pp. 793–800. MIT Press.

Bishop, C. M. and M. Svensén (2003). Bayesian hierarchical mixtures of experts. In U. Kjaerulff and C. Meek (Eds.), *Proceedings Nineteenth Conference on Uncertainty in Artificial Intelligence*, pp. 57–64. Morgan Kaufmann.

Bishop, C. M., M. Svensén, and G. E. Hinton (2004). Distinguishing text from graphics in online handwritten ink. In F. Kimura and H. Fujisawa (Eds.), *Proceedings Ninth International Workshop on Frontiers in Handwriting Recognition, IWFHR-9*, Tokyo, Japan, pp. 142–147.

Bishop, C. M., M. Svensén, and C. K. I. Williams (1996). EM optimization of latent variable density models. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo (Eds.), *Advances in Neural Information Processing Systems*, Volume 8, pp. 465–471. MIT Press.

Bishop, C. M., M. Svensén, and C. K. I. Williams (1997a). GTM: a principled alternative to the Self-Organizing Map. In M. C. Mozer, M. I. Jordan, and T. Petche (Eds.), *Advances in Neural Information Processing Systems*, Volume 9, pp. 354–360. MIT Press.

Bishop, C. M., M. Svensén, and C. K. I. Williams (1997b). Magnification factors for the GTM algorithm. In *Proceedings IEE Fifth International Conference on Artificial Neural Networks, Cambridge, U.K.*, pp. 64–69. Institute of Electrical Engineers.

Bishop, C. M., M. Svensén, and C. K. I. Williams (1998a). Developments of the Generative Topographic Mapping. *Neurocomputing* **21**, 203–224.

Bishop, C. M., M. Svensén, and C. K. I. Williams (1998b). GTM: the Generative Topographic Mapping. *Neural Computation* **10**(1), 215–234.

Bishop, C. M. and M. E. Tipping (1998). A hierarchical latent variable model for data visualization. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **20**(3), 281–293.

Bishop, C. M. and J. Winn (2000). Non-linear Bayesian image modelling. In *Proceedings Sixth European Conference on Computer Vision, Dublin*, Volume 1, pp. 3–17. Springer.

Blei, D. M., M. I. Jordan, and A. Y. Ng (2003). Hierarchical Bayesian models for applications in information retrieval. In J. M. B. *et al.* (Ed.), *Bayesian Statistics, 7*, pp. 25–43. Oxford University Press.

Block, H. D. (1962). The perceptron: a model for brain functioning. *Reviews of Modern Physics* **34**(1), 123–135. Reprinted in Anderson and Rosenfeld (1988).

Blum, J. A. (1965). Multidimensional stochastic approximation methods. *Annals of Mathematical Statistics* **25**, 737–744.

Bodlaender, H. (1993). A tourist guide through treewidth. *Acta Cybernetica* **11**, 1–21.

Boser, B. E., I. M. Guyon, and V. N. Vapnik (1992). A training algorithm for optimal margin classifiers. In D. Haussler (Ed.), *Proceedings Fifth Annual Workshop on Computational Learning Theory (COLT)*, pp. 144–152. ACM.

Bourlard, H. and Y. Kamp (1988). Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics* **59**, 291–294.

Box, G. E. P., G. M. Jenkins, and G. C. Reinsel (1994). *Time Series Analysis*. Prentice Hall.

Box, G. E. P. and G. C. Tao (1973). *Bayesian Inference in Statistical Analysis*. Wiley.

Boyd, S. and L. Vandenberghe (2004). *Convex Optimization*. Cambridge University Press.

Boyen, X. and D. Koller (1998). Tractable inference for complex stochastic processes. In G. F. Cooper and S. Moral (Eds.), *Proceedings 14th Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 33–42. Morgan Kaufmann.

Boykov, Y., O. Veksler, and R. Zabih (2001). Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **23**(11), 1222–1239.

Breiman, L. (1996). Bagging predictors. *Machine Learning* **26**, 123–140.

Breiman, L., J. H. Friedman, R. A. Olshen, and P. J. Stone (1984). *Classification and Regression Trees*. Wadsworth.

Brooks, S. P. (1998). Markov chain Monte Carlo method and its application. *The Statistician* **47**(1), 69–100.

Broomhead, D. S. and D. Lowe (1988). Multivariable functional interpolation and adaptive networks. *Complex Systems* **2**, 321–355.

Buntine, W. and A. Weigend (1991). Bayesian backpropagation. *Complex Systems* **5**, 603–643.

Buntine, W. L. and A. S. Weigend (1993). Computing second derivatives in feed-forward networks: a review. *IEEE Transactions on Neural Networks* **5**(3), 480–488.

Burges, C. J. C. (1998). A tutorial on support vector machines for pattern recognition. *Knowledge Discovery and Data Mining* **2**(2), 121–167.

Cardoso, J.-F. (1998). Blind signal separation: statistical principles. *Proceedings of the IEEE* **9**(10), 2009–2025.

Casella, G. and R. L. Berger (2002). *Statistical Inference* (Second ed.). Duxbury.

Castillo, E., J. M. Gutiérrez, and A. S. Hadi (1997). *Expert Systems and Probabilistic Network Models*. Springer.

Chan, K., T. Lee, and T. J. Sejnowski (2003). Variational Bayesian learning of ICA with missing data. *Neural Computation* **15**(8), 1991–2011.

Chen, A. M., H. Lu, and R. Hecht-Nielsen (1993). On the geometry of feedforward neural network error surfaces. *Neural Computation* **5**(6), 910–927.

Chen, M. H., Q. M. Shao, and J. G. Ibrahim (Eds.) (2001). *Monte Carlo Methods for Bayesian Computation*. Springer.

Chen, S., C. F. N. Cowan, and P. M. Grant (1991). Orthogonal least squares learning algorithm for radial basis function networks. *IEEE Transactions on Neural Networks* **2**(2), 302–309.

Choudrey, R. A. and S. J. Roberts (2003). Variational mixture of Bayesian independent component analyzers. *Neural Computation* **15**(1), 213–252.

Clifford, P. (1990). Markov random fields in statistics. In G. R. Grimmett and D. J. A. Welsh (Eds.), *Disorder in Physical Systems. A Volume in Honour of John M. Hammersley*, pp. 19–32. Oxford University Press.

Collins, M., S. Dasgupta, and R. E. Schapire (2002). A generalization of principal component analysis to the exponential family. In T. G. Dietterich, S. Becker, and Z. Ghahramani (Eds.), *Advances in Neural Information Processing Systems*, Volume 14, pp. 617–624. MIT Press.

Comon, P., C. Jutten, and J. Herault (1991). Blind source separation, 2: problems statement. *Signal Processing* **24**(1), 11–20.

Corduneanu, A. and C. M. Bishop (2001). Variational Bayesian model selection for mixture distributions. In T. Richardson and T. Jaakkola (Eds.), *Proceedings Eighth International Conference on Artificial Intelligence and Statistics*, pp. 27–34. Morgan Kaufmann.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2001). *Introduction to Algorithms* (Second ed.). MIT Press.

Cortes, C. and V. N. Vapnik (1995). Support vector networks. *Machine Learning* **20**, 273–297.

Cotter, N. E. (1990). The Stone-Weierstrass theorem and its application to neural networks. *IEEE Transactions on Neural Networks* **1**(4), 290–295.

Cover, T. and P. Hart (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory* **IT-11**, 21–27.

Cover, T. M. and J. A. Thomas (1991). *Elements of Information Theory*. Wiley.

Cowell, R. G., A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter (1999). *Probabilistic Networks and Expert Systems*. Springer.

Cox, R. T. (1946). Probability, frequency and reasonable expectation. *American Journal of Physics* **14**(1), 1–13.

Cox, T. F. and M. A. A. Cox (2000). *Multidimensional Scaling* (Second ed.). Chapman and Hall.

Cressie, N. (1993). *Statistics for Spatial Data*. Wiley.

Cristianini, N. and J. Shawe-Taylor (2000). *Support vector machines and other kernel-based learning methods*. Cambridge University Press.

Csató, L. and M. Opper (2002). Sparse on-line Gaussian processes. *Neural Computation* **14**(3), 641–668.

Csiszàr, I. and G. Tusnàdy (1984). Information geometry and alternating minimization procedures. *Statistics and Decisions* **1**(1), 205–237.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems* **2**, 304–314.

Dawid, A. P. (1979). Conditional independence in statistical theory (with discussion). *Journal of the Royal Statistical Society, Series B* **4**, 1–31.

Dawid, A. P. (1980). Conditional independence for statistical operations. *Annals of Statistics* **8**, 598–617.

deFinetti, B. (1970). *Theory of Probability*. Wiley and Sons.

Dempster, A. P., N. M. Laird, and D. B. Rubin (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, B* **39**(1), 1–38.

Denison, D. G. T., C. C. Holmes, B. K. Mallick, and A. F. M. Smith (2002). *Bayesian Methods for Nonlinear Classification and Regression*. Wiley.

Diaconis, P. and L. Saloff-Coste (1998). What do we know about the Metropolis algorithm? *Journal of Computer and System Sciences* **57**, 20–36.

Dietterich, T. G. and G. Bakiri (1995). Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research* **2**, 263–286.

Duane, S., A. D. Kennedy, B. J. Pendleton, and D. Roweth (1987). Hybrid Monte Carlo. *Physics Letters B* **195**(2), 216–222.

Duda, R. O. and P. E. Hart (1973). *Pattern Classification and Scene Analysis*. Wiley.

Duda, R. O., P. E. Hart, and D. G. Stork (2001). *Pattern Classification* (Second ed.). Wiley.

Durbin, R., S. Eddy, A. Krogh, and G. Mitchison (1998). *Biological Sequence Analysis*. Cambridge University Press.

Dybowski, R. and S. Roberts (2005). An anthology of probabilistic models for medical informatics. In D. Husmeier, R. Dybowski, and S. Roberts (Eds.), *Probabilistic Modeling in Bioinformatics and Medical Informatics*, pp. 297–349. Springer.

Efron, B. (1979). Bootstrap methods: another look at the jackknife. *Annals of Statistics* **7**, 1–26.

Elkan, C. (2003). Using the triangle inequality to accelerate k-means. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 147–153. AAAI.

Elliott, R. J., L. Aggoun, and J. B. Moore (1995). *Hidden Markov Models: Estimation and Control*. Springer.

Ephraim, Y., D. Malah, and B. H. Juang (1989). On the application of hidden Markov models for enhancing noisy speech. *IEEE Transactions on Acoustics, Speech and Signal Processing* **37**(12), 1846–1856.

Erwin, E., K. Obermayer, and K. Schulten (1992). Self-organizing maps: ordering, convergence properties and energy functions. *Biological Cybernetics* **67**, 47–55.

Everitt, B. S. (1984). *An Introduction to Latent Variable Models*. Chapman and Hall.

Faul, A. C. and M. E. Tipping (2002). Analysis of sparse Bayesian learning. In T. G. Dietterich, S. Becker, and Z. Ghahramani (Eds.), *Advances in Neural Information Processing Systems*, Volume 14, pp. 383–389. MIT Press.

Feller, W. (1966). *An Introduction to Probability Theory and its Applications* (Second ed.), Volume 2. Wiley.

Feynman, R. P., R. B. Leighton, and M. Sands (1964). *The Feynman Lectures of Physics*, Volume Two. Addison-Wesley. Chapter 19.

Fletcher, R. (1987). *Practical Methods of Optimization* (Second ed.). Wiley.

Forsyth, D. A. and J. Ponce (2003). *Computer Vision: A Modern Approach*. Prentice Hall.

Freund, Y. and R. E. Schapire (1996). Experiments with a new boosting algorithm. In L. Saitta (Ed.), *Thirteenth International Conference on Machine Learning*, pp. 148–156. Morgan Kaufmann.

Frey, B. J. (1998). *Graphical Models for Machine Learning and Digital Communication*. MIT Press.

Frey, B. J. and D. J. C. MacKay (1998). A revolution: Belief propagation in graphs with cycles. In M. I. Jordan, M. J. Kearns, and S. A. Solla (Eds.), *Advances in Neural Information Processing Systems*, Volume 10. MIT Press.

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of Statistics* **29**(5), 1189–1232.

Friedman, J. H., T. Hastie, and R. Tibshirani (2000). Additive logistic regression: a statistical view of boosting. *Annals of Statistics* **28**, 337–407.

Friedman, N. and D. Koller (2003). Being Bayesian about network structure: A Bayesian approach to structure discovery in Bayesian networks. *Machine Learning* **50**, 95–126.

Frydenberg, M. (1990). The chain graph Markov property. *Scandinavian Journal of Statistics* **17**, 333–353.

Fukunaga, K. (1990). *Introduction to Statistical Pattern Recognition* (Second ed.). Academic Press.

Funahashi, K. (1989). On the approximate realization of continuous mappings by neural networks. *Neural Networks* **2**(3), 183–192.

Fung, R. and K. C. Chang (1990). Weighting and integrating evidence for stochastic simulation in Bayesian networks. In P. P. Bonissone, M. Henrion, L. N. Kanal, and J. F. Lemmer (Eds.), *Uncertainty in Artificial Intelligence*, Volume 5, pp. 208–219. Elsevier.

Gallager, R. G. (1963). *Low-Density Parity-Check Codes*. MIT Press.

Gamerman, D. (1997). *Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference*. Chapman and Hall.

Gelman, A., J. B. Carlin, H. S. Stern, and D. B. Rubin (2004). *Bayesian Data Analysis* (Second ed.). Chapman and Hall.

Geman, S. and D. Geman (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **6**(1), 721–741.

Ghahramani, Z. and M. J. Beal (2000). Variational inference for Bayesian mixtures of factor analyzers. In S. A. Solla, T. K. Leen, and K. R. Müller (Eds.), *Advances in Neural Information Processing Systems*, Volume 12, pp. 449–455. MIT Press.

Ghahramani, Z. and G. E. Hinton (1996a). The EM algorithm for mixtures of factor analyzers. Technical Report CRG-TR-96-1, University of Toronto.

Ghahramani, Z. and G. E. Hinton (1996b). Parameter estimation for linear dynamical systems. Technical Report CRG-TR-96-2, University of Toronto.

Ghahramani, Z. and G. E. Hinton (1998). Variational learning for switching state-space models. *Neural Computation* **12**(4), 963–996.

Ghahramani, Z. and M. I. Jordan (1994). Supervised learning from incomplete data via an EM appproach. In J. D. Cowan, G. T. Tesauro, and J. Alspector (Eds.), *Advances in Neural Information Processing Systems*, Volume 6, pp. 120–127. Morgan Kaufmann.

Ghahramani, Z. and M. I. Jordan (1997). Factorial hidden Markov models. *Machine Learning* **29**, 245–275.

Gibbs, M. N. (1997). *Bayesian Gaussian processes for regression and classification*. Phd thesis, University of Cambridge.

Gibbs, M. N. and D. J. C. MacKay (2000). Variational Gaussian process classifiers. *IEEE Transactions on Neural Networks* **11**, 1458–1464.

Gilks, W. R. (1992). Derivative-free adaptive rejection sampling for Gibbs sampling. In J. Bernardo, J. Berger, A. P. Dawid, and A. F. M. Smith (Eds.), *Bayesian Statistics*, Volume 4. Oxford University Press.

Gilks, W. R., N. G. Best, and K. K. C. Tan (1995). Adaptive rejection Metropolis sampling. *Applied Statistics* **44**, 455–472.

Gilks, W. R., S. Richardson, and D. J. Spiegelhalter (Eds.) (1996). *Markov Chain Monte Carlo in Practice*. Chapman and Hall.

Gilks, W. R. and P. Wild (1992). Adaptive rejection sampling for Gibbs sampling. *Applied Statistics* **41**, 337–348.

Gill, P. E., W. Murray, and M. H. Wright (1981). *Practical Optimization*. Academic Press.

Goldberg, P. W., C. K. I. Williams, and C. M. Bishop (1998). Regression with input-dependent noise: A Gaussian process treatment. In *Advances in Neural Information Processing Systems*, Volume 10, pp. 493–499. MIT Press.

Golub, G. H. and C. F. Van Loan (1996). *Matrix Computations* (Third ed.). John Hopkins University Press.

Good, I. (1950). *Probability and the Weighing of Evidence*. Hafners.

Gordon, N. J., D. J. Salmond, and A. F. M. Smith (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Proceedings-F* **140**(2), 107–113.

Graepel, T. (2003). Solving noisy linear operator equations by Gaussian processes: Application to ordinary and partial differential equations. In *Proceedings of the Twentieth International Conference on Machine Learning*, pp. 234–241.

Greig, D., B. Porteous, and A. Seheult (1989). Exact maximum a-posteriori estimation for binary images. *Journal of the Royal Statistical Society, Series B* **51**(2), 271–279.

Gull, S. F. (1989). Developments in maximum entropy data analysis. In J. Skilling (Ed.), *Maximum Entropy and Bayesian Methods*, pp. 53–71. Kluwer.

Hassibi, B. and D. G. Stork (1993). Second order derivatives for network pruning: optimal brain surgeon. In S. J. Hanson, J. D. Cowan, and C. L. Giles (Eds.), *Advances in Neural Information Processing Systems*, Volume 5, pp. 164–171. Morgan Kaufmann.

Hastie, T. and W. Stuetzle (1989). Principal curves. *Journal of the American Statistical Association* **84**(106), 502–516.

Hastie, T., R. Tibshirani, and J. Friedman (2001). *The Elements of Statistical Learning*. Springer.

Hastings, W. K. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* **57**, 97–109.

Hathaway, R. J. (1986). Another interpretation of the EM algorithm for mixture distributions. *Statistics and Probability Letters* **4**, 53–56.

Haussler, D. (1999). Convolution kernels on discrete structures. Technical Report UCSC-CRL-99-10, University of California, Santa Cruz, Computer Science Department.

Henrion, M. (1988). Propagation of uncertainty by logic sampling in Bayes' networks. In J. F. Lemmer and L. N. Kanal (Eds.), *Uncertainty in Artificial Intelligence*, Volume 2, pp. 149–164. North Holland.

Herbrich, R. (2002). *Learning Kernel Classifiers*. MIT Press.

Hertz, J., A. Krogh, and R. G. Palmer (1991). *Introduction to the Theory of Neural Computation*. Addison Wesley.

Hinton, G. E., P. Dayan, and M. Revow (1997). Modelling the manifolds of images of handwritten digits. *IEEE Transactions on Neural Networks* **8**(1), 65–74.

Hinton, G. E. and D. van Camp (1993). Keeping neural networks simple by minimizing the description length of the weights. In *Proceedings of the Sixth Annual Conference on Computational Learning Theory*, pp. 5–13. ACM.

Hinton, G. E., M. Welling, Y. W. Teh, and S. Osindero (2001). A new view of ICA. In *Proceedings of the International Conference on Independent Component Analysis and Blind Signal Separation*, Volume 3.

Hodgson, M. E. (1998). Reducing computational requirements of the minimum-distance classifier. *Remote Sensing of Environments* **25**, 117–128.

Hoerl, A. E. and R. Kennard (1970). Ridge regression: biased estimation for nonorthogonal problems. *Technometrics* **12**, 55–67.

Hofmann, T. (2000). Learning the similarity of documents: an information-geometric approach to document retrieval and classification. In S. A. Solla, T. K. Leen, and K. R. Müller (Eds.), *Advances in Neural Information Processing Systems*, Volume 12, pp. 914–920. MIT Press.

Hojen-Sorensen, P. A., O. Winther, and L. K. Hansen (2002). Mean field approaches to independent component analysis. *Neural Computation* **14**(4), 889–918.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks* **4**(2), 251–257.

Hornik, K., M. Stinchcombe, and H. White (1989). Multilayer feedforward networks are universal approximators. *Neural Networks* **2**(5), 359–366.

Hotelling, H. (1933). Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology* **24**, 417–441.

Hotelling, H. (1936). Relations between two sets of variables. *Biometrika* **28**, 321–377.

Hyvärinen, A. and E. Oja (1997). A fast fixed-point algorithm for independent component analysis. *Neural Computation* **9**(7), 1483–1492.

Isard, M. and A. Blake (1998). CONDENSATION – conditional density propagation for visual tracking. *International Journal of Computer Vision* **29**(1), 5–18.

Ito, Y. (1991). Representation of functions by superpositions of a step or sigmoid function and their applications to neural network theory. *Neural Networks* **4**(3), 385–394.

Jaakkola, T. and M. I. Jordan (2000). Bayesian parameter estimation via variational methods. *Statistics and Computing* **10**, 25–37.

Jaakkola, T. S. (2001). Tutorial on variational approximation methods. In M. Opper and D. Saad (Eds.), *Advances in Mean Field Methods*, pp. 129–159. MIT Press.

Jaakkola, T. S. and D. Haussler (1999). Exploiting generative models in discriminative classifiers. In M. S. Kearns, S. A. Solla, and D. A. Cohn (Eds.), *Advances in Neural Information Processing Systems*, Volume 11. MIT Press.

Jacobs, R. A., M. I. Jordan, S. J. Nowlan, and G. E. Hinton (1991). Adaptive mixtures of local experts. *Neural Computation* **3**(1), 79–87.

Jaynes, E. T. (2003). *Probability Theory: The Logic of Science*. Cambridge University Press.

Jebara, T. (2004). *Machine Learning: Discriminative and Generative*. Kluwer.

Jeffries, H. (1946). An invariant form for the prior probability in estimation problems. *Pro. Roy. Soc. AA* **186**, 453–461.

Jelinek, F. (1997). *Statistical Methods for Speech Recognition*. MIT Press.

Jensen, C., A. Kong, and U. Kjaerulff (1995). Blocking gibbs sampling in very large probabilistic expert systems. *International Journal of Human Computer Studies. Special Issue on Real-World Applications of Uncertain Reasoning.* **42**, 647–666.

Jensen, F. V. (1996). *An Introduction to Bayesian Networks*. UCL Press.

Jerrum, M. and A. Sinclair (1996). The Markov chain Monte Carlo method: an approach to approximate counting and integration. In D. S. Hochbaum (Ed.), *Approximation Algorithms for NP-Hard Problems*. PWS Publishing.

Jolliffe, I. T. (2002). *Principal Component Analysis* (Second ed.). Springer.

Jordan, M. I. (1999). *Learning in Graphical Models*. MIT Press.

Jordan, M. I. (2007). *An Introduction to Probabilistic Graphical Models*. In preparation.

Jordan, M. I., Z. Ghahramani, T. S. Jaakkola, and L. K. Saul (1999). An introduction to variational methods for graphical models. In M. I. Jordan (Ed.), *Learning in Graphical Models*, pp. 105–162. MIT Press.

Jordan, M. I. and R. A. Jacobs (1994). Hierarchical mixtures of experts and the EM algorithm. *Neural Computation* **6**(2), 181–214.

Jutten, C. and J. Herault (1991). Blind separation of sources, 1: An adaptive algorithm based on neuromimetic architecture. *Signal Processing* **24**(1), 1–10.

Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *Transactions of the American Society for Mechanical Engineering, Series D, Journal of Basic Engineering* **82**, 35–45.

Kambhatla, N. and T. K. Leen (1997). Dimension reduction by local principal component analysis. *Neural Computation* **9**(7), 1493–1516.

Kanazawa, K., D. Koller, and S. Russel (1995). Stochastic simulation algorithms for dynamic probabilistic networks. In *Uncertainty in Artificial Intelligence*, Volume 11. Morgan Kaufmann.

Kapadia, S. (1998). *Discriminative Training of Hidden Markov Models*. Phd thesis, University of Cambridge, U.K.

Kapur, J. (1989). *Maximum entropy methods in science and engineering*. Wiley.

Karush, W. (1939). Minima of functions of several variables with inequalities as side constraints. Master's thesis, Department of Mathematics, University of Chicago.

Kass, R. E. and A. E. Raftery (1995). Bayes factors. *Journal of the American Statistical Association* **90**, 377–395.

Kearns, M. J. and U. V. Vazirani (1994). *An Introduction to Computational Learning Theory*. MIT Press.

Kindermann, R. and J. L. Snell (1980). *Markov Random Fields and Their Applications*. American Mathematical Society.

Kittler, J. and J. Föglein (1984). Contextual classification of multispectral pixel data. *Image and Vision Computing* **2**, 13–29.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics* **43**, 59–69.

Kohonen, T. (1995). *Self-Organizing Maps*. Springer.

Kolmogorov, V. and R. Zabih (2004). What energy functions can be minimized via graph cuts? *IEEE Transactions on Pattern Analysis and Machine Intelligence* **26**(2), 147–159.

Kreinovich, V. Y. (1991). Arbitrary nonlinearity is sufficient to represent all functions by neural networks: a theorem. *Neural Networks* **4**(3), 381–383.

Krogh, A., M. Brown, I. S. Mian, K. Sjölander, and D. Haussler (1994). Hidden Markov models in computational biology: Applications to protein modelling. *Journal of Molecular Biology* **235**, 1501–1531.

Kschischnang, F. R., B. J. Frey, and H. A. Loeliger (2001). Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory* **47**(2), 498–519.

Kuhn, H. W. and A. W. Tucker (1951). Nonlinear programming. In *Proceedings of the 2nd Berkeley Symposium on Mathematical Statistics and Probabilities*, pp. 481–492. University of California Press.

Kullback, S. and R. A. Leibler (1951). On information and sufficiency. *Annals of Mathematical Statistics* **22**(1), 79–86.

Kůrková, V. and P. C. Kainen (1994). Functionally equivalent feed-forward neural networks. *Neural Computation* **6**(3), 543–558.

Kuss, M. and C. Rasmussen (2006). Assessing approximations for Gaussian process classification. In *Advances in Neural Information Processing Systems*, Number 18. MIT Press. in press.

Lasserre, J., C. M. Bishop, and T. Minka (2006). Principled hybrids of generative and discriminative models. In *Proceedings 2006 IEEE Conference on Computer Vision and Pattern Recognition, New York*.

Lauritzen, S. and N. Wermuth (1989). Graphical models for association between variables, some of which are qualitative some quantitative. *Annals of Statistics* **17**, 31–57.

Lauritzen, S. L. (1992). Propagation of probabilities, means and variances in mixed graphical association models. *Journal of the American Statistical Association* **87**, 1098–1108.

Lauritzen, S. L. (1996). *Graphical Models*. Oxford University Press.

Lauritzen, S. L. and D. J. Spiegelhalter (1988). Local computations with probabailities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society* **50**, 157–224.

Lawley, D. N. (1953). A modified method of estimation in factor analysis and some large sample results. In *Uppsala Symposium on Psychological Factor Analysis*, Number 3 in Nordisk Psykologi Monograph Series, pp. 35–42. Uppsala: Almqvist and Wiksell.

Lawrence, N. D., A. I. T. Rowstron, C. M. Bishop, and M. J. Taylor (2002). Optimising synchronisation times for mobile devices. In T. G. Dietterich, S. Becker, and Z. Ghahramani (Eds.), *Advances in Neural Information Processing Systems*, Volume 14, pp. 1401–1408. MIT Press.

Lazarsfeld, P. F. and N. W. Henry (1968). *Latent Structure Analysis*. Houghton Mifflin.

Le Cun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation* **1**(4), 541–551.

Le Cun, Y., J. S. Denker, and S. A. Solla (1990). Optimal brain damage. In D. S. Touretzky (Ed.),

Advances in Neural Information Processing Systems, Volume 2, pp. 598–605. Morgan Kaufmann.

LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE* **86**, 2278–2324.

Lee, Y., Y. Lin, and G. Wahba (2001). Multicategory support vector machines. Technical Report 1040, Department of Statistics, University of Madison, Wisconsin.

Leen, T. K. (1995). From data distributions to regularization in invariant learning. *Neural Computation* **7**, 974–981.

Lindley, D. V. (1982). Scoring rules and the inevitability of probability. *International Statistical Review* **50**, 1–26.

Liu, J. S. (Ed.) (2001). *Monte Carlo Strategies in Scientific Computing*. Springer.

Lloyd, S. P. (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory* **28**(2), 129–137.

Lütkepohl, H. (1996). *Handbook of Matrices*. Wiley.

MacKay, D. J. C. (1992a). Bayesian interpolation. *Neural Computation* **4**(3), 415–447.

MacKay, D. J. C. (1992b). The evidence framework applied to classification networks. *Neural Computation* **4**(5), 720–736.

MacKay, D. J. C. (1992c). A practical Bayesian framework for back-propagation networks. *Neural Computation* **4**(3), 448–472.

MacKay, D. J. C. (1994). Bayesian methods for backprop networks. In E. Domany, J. L. van Hemmen, and K. Schulten (Eds.), *Models of Neural Networks, III*, Chapter 6, pp. 211–254. Springer.

MacKay, D. J. C. (1995). Bayesian neural networks and density networks. *Nuclear Instruments and Methods in Physics Research, A* **354**(1), 73–80.

MacKay, D. J. C. (1997). Ensemble learning for hidden Markov models. Unpublished manuscript, Department of Physics, University of Cambridge.

MacKay, D. J. C. (1998). Introduction to Gaussian processes. In C. M. Bishop (Ed.), *Neural Networks and Machine Learning*, pp. 133–166. Springer.

MacKay, D. J. C. (1999). Comparison of approximate methods for handling hyperparameters. *Neural Computation* **11**(5), 1035–1068.

MacKay, D. J. C. (2003). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press.

MacKay, D. J. C. and M. N. Gibbs (1999). Density networks. In J. W. Kay and D. M. Titterington (Eds.), *Statistics and Neural Networks: Advances at the Interface*, Chapter 5, pp. 129–145. Oxford University Press.

MacKay, D. J. C. and R. M. Neal (1999). Good error-correcting codes based on very sparse matrices. *IEEE Transactions on Information Theory* **45**, 399–431.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In L. M. LeCam and J. Neyman (Eds.), *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, Volume I, pp. 281–297. University of California Press.

Magnus, J. R. and H. Neudecker (1999). *Matrix Differential Calculus with Applications in Statistics and Econometrics*. Wiley.

Mallat, S. (1999). *A Wavelet Tour of Signal Processing* (Second ed.). Academic Press.

Manning, C. D. and H. Schütze (1999). *Foundations of Statistical Natural Language Processing*. MIT Press.

Mardia, K. V. and P. E. Jupp (2000). *Directional Statistics*. Wiley.

Maybeck, P. S. (1982). *Stochastic models, estimation and control*. Academic Press.

McAllester, D. A. (2003). PAC-Bayesian stochastic model selection. *Machine Learning* **51**(1), 5–21.

McCullagh, P. and J. A. Nelder (1989). *Generalized Linear Models* (Second ed.). Chapman and Hall.

McCulloch, W. S. and W. Pitts (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics* **5**, 115–133. Reprinted in Anderson and Rosenfeld (1988).

McEliece, R. J., D. J. C. MacKay, and J. F. Cheng (1998). Turbo decoding as an instance of Pearl's 'Belief Ppropagation' algorithm. *IEEE Journal on Selected Areas in Communications* **16**, 140–152.

McLachlan, G. J. and K. E. Basford (1988). *Mixture Models: Inference and Applications to Clustering*. Marcel Dekker.

McLachlan, G. J. and T. Krishnan (1997). *The EM Algorithm and its Extensions*. Wiley.

McLachlan, G. J. and D. Peel (2000). *Finite Mixture Models*. Wiley.

Meng, X. L. and D. B. Rubin (1993). Maximum likelihood estimation via the ECM algorithm: a general framework. *Biometrika* **80**, 267–278.

Metropolis, N., A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller (1953). Equation of state calculations by fast computing machines. *Journal of Chemical Physics* **21**(6), 1087–1092.

Metropolis, N. and S. Ulam (1949). The Monte Carlo method. *Journal of the American Statistical Association* **44**(247), 335–341.

Mika, S., G. Rätsch, J. Weston, and B. Schölkopf (1999). Fisher discriminant analysis with kernels. In Y. H. Hu, J. Larsen, E. Wilson, and S. Douglas (Eds.), *Neural Networks for Signal Processing IX*, pp. 41–48. IEEE.

Minka, T. (2001a). Expectation propagation for approximate Bayesian inference. In J. Breese and D. Koller (Eds.), *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, pp. 362–369. Morgan Kaufmann.

Minka, T. (2001b). *A family of approximate algorithms for Bayesian inference*. Ph. D. thesis, MIT.

Minka, T. (2004). Power EP. Technical Report MSR-TR-2004-149, Microsoft Research Cambridge.

Minka, T. (2005). Divergence measures and message passing. Technical Report MSR-TR-2005-173, Microsoft Research Cambridge.

Minka, T. P. (2001c). Automatic choice of dimensionality for PCA. In T. K. Leen, T. G. Dietterich, and V. Tresp (Eds.), *Advances in Neural Information Processing Systems*, Volume 13, pp. 598–604. MIT Press.

Minsky, M. L. and S. A. Papert (1969). *Perceptrons*. MIT Press. Expanded edition 1990.

Miskin, J. W. and D. J. C. MacKay (2001). Ensemble learning for blind source separation. In S. J. Roberts and R. M. Everson (Eds.), *Independent Component Analysis: Principles and Practice*. Cambridge University Press.

Møller, M. (1993). Efficient Training of Feed-Forward Neural Networks. Ph. D. thesis, Aarhus University, Denmark.

Moody, J. and C. J. Darken (1989). Fast learning in networks of locally-tuned processing units. *Neural Computation* **1**(2), 281–294.

Moore, A. W. (2000). The anchors hierarch: using the triangle inequality to survive high dimensional data. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pp. 397–405.

Müller, K. R., S. Mika, G. Rätsch, K. Tsuda, and B. Schölkopf (2001). An introduction to kernel-based learning algorithms. *IEEE Transactions on Neural Networks* **12**(2), 181–202.

Müller, P. and F. A. Quintana (2004). Nonparametric Bayesian data analysis. *Statistical Science* **19**(1), 95–110.

Nabney, I. T. (2002). *Netlab: Algorithms for Pattern Recognition*. Springer.

Nadaraya, É. A. (1964). On estimating regression. *Theory of Probability and its Applications* **9**(1), 141–142.

Nag, R., K. Wong, and F. Fallside (1986). Script recognition using hidden markov models. In *ICASSP86*, pp. 2071–2074. IEEE.

Neal, R. M. (1993). Probabilistic inference using Markov chain Monte Carlo methods. Technical Report CRG-TR-93-1, Department of Computer Science, University of Toronto, Canada.

Neal, R. M. (1996). *Bayesian Learning for Neural Networks*. Springer. Lecture Notes in Statistics 118.

Neal, R. M. (1997). Monte Carlo implementation of Gaussian process models for Bayesian regression and classification. Technical Report 9702, Department of Computer Statistics, University of Toronto.

Neal, R. M. (1999). Suppressing random walks in Markov chain Monte Carlo using ordered overrelaxation. In M. I. Jordan (Ed.), *Learning in Graphical Models*, pp. 205–228. MIT Press.

Neal, R. M. (2000). Markov chain sampling for Dirichlet process mixture models. *Journal of Computational and Graphical Statistics* **9**, 249–265.

Neal, R. M. (2003). Slice sampling. *Annals of Statistics* **31**, 705–767.

Neal, R. M. and G. E. Hinton (1999). A new view of the EM algorithm that justifies incremental and other variants. In M. I. Jordan (Ed.), *Learning in Graphical Models*, pp. 355–368. MIT Press.

Nelder, J. A. and R. W. M. Wedderburn (1972). Generalized linear models. *Journal of the Royal Statistical Society, A* **135**, 370–384.

Nilsson, N. J. (1965). *Learning Machines*. McGraw-Hill. Reprinted as *The Mathematical Foundations of Learning Machines*, Morgan Kaufmann, (1990).

Nocedal, J. and S. J. Wright (1999). *Numerical Optimization*. Springer.

Nowlan, S. J. and G. E. Hinton (1992). Simplifying neural networks by soft weight sharing. *Neural Computation* **4**(4), 473–493.

Ogden, R. T. (1997). *Essential Wavelets for Statistical Applications and Data Analysis*. Birkhäuser.

Opper, M. and O. Winther (1999). A Bayesian approach to on-line learning. In D. Saad (Ed.), *On-Line Learning in Neural Networks*, pp. 363–378. Cambridge University Press.

Opper, M. and O. Winther (2000a). Gaussian processes and SVM: mean field theory and leave-one-out. In A. J. Smola, P. L. Bartlett, B. Schölkopf, and D. Shuurmans (Eds.), *Advances in Large Margin Classifiers*, pp. 311–326. MIT Press.

Opper, M. and O. Winther (2000b). Gaussian processes for classification. *Neural Computation* **12**(11), 2655–2684.

Osuna, E., R. Freund, and F. Girosi (1996). Support vector machines: training and applications. A.I. Memo AIM-1602, MIT.

Papoulis, A. (1984). *Probability, Random Variables, and Stochastic Processes* (Second ed.). McGraw-Hill.

Parisi, G. (1988). *Statistical Field Theory*. Addison-Wesley.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann.

Pearlmutter, B. A. (1994). Fast exact multiplication by the Hessian. *Neural Computation* **6**(1), 147–160.

Pearlmutter, B. A. and L. C. Parra (1997). Maximum likelihood source separation: a context-sensitive generalization of ICA. In M. C. Mozer, M. I. Jordan, and T. Petsche (Eds.), *Advances in Neural Information Processing Systems*, Volume 9, pp. 613–619. MIT Press.

Pearson, K. (1901). On lines and planes of closest fit to systems of points in space. *The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science, Sixth Series* **2**, 559–572.

Platt, J. C. (1999). Fast training of support vector machines using sequential minimal optimization. In B. Schölkopf, C. J. C. Burges, and A. J. Smola (Eds.), *Advances in Kernel Methods – Support Vector Learning*, pp. 185–208. MIT Press.

Platt, J. C. (2000). Probabilities for SV machines. In A. J. Smola, P. L. Bartlett, B. Schölkopf, and D. Shuurmans (Eds.), *Advances in Large Margin Classifiers*, pp. 61–73. MIT Press.

Platt, J. C., N. Cristianini, and J. Shawe-Taylor (2000). Large margin DAGs for multiclass classification. In S. A. Solla, T. K. Leen, and K. R. Müller (Eds.), *Advances in Neural Information Processing Systems*, Volume 12, pp. 547–553. MIT Press.

Poggio, T. and F. Girosi (1990). Networks for approximation and learning. *Proceedings of the IEEE* **78**(9), 1481–1497.

Powell, M. J. D. (1987). Radial basis functions for multivariable interpolation: a review. In J. C. Mason and M. G. Cox (Eds.), *Algorithms for Approximation*, pp. 143–167. Oxford University Press.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (1992). *Numerical Recipes in C: The Art of Scientific Computing* (Second ed.). Cambridge University Press.

Qazaz, C. S., C. K. I. Williams, and C. M. Bishop (1997). An upper bound on the Bayesian error bars for generalized linear regression. In S. W. Ellacott, J. C. Mason, and I. J. Anderson (Eds.), *Mathematics of Neural Networks: Models, Algorithms and Applications*, pp. 295–299. Kluwer.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning* **1**(1), 81–106.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

Rabiner, L. and B. H. Juang (1993). *Fundamentals of Speech Recognition*. Prentice Hall.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE* **77**(2), 257–285.

Ramasubramanian, V. and K. K. Paliwal (1990). A generalized optimization of the k-d tree for fast nearest-neighbour search. In *Proceedings Fourth IEEE Region 10 International Conference (TENCON'89)*, pp. 565–568.

Ramsey, F. (1931). Truth and probability. In R. Braithwaite (Ed.), *The Foundations of Mathematics and other Logical Essays*. Humanities Press.

Rao, C. R. and S. K. Mitra (1971). *Generalized Inverse of Matrices and Its Applications*. Wiley.

Rasmussen, C. E. (1996). *Evaluation of Gaussian Processes and Other Methods for Non-Linear Regression*. Ph. D. thesis, University of Toronto.

Rasmussen, C. E. and J. Quiñonero-Candela (2005). Healing the relevance vector machine by augmentation. In L. D. Raedt and S. Wrobel (Eds.), *Proceedings of the 22nd International Conference on Machine Learning*, pp. 689–696.

Rasmussen, C. E. and C. K. I. Williams (2006). *Gaussian Processes for Machine Learning*. MIT Press.

Rauch, H. E., F. Tung, and C. T. Striebel (1965). Maximum likelihood estimates of linear dynamical systems. *AIAA Journal* **3**, 1445–1450.

Ricotti, L. P., S. Ragazzini, and G. Martinelli (1988). Learning of word stress in a sub-optimal second order backpropagation neural network. In *Proceedings of the IEEE International Conference on Neural Networks*, Volume 1, pp. 355–361. IEEE.

Ripley, B. D. (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press.

Robbins, H. and S. Monro (1951). A stochastic approximation method. *Annals of Mathematical Statistics* **22**, 400–407.

Robert, C. P. and G. Casella (1999). *Monte Carlo Statistical Methods*. Springer.

Rockafellar, R. (1972). *Convex Analysis*. Princeton University Press.

Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan.

Roth, V. and V. Steinhage (2000). Nonlinear discriminant analysis using kernel functions. In S. A.

Solla, T. K. Leen, and K. R. Müller (Eds.), *Advances in Neural Information Processing Systems*, Volume 12. MIT Press.

Roweis, S. (1998). EM algorithms for PCA and SPCA. In M. I. Jordan, M. J. Kearns, and S. A. Solla (Eds.), *Advances in Neural Information Processing Systems*, Volume 10, pp. 626–632. MIT Press.

Roweis, S. and Z. Ghahramani (1999). A unifying review of linear Gaussian models. *Neural Computation* **11**(2), 305–345.

Roweis, S. and L. Saul (2000, December). Nonlinear dimensionality reduction by locally linear embedding. *Science* **290**, 2323–2326.

Rubin, D. B. (1983). Iteratively reweighted least squares. In *Encyclopedia of Statistical Sciences*, Volume 4, pp. 272–275. Wiley.

Rubin, D. B. and D. T. Thayer (1982). EM algorithms for ML factor analysis. *Psychometrika* **47**(1), 69–76.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, and the PDP Research Group (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1: Foundations, pp. 318–362. MIT Press. Reprinted in Anderson and Rosenfeld (1988).

Rumelhart, D. E., J. L. McClelland, and the PDP Research Group (Eds.) (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1: Foundations. MIT Press.

Sagan, H. (1969). *Introduction to the Calculus of Variations*. Dover.

Savage, L. J. (1961). The subjective basis of statistical practice. Technical report, Department of Statistics, University of Michigan, Ann Arbor.

Schölkopf, B., J. Platt, J. Shawe-Taylor, A. Smola, and R. C. Williamson (2001). Estimating the support of a high-dimensional distribution. *Neural Computation* **13**(7), 1433–1471.

Schölkopf, B., A. Smola, and K.-R. Müller (1998). Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation* **10**(5), 1299–1319.

Schölkopf, B., A. Smola, R. C. Williamson, and P. L. Bartlett (2000). New support vector algorithms. *Neural Computation* **12**(5), 1207–1245.

Schölkopf, B. and A. J. Smola (2002). *Learning with Kernels*. MIT Press.

Schwarz, G. (1978). Estimating the dimension of a model. *Annals of Statistics* **6**, 461–464.

Schwarz, H. R. (1988). *Finite element methods*. Academic Press.

Seeger, M. (2003). *Bayesian Gaussian Process Models: PAC-Bayesian Generalization Error Bounds and Sparse Approximations*. Ph. D. thesis, University of Edinburg.

Seeger, M., C. K. I. Williams, and N. Lawrence (2003). Fast forward selection to speed up sparse Gaussian processes. In C. M. Bishop and B. Frey (Eds.), *Proceedings Ninth International Workshop on Artificial Intelligence and Statistics, Key West, Florida*.

Shachter, R. D. and M. Peot (1990). Simulation approaches to general probabilistic inference on belief networks. In P. P. Bonissone, M. Henrion, L. N. Kanal, and J. F. Lemmer (Eds.), *Uncertainty in Artificial Intelligence*, Volume 5. Elsevier.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal* **27**(3), 379–423 and 623–656.

Shawe-Taylor, J. and N. Cristianini (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press.

Sietsma, J. and R. J. F. Dow (1991). Creating artificial neural networks that generalize. *Neural Networks* **4**(1), 67–79.

Simard, P., Y. Le Cun, and J. Denker (1993). Efficient pattern recognition using a new transformation distance. In S. J. Hanson, J. D. Cowan, and

C. L. Giles (Eds.), *Advances in Neural Information Processing Systems*, Volume 5, pp. 50–58. Morgan Kaufmann.

Simard, P., B. Victorri, Y. Le Cun, and J. Denker (1992). Tangent prop – a formalism for specifying selected invariances in an adaptive network. In J. E. Moody, S. J. Hanson, and R. P. Lippmann (Eds.), *Advances in Neural Information Processing Systems*, Volume 4, pp. 895–903. Morgan Kaufmann.

Simard, P. Y., D. Steinkraus, and J. Platt (2003). Best practice for convolutional neural networks applied to visual document analysis. In *Proceedings International Conference on Document Analysis and Recognition (ICDAR)*, pp. 958–962. IEEE Computer Society.

Sirovich, L. (1987). Turbulence and the dynamics of coherent structures. *Quarterly Applied Mathematics* **45**(3), 561–590.

Smola, A. J. and P. Bartlett (2001). Sparse greedy Gaussian process regression. In T. K. Leen, T. G. Dietterich, and V. Tresp (Eds.), *Advances in Neural Information Processing Systems*, Volume 13, pp. 619–625. MIT Press.

Spiegelhalter, D. and S. Lauritzen (1990). Sequential updating of conditional probabilities on directed graphical structures. *Networks* **20**, 579–605.

Stinchecombe, M. and H. White (1989). Universal approximation using feed-forward networks with non-sigmoid hidden layer activation functions. In *International Joint Conference on Neural Networks*, Volume 1, pp. 613–618. IEEE.

Stone, J. V. (2004). *Independent Component Analysis: A Tutorial Introduction*. MIT Press.

Sung, K. K. and T. Poggio (1994). Example-based learning for view-based human face detection. A.I. Memo 1521, MIT.

Sutton, R. S. and A. G. Barto (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Svensén, M. and C. M. Bishop (2004). Robust Bayesian mixture modelling. *Neurocomputing* **64**, 235–252.

Tarassenko, L. (1995). Novelty detection for the identification of masses in mamograms. In *Proceedings Fourth IEE International Conference on Artificial Neural Networks*, Volume 4, pp. 442–447. IEE.

Tax, D. and R. Duin (1999). Data domain description by support vectors. In M. Verleysen (Ed.), *Proceedings European Symposium on Artificial Neural Networks, ESANN*, pp. 251–256. D. Facto Press.

Teh, Y. W., M. I. Jordan, M. J. Beal, and D. M. Blei (2006). Hierarchical Dirichlet processes. *Journal of the Americal Statistical Association*. to appear.

Tenenbaum, J. B., V. de Silva, and J. C. Langford (2000, December). A global framework for nonlinear dimensionality reduction. *Science* **290**, 2319–2323.

Tesauro, G. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation* **6**(2), 215–219.

Thiesson, B., D. M. Chickering, D. Heckerman, and C. Meek (2004). ARMA time-series modelling with graphical models. In M. Chickering and J. Halpern (Eds.), *Proceedings of the Twentieth Conference on Uncertainty in Artificial Intelligence, Banff, Canada*, pp. 552–560. AUAI Press.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, B* **58**, 267–288.

Tierney, L. (1994). Markov chains for exploring posterior distributions. *Annals of Statistics* **22**(4), 1701–1762.

Tikhonov, A. N. and V. Y. Arsenin (1977). *Solutions of Ill-Posed Problems*. V. H. Winston.

Tino, P. and I. T. Nabney (2002). Hierarchical GTM: constructing localized non-linear projection manifolds in a principled way. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **24**(5), 639–656.

Tino, P., I. T. Nabney, and Y. Sun (2001). Using directional curvatures to visualize folding patterns of the GTM projection manifolds. In

G. Dorffner, H. Bischof, and K. Hornik (Eds.), *Artificial Neural Networks – ICANN 2001*, pp. 421–428. Springer.

Tipping, M. E. (1999). Probabilistic visualisation of high-dimensional binary data. In M. S. Kearns, S. A. Solla, and D. A. Cohn (Eds.), *Advances in Neural Information Processing Systems*, Volume 11, pp. 592–598. MIT Press.

Tipping, M. E. (2001). Sparse Bayesian learning and the relevance vector machine. *Journal of Machine Learning Research* **1**, 211–244.

Tipping, M. E. and C. M. Bishop (1997). Probabilistic principal component analysis. Technical Report NCRG/97/010, Neural Computing Research Group, Aston University.

Tipping, M. E. and C. M. Bishop (1999a). Mixtures of probabilistic principal component analyzers. *Neural Computation* **11**(2), 443–482.

Tipping, M. E. and C. M. Bishop (1999b). Probabilistic principal component analysis. *Journal of the Royal Statistical Society, Series B* **21**(3), 611–622.

Tipping, M. E. and A. Faul (2003). Fast marginal likelihood maximization for sparse Bayesian models. In C. M. Bishop and B. Frey (Eds.), *Proceedings Ninth International Workshop on Artificial Intelligence and Statistics, Key West, Florida*.

Tong, S. and D. Koller (2000). Restricted Bayes optimal classifiers. In *Proceedings 17th National Conference on Artificial Intelligence*, pp. 658–664. AAAI.

Tresp, V. (2001). Scaling kernel-based systems to large data sets. *Data Mining and Knowledge Discovery* **5**(3), 197–211.

Uhlenbeck, G. E. and L. S. Ornstein (1930). On the theory of Brownian motion. *Phys. Rev.* **36**, 823–841.

Valiant, L. G. (1984). A theory of the learnable. *Communications of the Association for Computing Machinery* **27**, 1134–1142.

Vapnik, V. N. (1982). *Estimation of dependences based on empirical data*. Springer.

Vapnik, V. N. (1995). *The nature of statistical learning theory*. Springer.

Vapnik, V. N. (1998). *Statistical learning theory*. Wiley.

Veropoulos, K., C. Campbell, and N. Cristianini (1999). Controlling the sensitivity of support vector machines. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI99), Workshop ML3*, pp. 55–60.

Vidakovic, B. (1999). *Statistical Modelling by Wavelets*. Wiley.

Viola, P. and M. Jones (2004). Robust real-time face detection. *International Journal of Computer Vision* **57**(2), 137–154.

Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory* **IT-13**, 260–267.

Viterbi, A. J. and J. K. Omura (1979). *Principles of Digital Communication and Coding*. McGraw-Hill.

Wahba, G. (1975). A comparison of GCV and GML for choosing the smoothing parameter in the generalized spline smoothing problem. *Numerical Mathematics* **24**, 383–393.

Wainwright, M. J., T. S. Jaakkola, and A. S. Willsky (2005). A new class of upper bounds on the log partition function. *IEEE Transactions on Information Theory* **51**, 2313–2335.

Walker, A. M. (1969). On the asymptotic behaviour of posterior distributions. *Journal of the Royal Statistical Society, B* **31**(1), 80–88.

Walker, S. G., P. Damien, P. W. Laud, and A. F. M. Smith (1999). Bayesian nonparametric inference for random distributions and related functions (with discussion). *Journal of the Royal Statistical Society, B* **61**(3), 485–527.

Watson, G. S. (1964). Smooth regression analysis. *Sankhyā: The Indian Journal of Statistics. Series A* **26**, 359–372.

Webb, A. R. (1994). Functional approximation by feed-forward networks: a least-squares approach to generalisation. *IEEE Transactions on Neural Networks* **5**(3), 363–371.

Weisstein, E. W. (1999). *CRC Concise Encyclopedia of Mathematics*. Chapman and Hall, and CRC.

Weston, J. and C. Watkins (1999). Multi-class support vector machines. In M. Verlysen (Ed.), *Proceedings ESANN'99, Brussels*. D-Facto Publications.

Whittaker, J. (1990). *Graphical Models in Applied Multivariate Statistics*. Wiley.

Widrow, B. and M. E. Hoff (1960). Adaptive switching circuits. In *IRE WESCON Convention Record*, Volume 4, pp. 96–104. Reprinted in Anderson and Rosenfeld (1988).

Widrow, B. and M. A. Lehr (1990). 30 years of adaptive neural networks: perceptron, madeline, and backpropagation. *Proceedings of the IEEE* **78**(9), 1415–1442.

Wiegerinck, W. and T. Heskes (2003). Fractional belief propagation. In S. Becker, S. Thrun, and K. Obermayer (Eds.), *Advances in Neural Information Processing Systems*, Volume 15, pp. 455–462. MIT Press.

Williams, C. K. I. (1998). Computation with infinite neural networks. *Neural Computation* **10**(5), 1203–1216.

Williams, C. K. I. (1999). Prediction with Gaussian processes: from linear regression to linear prediction and beyond. In M. I. Jordan (Ed.), *Learning in Graphical Models*, pp. 599–621. MIT Press.

Williams, C. K. I. and D. Barber (1998). Bayesian classification with Gaussian processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **20**, 1342–1351.

Williams, C. K. I. and M. Seeger (2001). Using the Nystrom method to speed up kernel machines. In T. K. Leen, T. G. Dietterich, and V. Tresp (Eds.), *Advances in Neural Information Processing Systems*, Volume 13, pp. 682–688. MIT Press.

Williams, O., A. Blake, and R. Cipolla (2005). Sparse Bayesian learning for efficient visual tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **27**(8), 1292–1304.

Williams, P. M. (1996). Using neural networks to model conditional multivariate densities. *Neural Computation* **8**(4), 843–854.

Winn, J. and C. M. Bishop (2005). Variational message passing. *Journal of Machine Learning Research* **6**, 661–694.

Zarchan, P. and H. Musoff (2005). *Fundamentals of Kalman Filtering: A Practical Approach* (Second ed.). AIAA.

Index

Page numbers in **bold** indicate the primary source of information for the corresponding topic.

1-of-K coding scheme, 424

acceptance criterion, **538**, 541, 544
activation function, **180**, 213, 227
active constraint, 328, **709**
AdaBoost, 657, **658**
adaline, 196
adaptive rejection sampling, 530
ADF, *see* assumed density filtering
AIC, *see* Akaike information criterion
Akaike information criterion, **33**, 217
α family of divergences, 469
α recursion, 620
ancestral sampling, **365**, 525, 613
annular flow, 679
AR model, *see* autoregressive model
arc, 360
ARD, *see* automatic relevance determination
ARMA, *see* autoregressive moving average
assumed density filtering, 510
autoassociative networks, 592
automatic relevance determination, 259, 312, **349**, 485, 582
autoregressive hidden Markov model, 632
autoregressive model, 609
autoregressive moving average, 304

back-tracking, **415**, 630
backgammon, 3
backpropagation, 241
bagging, 656
basis function, **138**, 172, 204, 227
batch training, 240
Baum-Welch algorithm, 618
Bayes' theorem, 15
Bayes, Thomas, 21
Bayesian analysis, vii, 9, **21**
    hierarchical, 372
    model averaging, 654
Bayesian information criterion, 33, **216**
Bayesian model comparison, **161**, 473, 483
Bayesian network, 360
Bayesian probability, 21
belief propagation, 403
Bernoulli distribution, **69**, 113, 685
    mixture model, 444
Bernoulli, Jacob, 69
beta distribution, **71**, 686
beta recursion, 621
between-class covariance, 189
bias, **27**, 149
bias parameter, **138**, 181, 227, 346
bias-variance trade-off, 147
BIC, *see* Bayesian information criterion
binary entropy, 495
binomial distribution, **70**, 686

biological sequence, 610
bipartite graph, 401
bits, 49
blind source separation, 591
blocked path, 374, **378**, 384
Boltzmann distribution, 387
Boltzmann, Ludwig Eduard, 53
Boolean logic, 21
boosting, 657
bootstrap, **23**, 656
bootstrap filter, 646
box constraints, **333**, 342
Box-Muller method, 527

C4.5, 663
calculus of variations, 462
canonical correlation analysis, 565
canonical link function, 212
CART, *see* classification and regression trees
Cauchy distribution, **527**, 529, 692
causality, 366
CCA, *see* canonical correlation analysis
central differences, 246
central limit theorem, 78
chain graph, 393
chaining, 555
Chapman-Kolmogorov equations, 397
child node, 361
Cholesky decomposition, 528
chunking, 335
circular normal, *see* von Mises distribution
classical probability, 21
classification, 3
classification and regression trees, 663
clique, 385
clustering, 3
clutter problem, 511
co-parents, **383**, 492
code-book vectors, 429
combining models, 45, **653**
committee, 655
complete data set, 440
completing the square, 86
computational learning theory, 326, 344
concave function, 56
concentration parameter, **108**, 693
condensation algorithm, 646
conditional entropy, 55
conditional expectation, 20
conditional independence, 46, **372**, 383
conditional mixture model, *see* mixture model
conditional probability, 14
conjugate prior, 68, 98, **117**, 490
convex duality, 494
convex function, **55**, 493
convolutional neural network, 267
correlation matrix, 567
cost function, 41
covariance, 20
    between-class, 189
    within-class, 189
covariance matrix
    diagonal, 84
    isotropic, 84
    partitioned, **85**, 307
    positive definite, 308
Cox's axioms, 21
credit assignment, 3
cross-entropy error function, **206**, 209, 235, 631, 666
cross-validation, **32**, 161
cumulative distribution function, 18
curse of dimensionality, **33**, 36
curve fitting, 4

D map, *see* dependency map
d-separation, 373, **378**, 443
DAG, *see* directed acyclic graph
DAGSVM, 339
data augmentation, 537
data compression, 429
decision boundary, **39**, 179
decision region, **39**, 179
decision surface, *see* decision boundary
decision theory, 38
decision tree, 654, **663**, 673
decomposition methods, 335
degrees of freedom, 559
degrees-of-freedom parameter, **102**, 693
density estimation, 3, **67**

density network, 597
dependency map, 392
descendant node, 376
design matrix, **142**, 347
differential entropy, 53
digamma function, 687
directed acyclic graph, 362
directed cycle, 362
directed factorization, 381
Dirichlet distribution, **76**, 687
Dirichlet, Lejeune, 77
discriminant function, 43, 180, **181**
discriminative model, **43**, 203
distortion measure, 424
distributive law of multiplication, 396
DNA, 610
document retrieval, 299
dual representation, **293**, 329
dual-energy gamma densitometry, 678
dynamic programming, 411
dynamical system, 548

E step, *see* expectation step
early stopping, 259
ECM, *see* expectation conditional maximization
edge, 360
effective number of observations, **72**, 101
effective number of parameters, 9, **170**, 281
elliptical K-means, 444
EM, *see* expectation maximization
emission probability, 611
empirical Bayes, *see* evidence approximation
energy function, 387
entropy, 49
    conditional, 55
    differential, 53
    relative, 55
EP, *see* expectation propagation
ϵ-tube, 341
ϵ-insensitive error function, 340
equality constraint, 709
equivalent kernel, **159**, 301
erf function, 211
error backpropagation, *see* backpropagation
error function, **5**, 23
error-correcting output codes, 339
Euler, Leonhard, 465
Euler-Lagrange equations, 705
evidence approximation, **165**, 347, 581
evidence function, 161
expectation, 19
expectation conditional maximization, 454
expectation maximization, 113, **423**, 440
    Gaussian mixture, 435
    generalized, 454
    sampling methods, 536
expectation propagation, 315, 468, **505**
expectation step, 437
explaining away, 378
exploitation, 3
exploration, 3
exponential distribution, **526**, 688
exponential family, 68, **113**, 202, 490
extensive variables, 490

face detection, 2
face tracking, 355
factor analysis, 583
    mixture model, 595
factor graph, 360, **399**, 625
factor loading, 584
factorial hidden Markov model, 633
factorized distribution, **464**, 476
feature extraction, 2
feature map, 268
feature space, **292**, 586
Fisher information matrix, 298
Fisher kernel, 298
Fisher's linear discriminant, 186
flooding schedule, 417
forward kinematics, 272
forward problem, 272
forward propagation, **228**, 243
forward-backward algorithm, 618
fractional belief propagation, 517
frequentist probability, 21
fuel system, 376
function interpolation, 299
functional, 462, **703**
    derivative, 463

gamma densitometry, 678
gamma distribution, 529, **688**
gamma function, 71
gating function, 672
Gauss, Carl Friedrich, 79
Gaussian, 24, **78**, 688
    conditional, **85**, 93
    marginal, **88**, 93
    maximum likelihood, 93
    mixture, 110, 270, 273, **430**
    sequential estimation, 94
    sufficient statistics, 93
    wrapped, 110
Gaussian kernel, 296
Gaussian process, 160, **303**
Gaussian random field, 305
Gaussian-gamma distribution, **101**, 690
Gaussian-Wishart distribution, **102**, 475, 478, **690**
GEM, *see* expectation maximization, generalized
generalization, 2
generalized linear model, **180**, 213
generalized maximum likelihood, *see* evidence approximation
generative model, **43**, 196, 297, 365, 572, 631
generative topographic mapping, 597
    directional curvature, 599
    magnification factor, 599
geodesic distance, 596
Gibbs sampling, 542
    blocking, 546
Gibbs, Josiah Willard, 543
Gini index, 666
global minimum, 237
gradient descent, 240
Gram matrix, 293
graph-cut algorithm, 390
graphical model, 359
    bipartite, 401
    directed, 360
    factorization, 362, 384
    fully connected, 361
    inference, 393
    tree, 398
    treewidth, 417
    triangulated, 416
    undirected, 360
Green's function, 299
GTM, *see* generative topographic mapping

Hamilton, William Rowan, 549
Hamiltonian dynamics, 548
Hamiltonian function, 549
Hammersley-Clifford theorem, 387
handwriting recognition, 1, 610, 614
handwritten digit, 565, 614, **677**
head-to-head path, 376
head-to-tail path, 375
Heaviside step function, 206
Hellinger distance, 470
Hessian matrix, 167, 215, 217, 238, **249**
    diagonal approximation, 250
    exact evaluation, 253
    fast multiplication, 254
    finite differences, 252
    inverse, 252
    outer product approximation, 251
heteroscedastic, **273**, 311
hidden Markov model, 297, **610**
    autoregressive, 632
    factorial, 633
    forward-backward algorithm, 618
    input-output, 633
    left-to-right, 613
    maximum likelihood, 615
    scaling factor, 627
    sum-product algorithm, 625
    switching, 644
    variational inference, 625
hidden unit, 227
hidden variable, 84, **364**, 430, 559
hierarchical Bayesian model, 372
hierarchical mixture of experts, 673
hinge error function, 337
Hinton diagram, 584
histogram density estimation, 120
HME, *see* hierarchical mixture of experts
hold-out set, 11
homogeneous flow, 679
homogeneous kernel, 292
homogeneous Markov chain, **540**, 608

Hooke's law, 580
hybrid Monte Carlo, 548
hyperparameter, **71**, 280, 311, 346, 372, 502
hyperprior, 372

I map, *see* independence map
i.i.d., *see* independent identically distributed
ICA, *see* independent component analysis
ICM, *see* iterated conditional modes
ID3, 663
identifiability, 435
image de-noising, 387
importance sampling, 525, **532**
importance weights, 533
improper prior, **118**, 259, 472
imputation step, 537
imputation-posterior algorithm, 537
inactive constraint, 328, **709**
incomplete data set, 440
independence map, 392
independent component analysis, 591
independent factor analysis, 592
independent identically distributed, **26**, 379
independent variables, 17
independent, identically distributed, 605
induced factorization, 485
inequality constraint, 709
inference, 38, **42**
information criterion, 33
information geometry, 298
information theory, 48
input-output hidden Markov model, 633
intensive variables, 490
intrinsic dimensionality, 559
invariance, 261
inverse gamma distribution, 101
inverse kinematics, 272
inverse problem, 272
inverse Wishart distribution, 102
IP algorithm, *see* imputation-posterior algorithm
IRLS, *see* iterative reweighted least squares
Ising model, 389
isomap, 596
isometric feature map, 596
iterated conditional modes, **389**, 415
iterative reweighted least squares, **207**, 210, 316, 354, 672

Jacobian matrix, **247**, 264
Jensen's inequality, 56
join tree, 416
junction tree algorithm, 392, **416**

K nearest neighbours, 125
K-means clustering algorithm, **424**, 443
K-medoids algorithm, 428
Kalman filter, 304, **637**
    extended, 644
Kalman gain matrix, 639
Kalman smoother, 637
Karhunen-Loève transform, 561
Karush-Kuhn-Tucker conditions, 330, 333, 342, **710**
kernel density estimator, **122**, 326
kernel function, 123, **292**, 294
    Fisher, 298
    Gaussian, 296
    homogeneous, 292
    nonvectorial inputs, 297
    stationary, 292
kernel PCA, 586
kernel regression, 300, **302**
kernel substitution, 292
kernel trick, 292
kinetic energy, 549
KKT, *see* Karush-Kuhn-Tucker conditions
KL divergence, *see* Kullback-Leibler divergence
kriging, *see* Gaussian process
Kullback-Leibler divergence, **55**, 451, 468, 505

Lagrange multiplier, 707
Lagrange, Joseph-Louis, 329
Lagrangian, 328, 332, 341, **708**
laminar flow, 678
Laplace approximation, **213**, 217, 278, 315, 354
Laplace, Pierre-Simon, 24
large margin, *see* margin
lasso, 145
latent class analysis, 444
latent trait model, 597
latent variable, 84, **364**, 430, 559

lattice diagram, **414**, 611, 621, 629
LDS, *see* linear dynamical system
leapfrog discretization, 551
learning, 2
learning rate parameter, 240
least-mean-squares algorithm, 144
leave-one-out, 33
likelihood function, 22
likelihood weighted sampling, 534
linear discriminant, 181
    Fisher, 186
linear dynamical system, 84, **635**
    inference, 638
linear independence, 696
linear regression, 138
    EM, 448
    mixture model, 667
    variational, 486
linear smoother, 159
linear-Gaussian model, 87, **370**
linearly separable, 179
link, 360
link function, 180, 213
Liouville's Theorem, 550
LLE, *see* locally linear embedding
LMS algorithm, *see* least-mean-squares algorithm
local minimum, 237
local receptive field, 268
locally linear embedding, 596
location parameter, 118
log odds, 197
logic sampling, 525
logistic regression, **205**, 336
    Bayesian, 217, 498
    mixture model, 670
    multiclass, 209
logistic sigmoid, 114, 139, **197**, 205, 220, 227, 495
logit function, 197
loopy belief propagation, 417
loss function, 41
loss matrix, 41
lossless data compression, 429
lossy data compression, 429
lower bound, 484

M step, *see* maximization step
machine learning, vii
macrostate, 51
Mahalanobis distance, 80
manifold, **38**, 590, 595, 681
MAP, *see* maximum posterior
margin, 326, **327**, 502
    error, 334
    soft, 332
marginal likelihood, **162**, 165
marginal probability, 14
Markov blanket, **382**, 384, 545
Markov boundary, *see* Markov blanket
Markov chain, 397, **539**
    first order, 607
    homogeneous, **540**, 608
    second order, 608
Markov chain Monte Carlo, 537
Markov model, 607
    homogeneous, 612
Markov network, *see* Markov random field
Markov random field, 84, 360, **383**
max-sum algorithm, **411**, 629
maximal clique, 385
maximal spanning tree, 416
maximization step, 437
maximum likelihood, 9, **23**, 26, 116
    Gaussian mixture, 432
    singularities, 480
    type 2, *see* evidence approximation
maximum margin, *see* margin
maximum posterior, **30**, 441
MCMC, *see* Markov chain Monte Carlo
MDN, *see* mixture density network
MDS, *see* multidimensional scaling
mean, 24
mean field theory, 465
mean value theorem, 52
measure theory, 19
memory-based methods, 292
message passing, 396
    pending message, 417
    schedule, 417
    variational, 491
Metropolis algorithm, 538
Metropolis-Hastings algorithm, 541

microstate, 51
minimum risk, 44
Minkowski loss, 48
missing at random, **441**, 579
missing data, 579
mixing coefficient, 111
mixture component, 111
mixture density network, **272**, 673
mixture distribution, *see* mixture model
mixture model, 162, **423**
    conditional, 273, **666**
    linear regression, 667
    logistic regression, 670
    symmetries, 483
mixture of experts, 672
mixture of Gaussians, 110, 270, 273, **430**
MLP, *see* multilayer perceptron
MNIST data, 677
model comparison, 6, 32, **161**, 473, 483
model evidence, 161
model selection, 162
moment matching, **506**, 510
momentum variable, 548
Monte Carlo EM algorithm, 536
Monte Carlo sampling, 24, **523**
Moore-Penrose pseudo-inverse, *see* pseudo-inverse
moralization, **391**, 401
MRF, *see* Markov random field
multidimensional scaling, 596
multilayer perceptron, 226, **229**
multimodality, 272
multinomial distribution, 76, 114, **690**
multiplicity, 51
mutual information, 55, **57**

Nadaraya-Watson, *see* kernel regression
naive Bayes model, 46, **380**
nats, 50
natural language modelling, 610
natural parameters, 113
nearest-neighbour methods, 124
neural network, 225
    convolutional, 267
    regularization, 256
    relation to Gaussian process, 319
Newton-Raphson, **207**, 317
node, 360
noiseless coding theorem, 50
nonidentifiability, 585
noninformative prior, 23, **117**
nonparametric methods, 68, **120**
normal distribution, *see* Gaussian
normal equations, 142
normal-gamma distribution, **101**, 691
normal-Wishart distribution, **102**, 475, 478, 691
normalized exponential, *see* softmax function
novelty detection, 44
ν-SVM, 334

object recognition, 366
observed variable, 364
Occam factor, 217
oil flow data, 34, 560, 568, **678**
Old Faithful data, 110, 479, 484, **681**
on-line learning, *see* sequential learning
one-versus-one classifier, **183**, 339
one-versus-the-rest classifier, **182**, 338
ordered over-relaxation, 545
Ornstein-Uhlenbeck process, 305
orthogonal least squares, 301
outlier, 44, 185, 212
outliers, **103**
over-fitting, **6**, 147, 434, 464
over-relaxation, 544

PAC learning, *see* probably approximately correct
PAC-Bayesian framework, 345
parameter shrinkage, 144
parent node, 361
particle filter, 645
partition function, **386**, 554
Parzen estimator, *see* kernel density estimator
Parzen window, 123
pattern recognition, vii
PCA, *see* principal component analysis
pending message, 417
perceptron, 192
    convergence theorem, 194
    hardware, 196
perceptron criterion, 193
perfect map, 392

periodic variable, 105
phase space, 549
photon noise, 680
plate, 363
polynomial curve fitting, 4, 362
polytree, 399
position variable, 548
positive definite covariance, 81
positive definite matrix, 701
positive semidefinite covariance, 81
positive semidefinite matrix, 701
posterior probability, 17
posterior step, 537
potential energy, 549
potential function, 386
power EP, 517
power method, 563
precision matrix, 85
precision parameter, 24
predictive distribution, **30**, 156
preprocessing, 2
principal component analysis, **561**, 572, 593
    Bayesian, 580
    EM algorithm, 577
    Gibbs sampling, 583
    mixture distribution, 595
    physical analogy, 580
principal curve, 595
principal subspace, 561
principal surface, 596
prior, 17
    conjugate, 68, 98, **117**, 490
    consistent, 257
    improper, **118**, 259, 472
    noninformative, 23, **117**
probabilistic graphical model, *see* graphical model
probabilistic PCA, 570
probability, 12
    Bayesian, 21
    classical, 21
    density, 17
    frequentist, 21
    mass function, 19
    prior, 45
    product rule, 13, **14**, 359
    sum rule, 13, **14**, 359
    theory, 12
probably approximately correct, 344
probit function, **211**, 219
probit regression, 210
product rule of probability, 13, **14**, 359
proposal distribution, **528**, 532, 538
protected conjugate gradients, 335
protein sequence, 610
pseudo-inverse, **142**, 185
pseudo-random numbers, 526

quadratic discriminant, 199
quality parameter, 351

radial basis function, 292, **299**
Rauch-Tung-Striebel equations, 637
regression, 3
regression function, **47**, 95
regularization, 10
    Tikhonov, 267
regularized least squares, 144
reinforcement learning, 3
reject option, **42**, 45
rejection sampling, 528
relative entropy, 55
relevance vector, 348
relevance vector machine, 161, **345**
responsibility, 112, **432**, 477
ridge regression, 10
RMS error, *see* root-mean-square error
Robbins-Monro algorithm, 95
robot arm, 272
robustness, **103**, 185
root node, 399
root-mean-square error, 6
Rosenblatt, Frank, 193
rotation invariance, **573**, 585
RTS equations, *see* Rauch-Tung-Striebel equations
running intersection property, 416
RVM, *see* relevance vector machine

sample mean, 27
sample variance, 27
sampling-importance-resampling, 534
scale invariance, 119, **261**

scale parameter, 119
scaling factor, 627
Schwarz criterion, *see* Bayesian information criterion
self-organizing map, 598
sequential data, 605
sequential estimation, 94
sequential gradient descent, 144, 240
sequential learning, **73**, 143
sequential minimal optimization, 335
serial message passing schedule, 417
Shannon, Claude, 55
shared parameters, 368
shrinkage, 10
Shur complement, 87
sigmoid, *see* logistic sigmoid
simplex, 76
single-class support vector machine, 339
singular value decomposition, 143
sinusoidal data, 682
SIR, *see* sampling-importance-resampling
skip-layer connection, 229
slack variable, 331
slice sampling, 546
SMO, *see* sequential minimal optimization
smoother matrix, 159
smoothing parameter, 122
soft margin, 332
soft weight sharing, 269
softmax function, 115, **198**, 236, 274, 356, 497
SOM, *see* self-organizing map
sparsity, 145, 347, **349**, 582
sparsity parameter, 351
spectrogram, 606
speech recognition, **605**, 610
sphereing, 568
spline functions, 139
standard deviation, 24
standardizing, 425, **567**
state space model, 609
    switching, 644
stationary kernel, 292
statistical bias, *see* bias
statistical independence, *see* independent variables
statistical learning theory, *see* computational learning theory, 326, 344
steepest descent, 240
Stirling's approximation, 51
stochastic, 5
stochastic EM, 536
stochastic gradient descent, 144, 240
stochastic process, 305
stratified flow, 678
Student's t-distribution, **102**, 483, 691
subsampling, 268
sufficient statistics, 69, 75, **116**
sum rule of probability, 13, **14**, 359
sum-of-squares error, **5**, 29, 184, 232, 662
sum-product algorithm, 399, **402**
    for hidden Markov model, 625
supervised learning, 3
support vector, 330
support vector machine, 225
    for regression, 339
    multiclass, 338
survival of the fittest, 646
SVD, *see* singular value decomposition
SVM, *see* support vector machine
switching hidden Markov model, 644
switching state space model, 644
synthetic data sets, 682

tail-to-tail path, 374
tangent distance, 265
tangent propagation, 262, **263**
tapped delay line, 609
target vector, 2
test set, 2, **32**
threshold parameter, 181
tied parameters, 368
Tikhonov regularization, 267
time warping, 615
tomography, 679
training, 2
training set, 2
transition probability, **540**, 610
translation invariance, 118, **261**
tree-reweighted message passing, 517
treewidth, 417

trellis diagram, *see* lattice diagram
triangulated graph, 416
type 2 maximum likelihood, *see* evidence approximation

undetermined multiplier, *see* Lagrange multiplier
undirected graph, *see* Markov random field
uniform distribution, 692
uniform sampling, 534
uniquenesses, 584
unobserved variable, *see* latent variable
unsupervised learning, 3
utility function, 41

validation set, 11, **32**
Vapnik-Chervonenkis dimension, 344
variance, **20**, 24, 149
variational inference, 315, **462**, 635
  for Gaussian mixture, 474
  for hidden Markov model, 625
  local, 493
VC dimension, *see* Vapnik-Chervonenkis dimension
vector quantization, 429
vertex, *see* node
visualization, 3
Viterbi algorithm, 415, **629**
von Mises distribution, **108**, 693

wavelets, 139
weak learner, 657
weight decay, 10, **144**, 257
weight parameter, 227
weight sharing, 268
  soft, 269
weight vector, 181
weight-space symmetry, **231**, 281
weighted least squares, 668
well-determined parameters, 170
whitening, 299, **568**
Wishart distribution, **102**, 693
within-class covariance, 189
Woodbury identity, 696
wrapped distribution, 110

Yellowstone National Park, 110, **681**

O'REILLY®

2nd Edition
Updated for
TensorFlow 2

Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow

Concepts, Tools, and Techniques to Build Intelligent Systems

powered by

jupyter™

Aurélien Géron

O'REILLY®

Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

Through a series of breakthroughs, Deep Learning has boosted the entire field of machine learning. Now, even programmers who know close to nothing about this technology can use simple, efficient tools to implement programs capable of learning from data. The updated edition of this best-selling book uses concrete examples, minimal theory, and production-ready Python frameworks to help you gain an intuitive understanding of the concepts and tools for building intelligent systems.

You'll learn a range of techniques that you can quickly put to use. With exercises in each chapter to help you apply what you've learned, all you need is programming experience to get started. All code is available on GitHub. It has been updated to TensorFlow 2 and the latest version of Scikit-Learn.

- Learn Machine Learning fundamentals through an end-to-end project using Scikit-Learn and pandas
- Build and train many neural network architectures for classification and regression using TensorFlow 2
- Discover object detection, semantic segmentation, attention mechanisms, language models, GANs, and more
- Explore the Keras API, the official high-level API for TensorFlow 2
- Productionize TensorFlow models using TensorFlow's Data API, distribution strategies API, TF Transform, and TF-Serving
- Deploy on Google Cloud AI Platform or on mobile devices
- Exploit unsupervised learning techniques such as dimensionality reduction, clustering, and anomaly detection
- Create autonomous learning agents with Reinforcement Learning, including using the TF-Agents library

"An exceptional resource to study Machine Learning. You will find clear-minded, intuitive explanations, and a wealth of practical tips."

—**François Chollet**
Author of Keras, author of *Deep Learning with Python*

"This book is a great introduction to the theory and practice of solving problems with neural networks; I recommend it to anyone interested in learning about practical ML."

—**Pete Warden**
Mobile Lead for TensorFlow

Aurélien Géron is a machine learning consultant and trainer. A former Googler, he led YouTube's video classification team from 2013 to 2016. He was also a founder and CTO of Wifirst (a leading Wireless ISP in France) from 2002 to 2012.

DATA | DATA SCIENCE | DATA ANALYTICS
MACHINE LEARNING | DEEP LEARNING
PYTHON MACHINE LEARNING

US $74.99 CAN $98.99

ISBN: 978-1-492-03264-9

Twitter: @oreillymedia
facebook.com/oreilly

SECOND EDITION

Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

Concepts, Tools, and Techniques to Build Intelligent Systems

Aurélien Géron

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow
by Aurélien Géron

Copyright © 2019 Kiwisoft S.A.S. All rights reserved.

Printed in Canada.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Rachel Roumeliotis and Nicole Tache
Production Editor: Kristen Brown
Copyeditor: Amanda Kersey
Proofreader: Rachel Head
Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

September 2019: Second Edition

Revision History for the Second Edition

2019-09-05: First Release
2019-10-11: Second Release
2019-11-22: Third Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492032649* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-03264-9

[TI]

Table of Contents

Preface ..... **xv**

Part I. The Fundamentals of Machine Learning

1. The Machine Learning Landscape **1**
What Is Machine Learning? 2
Why Use Machine Learning? 2
Examples of Applications 5
Types of Machine Learning Systems 7
Supervised/Unsupervised Learning 7
Batch and Online Learning 14
Instance-Based Versus Model-Based Learning 17
Main Challenges of Machine Learning 23
Insufficient Quantity of Training Data 23
Nonrepresentative Training Data 25
Poor-Quality Data 26
Irrelevant Features 27
Overfitting the Training Data 27
Underfitting the Training Data 29
Stepping Back 30
Testing and Validating 30
Hyperparameter Tuning and Model Selection 31
Data Mismatch 32
Exercises 33

2. End-to-End Machine Learning Project **35**
Working with Real Data 35

Look at the Big Picture 37
Frame the Problem 37
Select a Performance Measure 39
Check the Assumptions 42
Get the Data 42
Create the Workspace 42
Download the Data 46
Take a Quick Look at the Data Structure 47
Create a Test Set 51
Discover and Visualize the Data to Gain Insights 56
Visualizing Geographical Data 56
Looking for Correlations 58
Experimenting with Attribute Combinations 61
Prepare the Data for Machine Learning Algorithms 62
Data Cleaning 63
Handling Text and Categorical Attributes 65
Custom Transformers 68
Feature Scaling 69
Transformation Pipelines 70
Select and Train a Model 72
Training and Evaluating on the Training Set 72
Better Evaluation Using Cross-Validation 73
Fine-Tune Your Model 75
Grid Search 76
Randomized Search 78
Ensemble Methods 78
Analyze the Best Models and Their Errors 78
Evaluate Your System on the Test Set 79
Launch, Monitor, and Maintain Your System 80
Try It Out! 83
Exercises 84

3. Classification. 85
MNIST 85
Training a Binary Classifier 88
Performance Measures 88
Measuring Accuracy Using Cross-Validation 89
Confusion Matrix 90
Precision and Recall 92
Precision/Recall Trade-off 93
The ROC Curve 97
Multiclass Classification 100

Error Analysis 102
Multilabel Classification 106
Multioutput Classification 107
Exercises 108

4. Training Models. 111
Linear Regression 112
The Normal Equation 114
Computational Complexity 117
Gradient Descent 118
Batch Gradient Descent 121
Stochastic Gradient Descent 124
Mini-batch Gradient Descent 127
Polynomial Regression 128
Learning Curves 130
Regularized Linear Models 134
Ridge Regression 135
Lasso Regression 137
Elastic Net 140
Early Stopping 141
Logistic Regression 142
Estimating Probabilities 143
Training and Cost Function 144
Decision Boundaries 145
Softmax Regression 148
Exercises 151

5. Support Vector Machines. 153
Linear SVM Classification 153
Soft Margin Classification 154
Nonlinear SVM Classification 157
Polynomial Kernel 158
Similarity Features 159
Gaussian RBF Kernel 160
Computational Complexity 162
SVM Regression 162
Under the Hood 164
Decision Function and Predictions 165
Training Objective 166
Quadratic Programming 167
The Dual Problem 168
Kernelized SVMs 169

Online SVMs 172
Exercises 174

6. Decision Trees. 175
Training and Visualizing a Decision Tree 175
Making Predictions 176
Estimating Class Probabilities 178
The CART Training Algorithm 179
Computational Complexity 180
Gini Impurity or Entropy? 180
Regularization Hyperparameters 181
Regression 183
Instability 185
Exercises 186

7. Ensemble Learning and Random Forests. 189
Voting Classifiers 189
Bagging and Pasting 192
Bagging and Pasting in Scikit-Learn 194
Out-of-Bag Evaluation 195
Random Patches and Random Subspaces 196
Random Forests 197
Extra-Trees 198
Feature Importance 198
Boosting 199
AdaBoost 200
Gradient Boosting 203
Stacking 208
Exercises 211

8. Dimensionality Reduction. 213
The Curse of Dimensionality 214
Main Approaches for Dimensionality Reduction 215
Projection 215
Manifold Learning 218
PCA 219
Preserving the Variance 219
Principal Components 220
Projecting Down to d Dimensions 221
Using Scikit-Learn 222
Explained Variance Ratio 222
Choosing the Right Number of Dimensions 223

PCA for Compression 224
Randomized PCA 225
Incremental PCA 225
Kernel PCA 226
Selecting a Kernel and Tuning Hyperparameters 227
LLE 230
Other Dimensionality Reduction Techniques 232
Exercises 233

9. Unsupervised Learning Techniques. 235
Clustering 236
K-Means 238
Limits of K-Means 248
Using Clustering for Image Segmentation 249
Using Clustering for Preprocessing 251
Using Clustering for Semi-Supervised Learning 253
DBSCAN 255
Other Clustering Algorithms 258
Gaussian Mixtures 260
Anomaly Detection Using Gaussian Mixtures 266
Selecting the Number of Clusters 267
Bayesian Gaussian Mixture Models 270
Other Algorithms for Anomaly and Novelty Detection 274
Exercises 275

Part II. Neural Networks and Deep Learning

10. Introduction to Artificial Neural Networks with Keras. 279
From Biological to Artificial Neurons 280
Biological Neurons 281
Logical Computations with Neurons 283
The Perceptron 284
The Multilayer Perceptron and Backpropagation 289
Regression MLPs 292
Classification MLPs 294
Implementing MLPs with Keras 295
Installing TensorFlow 2 296
Building an Image Classifier Using the Sequential API 297
Building a Regression MLP Using the Sequential API 307
Building Complex Models Using the Functional API 308
Using the Subclassing API to Build Dynamic Models 313

Saving and Restoring a Model 314
Using Callbacks 315
Using TensorBoard for Visualization 317
Fine-Tuning Neural Network Hyperparameters 320
Number of Hidden Layers 323
Number of Neurons per Hidden Layer 324
Learning Rate, Batch Size, and Other Hyperparameters 325
Exercises 327

11. Training Deep Neural Networks. 331
The Vanishing/Exploding Gradients Problems 332
Glorot and He Initialization 333
Nonsaturating Activation Functions 335
Batch Normalization 338
Gradient Clipping 345
Reusing Pretrained Layers 345
Transfer Learning with Keras 347
Unsupervised Pretraining 349
Pretraining on an Auxiliary Task 350
Faster Optimizers 351
Momentum Optimization 351
Nesterov Accelerated Gradient 353
AdaGrad 354
RMSProp 355
Adam and Nadam Optimization 356
Learning Rate Scheduling 359
Avoiding Overfitting Through Regularization 364
ℓ_1 and ℓ_2 Regularization 364
Dropout 365
Monte Carlo (MC) Dropout 368
Max-Norm Regularization 370
Summary and Practical Guidelines 371
Exercises 373

12. Custom Models and Training with TensorFlow. 375
A Quick Tour of TensorFlow 376
Using TensorFlow like NumPy 379
Tensors and Operations 379
Tensors and NumPy 381
Type Conversions 381
Variables 382
Other Data Structures 383

Customizing Models and Training Algorithms 384
Custom Loss Functions 384
Saving and Loading Models That Contain Custom Components 385
Custom Activation Functions, Initializers, Regularizers, and Constraints 387
Custom Metrics 388
Custom Layers 391
Custom Models 394
Losses and Metrics Based on Model Internals 397
Computing Gradients Using Autodiff 399
Custom Training Loops 402
TensorFlow Functions and Graphs 405
AutoGraph and Tracing 407
TF Function Rules 409
Exercises 410

13. Loading and Preprocessing Data with TensorFlow. 413
The Data API 414
Chaining Transformations 415
Shuffling the Data 416
Preprocessing the Data 419
Putting Everything Together 420
Prefetching 421
Using the Dataset with tf.keras 423
The TFRecord Format 424
Compressed TFRecord Files 425
A Brief Introduction to Protocol Buffers 425
TensorFlow Protobufs 427
Loading and Parsing Examples 428
Handling Lists of Lists Using the SequenceExample Protobuf 429
Preprocessing the Input Features 430
Encoding Categorical Features Using One-Hot Vectors 431
Encoding Categorical Features Using Embeddings 433
Keras Preprocessing Layers 437
TF Transform 439
The TensorFlow Datasets (TFDS) Project 441
Exercises 442

14. Deep Computer Vision Using Convolutional Neural Networks. 445
The Architecture of the Visual Cortex 446
Convolutional Layers 448
Filters 450
Stacking Multiple Feature Maps 451

TensorFlow Implementation 453
Memory Requirements 456
Pooling Layers 456
TensorFlow Implementation 458
CNN Architectures 460
LeNet-5 463
AlexNet 464
GoogLeNet 466
VGGNet 470
ResNet 471
Xception 474
SENet 476
Implementing a ResNet-34 CNN Using Keras 478
Using Pretrained Models from Keras 479
Pretrained Models for Transfer Learning 481
Classification and Localization 483
Object Detection 485
Fully Convolutional Networks 487
You Only Look Once (YOLO) 489
Semantic Segmentation 492
Exercises 496

15. Processing Sequences Using RNNs and CNNs. 497
Recurrent Neurons and Layers 498
Memory Cells 500
Input and Output Sequences 501
Training RNNs 502
Forecasting a Time Series 503
Baseline Metrics 505
Implementing a Simple RNN 505
Deep RNNs 506
Forecasting Several Time Steps Ahead 508
Handling Long Sequences 511
Fighting the Unstable Gradients Problem 512
Tackling the Short-Term Memory Problem 514
Exercises 523

16. Natural Language Processing with RNNs and Attention. 525
Generating Shakespearean Text Using a Character RNN 526
Creating the Training Dataset 527
How to Split a Sequential Dataset 527
Chopping the Sequential Dataset into Multiple Windows 528

Building and Training the Char-RNN Model 530
Using the Char-RNN Model 531
Generating Fake Shakespearean Text 531
Stateful RNN 532
Sentiment Analysis 534
Masking 538
Reusing Pretrained Embeddings 540
An Encoder–Decoder Network for Neural Machine Translation 542
Bidirectional RNNs 546
Beam Search 547
Attention Mechanisms 549
Visual Attention 552
Attention Is All You Need: The Transformer Architecture 554
Recent Innovations in Language Models 563
Exercises 565

17. Representation Learning and Generative Learning Using Autoencoders and GANs. 567
Efficient Data Representations 569
Performing PCA with an Undercomplete Linear Autoencoder 570
Stacked Autoencoders 572
Implementing a Stacked Autoencoder Using Keras 572
Visualizing the Reconstructions 574
Visualizing the Fashion MNIST Dataset 574
Unsupervised Pretraining Using Stacked Autoencoders 576
Tying Weights 577
Training One Autoencoder at a Time 578
Convolutional Autoencoders 579
Recurrent Autoencoders 580
Denoising Autoencoders 581
Sparse Autoencoders 582
Variational Autoencoders 586
Generating Fashion MNIST Images 590
Generative Adversarial Networks 592
The Difficulties of Training GANs 596
Deep Convolutional GANs 598
Progressive Growing of GANs 601
StyleGANs 604
Exercises 607

18. Reinforcement Learning. 609
Learning to Optimize Rewards 610
Policy Search 612

Introduction to OpenAI Gym 613
Neural Network Policies 617
Evaluating Actions: The Credit Assignment Problem 619
Policy Gradients 620
Markov Decision Processes 625
Temporal Difference Learning 629
Q-Learning 630
Exploration Policies 632
Approximate Q-Learning and Deep Q-Learning 633
Implementing Deep Q-Learning 634
Deep Q-Learning Variants 639
Fixed Q-Value Targets 639
Double DQN 640
Prioritized Experience Replay 640
Dueling DQN 641
The TF-Agents Library 642
Installing TF-Agents 643
TF-Agents Environments 643
Environment Specifications 644
Environment Wrappers and Atari Preprocessing 645
Training Architecture 649
Creating the Deep Q-Network 650
Creating the DQN Agent 652
Creating the Replay Buffer and the Corresponding Observer 654
Creating Training Metrics 655
Creating the Collect Driver 656
Creating the Dataset 658
Creating the Training Loop 661
Overview of Some Popular RL Algorithms 662
Exercises 664

19. Training and Deploying TensorFlow Models at Scale. 667
Serving a TensorFlow Model 668
Using TensorFlow Serving 668
Creating a Prediction Service on GCP AI Platform 677
Using the Prediction Service 682
Deploying a Model to a Mobile or Embedded Device 685
Using GPUs to Speed Up Computations 689
Getting Your Own GPU 690
Using a GPU-Equipped Virtual Machine 692
Colaboratory 693
Managing the GPU RAM 694

Placing Operations and Variables on Devices 697
Parallel Execution Across Multiple Devices 699
Training Models Across Multiple Devices 701
Model Parallelism 701
Data Parallelism 704
Training at Scale Using the Distribution Strategies API 709
Training a Model on a TensorFlow Cluster 711
Running Large Training Jobs on Google Cloud AI Platform 714
Black Box Hyperparameter Tuning on AI Platform 716
Exercises 717
Thank You! 718

A. Exercise Solutions **719**

B. Machine Learning Project Checklist **755**

C. SVM Dual Problem **761**

D. Autodiff **765**

E. Other Popular ANN Architectures **773**

F. Special Data Structures **783**

G. TensorFlow Graphs **791**

Index **801**

Preface

The Machine Learning Tsunami

In 2006, Geoffrey Hinton et al. published a paper[1] showing how to train a deep neural network capable of recognizing handwritten digits with state-of-the-art precision (>98%). They branded this technique "Deep Learning." A deep neural network is a (very) simplified model of our cerebral cortex, composed of a stack of layers of artificial neurons. Training a deep neural net was widely considered impossible at the time,[2] and most researchers had abandoned the idea in the late 1990s. This paper revived the interest of the scientific community, and before long many new papers demonstrated that Deep Learning was not only possible, but capable of mind-blowing achievements that no other Machine Learning (ML) technique could hope to match (with the help of tremendous computing power and great amounts of data). This enthusiasm soon extended to many other areas of Machine Learning.

A decade or so later, Machine Learning has conquered the industry: it is at the heart of much of the magic in today's high-tech products, ranking your web search results, powering your smartphone's speech recognition, recommending videos, and beating the world champion at the game of Go. Before you know it, it will be driving your car.

Machine Learning in Your Projects

So, naturally you are excited about Machine Learning and would love to join the party!

1 Geoffrey E. Hinton et al., "A Fast Learning Algorithm for Deep Belief Nets," *Neural Computation* 18 (2006): 1527–1554.

2 Despite the fact that Yann LeCun's deep convolutional neural networks had worked well for image recognition since the 1990s, although they were not as general-purpose.

Perhaps you would like to give your homemade robot a brain of its own? Make it recognize faces? Or learn to walk around?

Or maybe your company has tons of data (user logs, financial data, production data, machine sensor data, hotline stats, HR reports, etc.), and more than likely you could unearth some hidden gems if you just knew where to look. With Machine Learning, you could accomplish the following and more:

- Segment customers and find the best marketing strategy for each group.
- Recommend products for each client based on what similar clients bought.
- Detect which transactions are likely to be fraudulent.
- Forecast next year's revenue.

Whatever the reason, you have decided to learn Machine Learning and implement it in your projects. Great idea!

Objective and Approach

This book assumes that you know close to nothing about Machine Learning. Its goal is to give you the concepts, tools, and intuition you need to implement programs capable of *learning from data*.

We will cover a large number of techniques, from the simplest and most commonly used (such as Linear Regression) to some of the Deep Learning techniques that regularly win competitions.

Rather than implementing our own toy versions of each algorithm, we will be using production-ready Python frameworks:

- Scikit-Learn is very easy to use, yet it implements many Machine Learning algorithms efficiently, so it makes for a great entry point to learning Machine Learning. It was created by David Cournapeau in 2007, and is now led by a team of researchers at the French Institute for Research in Computer Science and Automation (Inria).
- TensorFlow is a more complex library for distributed numerical computation. It makes it possible to train and run very large neural networks efficiently by distributing the computations across potentially hundreds of multi-GPU (graphics processing unit) servers. TensorFlow (TF) was created at Google and supports many of its large-scale Machine Learning applications. It was open sourced in November 2015, and version 2.0 was released in September 2019.
- Keras is a high-level Deep Learning API that makes it very simple to train and run neural networks. It can run on top of either TensorFlow, Theano, or Microsoft Cognitive Toolkit (formerly known as CNTK). TensorFlow comes with its

own implementation of this API, called *tf.keras*, which provides support for some advanced TensorFlow features (e.g., the ability to efficiently load data).

The book favors a hands-on approach, growing an intuitive understanding of Machine Learning through concrete working examples and just a little bit of theory. While you can read this book without picking up your laptop, I highly recommend you experiment with the code examples available online as Jupyter notebooks at *https://github.com/ageron/handson-ml2*.

Prerequisites

This book assumes that you have some Python programming experience and that you are familiar with Python's main scientific libraries—in particular, NumPy, pandas, and Matplotlib.

Also, if you care about what's under the hood, you should have a reasonable understanding of college-level math as well (calculus, linear algebra, probabilities, and statistics).

If you don't know Python yet, *http://learnpython.org/* is a great place to start. The official tutorial on Python.org is also quite good.

If you have never used Jupyter, Chapter 2 will guide you through installation and the basics: it is a powerful tool to have in your toolbox.

If you are not familiar with Python's scientific libraries, the provided Jupyter notebooks include a few tutorials. There is also a quick math tutorial for linear algebra.

Roadmap

This book is organized in two parts. Part I, *The Fundamentals of Machine Learning*, covers the following topics:

- What Machine Learning is, what problems it tries to solve, and the main categories and fundamental concepts of its systems
- The steps in a typical Machine Learning project
- Learning by fitting a model to data
- Optimizing a cost function
- Handling, cleaning, and preparing data
- Selecting and engineering features
- Selecting a model and tuning hyperparameters using cross-validation
- The challenges of Machine Learning, in particular underfitting and overfitting (the bias/variance trade-off)

- The most common learning algorithms: Linear and Polynomial Regression, Logistic Regression, k-Nearest Neighbors, Support Vector Machines, Decision Trees, Random Forests, and Ensemble methods
- Reducing the dimensionality of the training data to fight the "curse of dimensionality"
- Other unsupervised learning techniques, including clustering, density estimation, and anomaly detection

Part II, *Neural Networks and Deep Learning*, covers the following topics:

- What neural nets are and what they're good for
- Building and training neural nets using TensorFlow and Keras
- The most important neural net architectures: feedforward neural nets for tabular data, convolutional nets for computer vision, recurrent nets and long short-term memory (LSTM) nets for sequence processing, encoder/decoders and Transformers for natural language processing, autoencoders and generative adversarial networks (GANs) for generative learning
- Techniques for training deep neural nets
- How to build an agent (e.g., a bot in a game) that can learn good strategies through trial and error, using Reinforcement Learning
- Loading and preprocessing large amounts of data efficiently
- Training and deploying TensorFlow models at scale

The first part is based mostly on Scikit-Learn, while the second part uses TensorFlow and Keras.

Don't jump into deep waters too hastily: while Deep Learning is no doubt one of the most exciting areas in Machine Learning, you should master the fundamentals first. Moreover, most problems can be solved quite well using simpler techniques such as Random Forests and Ensemble methods (discussed in Part I). Deep Learning is best suited for complex problems such as image recognition, speech recognition, or natural language processing, provided you have enough data, computing power, and patience.

Changes in the Second Edition

This second edition has six main objectives:

1. Cover additional ML topics: more unsupervised learning techniques (including clustering, anomaly detection, density estimation, and mixture models); more techniques for training deep nets (including self-normalized networks); additional computer vision techniques (including Xception, SENet, object detection with YOLO, and semantic segmentation using R-CNN); handling sequences using covolutional neural networks (CNNs, including WaveNet); natural language processing using recurrent neural networks (RNNs), CNNs, and Transformers; and GANs.
2. Cover additional libraries and APIs (Keras, the Data API, TF-Agents for Reinforcement Learning) and training and deploying TF models at scale using the Distribution Strategies API, TF-Serving, and Google Cloud AI Platform. Also briefly introduce TF Transform, TFLite, TF Addons/Seq2Seq, and TensorFlow.js.
3. Discuss some of the latest important results from Deep Learning research.
4. Migrate all TensorFlow chapters to TensorFlow 2, and use TensorFlow's implementation of the Keras API (tf.keras) whenever possible.
5. Update the code examples to use the latest versions of Scikit-Learn, NumPy, pandas, Matplotlib, and other libraries.
6. Clarify some sections and fix some errors, thanks to plenty of great feedback from readers.

Some chapters were added, others were rewritten, and a few were reordered. See *https://homl.info/changes2* for more details on what changed in the second edition.

Other Resources

Many excellent resources are available to learn about Machine Learning. For example, Andrew Ng's ML course on Coursera is amazing, although it requires a significant time investment (think months).

There are also many interesting websites about Machine Learning, including of course Scikit-Learn's exceptional User Guide. You may also enjoy Dataquest, which provides very nice interactive tutorials, and ML blogs such as those listed on Quora. Finally, the Deep Learning website has a good list of resources to check out to learn more.

There are many other introductory books about Machine Learning. In particular:

- Joel Grus's *Data Science from Scratch* (O'Reilly) presents the fundamentals of Machine Learning and implements some of the main algorithms in pure Python (from scratch, as the name suggests).
- Stephen Marsland's *Machine Learning: An Algorithmic Perspective* (Chapman & Hall) is a great introduction to Machine Learning, covering a wide range of topics in depth with code examples in Python (also from scratch, but using NumPy).
- Sebastian Raschka's *Python Machine Learning* (Packt Publishing) is also a great introduction to Machine Learning and leverages Python open source libraries (Pylearn 2 and Theano).
- François Chollet's *Deep Learning with Python* (Manning) is a very practical book that covers a large range of topics in a clear and concise way, as you might expect from the author of the excellent Keras library. It favors code examples over mathematical theory.
- Andriy Burkov's *The Hundred-Page Machine Learning Book* is very short and covers an impressive range of topics, introducing them in approachable terms without shying away from the math equations.
- Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin's *Learning from Data* (AMLBook) is a rather theoretical approach to ML that provides deep insights, in particular on the bias/variance trade-off (see Chapter 4).
- Stuart Russell and Peter Norvig's *Artificial Intelligence: A Modern Approach*, 3rd Edition (Pearson), is a great (and huge) book covering an incredible amount of topics, including Machine Learning. It helps put ML into perspective.

Finally, joining ML competition websites such as Kaggle.com will allow you to practice your skills on real-world problems, with help and insights from some of the best ML professionals out there.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Code Examples

There is a series of Jupyter notebooks full of supplemental material, such as code examples and exercises, available for download at *https://github.com/ageron/handson-ml2*.

Some of the code examples in the book leave out repetitive sections or details that are obvious or unrelated to Machine Learning. This keeps the focus on the important parts of the code and saves space to cover more topics. If you want the full code examples, they are all available in the Jupyter notebooks.

Note that when the code examples display some outputs, these code examples are shown with Python prompts (`>>>` and `...`), as in a Python shell, to clearly distinguish the code from the outputs. For example, this code defines the `square()` function, then it computes and displays the square of 3:

```
>>> def square(x):
...     return x ** 2
...
>>> result = square(3)
>>> result
9
```

When code does not display anything, prompts are not used. However, the result may sometimes be shown as a comment, like this:

```
def square(x):
    return x ** 2

result = square(3)  # result is 9
```

Using Code Examples

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd Edition, by Aurélien Géron (O'Reilly). Copyright 2019 Kiwisoft S.A.S., 978-1-492-03264-9." If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For almost 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://homl.info/oreilly2*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Never in my wildest dreams did I imagine that the first edition of this book would get such a large audience. I received so many messages from readers, many asking questions, some kindly pointing out errata, and most sending me encouraging words. I cannot express how grateful I am to all these readers for their tremendous support. Thank you all so very much! Please do not hesitate to file issues on GitHub if you find errors in the code examples (or just to ask questions), or to submit errata if you find errors in the text. Some readers also shared how this book helped them get their first job, or how it helped them solve a concrete problem they were working on. I find such feedback incredibly motivating. If you find this book helpful, I would love it if you could share your story with me, either privately (e.g., via LinkedIn) or publicly (e.g., in a tweet or through an Amazon review).

I am also incredibly thankful to all the amazing people who took time out of their busy lives to review my book with such care. In particular, I would like to thank François Chollet for reviewing all the chapters based on Keras and TensorFlow and giving me some great in-depth feedback. Since Keras is one of the main additions to this second edition, having its author review the book was invaluable. I highly recommend

François's book *Deep Learning with Python* (Manning): it has the conciseness, clarity, and depth of the Keras library itself. Special thanks as well to Ankur Patel, who reviewed every chapter of this second edition and gave me excellent feedback, in particular on Chapter 9, which covers unsupervised learning techniques. He could write a whole book on the topic… oh, wait, he did! Do check out *Hands-On Unsupervised Learning Using Python: How to Build Applied Machine Learning Solutions from Unlabeled Data* (O'Reilly). Huge thanks as well to Olzhas Akpambetov, who reviewed all the chapters in the second part of the book, tested much of the code, and offered many great suggestions. I'm grateful to Mark Daoust, Jon Krohn, Dominic Monn, and Josh Patterson for reviewing the second part of this book so thoroughly and offering their expertise. They left no stone unturned and provided amazingly useful feedback.

While writing this second edition, I was fortunate enough to get plenty of help from members of the TensorFlow team—in particular Martin Wicke, who tirelessly answered dozens of my questions and dispatched the rest to the right people, including Karmel Allison, Paige Bailey, Eugene Brevdo, William Chargin, Daniel "Wolff" Dobson, Nick Felt, Bruce Fontaine, Goldie Gadde, Sandeep Gupta, Priya Gupta, Kevin Haas, Konstantinos Katsiapis ,Viacheslav Kovalevskyi, Allen Lavoie, Clemens Mewald, Dan Moldovan, Sean Morgan, Tom O'Malley, Alexandre Passos, André Susano Pinto, Anthony Platanios, Oscar Ramirez, Anna Revinskaya, Saurabh Saxena, Ryan Sepassi, Jiri Simsa, Xiaodan Song, Christina Sorokin, Dustin Tran, Todd Wang, Pete Warden (who also reviewed the first edition) Edd Wilder-James, and Yuefeng Zhou, all of whom were tremendously helpful. Huge thanks to all of you, and to all other members of the TensorFlow team, not just for your help, but also for making such a great library! Special thanks to Irene Giannoumis and Robert Crowe of the TFX team for reviewing Chapters 13 and 19 in depth.

Many thanks as well to O'Reilly's fantastic staff, in particular Nicole Taché, who gave me insightful feedback and was always cheerful, encouraging, and helpful: I could not dream of a better editor. Big thanks to Michele Cronin as well, who was very helpful (and patient) at the start of this second edition, and to Kristen Brown, the production editor for the second edition, who saw it through all the steps (she also coordinated fixes and updates for each reprint of the first edition). Thanks as well to Rachel Monaghan and Amanda Kersey for their thorough copyediting (respectively for the first and second edition), and to Johnny O'Toole who managed the relationship with Amazon and answered many of my questions. Thanks to Marie Beaugureau, Ben Lorica, Mike Loukides, and Laurel Ruma for believing in this project and helping me define its scope. Thanks to Matt Hacker and all of the Atlas team for answering all my technical questions regarding formatting, AsciiDoc, and LaTeX, and thanks to Nick Adams, Rebecca Demarest, Rachel Head, Judith McConville, Helen Monroe, Karen Montgomery, Rachel Roumeliotis, and everyone else at O'Reilly who contributed to this book.

I would also like to thank my former Google colleagues, in particular the YouTube video classification team, for teaching me so much about Machine Learning. I could never have started the first edition without them. Special thanks to my personal ML gurus: Clément Courbet, Julien Dubois, Mathias Kende, Daniel Kitachewsky, James Pack, Alexander Pak, Anosh Raj, Vitor Sessak, Wiktor Tomczak, Ingrid von Glehn, and Rich Washington. And thanks to everyone else I worked with at YouTube and in the amazing Google research teams in Mountain View. Many thanks as well to Martin Andrews, Sam Witteveen, and Jason Zaman for welcoming me into their Google Developer Experts group in Singapore, with the kind support of Soonson Kwon, and for all the great discussions we had about Deep Learning and TensorFlow. Anyone interested in Deep Learning in Singapore should definitely join their Deep Learning Singapore meetup. Jason deserves special thanks for sharing some of his TFLite expertise for Chapter 19!

I will never forget the kind people who reviewed the first edition of this book, including David Andrzejewski, Lukas Biewald, Justin Francis, Vincent Guilbeau, Eddy Hung, Karim Matrah, Grégoire Mesnil, Salim Sémaoune, Iain Smears, Michel Tessier, Ingrid von Glehn, Pete Warden, and of course my dear brother Sylvain. Special thanks to Haesun Park, who gave me plenty of excellent feedback and caught several errors while he was writing the Korean translation of the first edition of this book. He also translated the Jupyter notebooks into Korean, not to mention TensorFlow's documentation. I do not speak Korean, but judging by the quality of his feedback, all his translations must be truly excellent! Haesun also kindly contributed some of the solutions to the exercises in this second edition.

Last but not least, I am infinitely grateful to my beloved wife, Emmanuelle, and to our three wonderful children, Alexandre, Rémi, and Gabrielle, for encouraging me to work hard on this book. I'm also thankful to them for their insatiable curiosity: explaining some of the most difficult concepts in this book to my wife and children helped me clarify my thoughts and directly improved many parts of it. And they keep bringing me cookies and coffee! What more can one dream of?

PART I

The Fundamentals of Machine Learning

CHAPTER 1
The Machine Learning Landscape

When most people hear "Machine Learning," they picture a robot: a dependable butler or a deadly Terminator, depending on who you ask. But Machine Learning is not just a futuristic fantasy; it's already here. In fact, it has been around for decades in some specialized applications, such as Optical Character Recognition (OCR). But the first ML application that really became mainstream, improving the lives of hundreds of millions of people, took over the world back in the 1990s: the *spam filter*. It's not exactly a self-aware Skynet, but it does technically qualify as Machine Learning (it has actually learned so well that you seldom need to flag an email as spam anymore). It was followed by hundreds of ML applications that now quietly power hundreds of products and features that you use regularly, from better recommendations to voice search.

Where does Machine Learning start and where does it end? What exactly does it mean for a machine to *learn* something? If I download a copy of Wikipedia, has my computer really learned something? Is it suddenly smarter? In this chapter we will start by clarifying what Machine Learning is and why you may want to use it.

Then, before we set out to explore the Machine Learning continent, we will take a look at the map and learn about the main regions and the most notable landmarks: supervised versus unsupervised learning, online versus batch learning, instance-based versus model-based learning. Then we will look at the workflow of a typical ML project, discuss the main challenges you may face, and cover how to evaluate and fine-tune a Machine Learning system.

This chapter introduces a lot of fundamental concepts (and jargon) that every data scientist should know by heart. It will be a high-level overview (it's the only chapter without much code), all rather simple, but you should make sure everything is crystal clear to you before continuing on to the rest of the book. So grab a coffee and let's get started!

If you already know all the Machine Learning basics, you may want to skip directly to Chapter 2. If you are not sure, try to answer all the questions listed at the end of the chapter before moving on.

What Is Machine Learning?

Machine Learning is the science (and art) of programming computers so they can *learn from data*.

Here is a slightly more general definition:

> [Machine Learning is the] field of study that gives computers the ability to learn without being explicitly programmed.
>
> —Arthur Samuel, 1959

And a more engineering-oriented one:

> A computer program is said to learn from experience *E* with respect to some task *T* and some performance measure *P*, if its performance on *T*, as measured by *P*, improves with experience *E*.
>
> —Tom Mitchell, 1997

Your spam filter is a Machine Learning program that, given examples of spam emails (e.g., flagged by users) and examples of regular (nonspam, also called "ham") emails, can learn to flag spam. The examples that the system uses to learn are called the *training set*. Each training example is called a *training instance* (or *sample*). In this case, the task *T* is to flag spam for new emails, the experience *E* is the *training data*, and the performance measure *P* needs to be defined; for example, you can use the ratio of correctly classified emails. This particular performance measure is called *accuracy*, and it is often used in classification tasks.

If you just download a copy of Wikipedia, your computer has a lot more data, but it is not suddenly better at any task. Thus, downloading a copy of Wikipedia is not Machine Learning.

Why Use Machine Learning?

Consider how you would write a spam filter using traditional programming techniques (Figure 1-1):

1. First you would consider what spam typically looks like. You might notice that some words or phrases (such as "4U," "credit card," "free," and "amazing") tend to come up a lot in the subject line. Perhaps you would also notice a few other patterns in the sender's name, the email's body, and other parts of the email.

2. You would write a detection algorithm for each of the patterns that you noticed, and your program would flag emails as spam if a number of these patterns were detected.
3. You would test your program and repeat steps 1 and 2 until it was good enough to launch.

Figure 1-1. The traditional approach

Since the problem is difficult, your program will likely become a long list of complex rules—pretty hard to maintain.

In contrast, a spam filter based on Machine Learning techniques automatically learns which words and phrases are good predictors of spam by detecting unusually frequent patterns of words in the spam examples compared to the ham examples (Figure 1-2). The program is much shorter, easier to maintain, and most likely more accurate.

What if spammers notice that all their emails containing "4U" are blocked? They might start writing "For U" instead. A spam filter using traditional programming techniques would need to be updated to flag "For U" emails. If spammers keep working around your spam filter, you will need to keep writing new rules forever.

In contrast, a spam filter based on Machine Learning techniques automatically notices that "For U" has become unusually frequent in spam flagged by users, and it starts flagging them without your intervention (Figure 1-3).

Figure 1-2. The Machine Learning approach

Figure 1-3. Automatically adapting to change

Another area where Machine Learning shines is for problems that either are too complex for traditional approaches or have no known algorithm. For example, consider speech recognition. Say you want to start simple and write a program capable of distinguishing the words "one" and "two." You might notice that the word "two" starts with a high-pitch sound ("T"), so you could hardcode an algorithm that measures high-pitch sound intensity and use that to distinguish ones and twos—but obviously this technique will not scale to thousands of words spoken by millions of very different people in noisy environments and in dozens of languages. The best solution (at least today) is to write an algorithm that learns by itself, given many example recordings for each word.

Finally, Machine Learning can help humans learn (Figure 1-4). ML algorithms can be inspected to see what they have learned (although for some algorithms this can be tricky). For instance, once a spam filter has been trained on enough spam, it can easily be inspected to reveal the list of words and combinations of words that it believes are the best predictors of spam. Sometimes this will reveal unsuspected

correlations or new trends, and thereby lead to a better understanding of the problem. Applying ML techniques to dig into large amounts of data can help discover patterns that were not immediately apparent. This is called *data mining*.

Figure 1-4. Machine Learning can help humans learn

To summarize, Machine Learning is great for:

- Problems for which existing solutions require a lot of fine-tuning or long lists of rules: one Machine Learning algorithm can often simplify code and perform better than the traditional approach.
- Complex problems for which using a traditional approach yields no good solution: the best Machine Learning techniques can perhaps find a solution.
- Fluctuating environments: a Machine Learning system can adapt to new data.
- Getting insights about complex problems and large amounts of data.

Examples of Applications

Let's look at some concrete examples of Machine Learning tasks, along with the techniques that can tackle them:

Analyzing images of products on a production line to automatically classify them
: This is image classification, typically performed using convolutional neural networks (CNNs; see Chapter 14).

Detecting tumors in brain scans
: This is semantic segmentation, where each pixel in the image is classified (as we want to determine the exact location and shape of tumors), typically using CNNs as well.

Automatically classifying news articles
: This is natural language processing (NLP), and more specifically text classification, which can be tackled using recurrent neural networks (RNNs), CNNs, or Transformers (see Chapter 16).

Automatically flagging offensive comments on discussion forums
: This is also text classification, using the same NLP tools.

Summarizing long documents automatically
: This is a branch of NLP called text summarization, again using the same tools.

Creating a chatbot or a personal assistant
: This involves many NLP components, including natural language understanding (NLU) and question-answering modules.

Forecasting your company's revenue next year, based on many performance metrics
: This is a regression task (i.e., predicting values) that may be tackled using any regression model, such as a Linear Regression or Polynomial Regression model (see Chapter 4), a regression SVM (see Chapter 5), a regression Random Forest (see Chapter 7), or an artificial neural network (see Chapter 10). If you want to take into account sequences of past performance metrics, you may want to use RNNs, CNNs, or Transformers (see Chapters 15 and 16).

Making your app react to voice commands
: This is speech recognition, which requires processing audio samples: since they are long and complex sequences, they are typically processed using RNNs, CNNs, or Transformers (see Chapters 15 and 16).

Detecting credit card fraud
: This is anomaly detection (see Chapter 9).

Segmenting clients based on their purchases so that you can design a different marketing strategy for each segment
: This is clustering (see Chapter 9).

Representing a complex, high-dimensional dataset in a clear and insightful diagram
: This is data visualization, often involving dimensionality reduction techniques (see Chapter 8).

Recommending a product that a client may be interested in, based on past purchases
: This is a recommender system. One approach is to feed past purchases (and other information about the client) to an artificial neural network (see Chap-

ter 10), and get it to output the most likely next purchase. This neural net would typically be trained on past sequences of purchases across all clients.

Building an intelligent bot for a game
: This is often tackled using Reinforcement Learning (RL; see Chapter 18), which is a branch of Machine Learning that trains agents (such as bots) to pick the actions that will maximize their rewards over time (e.g., a bot may get a reward every time the player loses some life points), within a given environment (such as the game). The famous AlphaGo program that beat the world champion at the game of Go was built using RL.

This list could go on and on, but hopefully it gives you a sense of the incredible breadth and complexity of the tasks that Machine Learning can tackle, and the types of techniques that you would use for each task.

Types of Machine Learning Systems

There are so many different types of Machine Learning systems that it is useful to classify them in broad categories, based on the following criteria:

- Whether or not they are trained with human supervision (supervised, unsupervised, semisupervised, and Reinforcement Learning)
- Whether or not they can learn incrementally on the fly (online versus batch learning)
- Whether they work by simply comparing new data points to known data points, or instead by detecting patterns in the training data and building a predictive model, much like scientists do (instance-based versus model-based learning)

These criteria are not exclusive; you can combine them in any way you like. For example, a state-of-the-art spam filter may learn on the fly using a deep neural network model trained using examples of spam and ham; this makes it an online, model-based, supervised learning system.

Let's look at each of these criteria a bit more closely.

Supervised/Unsupervised Learning

Machine Learning systems can be classified according to the amount and type of supervision they get during training. There are four major categories: supervised learning, unsupervised learning, semisupervised learning, and Reinforcement Learning.

Supervised learning

In *supervised learning*, the training set you feed to the algorithm includes the desired solutions, called *labels* (Figure 1-5).

Figure 1-5. A labeled training set for spam classification (an example of supervised learning)

A typical supervised learning task is *classification*. The spam filter is a good example of this: it is trained with many example emails along with their *class* (spam or ham), and it must learn how to classify new emails.

Another typical task is to predict a *target* numeric value, such as the price of a car, given a set of *features* (mileage, age, brand, etc.) called *predictors*. This sort of task is called *regression* (Figure 1-6).[1] To train the system, you need to give it many examples of cars, including both their predictors and their labels (i.e., their prices).

In Machine Learning an *attribute* is a data type (e.g., "mileage"), while a *feature* has several meanings, depending on the context, but generally means an attribute plus its value (e.g., "mileage = 15,000"). Many people use the words *attribute* and *feature* interchangeably.

Note that some regression algorithms can be used for classification as well, and vice versa. For example, *Logistic Regression* is commonly used for classification, as it can output a value that corresponds to the probability of belonging to a given class (e.g., 20% chance of being spam).

1 Fun fact: this odd-sounding name is a statistics term introduced by Francis Galton while he was studying the fact that the children of tall people tend to be shorter than their parents. Since the children were shorter, he called this *regression to the mean*. This name was then applied to the methods he used to analyze correlations between variables.

Figure 1-6. A regression problem: predict a value, given an input feature (there are usually multiple input features, and sometimes multiple output values)

Here are some of the most important supervised learning algorithms (covered in this book):

- k-Nearest Neighbors
- Linear Regression
- Logistic Regression
- Support Vector Machines (SVMs)
- Decision Trees and Random Forests
- Neural networks[2]

Unsupervised learning

In *unsupervised learning*, as you might guess, the training data is unlabeled (Figure 1-7). The system tries to learn without a teacher.

2 Some neural network architectures can be unsupervised, such as autoencoders and restricted Boltzmann machines. They can also be semisupervised, such as in deep belief networks and unsupervised pretraining.

Figure 1-7. An unlabeled training set for unsupervised learning

Here are some of the most important unsupervised learning algorithms (most of these are covered in Chapters 8 and 9):

- Clustering
  - K-Means
  - DBSCAN
  - Hierarchical Cluster Analysis (HCA)
- Anomaly detection and novelty detection
  - One-class SVM
  - Isolation Forest
- Visualization and dimensionality reduction
  - Principal Component Analysis (PCA)
  - Kernel PCA
  - Locally Linear Embedding (LLE)
  - t-Distributed Stochastic Neighbor Embedding (t-SNE)
- Association rule learning
  - Apriori
  - Eclat

For example, say you have a lot of data about your blog's visitors. You may want to run a *clustering* algorithm to try to detect groups of similar visitors (Figure 1-8). At no point do you tell the algorithm which group a visitor belongs to: it finds those connections without your help. For example, it might notice that 40% of your visitors are males who love comic books and generally read your blog in the evening, while 20% are young sci-fi lovers who visit during the weekends. If you use a *hierarchical clustering* algorithm, it may also subdivide each group into smaller groups. This may help you target your posts for each group.

Figure 1-8. Clustering

Visualization algorithms are also good examples of unsupervised learning algorithms: you feed them a lot of complex and unlabeled data, and they output a 2D or 3D representation of your data that can easily be plotted (Figure 1-9). These algorithms try to preserve as much structure as they can (e.g., trying to keep separate clusters in the input space from overlapping in the visualization) so that you can understand how the data is organized and perhaps identify unsuspected patterns.

Figure 1-9. Example of a t-SNE visualization highlighting semantic clusters[3]

3 Notice how animals are rather well separated from vehicles and how horses are close to deer but far from birds. Figure reproduced with permission from Richard Socher et al., "Zero-Shot Learning Through Cross-Modal Transfer," *Proceedings of the 26th International Conference on Neural Information Processing Systems* 1 (2013): 935–943.

A related task is *dimensionality reduction*, in which the goal is to simplify the data without losing too much information. One way to do this is to merge several correlated features into one. For example, a car's mileage may be strongly correlated with its age, so the dimensionality reduction algorithm will merge them into one feature that represents the car's wear and tear. This is called *feature extraction*.

It is often a good idea to try to reduce the dimension of your training data using a dimensionality reduction algorithm before you feed it to another Machine Learning algorithm (such as a supervised learning algorithm). It will run much faster, the data will take up less disk and memory space, and in some cases it may also perform better.

Yet another important unsupervised task is *anomaly detection*—for example, detecting unusual credit card transactions to prevent fraud, catching manufacturing defects, or automatically removing outliers from a dataset before feeding it to another learning algorithm. The system is shown mostly normal instances during training, so it learns to recognize them; then, when it sees a new instance, it can tell whether it looks like a normal one or whether it is likely an anomaly (see Figure 1-10). A very similar task is *novelty detection*: it aims to detect new instances that look different from all instances in the training set. This requires having a very "clean" training set, devoid of any instance that you would like the algorithm to detect. For example, if you have thousands of pictures of dogs, and 1% of these pictures represent Chihuahuas, then a novelty detection algorithm should not treat new pictures of Chihuahuas as novelties. On the other hand, anomaly detection algorithms may consider these dogs as so rare and so different from other dogs that they would likely classify them as anomalies (no offense to Chihuahuas).

Figure 1-10. Anomaly detection

Finally, another common unsupervised task is *association rule learning*, in which the goal is to dig into large amounts of data and discover interesting relations between

attributes. For example, suppose you own a supermarket. Running an association rule on your sales logs may reveal that people who purchase barbecue sauce and potato chips also tend to buy steak. Thus, you may want to place these items close to one another.

Semisupervised learning

Since labeling data is usually time-consuming and costly, you will often have plenty of unlabeled instances, and few labeled instances. Some algorithms can deal with data that's partially labeled. This is called *semisupervised learning* (Figure 1-11).

Figure 1-11. Semisupervised learning with two classes (triangles and squares): the unlabeled examples (circles) help classify a new instance (the cross) into the triangle class rather than the square class, even though it is closer to the labeled squares

Some photo-hosting services, such as Google Photos, are good examples of this. Once you upload all your family photos to the service, it automatically recognizes that the same person A shows up in photos 1, 5, and 11, while another person B shows up in photos 2, 5, and 7. This is the unsupervised part of the algorithm (clustering). Now all the system needs is for you to tell it who these people are. Just add one label per person[4] and it is able to name everyone in every photo, which is useful for searching photos.

Most semisupervised learning algorithms are combinations of unsupervised and supervised algorithms. For example, *deep belief networks* (DBNs) are based on unsupervised components called *restricted Boltzmann machines* (RBMs) stacked on top of one another. RBMs are trained sequentially in an unsupervised manner, and then the whole system is fine-tuned using supervised learning techniques.

4 That's when the system works perfectly. In practice it often creates a few clusters per person, and sometimes mixes up two people who look alike, so you may need to provide a few labels per person and manually clean up some clusters.

Reinforcement Learning

Reinforcement Learning is a very different beast. The learning system, called an *agent* in this context, can observe the environment, select and perform actions, and get *rewards* in return (or *penalties* in the form of negative rewards, as shown in Figure 1-12). It must then learn by itself what is the best strategy, called a *policy*, to get the most reward over time. A policy defines what action the agent should choose when it is in a given situation.

Figure 1-12. Reinforcement Learning

For example, many robots implement Reinforcement Learning algorithms to learn how to walk. DeepMind's AlphaGo program is also a good example of Reinforcement Learning: it made the headlines in May 2017 when it beat the world champion Ke Jie at the game of Go. It learned its winning policy by analyzing millions of games, and then playing many games against itself. Note that learning was turned off during the games against the champion; AlphaGo was just applying the policy it had learned.

Batch and Online Learning

Another criterion used to classify Machine Learning systems is whether or not the system can learn incrementally from a stream of incoming data.

Batch learning

In *batch learning*, the system is incapable of learning incrementally: it must be trained using all the available data. This will generally take a lot of time and computing resources, so it is typically done offline. First the system is trained, and then it is launched into production and runs without learning anymore; it just applies what it has learned. This is called *offline learning*.

If you want a batch learning system to know about new data (such as a new type of spam), you need to train a new version of the system from scratch on the full dataset (not just the new data, but also the old data), then stop the old system and replace it with the new one.

Fortunately, the whole process of training, evaluating, and launching a Machine Learning system can be automated fairly easily (as shown in Figure 1-3), so even a batch learning system can adapt to change. Simply update the data and train a new version of the system from scratch as often as needed.

This solution is simple and often works fine, but training using the full set of data can take many hours, so you would typically train a new system only every 24 hours or even just weekly. If your system needs to adapt to rapidly changing data (e.g., to predict stock prices), then you need a more reactive solution.

Also, training on the full set of data requires a lot of computing resources (CPU, memory space, disk space, disk I/O, network I/O, etc.). If you have a lot of data and you automate your system to train from scratch every day, it will end up costing you a lot of money. If the amount of data is huge, it may even be impossible to use a batch learning algorithm.

Finally, if your system needs to be able to learn autonomously and it has limited resources (e.g., a smartphone application or a rover on Mars), then carrying around large amounts of training data and taking up a lot of resources to train for hours every day is a showstopper.

Fortunately, a better option in all these cases is to use algorithms that are capable of learning incrementally.

Online learning

In *online learning*, you train the system incrementally by feeding it data instances sequentially, either individually or in small groups called *mini-batches*. Each learning step is fast and cheap, so the system can learn about new data on the fly, as it arrives (see Figure 1-13).

Figure 1-13. In online learning, a model is trained and launched into production, and then it keeps learning as new data comes in

Online learning is great for systems that receive data as a continuous flow (e.g., stock prices) and need to adapt to change rapidly or autonomously. It is also a good option if you have limited computing resources: once an online learning system has learned about new data instances, it does not need them anymore, so you can discard them (unless you want to be able to roll back to a previous state and "replay" the data). This can save a huge amount of space.

Online learning algorithms can also be used to train systems on huge datasets that cannot fit in one machine's main memory (this is called *out-of-core* learning). The algorithm loads part of the data, runs a training step on that data, and repeats the process until it has run on all of the data (see Figure 1-14).

Out-of-core learning is usually done offline (i.e., not on the live system), so *online learning* can be a confusing name. Think of it as *incremental learning*.

One important parameter of online learning systems is how fast they should adapt to changing data: this is called the *learning rate*. If you set a high learning rate, then your system will rapidly adapt to new data, but it will also tend to quickly forget the old data (you don't want a spam filter to flag only the latest kinds of spam it was shown). Conversely, if you set a low learning rate, the system will have more inertia; that is, it will learn more slowly, but it will also be less sensitive to noise in the new data or to sequences of nonrepresentative data points (outliers).

Figure 1-14. Using online learning to handle huge datasets

A big challenge with online learning is that if bad data is fed to the system, the system's performance will gradually decline. If it's a live system, your clients will notice. For example, bad data could come from a malfunctioning sensor on a robot, or from someone spamming a search engine to try to rank high in search results. To reduce this risk, you need to monitor your system closely and promptly switch learning off (and possibly revert to a previously working state) if you detect a drop in performance. You may also want to monitor the input data and react to abnormal data (e.g., using an anomaly detection algorithm).

Instance-Based Versus Model-Based Learning

One more way to categorize Machine Learning systems is by how they *generalize*. Most Machine Learning tasks are about making predictions. This means that given a number of training examples, the system needs to be able to make good predictions for (generalize to) examples it has never seen before. Having a good performance measure on the training data is good, but insufficient; the true goal is to perform well on new instances.

There are two main approaches to generalization: instance-based learning and model-based learning.

Instance-based learning

Possibly the most trivial form of learning is simply to learn by heart. If you were to create a spam filter this way, it would just flag all emails that are identical to emails that have already been flagged by users—not the worst solution, but certainly not the best.

Instead of just flagging emails that are identical to known spam emails, your spam filter could be programmed to also flag emails that are very similar to known spam emails. This requires a *measure of similarity* between two emails. A (very basic) similarity measure between two emails could be to count the number of words they have in common. The system would flag an email as spam if it has many words in common with a known spam email.

This is called *instance-based learning*: the system learns the examples by heart, then generalizes to new cases by using a similarity measure to compare them to the learned examples (or a subset of them). For example, in Figure 1-15 the new instance would be classified as a triangle because the majority of the most similar instances belong to that class.

Figure 1-15. Instance-based learning

Model-based learning

Another way to generalize from a set of examples is to build a model of these examples and then use that model to make *predictions*. This is called *model-based learning* (Figure 1-16).

Figure 1-16. Model-based learning

For example, suppose you want to know if money makes people happy, so you download the Better Life Index data from the OECD's website and stats about gross domestic product (GDP) per capita from the IMF's website. Then you join the tables and sort by GDP per capita. Table 1-1 shows an excerpt of what you get.

Table 1-1. Does money make people happier?

Country	GDP per capita (USD)	Life satisfaction
Hungary	12,240	4.9
Korea	27,195	5.8
France	37,675	6.5
Australia	50,962	7.3
United States	55,805	7.2

Let's plot the data for these countries (Figure 1-17).

Figure 1-17. Do you see a trend here?

There does seem to be a trend here! Although the data is *noisy* (i.e., partly random), it looks like life satisfaction goes up more or less linearly as the country's GDP per capita increases. So you decide to model life satisfaction as a linear function of GDP per capita. This step is called *model selection*: you selected a *linear model* of life satisfaction with just one attribute, GDP per capita (Equation 1-1).

Equation 1-1. A simple linear model

$$\text{life_satisfaction} = \theta_0 + \theta_1 \times \text{GDP_per_capita}$$

This model has two *model parameters*, θ_0 and θ_1.[5] By tweaking these parameters, you can make your model represent any linear function, as shown in Figure 1-18.

Figure 1-18. A few possible linear models

Before you can use your model, you need to define the parameter values θ_0 and θ_1. How can you know which values will make your model perform best? To answer this question, you need to specify a performance measure. You can either define a *utility function* (or *fitness function*) that measures how *good* your model is, or you can define a *cost function* that measures how *bad* it is. For Linear Regression problems, people typically use a cost function that measures the distance between the linear model's predictions and the training examples; the objective is to minimize this distance.

This is where the Linear Regression algorithm comes in: you feed it your training examples, and it finds the parameters that make the linear model fit best to your data. This is called *training* the model. In our case, the algorithm finds that the optimal parameter values are $\theta_0 = 4.85$ and $\theta_1 = 4.91 \times 10^{-5}$.

Confusingly, the same word "model" can refer to a *type of model* (e.g., Linear Regression), to a *fully specified model architecture* (e.g., Linear Regression with one input and one output), or to the *final trained model* ready to be used for predictions (e.g., Linear Regression with one input and one output, using $\theta_0 = 4.85$ and $\theta_1 = 4.91 \times 10^{-5}$). Model selection consists in choosing the type of model and fully specifying its architecture. Training a model means running an algorithm to find the model parameters that will make it best fit the training data (and hopefully make good predictions on new data).

5 By convention, the Greek letter θ (theta) is frequently used to represent model parameters.

Now the model fits the training data as closely as possible (for a linear model), as you can see in Figure 1-19.

Figure 1-19. The linear model that fits the training data best

You are finally ready to run the model to make predictions. For example, say you want to know how happy Cypriots are, and the OECD data does not have the answer. Fortunately, you can use your model to make a good prediction: you look up Cyprus's GDP per capita, find $22,587, and then apply your model and find that life satisfaction is likely to be somewhere around $4.85 + 22{,}587 \times 4.91 \times 10^{-5} = 5.96$.

To whet your appetite, Example 1-1 shows the Python code that loads the data, prepares it,[6] creates a scatterplot for visualization, and then trains a linear model and makes a prediction.[7]

Example 1-1. Training and running a linear model using Scikit-Learn

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.linear_model

# Load the data
oecd_bli = pd.read_csv("oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv("gdp_per_capita.csv",thousands=',',delimiter='\t',
                             encoding='latin1', na_values="n/a")
```

6 The `prepare_country_stats()` function's definition is not shown here (see this chapter's Jupyter notebook if you want all the gory details). It's just boring pandas code that joins the life satisfaction data from the OECD with the GDP per capita data from the IMF.

7 It's OK if you don't understand all the code yet; we will present Scikit-Learn in the following chapters.

```
# Prepare the data
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]

# Visualize the data
country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction')
plt.show()

# Select a linear model
model = sklearn.linear_model.LinearRegression()

# Train the model
model.fit(X, y)

# Make a prediction for Cyprus
X_new = [[22587]]  # Cyprus's GDP per capita
print(model.predict(X_new)) # outputs [[ 5.96242338]]
```

If you had used an instance-based learning algorithm instead, you would have found that Slovenia has the closest GDP per capita to that of Cyprus ($20,732), and since the OECD data tells us that Slovenians' life satisfaction is 5.7, you would have predicted a life satisfaction of 5.7 for Cyprus. If you zoom out a bit and look at the two next-closest countries, you will find Portugal and Spain with life satisfactions of 5.1 and 6.5, respectively. Averaging these three values, you get 5.77, which is pretty close to your model-based prediction. This simple algorithm is called *k-Nearest Neighbors* regression (in this example, $k = 3$).

Replacing the Linear Regression model with k-Nearest Neighbors regression in the previous code is as simple as replacing these two lines:

```
import sklearn.linear_model
model = sklearn.linear_model.LinearRegression()
```

with these two:

```
import sklearn.neighbors
model = sklearn.neighbors.KNeighborsRegressor(
    n_neighbors=3)
```

If all went well, your model will make good predictions. If not, you may need to use more attributes (employment rate, health, air pollution, etc.), get more or better-quality training data, or perhaps select a more powerful model (e.g., a Polynomial Regression model).

In summary:

- You studied the data.
- You selected a model.
- You trained it on the training data (i.e., the learning algorithm searched for the model parameter values that minimize a cost function).
- Finally, you applied the model to make predictions on new cases (this is called *inference*), hoping that this model will generalize well.

This is what a typical Machine Learning project looks like. In Chapter 2 you will experience this firsthand by going through a project end to end.

We have covered a lot of ground so far: you now know what Machine Learning is really about, why it is useful, what some of the most common categories of ML systems are, and what a typical project workflow looks like. Now let's look at what can go wrong in learning and prevent you from making accurate predictions.

Main Challenges of Machine Learning

In short, since your main task is to select a learning algorithm and train it on some data, the two things that can go wrong are "bad algorithm" and "bad data." Let's start with examples of bad data.

Insufficient Quantity of Training Data

For a toddler to learn what an apple is, all it takes is for you to point to an apple and say "apple" (possibly repeating this procedure a few times). Now the child is able to recognize apples in all sorts of colors and shapes. Genius.

Machine Learning is not quite there yet; it takes a lot of data for most Machine Learning algorithms to work properly. Even for very simple problems you typically need thousands of examples, and for complex problems such as image or speech recognition you may need millions of examples (unless you can reuse parts of an existing model).

The Unreasonable Effectiveness of Data

In a famous paper published in 2001, Microsoft researchers Michele Banko and Eric Brill showed that very different Machine Learning algorithms, including fairly simple ones, performed almost identically well on a complex problem of natural language disambiguation[8] once they were given enough data (as you can see in Figure 1-20).

Figure 1-20. The importance of data versus algorithms[9]

As the authors put it, "these results suggest that we may want to reconsider the trade-off between spending time and money on algorithm development versus spending it on corpus development."

The idea that data matters more than algorithms for complex problems was further popularized by Peter Norvig et al. in a paper titled "The Unreasonable Effectiveness of Data", published in 2009.[10] It should be noted, however, that small- and medium-sized datasets are still very common, and it is not always easy or cheap to get extra training data—so don't abandon algorithms just yet.

8 For example, knowing whether to write "to," "two," or "too," depending on the context.

9 Figure reproduced with permission from Michele Banko and Eric Brill, "Scaling to Very Very Large Corpora for Natural Language Disambiguation," *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics* (2001): 26–33.

10 Peter Norvig et al., "The Unreasonable Effectiveness of Data," *IEEE Intelligent Systems* 24, no. 2 (2009): 8–12.

Nonrepresentative Training Data

In order to generalize well, it is crucial that your training data be representative of the new cases you want to generalize to. This is true whether you use instance-based learning or model-based learning.

For example, the set of countries we used earlier for training the linear model was not perfectly representative; a few countries were missing. Figure 1-21 shows what the data looks like when you add the missing countries.

Figure 1-21. A more representative training sample

If you train a linear model on this data, you get the solid line, while the old model is represented by the dotted line. As you can see, not only does adding a few missing countries significantly alter the model, but it makes it clear that such a simple linear model is probably never going to work well. It seems that very rich countries are not happier than moderately rich countries (in fact, they seem unhappier), and conversely some poor countries seem happier than many rich countries.

By using a nonrepresentative training set, we trained a model that is unlikely to make accurate predictions, especially for very poor and very rich countries.

It is crucial to use a training set that is representative of the cases you want to generalize to. This is often harder than it sounds: if the sample is too small, you will have *sampling noise* (i.e., nonrepresentative data as a result of chance), but even very large samples can be nonrepresentative if the sampling method is flawed. This is called *sampling bias*.

Examples of Sampling Bias

Perhaps the most famous example of sampling bias happened during the US presidential election in 1936, which pitted Landon against Roosevelt: the *Literary Digest* conducted a very large poll, sending mail to about 10 million people. It got 2.4 million answers, and predicted with high confidence that Landon would get 57% of the votes. Instead, Roosevelt won with 62% of the votes. The flaw was in the *Literary Digest*'s sampling method:

- First, to obtain the addresses to send the polls to, the *Literary Digest* used telephone directories, lists of magazine subscribers, club membership lists, and the like. All of these lists tended to favor wealthier people, who were more likely to vote Republican (hence Landon).
- Second, less than 25% of the people who were polled answered. Again this introduced a sampling bias, by potentially ruling out people who didn't care much about politics, people who didn't like the *Literary Digest*, and other key groups. This is a special type of sampling bias called *nonresponse bias*.

Here is another example: say you want to build a system to recognize funk music videos. One way to build your training set is to search for "funk music" on YouTube and use the resulting videos. But this assumes that YouTube's search engine returns a set of videos that are representative of all the funk music videos on YouTube. In reality, the search results are likely to be biased toward popular artists (and if you live in Brazil you will get a lot of "funk carioca" videos, which sound nothing like James Brown). On the other hand, how else can you get a large training set?

Poor-Quality Data

Obviously, if your training data is full of errors, outliers, and noise (e.g., due to poor-quality measurements), it will make it harder for the system to detect the underlying patterns, so your system is less likely to perform well. It is often well worth the effort to spend time cleaning up your training data. The truth is, most data scientists spend a significant part of their time doing just that. The following are a couple of examples of when you'd want to clean up training data:

- If some instances are clearly outliers, it may help to simply discard them or try to fix the errors manually.
- If some instances are missing a few features (e.g., 5% of your customers did not specify their age), you must decide whether you want to ignore this attribute altogether, ignore these instances, fill in the missing values (e.g., with the median age), or train one model with the feature and one model without it.

Irrelevant Features

As the saying goes: garbage in, garbage out. Your system will only be capable of learning if the training data contains enough relevant features and not too many irrelevant ones. A critical part of the success of a Machine Learning project is coming up with a good set of features to train on. This process, called *feature engineering*, involves the following steps:

- *Feature selection* (selecting the most useful features to train on among existing features)
- *Feature extraction* (combining existing features to produce a more useful one—as we saw earlier, dimensionality reduction algorithms can help)
- Creating new features by gathering new data

Now that we have looked at many examples of bad data, let's look at a couple of examples of bad algorithms.

Overfitting the Training Data

Say you are visiting a foreign country and the taxi driver rips you off. You might be tempted to say that *all* taxi drivers in that country are thieves. Overgeneralizing is something that we humans do all too often, and unfortunately machines can fall into the same trap if we are not careful. In Machine Learning this is called *overfitting*: it means that the model performs well on the training data, but it does not generalize well.

Figure 1-22 shows an example of a high-degree polynomial life satisfaction model that strongly overfits the training data. Even though it performs much better on the training data than the simple linear model, would you really trust its predictions?

Figure 1-22. Overfitting the training data

Complex models such as deep neural networks can detect subtle patterns in the data, but if the training set is noisy, or if it is too small (which introduces sampling noise), then the model is likely to detect patterns in the noise itself. Obviously these patterns will not generalize to new instances. For example, say you feed your life satisfaction model many more attributes, including uninformative ones such as the country's name. In that case, a complex model may detect patterns like the fact that all countries in the training data with a *w* in their name have a life satisfaction greater than 7: New Zealand (7.3), Norway (7.4), Sweden (7.2), and Switzerland (7.5). How confident are you that the *w*-satisfaction rule generalizes to Rwanda or Zimbabwe? Obviously this pattern occurred in the training data by pure chance, but the model has no way to tell whether a pattern is real or simply the result of noise in the data.

Overfitting happens when the model is too complex relative to the amount and noisiness of the training data. Here are possible solutions:

- Simplify the model by selecting one with fewer parameters (e.g., a linear model rather than a high-degree polynomial model), by reducing the number of attributes in the training data, or by constraining the model.
- Gather more training data.
- Reduce the noise in the training data (e.g., fix data errors and remove outliers).

Constraining a model to make it simpler and reduce the risk of overfitting is called *regularization*. For example, the linear model we defined earlier has two parameters, θ_0 and θ_1. This gives the learning algorithm two *degrees of freedom* to adapt the model to the training data: it can tweak both the height (θ_0) and the slope (θ_1) of the line. If we forced $\theta_1 = 0$, the algorithm would have only one degree of freedom and would have a much harder time fitting the data properly: all it could do is move the line up or down to get as close as possible to the training instances, so it would end up around the mean. A very simple model indeed! If we allow the algorithm to modify θ_1 but we force it to keep it small, then the learning algorithm will effectively have somewhere in between one and two degrees of freedom. It will produce a model that's simpler than one with two degrees of freedom, but more complex than one with just one. You want to find the right balance between fitting the training data perfectly and keeping the model simple enough to ensure that it will generalize well.

Figure 1-23 shows three models. The dotted line represents the original model that was trained on the countries represented as circles (without the countries represented as squares), the dashed line is our second model trained with all countries (circles and squares), and the solid line is a model trained with the same data as the first model

but with a regularization constraint. You can see that regularization forced the model to have a smaller slope: this model does not fit the training data (circles) as well as the first model, but it actually generalizes better to new examples that it did not see during training (squares).

Figure 1-23. Regularization reduces the risk of overfitting

The amount of regularization to apply during learning can be controlled by a *hyperparameter*. A hyperparameter is a parameter of a learning algorithm (not of the model). As such, it is not affected by the learning algorithm itself; it must be set prior to training and remains constant during training. If you set the regularization hyperparameter to a very large value, you will get an almost flat model (a slope close to zero); the learning algorithm will almost certainly not overfit the training data, but it will be less likely to find a good solution. Tuning hyperparameters is an important part of building a Machine Learning system (you will see a detailed example in the next chapter).

Underfitting the Training Data

As you might guess, *underfitting* is the opposite of overfitting: it occurs when your model is too simple to learn the underlying structure of the data. For example, a linear model of life satisfaction is prone to underfit; reality is just more complex than the model, so its predictions are bound to be inaccurate, even on the training examples.

Here are the main options for fixing this problem:

- Select a more powerful model, with more parameters.
- Feed better features to the learning algorithm (feature engineering).
- Reduce the constraints on the model (e.g., reduce the regularization hyperparameter).

Stepping Back

By now you know a lot about Machine Learning. However, we went through so many concepts that you may be feeling a little lost, so let's step back and look at the big picture:

- Machine Learning is about making machines get better at some task by learning from data, instead of having to explicitly code rules.
- There are many different types of ML systems: supervised or not, batch or online, instance-based or model-based.
- In an ML project you gather data in a training set, and you feed the training set to a learning algorithm. If the algorithm is model-based, it tunes some parameters to fit the model to the training set (i.e., to make good predictions on the training set itself), and then hopefully it will be able to make good predictions on new cases as well. If the algorithm is instance-based, it just learns the examples by heart and generalizes to new instances by using a similarity measure to compare them to the learned instances.
- The system will not perform well if your training set is too small, or if the data is not representative, is noisy, or is polluted with irrelevant features (garbage in, garbage out). Lastly, your model needs to be neither too simple (in which case it will underfit) nor too complex (in which case it will overfit).

There's just one last important topic to cover: once you have trained a model, you don't want to just "hope" it generalizes to new cases. You want to evaluate it and fine-tune it if necessary. Let's see how to do that.

Testing and Validating

The only way to know how well a model will generalize to new cases is to actually try it out on new cases. One way to do that is to put your model in production and monitor how well it performs. This works well, but if your model is horribly bad, your users will complain—not the best idea.

A better option is to split your data into two sets: the *training set* and the *test set*. As these names imply, you train your model using the training set, and you test it using the test set. The error rate on new cases is called the *generalization error* (or *out-of-sample error*), and by evaluating your model on the test set, you get an estimate of this error. This value tells you how well your model will perform on instances it has never seen before.

If the training error is low (i.e., your model makes few mistakes on the training set) but the generalization error is high, it means that your model is overfitting the training data.

It is common to use 80% of the data for training and *hold out* 20% for testing. However, this depends on the size of the dataset: if it contains 10 million instances, then holding out 1% means your test set will contain 100,000 instances, probably more than enough to get a good estimate of the generalization error.

Hyperparameter Tuning and Model Selection

Evaluating a model is simple enough: just use a test set. But suppose you are hesitating between two types of models (say, a linear model and a polynomial model): how can you decide between them? One option is to train both and compare how well they generalize using the test set.

Now suppose that the linear model generalizes better, but you want to apply some regularization to avoid overfitting. The question is, how do you choose the value of the regularization hyperparameter? One option is to train 100 different models using 100 different values for this hyperparameter. Suppose you find the best hyperparameter value that produces a model with the lowest generalization error—say, just 5% error. You launch this model into production, but unfortunately it does not perform as well as expected and produces 15% errors. What just happened?

The problem is that you measured the generalization error multiple times on the test set, and you adapted the model and hyperparameters to produce the best model *for that particular set*. This means that the model is unlikely to perform as well on new data.

A common solution to this problem is called *holdout validation*: you simply hold out part of the training set to evaluate several candidate models and select the best one. The new held-out set is called the *validation set* (or sometimes the *development set*, or *dev set*). More specifically, you train multiple models with various hyperparameters on the reduced training set (i.e., the full training set minus the validation set), and you select the model that performs best on the validation set. After this holdout validation process, you train the best model on the full training set (including the validation set), and this gives you the final model. Lastly, you evaluate this final model on the test set to get an estimate of the generalization error.

This solution usually works quite well. However, if the validation set is too small, then model evaluations will be imprecise: you may end up selecting a suboptimal model by mistake. Conversely, if the validation set is too large, then the remaining training set will be much smaller than the full training set. Why is this bad? Well, since the final model will be trained on the full training set, it is not ideal to compare candidate models trained on a much smaller training set. It would be like selecting the fastest sprinter to participate in a marathon. One way to solve this problem is to perform repeated *cross-validation*, using many small validation sets. Each model is evaluated once per validation set after it is trained on the rest of the data. By averaging out all

the evaluations of a model, you get a much more accurate measure of its performance. There is a drawback, however: the training time is multiplied by the number of validation sets.

Data Mismatch

In some cases, it's easy to get a large amount of data for training, but this data probably won't be perfectly representative of the data that will be used in production. For example, suppose you want to create a mobile app to take pictures of flowers and automatically determine their species. You can easily download millions of pictures of flowers on the web, but they won't be perfectly representative of the pictures that will actually be taken using the app on a mobile device. Perhaps you only have 10,000 representative pictures (i.e., actually taken with the app). In this case, the most important rule to remember is that the validation set and the test set must be as representative as possible of the data you expect to use in production, so they should be composed exclusively of representative pictures: you can shuffle them and put half in the validation set and half in the test set (making sure that no duplicates or near-duplicates end up in both sets). But after training your model on the web pictures, if you observe that the performance of the model on the validation set is disappointing, you will not know whether this is because your model has overfit the training set, or whether this is just due to the mismatch between the web pictures and the mobile app pictures. One solution is to hold out some of the training pictures (from the web) in yet another set that Andrew Ng calls the *train-dev set*. After the model is trained (on the training set, *not* on the train-dev set), you can evaluate it on the train-dev set. If it performs well, then the model is not overfitting the training set. If it performs poorly on the validation set, the problem must be coming from the data mismatch. You can try to tackle this problem by preprocessing the web images to make them look more like the pictures that will be taken by the mobile app, and then retraining the model. Conversely, if the model performs poorly on the train-dev set, then it must have overfit the training set, so you should try to simplify or regularize the model, get more training data, and clean up the training data.

No Free Lunch Theorem

A model is a simplified version of the observations. The simplifications are meant to discard the superfluous details that are unlikely to generalize to new instances. To decide what data to discard and what data to keep, you must make *assumptions*. For example, a linear model makes the assumption that the data is fundamentally linear and that the distance between the instances and the straight line is just noise, which can safely be ignored.

In a famous 1996 paper,[11] David Wolpert demonstrated that if you make absolutely no assumption about the data, then there is no reason to prefer one model over any other. This is called the *No Free Lunch* (NFL) theorem. For some datasets the best model is a linear model, while for other datasets it is a neural network. There is no model that is *a priori* guaranteed to work better (hence the name of the theorem). The only way to know for sure which model is best is to evaluate them all. Since this is not possible, in practice you make some reasonable assumptions about the data and evaluate only a few reasonable models. For example, for simple tasks you may evaluate linear models with various levels of regularization, and for a complex problem you may evaluate various neural networks.

Exercises

In this chapter we have covered some of the most important concepts in Machine Learning. In the next chapters we will dive deeper and write more code, but before we do, make sure you know how to answer the following questions:

1. How would you define Machine Learning?
2. Can you name four types of problems where it shines?
3. What is a labeled training set?
4. What are the two most common supervised tasks?
5. Can you name four common unsupervised tasks?
6. What type of Machine Learning algorithm would you use to allow a robot to walk in various unknown terrains?
7. What type of algorithm would you use to segment your customers into multiple groups?
8. Would you frame the problem of spam detection as a supervised learning problem or an unsupervised learning problem?

11 David Wolpert, "The Lack of A Priori Distinctions Between Learning Algorithms," *Neural Computation* 8, no. 7 (1996): 1341–1390.

9. What is an online learning system?
10. What is out-of-core learning?
11. What type of learning algorithm relies on a similarity measure to make predictions?
12. What is the difference between a model parameter and a learning algorithm's hyperparameter?
13. What do model-based learning algorithms search for? What is the most common strategy they use to succeed? How do they make predictions?
14. Can you name four of the main challenges in Machine Learning?
15. If your model performs great on the training data but generalizes poorly to new instances, what is happening? Can you name three possible solutions?
16. What is a test set, and why would you want to use it?
17. What is the purpose of a validation set?
18. What is the train-dev set, when do you need it, and how do you use it?
19. What can go wrong if you tune hyperparameters using the test set?

Solutions to these exercises are available in Appendix A.

CHAPTER 2

End-to-End Machine Learning Project

In this chapter you will work through an example project end to end, pretending to be a recently hired data scientist at a real estate company.[1] Here are the main steps you will go through:

1. Look at the big picture.
2. Get the data.
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
5. Select a model and train it.
6. Fine-tune your model.
7. Present your solution.
8. Launch, monitor, and maintain your system.

Working with Real Data

When you are learning about Machine Learning, it is best to experiment with real-world data, not artificial datasets. Fortunately, there are thousands of open datasets to choose from, ranging across all sorts of domains. Here are a few places you can look to get data:

1 The example project is fictitious; the goal is to illustrate the main steps of a Machine Learning project, not to learn anything about the real estate business.

- Popular open data repositories
  - — UC Irvine Machine Learning Repository
  - — Kaggle datasets
  - — Amazon's AWS datasets
- Meta portals (they list open data repositories)
  - — Data Portals
  - — OpenDataMonitor
  - — Quandl
- Other pages listing many popular open data repositories
  - — Wikipedia's list of Machine Learning datasets
  - — Quora.com
  - — The datasets subreddit

In this chapter we'll use the California Housing Prices dataset from the StatLib repository[2] (see Figure 2-1). This dataset is based on data from the 1990 California census. It is not exactly recent (a nice house in the Bay Area was still affordable at the time), but it has many qualities for learning, so we will pretend it is recent data. For teaching purposes I've added a categorical attribute and removed a few features.

Figure 2-1. California housing prices

2 The original dataset appeared in R. Kelley Pace and Ronald Barry, "Sparse Spatial Autoregressions," *Statistics & Probability Letters* 33, no. 3 (1997): 291–297.

Look at the Big Picture

Welcome to the Machine Learning Housing Corporation! Your first task is to use California census data to build a model of housing prices in the state. This data includes metrics such as the population, median income, and median housing price for each block group in California. Block groups are the smallest geographical unit for which the US Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people). We will call them "districts" for short.

Your model should learn from this data and be able to predict the median housing price in any district, given all the other metrics.

Since you are a well-organized data scientist, the first thing you should do is pull out your Machine Learning project checklist. You can start with the one in Appendix B; it should work reasonably well for most Machine Learning projects, but make sure to adapt it to your needs. In this chapter we will go through many checklist items, but we will also skip a few, either because they are self-explanatory or because they will be discussed in later chapters.

Frame the Problem

The first question to ask your boss is what exactly the business objective is. Building a model is probably not the end goal. How does the company expect to use and benefit from this model? Knowing the objective is important because it will determine how you frame the problem, which algorithms you will select, which performance measure you will use to evaluate your model, and how much effort you will spend tweaking it.

Your boss answers that your model's output (a prediction of a district's median housing price) will be fed to another Machine Learning system (see Figure 2-2), along with many other signals.[3] This downstream system will determine whether it is worth investing in a given area or not. Getting this right is critical, as it directly affects revenue.

3 A piece of information fed to a Machine Learning system is often called a *signal*, in reference to Claude Shannon's information theory, which he developed at Bell Labs to improve telecommunications. His theory: you want a high signal-to-noise ratio.

Figure 2-2. A Machine Learning pipeline for real estate investments

Pipelines

A sequence of data processing components is called a data *pipeline*. Pipelines are very common in Machine Learning systems, since there is a lot of data to manipulate and many data transformations to apply.

Components typically run asynchronously. Each component pulls in a large amount of data, processes it, and spits out the result in another data store. Then, some time later, the next component in the pipeline pulls this data and spits out its own output. Each component is fairly self-contained: the interface between components is simply the data store. This makes the system simple to grasp (with the help of a data flow graph), and different teams can focus on different components. Moreover, if a component breaks down, the downstream components can often continue to run normally (at least for a while) by just using the last output from the broken component. This makes the architecture quite robust.

On the other hand, a broken component can go unnoticed for some time if proper monitoring is not implemented. The data gets stale and the overall system's performance drops.

The next question to ask your boss is what the current solution looks like (if any). The current situation will often give you a reference for performance, as well as insights on how to solve the problem. Your boss answers that the district housing prices are currently estimated manually by experts: a team gathers up-to-date information about a district, and when they cannot get the median housing price, they estimate it using complex rules.

This is costly and time-consuming, and their estimates are not great; in cases where they manage to find out the actual median housing price, they often realize that their estimates were off by more than 20%. This is why the company thinks that it would be useful to train a model to predict a district's median housing price, given other

data about that district. The census data looks like a great dataset to exploit for this purpose, since it includes the median housing prices of thousands of districts, as well as other data.

With all this information, you are now ready to start designing your system. First, you need to frame the problem: is it supervised, unsupervised, or Reinforcement Learning? Is it a classification task, a regression task, or something else? Should you use batch learning or online learning techniques? Before you read on, pause and try to answer these questions for yourself.

Have you found the answers? Let's see: it is clearly a typical supervised learning task, since you are given *labeled* training examples (each instance comes with the expected output, i.e., the district's median housing price). It is also a typical regression task, since you are asked to predict a value. More specifically, this is a *multiple regression* problem, since the system will use multiple features to make a prediction (it will use the district's population, the median income, etc.). It is also a *univariate regression* problem, since we are only trying to predict a single value for each district. If we were trying to predict multiple values per district, it would be a *multivariate regression* problem. Finally, there is no continuous flow of data coming into the system, there is no particular need to adjust to changing data rapidly, and the data is small enough to fit in memory, so plain batch learning should do just fine.

If the data were huge, you could either split your batch learning work across multiple servers (using the MapReduce technique) or use an online learning technique.

Select a Performance Measure

Your next step is to select a performance measure. A typical performance measure for regression problems is the Root Mean Square Error (RMSE). It gives an idea of how much error the system typically makes in its predictions, with a higher weight for large errors. Equation 2-1 shows the mathematical formula to compute the RMSE.

Equation 2-1. Root Mean Square Error (RMSE)

$$\mathrm{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2}$$

Notations

This equation introduces several very common Machine Learning notations that we will use throughout this book:

- m is the number of instances in the dataset you are measuring the RMSE on.
  - For example, if you are evaluating the RMSE on a validation set of 2,000 districts, then $m = 2{,}000$.
- $\mathbf{x}^{(i)}$ is a vector of all the feature values (excluding the label) of the ith instance in the dataset, and $y^{(i)}$ is its label (the desired output value for that instance).
  - For example, if the first district in the dataset is located at longitude −118.29°, latitude 33.91°, and it has 1,416 inhabitants with a median income of $38,372, and the median house value is $156,400 (ignoring the other features for now), then:

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1{,}416 \\ 38{,}372 \end{pmatrix}$$

    and:

$$y^{(1)} = 156{,}400$$

- $\mathbf{X}$ is a matrix containing all the feature values (excluding labels) of all instances in the dataset. There is one row per instance, and the ith row is equal to the transpose of $\mathbf{x}^{(i)}$, noted $(\mathbf{x}^{(i)})^{\mathsf{T}}$.[4]
  - For example, if the first district is as just described, then the matrix $\mathbf{X}$ looks like this:

$$\mathbf{X} = \begin{pmatrix} \left(\mathbf{x}^{(1)}\right)^{\mathsf{T}} \\ \left(\mathbf{x}^{(2)}\right)^{\mathsf{T}} \\ \vdots \\ \left(\mathbf{x}^{(1999)}\right)^{\mathsf{T}} \\ \left(\mathbf{x}^{(2000)}\right)^{\mathsf{T}} \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1{,}416 & 38{,}372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

4 Recall that the transpose operator flips a column vector into a row vector (and vice versa).

- h is your system's prediction function, also called a *hypothesis*. When your system is given an instance's feature vector $\mathbf{x}^{(i)}$, it outputs a predicted value $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$ for that instance ($\hat{y}$ is pronounced "y-hat").
  - For example, if your system predicts that the median housing price in the first district is $158,400, then $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158{,}400$. The prediction error for this district is $\hat{y}^{(1)} - y^{(1)} = 2{,}000$.
- RMSE($\mathbf{X}$,h) is the cost function measured on the set of examples using your hypothesis h.

We use lowercase italic font for scalar values (such as m or $y^{(i)}$) and function names (such as h), lowercase bold font for vectors (such as $\mathbf{x}^{(i)}$), and uppercase bold font for matrices (such as $\mathbf{X}$).

Even though the RMSE is generally the preferred performance measure for regression tasks, in some contexts you may prefer to use another function. For example, suppose that there are many outlier districts. In that case, you may consider using the *mean absolute error* (MAE, also called the average absolute deviation; see Equation 2-2):

Equation 2-2. Mean absolute error (MAE)

$$\mathrm{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\left|h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right|$$

Both the RMSE and the MAE are ways to measure the distance between two vectors: the vector of predictions and the vector of target values. Various distance measures, or *norms*, are possible:

- Computing the root of a sum of squares (RMSE) corresponds to the *Euclidean norm*: this is the notion of distance you are familiar with. It is also called the ℓ_2 *norm*, noted $\|\cdot\|_2$ (or just $\|\cdot\|$).
- Computing the sum of absolutes (MAE) corresponds to the ℓ_1 *norm*, noted $\|\cdot\|_1$. This is sometimes called the *Manhattan norm* because it measures the distance between two points in a city if you can only travel along orthogonal city blocks.
- More generally, the ℓ_k *norm* of a vector $\mathbf{v}$ containing n elements is defined as $\|\mathbf{v}\|_k = (|v_0|^k + |v_1|^k + \ldots + |v_n|^k)^{1/k}$. ℓ_0 gives the number of nonzero elements in the vector, and ℓ_∞ gives the maximum absolute value in the vector.
- The higher the norm index, the more it focuses on large values and neglects small ones. This is why the RMSE is more sensitive to outliers than the MAE. But when outliers are exponentially rare (like in a bell-shaped curve), the RMSE performs very well and is generally preferred.

Check the Assumptions

Lastly, it is good practice to list and verify the assumptions that have been made so far (by you or others); this can help you catch serious issues early on. For example, the district prices that your system outputs are going to be fed into a downstream Machine Learning system, and you assume that these prices are going to be used as such. But what if the downstream system converts the prices into categories (e.g., "cheap," "medium," or "expensive") and then uses those categories instead of the prices themselves? In this case, getting the price perfectly right is not important at all; your system just needs to get the category right. If that's so, then the problem should have been framed as a classification task, not a regression task. You don't want to find this out after working on a regression system for months.

Fortunately, after talking with the team in charge of the downstream system, you are confident that they do indeed need the actual prices, not just categories. Great! You're all set, the lights are green, and you can start coding now!

Get the Data

It's time to get your hands dirty. Don't hesitate to pick up your laptop and walk through the following code examples in a Jupyter notebook. The full Jupyter notebook is available at *https://github.com/ageron/handson-ml2*.

Create the Workspace

First you will need to have Python installed. It is probably already installed on your system. If not, you can get it at *https://www.python.org/*.[5]

Next you need to create a workspace directory for your Machine Learning code and datasets. Open a terminal and type the following commands (after the `$` prompts):

```
$ export ML_PATH="$HOME/ml"      # You can change the path if you prefer
$ mkdir -p $ML_PATH
```

You will need a number of Python modules: Jupyter, NumPy, pandas, Matplotlib, and Scikit-Learn. If you already have Jupyter running with all these modules installed, you can safely skip to "Download the Data" on page 46. If you don't have them yet, there are many ways to install them (and their dependencies). You can use your system's packaging system (e.g., apt-get on Ubuntu, or MacPorts or Homebrew on macOS), install a Scientific Python distribution such as Anaconda and use its packaging system, or just use Python's own packaging system, pip, which is included by

5 The latest version of Python 3 is recommended. Python 2.7+ may work too, but now that it's deprecated, all major scientific libraries are dropping support for it, so you should migrate to Python 3 as soon as possible.

default with the Python binary installers (since Python 2.7.9).[6] You can check to see if pip is installed by typing the following command:

```
$ python3 -m pip --version
pip 19.3.1 from [...]/lib/python3.7/site-packages/pip (python 3.7)
```

You should make sure you have a recent version of pip installed. To upgrade the pip module, type the following (the exact version may differ):[7]

```
$ python3 -m pip install --user -U pip
Collecting pip
[...]
Successfully installed pip-19.3.1
```

Creating an Isolated Environment

If you would like to work in an isolated environment (which is strongly recommended so that you can work on different projects without having conflicting library versions), install virtualenv[8] by running the following pip command (again, if you want virtualenv to be installed for all users on your machine, remove `--user` and run this command with administrator rights):

```
$ python3 -m pip install --user -U virtualenv
Collecting virtualenv
[...]
Successfully installed virtualenv-16.7.6
```

Now you can create an isolated Python environment by typing this:

```
$ cd $ML_PATH
$ python3 -m virtualenv my_env
Using base prefix '[...]'
New python executable in [...]/ml/my_env/bin/python3
Also creating executable in [...]/ml/my_env/bin/python
Installing setuptools, pip, wheel...done.
```

Now every time you want to activate this environment, just open a terminal and type the following:

6 I'll show the installation steps using pip in a bash shell on a Linux or macOS system. You may need to adapt these commands to your own system. On Windows, I recommend installing Anaconda instead.

7 If you want to upgrade pip for all users on your machine rather than just your own user, you should remove the `--user` option and make sure you have administrator rights (e.g., by adding `sudo` before the whole command on Linux or macOS).

8 Alternative tools include venv (very similar to virtualenv and included in the standard library), virtualenvwrapper (provides extra functionalities on top of virtualenv), pyenv (allows easy switching between Python versions), and pipenv (a great packaging tool by the same author as the popular `requests` library, built on top of pip and virtualenv).

```
$ cd $ML_PATH
$ source my_env/bin/activate # on Linux or macOS
$ .\my_env\Scripts\activate  # on Windows
```

To deactivate this environment, type **`deactivate`**. While the environment is active, any package you install using pip will be installed in this isolated environment, and Python will only have access to these packages (if you also want access to the system's packages, you should create the environment using virtualenv's `--system-site-packages` option). Check out virtualenv's documentation for more information.

Now you can install all the required modules and their dependencies using this simple pip command (if you are not using a virtualenv, you will need the `--user` option or administrator rights):

```
$ python3 -m pip install -U jupyter matplotlib numpy pandas scipy scikit-learn
Collecting jupyter
  Downloading https://[...]/jupyter-1.0.0-py2.py3-none-any.whl
Collecting matplotlib
  [...]
```

If you created a virtualenv, you need to register it to Jupyter and give it a name:

```
$ python3 -m ipykernel install --user --name=python3
```

Now you can fire up Jupyter by typing the following command:

```
$ jupyter notebook
[...] Serving notebooks from local directory: [...]/ml
[...] The Jupyter Notebook is running at:
[...] http://localhost:8888/?token=60995e108e44ac8d8865a[...]
[...]  or http://127.0.0.1:8889/?token=60995e108e44ac8d8865a[...]
[...] Use Control-C to stop this server and shut down all kernels [...]
```

A Jupyter server is now running in your terminal, listening to port 8888. You can visit this server by opening your web browser to *http://localhost:8888/* (this usually happens automatically when the server starts). You should see your empty workspace directory (containing only the *env* directory if you followed the preceding virtualenv instructions).

Now create a new Python notebook by clicking the New button and selecting the appropriate Python version[9] (see Figure 2-3). Doing that will create a new notebook file called *Untitled.ipynb* in your workspace, start a Jupyter Python kernel to run the notebook, and open this notebook in a new tab. You should start by renaming this notebook to "Housing" (this will automatically rename the file to *Housing.ipynb*) by clicking Untitled and typing the new name.

9 Note that Jupyter can handle multiple versions of Python, and even many other languages such as R or Octave.

Figure 2-3. Your workspace in Jupyter

A notebook contains a list of cells. Each cell can contain executable code or formatted text. Right now the notebook contains only one empty code cell, labeled "In [1]:". Try typing **`print("Hello world!")`** in the cell and clicking the play button (see Figure 2-4) or pressing Shift-Enter. This sends the current cell to this notebook's Python kernel, which runs it and returns the output. The result is displayed below the cell, and since you've reached the end of the notebook, a new cell is automatically created. Go through the User Interface Tour from Jupyter's Help menu to learn the basics.

Figure 2-4. Hello world Python notebook

Download the Data

In typical environments your data would be available in a relational database (or some other common data store) and spread across multiple tables/documents/files. To access it, you would first need to get your credentials and access authorizations[10] and familiarize yourself with the data schema. In this project, however, things are much simpler: you will just download a single compressed file, *housing.tgz*, which contains a comma-separated values (CSV) file called *housing.csv* with all the data.

You could use your web browser to download the file and run `tar xzf housing.tgz` to decompress it and extract the CSV file, but it is preferable to create a small function to do that. Having a function that downloads the data is useful in particular if the data changes regularly: you can write a small script that uses the function to fetch the latest data (or you can set up a scheduled job to do that automatically at regular intervals). Automating the process of fetching the data is also useful if you need to install the dataset on multiple machines.

Here is the function to fetch the data:[11]

```
import os
import tarfile
import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    os.makedirs(housing_path, exist_ok=True)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

Now when you call `fetch_housing_data()`, it creates a *datasets/housing* directory in your workspace, downloads the *housing.tgz* file, and extracts the *housing.csv* file from it in this directory.

10 You might also need to check legal constraints, such as private fields that should never be copied to unsafe data stores.

11 In a real project you would save this code in a Python file, but for now you can just write it in your Jupyter notebook.

Now let's load the data using pandas. Once again, you should write a small function to load the data:

```
import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

This function returns a pandas DataFrame object containing all the data.

Take a Quick Look at the Data Structure

Let's take a look at the top five rows using the DataFrame's `head()` method (see Figure 2-5).

```
In [5]: housing = load_housing_data()
        housing.head()
```

Out[5]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	populatio
0	-122.23	37.88	41.0	880.0	129.0	322.0
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0
2	-122.24	37.85	52.0	1467.0	190.0	496.0
3	-122.25	37.85	52.0	1274.0	235.0	558.0
4	-122.25	37.85	52.0	1627.0	280.0	565.0

Figure 2-5. Top five rows in the dataset

Each row represents one district. There are 10 attributes (you can see the first 6 in the screenshot): `longitude`, `latitude`, `housing_median_age`, `total_rooms`, `total_bedrooms`, `population`, `households`, `median_income`, `median_house_value`, and `ocean_proximity`.

The `info()` method is useful to get a quick description of the data, in particular the total number of rows, each attribute's type, and the number of nonnull values (see Figure 2-6).

```
In [6]: housing.info()

        <class 'pandas.core.frame.DataFrame'>
        RangeIndex: 20640 entries, 0 to 20639
        Data columns (total 10 columns):
        longitude             20640 non-null float64
        latitude              20640 non-null float64
        housing_median_age    20640 non-null float64
        total_rooms           20640 non-null float64
        total_bedrooms        20433 non-null float64
        population            20640 non-null float64
        households            20640 non-null float64
        median_income         20640 non-null float64
        median_house_value    20640 non-null float64
        ocean_proximity       20640 non-null object
        dtypes: float64(9), object(1)
        memory usage: 1.6+ MB
```

Figure 2-6. Housing info

There are 20,640 instances in the dataset, which means that it is fairly small by Machine Learning standards, but it's perfect to get started. Notice that the `total_bedrooms` attribute has only 20,433 nonnull values, meaning that 207 districts are missing this feature. We will need to take care of this later.

All attributes are numerical, except the `ocean_proximity` field. Its type is `object`, so it could hold any kind of Python object. But since you loaded this data from a CSV file, you know that it must be a text attribute. When you looked at the top five rows, you probably noticed that the values in the `ocean_proximity` column were repetitive, which means that it is probably a categorical attribute. You can find out what categories exist and how many districts belong to each category by using the `value_counts()` method:

```
>>> housing["ocean_proximity"].value_counts()
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

Let's look at the other fields. The `describe()` method shows a summary of the numerical attributes (Figure 2-7).

```
In [8]: housing.describe()
```

Out[8]:

	longitude	latitude	housing_median_age	total_rooms	total_bedr
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.385070
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.0000

Figure 2-7. Summary of each numerical attribute

The `count`, `mean`, `min`, and `max` rows are self-explanatory. Note that the null values are ignored (so, for example, the `count` of `total_bedrooms` is 20,433, not 20,640). The `std` row shows the *standard deviation*, which measures how dispersed the values are.[12] The 25%, 50%, and 75% rows show the corresponding *percentiles*: a percentile indicates the value below which a given percentage of observations in a group of observations fall. For example, 25% of the districts have a `housing_median_age` lower than 18, while 50% are lower than 29 and 75% are lower than 37. These are often called the 25th percentile (or first *quartile*), the median, and the 75th percentile (or third quartile).

Another quick way to get a feel of the type of data you are dealing with is to plot a histogram for each numerical attribute. A histogram shows the number of instances (on the vertical axis) that have a given value range (on the horizontal axis). You can either plot this one attribute at a time, or you can call the `hist()` method on the whole dataset (as shown in the following code example), and it will plot a histogram for each numerical attribute (see Figure 2-8):

```
%matplotlib inline   # only in a Jupyter notebook
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```

12 The standard deviation is generally denoted σ (the Greek letter sigma), and it is the square root of the *variance*, which is the average of the squared deviation from the mean. When a feature has a bell-shaped *normal distribution* (also called a *Gaussian distribution*), which is very common, the "68-95-99.7" rule applies: about 68% of the values fall within 1σ of the mean, 95% within 2σ, and 99.7% within 3σ.

The `hist()` method relies on Matplotlib, which in turn relies on a user-specified graphical backend to draw on your screen. So before you can plot anything, you need to specify which backend Matplotlib should use. The simplest option is to use Jupyter's magic command `%matplotlib inline`. This tells Jupyter to set up Matplotlib so it uses Jupyter's own backend. Plots are then rendered within the notebook itself. Note that calling `show()` is optional in a Jupyter notebook, as Jupyter will automatically display plots when a cell is executed.

Figure 2-8. A histogram for each numerical attribute

There are a few things you might notice in these histograms:

1. First, the median income attribute does not look like it is expressed in US dollars (USD). After checking with the team that collected the data, you are told that the data has been scaled and capped at 15 (actually, 15.0001) for higher median incomes, and at 0.5 (actually, 0.4999) for lower median incomes. The numbers represent roughly tens of thousands of dollars (e.g., 3 actually means about $30,000). Working with preprocessed attributes is common in Machine Learning,

and it is not necessarily a problem, but you should try to understand how the data was computed.

2. The housing median age and the median house value were also capped. The latter may be a serious problem since it is your target attribute (your labels). Your Machine Learning algorithms may learn that prices never go beyond that limit. You need to check with your client team (the team that will use your system's output) to see if this is a problem or not. If they tell you that they need precise predictions even beyond $500,000, then you have two options:
   a. Collect proper labels for the districts whose labels were capped.
   b. Remove those districts from the training set (and also from the test set, since your system should not be evaluated poorly if it predicts values beyond $500,000).
3. These attributes have very different scales. We will discuss this later in this chapter, when we explore feature scaling.
4. Finally, many histograms are *tail-heavy*: they extend much farther to the right of the median than to the left. This may make it a bit harder for some Machine Learning algorithms to detect patterns. We will try transforming these attributes later on to have more bell-shaped distributions.

Hopefully you now have a better understanding of the kind of data you are dealing with.

Wait! Before you look at the data any further, you need to create a test set, put it aside, and never look at it.

Create a Test Set

It may sound strange to voluntarily set aside part of the data at this stage. After all, you have only taken a quick glance at the data, and surely you should learn a whole lot more about it before you decide what algorithms to use, right? This is true, but your brain is an amazing pattern detection system, which means that it is highly prone to overfitting: if you look at the test set, you may stumble upon some seemingly interesting pattern in the test data that leads you to select a particular kind of Machine Learning model. When you estimate the generalization error using the test set, your estimate will be too optimistic, and you will launch a system that will not perform as well as expected. This is called *data snooping* bias.

Creating a test set is theoretically simple: pick some instances randomly, typically 20% of the dataset (or less if your dataset is very large), and set them aside:

```
import numpy as np

def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

You can then use this function like this:[13]

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> len(train_set)
16512
>>> len(test_set)
4128
```

Well, this works, but it is not perfect: if you run the program again, it will generate a different test set! Over time, you (or your Machine Learning algorithms) will get to see the whole dataset, which is what you want to avoid.

One solution is to save the test set on the first run and then load it in subsequent runs. Another option is to set the random number generator's seed (e.g., with `np.random.seed(42)`)[14] before calling `np.random.permutation()` so that it always generates the same shuffled indices.

But both these solutions will break the next time you fetch an updated dataset. To have a stable train/test split even after updating the dataset, a common solution is to use each instance's identifier to decide whether or not it should go in the test set (assuming instances have a unique and immutable identifier). For example, you could compute a hash of each instance's identifier and put that instance in the test set if the hash is lower than or equal to 20% of the maximum hash value. This ensures that the test set will remain consistent across multiple runs, even if you refresh the dataset. The new test set will contain 20% of the new instances, but it will not contain any instance that was previously in the training set.

Here is a possible implementation:

```
from zlib import crc32

def test_set_check(identifier, test_ratio):
    return crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32
```

13 In this book, when a code example contains a mix of code and outputs, as is the case here, it is formatted like in the Python interpreter, for better readability: the code lines are prefixed with `>>>` (or `...` for indented blocks), and the outputs have no prefix.

14 You will often see people set the random seed to 42. This number has no special property, other than to be the Answer to the Ultimate Question of Life, the Universe, and Everything.

```
def split_train_test_by_id(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

Unfortunately, the housing dataset does not have an identifier column. The simplest solution is to use the row index as the ID:

```
housing_with_id = housing.reset_index()   # adds an `index` column
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

If you use the row index as a unique identifier, you need to make sure that new data gets appended to the end of the dataset and that no row ever gets deleted. If this is not possible, then you can try to use the most stable features to build a unique identifier. For example, a district's latitude and longitude are guaranteed to be stable for a few million years, so you could combine them into an ID like so:[15]

```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways. The simplest function is `train_test_split()`, which does pretty much the same thing as the function `split_train_test()`, with a couple of additional features. First, there is a `random_state` parameter that allows you to set the random generator seed. Second, you can pass it multiple datasets with an identical number of rows, and it will split them on the same indices (this is very useful, for example, if you have a separate DataFrame for labels):

```
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

So far we have considered purely random sampling methods. This is generally fine if your dataset is large enough (especially relative to the number of attributes), but if it is not, you run the risk of introducing a significant sampling bias. When a survey company decides to call 1,000 people to ask them a few questions, they don't just pick 1,000 people randomly in a phone book. They try to ensure that these 1,000 people are representative of the whole population. For example, the US population is 51.3% females and 48.7% males, so a well-conducted survey in the US would try to maintain this ratio in the sample: 513 female and 487 male. This is called *stratified sampling*: the population is divided into homogeneous subgroups called *strata*, and the right number of instances are sampled from each stratum to guarantee that the test set is representative of the overall population. If the people running the survey used purely random sampling, there would be about a 12% chance of sampling a skewed test set

15 The location information is actually quite coarse, and as a result many districts will have the exact same ID, so they will end up in the same set (test or train). This introduces some unfortunate sampling bias.

that was either less than 49% female or more than 54% female. Either way, the survey results would be significantly biased.

Suppose you chatted with experts who told you that the median income is a very important attribute to predict median housing prices. You may want to ensure that the test set is representative of the various categories of incomes in the whole dataset. Since the median income is a continuous numerical attribute, you first need to create an income category attribute. Let's look at the median income histogram more closely (back in Figure 2-8): most median income values are clustered around 1.5 to 6 (i.e., $15,000–$60,000), but some median incomes go far beyond 6. It is important to have a sufficient number of instances in your dataset for each stratum, or else the estimate of a stratum's importance may be biased. This means that you should not have too many strata, and each stratum should be large enough. The following code uses the `pd.cut()` function to create an income category attribute with five categories (labeled from 1 to 5): category 1 ranges from 0 to 1.5 (i.e., less than $15,000), category 2 from 1.5 to 3, and so on:

```
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])
```

These income categories are represented in Figure 2-9:

```
housing["income_cat"].hist()
```

Figure 2-9. Histogram of income categories

Now you are ready to do stratified sampling based on the income category. For this you can use Scikit-Learn's `StratifiedShuffleSplit` class:

```
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

Let's see if this worked as expected. You can start by looking at the income category proportions in the test set:

```
>>> strat_test_set["income_cat"].value_counts() / len(strat_test_set)
3    0.350533
2    0.318798
4    0.176357
5    0.114583
1    0.039729
Name: income_cat, dtype: float64
```

With similar code you can measure the income category proportions in the full dataset. Figure 2-10 compares the income category proportions in the overall dataset, in the test set generated with stratified sampling, and in a test set generated using purely random sampling. As you can see, the test set generated using stratified sampling has income category proportions almost identical to those in the full dataset, whereas the test set generated using purely random sampling is skewed.

	Overall	Stratified	Random	Rand. %error	Strat. %error
1	0.039826	0.039729	0.040213	0.973236	-0.243309
2	0.318847	0.318798	0.324370	1.732260	-0.015195
3	0.350581	0.350533	0.358527	2.266446	-0.013820
4	0.176308	0.176357	0.167393	-5.056334	0.027480
5	0.114438	0.114583	0.109496	-4.318374	0.127011

Figure 2-10. Sampling bias comparison of stratified versus purely random sampling

Now you should remove the `income_cat` attribute so the data is back to its original state:

```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

We spent quite a bit of time on test set generation for a good reason: this is an often neglected but critical part of a Machine Learning project. Moreover, many of these ideas will be useful later when we discuss cross-validation. Now it's time to move on to the next stage: exploring the data.

Discover and Visualize the Data to Gain Insights

So far you have only taken a quick glance at the data to get a general understanding of the kind of data you are manipulating. Now the goal is to go into a little more depth.

First, make sure you have put the test set aside and you are only exploring the training set. Also, if the training set is very large, you may want to sample an exploration set, to make manipulations easy and fast. In our case, the set is quite small, so you can just work directly on the full set. Let's create a copy so that you can play with it without harming the training set:

```
housing = strat_train_set.copy()
```

Visualizing Geographical Data

Since there is geographical information (latitude and longitude), it is a good idea to create a scatterplot of all districts to visualize the data (Figure 2-11):

```
housing.plot(kind="scatter", x="longitude", y="latitude")
```

Figure 2-11. A geographical scatterplot of the data

This looks like California all right, but other than that it is hard to see any particular pattern. Setting the `alpha` option to `0.1` makes it much easier to visualize the places where there is a high density of data points (Figure 2-12):

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

Figure 2-12. A better visualization that highlights high-density areas

Now that's much better: you can clearly see the high-density areas, namely the Bay Area and around Los Angeles and San Diego, plus a long line of fairly high density in the Central Valley, in particular around Sacramento and Fresno.

Our brains are very good at spotting patterns in pictures, but you may need to play around with visualization parameters to make the patterns stand out.

Now let's look at the housing prices (Figure 2-13). The radius of each circle represents the district's population (option `s`), and the color represents the price (option `c`). We will use a predefined color map (option `cmap`) called `jet`, which ranges from blue (low values) to red (high prices):[16]

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
    s=housing["population"]/100, label="population", figsize=(10,7),
    c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```

16 If you are reading this in grayscale, grab a red pen and scribble over most of the coastline from the Bay Area down to San Diego (as you might expect). You can add a patch of yellow around Sacramento as well.

Figure 2-13. California housing prices: red is expensive, blue is cheap, larger circles indicate areas with a larger population

This image tells you that the housing prices are very much related to the location (e.g., close to the ocean) and to the population density, as you probably knew already. A clustering algorithm should be useful for detecting the main cluster and for adding new features that measure the proximity to the cluster centers. The ocean proximity attribute may be useful as well, although in Northern California the housing prices in coastal districts are not too high, so it is not a simple rule.

Looking for Correlations

Since the dataset is not too large, you can easily compute the *standard correlation coefficient* (also called *Pearson's r*) between every pair of attributes using the `corr()` method:

```
corr_matrix = housing.corr()
```

Now let's look at how much each attribute correlates with the median house value:

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.687170
total_rooms           0.135231
housing_median_age    0.114220
```

```
households            0.064702
total_bedrooms        0.047865
population           -0.026699
longitude            -0.047279
latitude             -0.142826
Name: median_house_value, dtype: float64
```

The correlation coefficient ranges from –1 to 1. When it is close to 1, it means that there is a strong positive correlation; for example, the median house value tends to go up when the median income goes up. When the coefficient is close to –1, it means that there is a strong negative correlation; you can see a small negative correlation between the latitude and the median house value (i.e., prices have a slight tendency to go down when you go north). Finally, coefficients close to 0 mean that there is no linear correlation. Figure 2-14 shows various plots along with the correlation coefficient between their horizontal and vertical axes.

Figure 2-14. Standard correlation coefficient of various datasets (source: Wikipedia; public domain image)

The correlation coefficient only measures linear correlations ("if *x* goes up, then *y* generally goes up/down"). It may completely miss out on nonlinear relationships (e.g., "if *x* is close to 0, then *y* generally goes up"). Note how all the plots of the bottom row have a correlation coefficient equal to 0, despite the fact that their axes are clearly not independent: these are examples of nonlinear relationships. Also, the second row shows examples where the correlation coefficient is equal to 1 or –1; notice that this has nothing to do with the slope. For example, your height in inches has a correlation coefficient of 1 with your height in feet or in nanometers.

Another way to check for correlation between attributes is to use the pandas `scatter_matrix()` function, which plots every numerical attribute against every

other numerical attribute. Since there are now 11 numerical attributes, you would get $11^2 = 121$ plots, which would not fit on a page—so let's just focus on a few promising attributes that seem most correlated with the median housing value (Figure 2-15):

```
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```

Figure 2-15. This scatter matrix plots every numerical attribute against every other numerical attribute, plus a histogram of each numerical attribute

The main diagonal (top left to bottom right) would be full of straight lines if pandas plotted each variable against itself, which would not be very useful. So instead pandas displays a histogram of each attribute (other options are available; see the pandas documentation for more details).

The most promising attribute to predict the median house value is the median income, so let's zoom in on their correlation scatterplot (Figure 2-16):

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1)
```

Figure 2-16. Median income versus median house value

This plot reveals a few things. First, the correlation is indeed very strong; you can clearly see the upward trend, and the points are not too dispersed. Second, the price cap that we noticed earlier is clearly visible as a horizontal line at $500,000. But this plot reveals other less obvious straight lines: a horizontal line around $450,000, another around $350,000, perhaps one around $280,000, and a few more below that. You may want to try removing the corresponding districts to prevent your algorithms from learning to reproduce these data quirks.

Experimenting with Attribute Combinations

Hopefully the previous sections gave you an idea of a few ways you can explore the data and gain insights. You identified a few data quirks that you may want to clean up before feeding the data to a Machine Learning algorithm, and you found interesting correlations between attributes, in particular with the target attribute. You also noticed that some attributes have a tail-heavy distribution, so you may want to transform them (e.g., by computing their logarithm). Of course, your mileage will vary considerably with each project, but the general ideas are similar.

One last thing you may want to do before preparing the data for Machine Learning algorithms is to try out various attribute combinations. For example, the total number of rooms in a district is not very useful if you don't know how many households there are. What you really want is the number of rooms per household. Similarly, the total number of bedrooms by itself is not very useful: you probably want to compare it to the number of rooms. And the population per household also seems like an interesting attribute combination to look at. Let's create these new attributes:

```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households"]
```

And now let's look at the correlation matrix again:

```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value          1.000000
median_income               0.687160
rooms_per_household         0.146285
total_rooms                 0.135097
housing_median_age          0.114110
households                  0.064506
total_bedrooms              0.047689
population_per_household   -0.021985
population                 -0.026920
longitude                  -0.047432
latitude                   -0.142724
bedrooms_per_room          -0.259984
Name: median_house_value, dtype: float64
```

Hey, not bad! The new `bedrooms_per_room` attribute is much more correlated with the median house value than the total number of rooms or bedrooms. Apparently houses with a lower bedroom/room ratio tend to be more expensive. The number of rooms per household is also more informative than the total number of rooms in a district—obviously the larger the houses, the more expensive they are.

This round of exploration does not have to be absolutely thorough; the point is to start off on the right foot and quickly gain insights that will help you get a first reasonably good prototype. But this is an iterative process: once you get a prototype up and running, you can analyze its output to gain more insights and come back to this exploration step.

Prepare the Data for Machine Learning Algorithms

It's time to prepare the data for your Machine Learning algorithms. Instead of doing this manually, you should write functions for this purpose, for several good reasons:

- This will allow you to reproduce these transformations easily on any dataset (e.g., the next time you get a fresh dataset).
- You will gradually build a library of transformation functions that you can reuse in future projects.
- You can use these functions in your live system to transform the new data before feeding it to your algorithms.

- This will make it possible for you to easily try various transformations and see which combination of transformations works best.

But first let's revert to a clean training set (by copying `strat_train_set` once again). Let's also separate the predictors and the labels, since we don't necessarily want to apply the same transformations to the predictors and the target values (note that `drop()` creates a copy of the data and does not affect `strat_train_set`):

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

Data Cleaning

Most Machine Learning algorithms cannot work with missing features, so let's create a few functions to take care of them. We saw earlier that the `total_bedrooms` attribute has some missing values, so let's fix this. You have three options:

1. Get rid of the corresponding districts.
2. Get rid of the whole attribute.
3. Set the values to some value (zero, the mean, the median, etc.).

You can accomplish these easily using DataFrame's `dropna()`, `drop()`, and `fillna()` methods:

```
housing.dropna(subset=["total_bedrooms"])    # option 1
housing.drop("total_bedrooms", axis=1)       # option 2
median = housing["total_bedrooms"].median()  # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

If you choose option 3, you should compute the median value on the training set and use it to fill the missing values in the training set. Don't forget to save the median value that you have computed. You will need it later to replace missing values in the test set when you want to evaluate your system, and also once the system goes live to replace missing values in new data.

Scikit-Learn provides a handy class to take care of missing values: `SimpleImputer`. Here is how to use it. First, you need to create a `SimpleImputer` instance, specifying that you want to replace each attribute's missing values with the median of that attribute:

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
```

Since the median can only be computed on numerical attributes, you need to create a copy of the data without the text attribute `ocean_proximity`:

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

Now you can fit the `imputer` instance to the training data using the `fit()` method:

```
imputer.fit(housing_num)
```

The `imputer` has simply computed the median of each attribute and stored the result in its `statistics_` instance variable. Only the `total_bedrooms` attribute had missing values, but we cannot be sure that there won't be any missing values in new data after the system goes live, so it is safer to apply the `imputer` to all the numerical attributes:

```
>>> imputer.statistics_
array([ -118.51 , 34.26 , 29. , 2119.5 , 433. , 1164. , 408. , 3.5409])
>>> housing_num.median().values
array([ -118.51 , 34.26 , 29. , 2119.5 , 433. , 1164. , 408. , 3.5409])
```

Now you can use this "trained" `imputer` to transform the training set by replacing missing values with the learned medians:

```
X = imputer.transform(housing_num)
```

The result is a plain NumPy array containing the transformed features. If you want to put it back into a pandas DataFrame, it's simple:

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing_num.index)
```

Scikit-Learn Design

Scikit-Learn's API is remarkably well designed. These are the main design principles:[17]

Consistency

All objects share a consistent and simple interface:

Estimators

Any object that can estimate some parameters based on a dataset is called an *estimator* (e.g., an `imputer` is an estimator). The estimation itself is performed by the `fit()` method, and it takes only a dataset as a parameter (or two for supervised learning algorithms; the second dataset contains the labels). Any other parameter needed to guide the estimation process is considered a hyperparameter (such as an `imputer`'s `strategy`), and it must be set as an instance variable (generally via a constructor parameter).

Transformers

Some estimators (such as an `imputer`) can also transform a dataset; these are called *transformers*. Once again, the API is simple: the transformation is performed by the `transform()` method with the dataset to transform as a

17 For more details on the design principles, see Lars Buitinck et al., "API Design for Machine Learning Software: Experiences from the Scikit-Learn Project" ," arXiv preprint arXiv:1309.0238 (2013).

parameter. It returns the transformed dataset. This transformation generally relies on the learned parameters, as is the case for an `imputer`. All transformers also have a convenience method called `fit_transform()` that is equivalent to calling `fit()` and then `transform()` (but sometimes `fit_transform()` is optimized and runs much faster).

Predictors
: Finally, some estimators, given a dataset, are capable of making predictions; they are called *predictors*. For example, the `LinearRegression` model in the previous chapter was a predictor: given a country's GDP per capita, it predicted life satisfaction. A predictor has a `predict()` method that takes a dataset of new instances and returns a dataset of corresponding predictions. It also has a `score()` method that measures the quality of the predictions, given a test set (and the corresponding labels, in the case of supervised learning algorithms).[18]

Inspection
: All the estimator's hyperparameters are accessible directly via public instance variables (e.g., `imputer.strategy`), and all the estimator's learned parameters are accessible via public instance variables with an underscore suffix (e.g., `imputer.statistics_`).

Nonproliferation of classes
: Datasets are represented as NumPy arrays or SciPy sparse matrices, instead of homemade classes. Hyperparameters are just regular Python strings or numbers.

Composition
: Existing building blocks are reused as much as possible. For example, it is easy to create a `Pipeline` estimator from an arbitrary sequence of transformers followed by a final estimator, as we will see.

Sensible defaults
: Scikit-Learn provides reasonable default values for most parameters, making it easy to quickly create a baseline working system.

Handling Text and Categorical Attributes

So far we have only dealt with numerical attributes, but now let's look at text attributes. In this dataset, there is just one: the `ocean_proximity` attribute. Let's look at its value for the first 10 instances:

18 Some predictors also provide methods to measure the confidence of their predictions.

```
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat.head(10)
      ocean_proximity
17606       <1H OCEAN
18632       <1H OCEAN
14650      NEAR OCEAN
3230           INLAND
3555        <1H OCEAN
19480          INLAND
8879        <1H OCEAN
13685          INLAND
4937        <1H OCEAN
4861        <1H OCEAN
```

It's not arbitrary text: there are a limited number of possible values, each of which represents a category. So this attribute is a categorical attribute. Most Machine Learning algorithms prefer to work with numbers, so let's convert these categories from text to numbers. For this, we can use Scikit-Learn's `OrdinalEncoder` class:[19]

```
>>> from sklearn.preprocessing import OrdinalEncoder
>>> ordinal_encoder = OrdinalEncoder()
>>> housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
>>> housing_cat_encoded[:10]
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

You can get the list of categories using the `categories_` instance variable. It is a list containing a 1D array of categories for each categorical attribute (in this case, a list containing a single array since there is just one categorical attribute):

```
>>> ordinal_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values. This may be fine in some cases (e.g., for ordered categories such as "bad," "average," "good," and "excellent"), but it is obviously not the case for the `ocean_proximity` column (for example, categories 0 and 4 are clearly more similar than categories 0 and 1). To fix this issue, a common solution

19 This class is available in Scikit-Learn 0.20 and later. If you use an earlier version, please consider upgrading, or use the pandas `Series.factorize()` method.

is to create one binary attribute per category: one attribute equal to 1 when the category is "<1H OCEAN" (and 0 otherwise), another attribute equal to 1 when the category is "INLAND" (and 0 otherwise), and so on. This is called *one-hot encoding*, because only one attribute will be equal to 1 (hot), while the others will be 0 (cold). The new attributes are sometimes called *dummy* attributes. Scikit-Learn provides a `OneHotEncoder` class to convert categorical values into one-hot vectors:[20]

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> cat_encoder = OneHotEncoder()
>>> housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
>>> housing_cat_1hot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
  with 16512 stored elements in Compressed Sparse Row format>
```

Notice that the output is a SciPy *sparse matrix*, instead of a NumPy array. This is very useful when you have categorical attributes with thousands of categories. After one-hot encoding, we get a matrix with thousands of columns, and the matrix is full of 0s except for a single 1 per row. Using up tons of memory mostly to store zeros would be very wasteful, so instead a sparse matrix only stores the location of the nonzero elements. You can use it mostly like a normal 2D array,[21] but if you really want to convert it to a (dense) NumPy array, just call the `toarray()` method:

```
>>> housing_cat_1hot.toarray()
array([[1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       ...,
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

Once again, you can get the list of categories using the encoder's `categories_` instance variable:

```
>>> cat_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

20 Before Scikit-Learn 0.20, the method could only encode integer categorical values, but since 0.20 it can also handle other types of inputs, including text categorical inputs.

21 See SciPy's documentation for more details.

If a categorical attribute has a large number of possible categories (e.g., country code, profession, species), then one-hot encoding will result in a large number of input features. This may slow down training and degrade performance. If this happens, you may want to replace the categorical input with useful numerical features related to the categories: for example, you could replace the `ocean_proximity` feature with the distance to the ocean (similarly, a country code could be replaced with the country's population and GDP per capita). Alternatively, you could replace each category with a learnable, low-dimensional vector called an *embedding*. Each category's representation would be learned during training. This is an example of *representation learning* (see Chapters 13 and 17 for more details).

Custom Transformers

Although Scikit-Learn provides many useful transformers, you will need to write your own for tasks such as custom cleanup operations or combining specific attributes. You will want your transformer to work seamlessly with Scikit-Learn functionalities (such as pipelines), and since Scikit-Learn relies on duck typing (not inheritance), all you need to do is create a class and implement three methods: `fit()` (returning `self`), `transform()`, and `fit_transform()`.

You can get the last one for free by simply adding `TransformerMixin` as a base class. If you add `BaseEstimator` as a base class (and avoid `*args` and `**kargs` in your constructor), you will also get two extra methods (`get_params()` and `set_params()`) that will be useful for automatic hyperparameter tuning.

For example, here is a small transformer class that adds the combined attributes we discussed earlier:

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                         bedrooms_per_room]
```

```
        else:
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

In this example the transformer has one hyperparameter, `add_bedrooms_per_room`, set to `True` by default (it is often helpful to provide sensible defaults). This hyperparameter will allow you to easily find out whether adding this attribute helps the Machine Learning algorithms or not. More generally, you can add a hyperparameter to gate any data preparation step that you are not 100% sure about. The more you automate these data preparation steps, the more combinations you can automatically try out, making it much more likely that you will find a great combination (and saving you a lot of time).

Feature Scaling

One of the most important transformations you need to apply to your data is *feature scaling*. With few exceptions, Machine Learning algorithms don't perform well when the input numerical attributes have very different scales. This is the case for the housing data: the total number of rooms ranges from about 6 to 39,320, while the median incomes only range from 0 to 15. Note that scaling the target values is generally not required.

There are two common ways to get all attributes to have the same scale: *min-max scaling* and *standardization*.

Min-max scaling (many people call this *normalization*) is the simplest: values are shifted and rescaled so that they end up ranging from 0 to 1. We do this by subtracting the min value and dividing by the max minus the min. Scikit-Learn provides a transformer called `MinMaxScaler` for this. It has a `feature_range` hyperparameter that lets you change the range if, for some reason, you don't want 0–1.

Standardization is different: first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the standard deviation so that the resulting distribution has unit variance. Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1). However, standardization is much less affected by outliers. For example, suppose a district had a median income equal to 100 (by mistake). Min-max scaling would then crush all the other values from 0–15 down to 0–0.15, whereas standardization would not be much affected. Scikit-Learn provides a transformer called `StandardScaler` for standardization.

As with all the transformations, it is important to fit the scalers to the training data only, not to the full dataset (including the test set). Only then can you use them to transform the training set and the test set (and new data).

Transformation Pipelines

As you can see, there are many data transformation steps that need to be executed in the right order. Fortunately, Scikit-Learn provides the `Pipeline` class to help with such sequences of transformations. Here is a small pipeline for the numerical attributes:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

The `Pipeline` constructor takes a list of name/estimator pairs defining a sequence of steps. All but the last estimator must be transformers (i.e., they must have a `fit_transform()` method). The names can be anything you like (as long as they are unique and don't contain double underscores, __); they will come in handy later for hyperparameter tuning.

When you call the pipeline's `fit()` method, it calls `fit_transform()` sequentially on all transformers, passing the output of each call as the parameter to the next call until it reaches the final estimator, for which it calls the `fit()` method.

The pipeline exposes the same methods as the final estimator. In this example, the last estimator is a `StandardScaler`, which is a transformer, so the pipeline has a `transform()` method that applies all the transforms to the data in sequence (and of course also a `fit_transform()` method, which is the one we used).

So far, we have handled the categorical columns and the numerical columns separately. It would be more convenient to have a single transformer able to handle all columns, applying the appropriate transformations to each column. In version 0.20, Scikit-Learn introduced the `ColumnTransformer` for this purpose, and the good news is that it works great with pandas DataFrames. Let's use it to apply all the transformations to the housing data:

```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
    ])

housing_prepared = full_pipeline.fit_transform(housing)
```

First we import the `ColumnTransformer` class, next we get the list of numerical column names and the list of categorical column names, and then we construct a `ColumnTransformer`. The constructor requires a list of tuples, where each tuple contains a name,[22] a transformer, and a list of names (or indices) of columns that the transformer should be applied to. In this example, we specify that the numerical columns should be transformed using the `num_pipeline` that we defined earlier, and the categorical columns should be transformed using a `OneHotEncoder`. Finally, we apply this `ColumnTransformer` to the housing data: it applies each transformer to the appropriate columns and concatenates the outputs along the second axis (the transformers must return the same number of rows).

Note that the `OneHotEncoder` returns a sparse matrix, while the `num_pipeline` returns a dense matrix. When there is such a mix of sparse and dense matrices, the `ColumnTransformer` estimates the density of the final matrix (i.e., the ratio of nonzero cells), and it returns a sparse matrix if the density is lower than a given threshold (by default, `sparse_threshold=0.3`). In this example, it returns a dense matrix. And that's it! We have a preprocessing pipeline that takes the full housing data and applies the appropriate transformations to each column.

Instead of using a transformer, you can specify the string `"drop"` if you want the columns to be dropped, or you can specify `"passthrough"` if you want the columns to be left untouched. By default, the remaining columns (i.e., the ones that were not listed) will be dropped, but you can set the `remainder` hyperparameter to any transformer (or to `"passthrough"`) if you want these columns to be handled differently.

If you are using Scikit-Learn 0.19 or earlier, you can use a third-party library such as `sklearn-pandas`, or you can roll out your own custom transformer to get the same functionality as the `ColumnTransformer`. Alternatively, you can use the `FeatureUnion`

22 Just like for pipelines, the name can be anything as long as it does not contain double underscores.

class, which can apply different transformers and concatenate their outputs. But you cannot specify different columns for each transformer; they all apply to the whole data. It is possible to work around this limitation using a custom transformer for column selection (see the Jupyter notebook for an example).

Select and Train a Model

At last! You framed the problem, you got the data and explored it, you sampled a training set and a test set, and you wrote transformation pipelines to clean up and prepare your data for Machine Learning algorithms automatically. You are now ready to select and train a Machine Learning model.

Training and Evaluating on the Training Set

The good news is that thanks to all these previous steps, things are now going to be much simpler than you might think. Let's first train a Linear Regression model, like we did in the previous chapter:

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

Done! You now have a working Linear Regression model. Let's try it out on a few instances from the training set:

```
>>> some_data = housing.iloc[:5]
>>> some_labels = housing_labels.iloc[:5]
>>> some_data_prepared = full_pipeline.transform(some_data)
>>> print("Predictions:", lin_reg.predict(some_data_prepared))
Predictions: [ 210644.6045  317768.8069  210956.4333  59218.9888  189747.5584]
>>> print("Labels:", list(some_labels))
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

It works, although the predictions are not exactly accurate (e.g., the first prediction is off by close to 40%!). Let's measure this regression model's RMSE on the whole training set using Scikit-Learn's `mean_squared_error()` function:

```
>>> from sklearn.metrics import mean_squared_error
>>> housing_predictions = lin_reg.predict(housing_prepared)
>>> lin_mse = mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse = np.sqrt(lin_mse)
>>> lin_rmse
68628.19819848922
```

This is better than nothing, but clearly not a great score: most districts' `median_housing_values` range between $120,000 and $265,000, so a typical prediction error of $68,628 is not very satisfying. This is an example of a model underfitting the training data. When this happens it can mean that the features do not provide enough

information to make good predictions, or that the model is not powerful enough. As we saw in the previous chapter, the main ways to fix underfitting are to select a more powerful model, to feed the training algorithm with better features, or to reduce the constraints on the model. This model is not regularized, which rules out the last option. You could try to add more features (e.g., the log of the population), but first let's try a more complex model to see how it does.

Let's train a `DecisionTreeRegressor`. This is a powerful model, capable of finding complex nonlinear relationships in the data (Decision Trees are presented in more detail in Chapter 6). The code should look familiar by now:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

Now that the model is trained, let's evaluate it on the training set:

```
>>> housing_predictions = tree_reg.predict(housing_prepared)
>>> tree_mse = mean_squared_error(housing_labels, housing_predictions)
>>> tree_rmse = np.sqrt(tree_mse)
>>> tree_rmse
0.0
```

Wait, what!? No error at all? Could this model really be absolutely perfect? Of course, it is much more likely that the model has badly overfit the data. How can you be sure? As we saw earlier, you don't want to touch the test set until you are ready to launch a model you are confident about, so you need to use part of the training set for training and part of it for model validation.

Better Evaluation Using Cross-Validation

One way to evaluate the Decision Tree model would be to use the `train_test_split()` function to split the training set into a smaller training set and a validation set, then train your models against the smaller training set and evaluate them against the validation set. It's a bit of work, but nothing too difficult, and it would work fairly well.

A great alternative is to use Scikit-Learn's *K-fold cross-validation* feature. The following code randomly splits the training set into 10 distinct subsets called *folds*, then it trains and evaluates the Decision Tree model 10 times, picking a different fold for evaluation every time and training on the other 9 folds. The result is an array containing the 10 evaluation scores:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```

Scikit-Learn's cross-validation features expect a utility function (greater is better) rather than a cost function (lower is better), so the scoring function is actually the opposite of the MSE (i.e., a negative value), which is why the preceding code computes `-scores` before calculating the square root.

Let's look at the results:

```
>>> def display_scores(scores):
...     print("Scores:", scores)
...     print("Mean:", scores.mean())
...     print("Standard deviation:", scores.std())
...
>>> display_scores(tree_rmse_scores)
Scores: [70194.33680785 66855.16363941 72432.58244769 70758.73896782
 71115.88230639 75585.14172901 70262.86139133 70273.6325285
 75366.87952553 71231.65726027]
Mean: 71407.68766037929
Standard deviation: 2439.4345041191004
```

Now the Decision Tree doesn't look as good as it did earlier. In fact, it seems to perform worse than the Linear Regression model! Notice that cross-validation allows you to get not only an estimate of the performance of your model, but also a measure of how precise this estimate is (i.e., its standard deviation). The Decision Tree has a score of approximately 71,407, generally ±2,439. You would not have this information if you just used one validation set. But cross-validation comes at the cost of training the model several times, so it is not always possible.

Let's compute the same scores for the Linear Regression model just to be sure:

```
>>> lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
...                              scoring="neg_mean_squared_error", cv=10)
...
>>> lin_rmse_scores = np.sqrt(-lin_scores)
>>> display_scores(lin_rmse_scores)
Scores: [66782.73843989 66960.118071   70347.95244419 74739.57052552
 68031.13388938 71193.84183426 64969.63056405 68281.61137997
 71552.91566558 67665.10082067]
Mean: 69052.46136345083
Standard deviation: 2731.674001798348
```

That's right: the Decision Tree model is overfitting so badly that it performs worse than the Linear Regression model.

Let's try one last model now: the `RandomForestRegressor`. As we will see in Chapter 7, Random Forests work by training many Decision Trees on random subsets of the features, then averaging out their predictions. Building a model on top of many other models is called *Ensemble Learning*, and it is often a great way to push ML algorithms even further. We will skip most of the code since it is essentially the same as for the other models:

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
>>> [...]
>>> forest_rmse
18603.515021376355
>>> display_scores(forest_rmse_scores)
Scores: [49519.80364233 47461.9115823  50029.02762854 52325.28068953
 49308.39426421 53446.37892622 48634.8036574  47585.73832311
 53490.10699751 50021.5852922 ]
Mean: 50182.303100336096
Standard deviation: 2097.0810550985693
```

Wow, this is much better: Random Forests look very promising. However, note that the score on the training set is still much lower than on the validation sets, meaning that the model is still overfitting the training set. Possible solutions for overfitting are to simplify the model, constrain it (i.e., regularize it), or get a lot more training data. Before you dive much deeper into Random Forests, however, you should try out many other models from various categories of Machine Learning algorithms (e.g., several Support Vector Machines with different kernels, and possibly a neural network), without spending too much time tweaking the hyperparameters. The goal is to shortlist a few (two to five) promising models.

You should save every model you experiment with so that you can come back easily to any model you want. Make sure you save both the hyperparameters and the trained parameters, as well as the cross-validation scores and perhaps the actual predictions as well. This will allow you to easily compare scores across model types, and compare the types of errors they make. You can easily save Scikit-Learn models by using Python's `pickle` module or by using the `joblib` library, which is more efficient at serializing large NumPy arrays (you can install this library using pip):

```
import joblib

joblib.dump(my_model, "my_model.pkl")
# and later...
my_model_loaded = joblib.load("my_model.pkl")
```

Fine-Tune Your Model

Let's assume that you now have a shortlist of promising models. You now need to fine-tune them. Let's look at a few ways you can do that.

Grid Search

One option would be to fiddle with the hyperparameters manually, until you find a great combination of hyperparameter values. This would be very tedious work, and you may not have time to explore many combinations.

Instead, you should get Scikit-Learn's `GridSearchCV` to search for you. All you need to do is tell it which hyperparameters you want it to experiment with and what values to try out, and it will use cross-validation to evaluate all the possible combinations of hyperparameter values. For example, the following code searches for the best combination of hyperparameter values for the `RandomForestRegressor`:

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)

grid_search.fit(housing_prepared, housing_labels)
```

When you have no idea what value a hyperparameter should have, a simple approach is to try out consecutive powers of 10 (or a smaller number if you want a more fine-grained search, as shown in this example with the `n_estimators` hyperparameter).

This `param_grid` tells Scikit-Learn to first evaluate all 3 × 4 = 12 combinations of `n_estimators` and `max_features` hyperparameter values specified in the first `dict` (don't worry about what these hyperparameters mean for now; they will be explained in Chapter 7), then try all 2 × 3 = 6 combinations of hyperparameter values in the second `dict`, but this time with the `bootstrap` hyperparameter set to `False` instead of `True` (which is the default value for this hyperparameter).

The grid search will explore 12 + 6 = 18 combinations of `RandomForestRegressor` hyperparameter values, and it will train each model 5 times (since we are using five-fold cross validation). In other words, all in all, there will be 18 × 5 = 90 rounds of training! It may take quite a long time, but when it is done you can get the best combination of parameters like this:

```
>>> grid_search.best_params_
{'max_features': 8, 'n_estimators': 30}
```

Since 8 and 30 are the maximum values that were evaluated, you should probably try searching again with higher values; the score may continue to improve.

You can also get the best estimator directly:

```
>>> grid_search.best_estimator_
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features=8, max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           n_estimators=30, n_jobs=None, oob_score=False, random_state=None,
           verbose=0, warm_start=False)
```

If `GridSearchCV` is initialized with `refit=True` (which is the default), then once it finds the best estimator using cross-validation, it retrains it on the whole training set. This is usually a good idea, since feeding it more data will likely improve its performance.

And of course the evaluation scores are also available:

```
>>> cvres = grid_search.cv_results_
>>> for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
...     print(np.sqrt(-mean_score), params)
...
63669.05791727153 {'max_features': 2, 'n_estimators': 3}
55627.16171305252 {'max_features': 2, 'n_estimators': 10}
53384.57867637289 {'max_features': 2, 'n_estimators': 30}
60965.99185930139 {'max_features': 4, 'n_estimators': 3}
52740.98248528835 {'max_features': 4, 'n_estimators': 10}
50377.344409590376 {'max_features': 4, 'n_estimators': 30}
58663.84733372485 {'max_features': 6, 'n_estimators': 3}
52006.15355973719 {'max_features': 6, 'n_estimators': 10}
50146.465964159885 {'max_features': 6, 'n_estimators': 30}
57869.25504027614 {'max_features': 8, 'n_estimators': 3}
51711.09443660957 {'max_features': 8, 'n_estimators': 10}
49682.25345942335 {'max_features': 8, 'n_estimators': 30}
62895.088889905004 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54658.14484390074 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59470.399594730654 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52725.01091081235 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57490.612956065226 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51009.51445842374 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

In this example, we obtain the best solution by setting the `max_features` hyperparameter to `8` and the `n_estimators` hyperparameter to `30`. The RMSE score for this combination is 49,682, which is slightly better than the score you got earlier using the

default hyperparameter values (which was 50,182). Congratulations, you have successfully fine-tuned your best model!

Don't forget that you can treat some of the data preparation steps as hyperparameters. For example, the grid search will automatically find out whether or not to add a feature you were not sure about (e.g., using the `add_bedrooms_per_room` hyperparameter of your `CombinedAttributesAdder` transformer). It may similarly be used to automatically find the best way to handle outliers, missing features, feature selection, and more.

Randomized Search

The grid search approach is fine when you are exploring relatively few combinations, like in the previous example, but when the hyperparameter search space is large, it is often preferable to use `RandomizedSearchCV` instead. This class can be used in much the same way as the `GridSearchCV` class, but instead of trying out all possible combinations, it evaluates a given number of random combinations by selecting a random value for each hyperparameter at every iteration. This approach has two main benefits:

- If you let the randomized search run for, say, 1,000 iterations, this approach will explore 1,000 different values for each hyperparameter (instead of just a few values per hyperparameter with the grid search approach).
- Simply by setting the number of iterations, you have more control over the computing budget you want to allocate to hyperparameter search.

Ensemble Methods

Another way to fine-tune your system is to try to combine the models that perform best. The group (or "ensemble") will often perform better than the best individual model (just like Random Forests perform better than the individual Decision Trees they rely on), especially if the individual models make very different types of errors. We will cover this topic in more detail in Chapter 7.

Analyze the Best Models and Their Errors

You will often gain good insights on the problem by inspecting the best models. For example, the `RandomForestRegressor` can indicate the relative importance of each attribute for making accurate predictions:

```
>>> feature_importances = grid_search.best_estimator_.feature_importances_
>>> feature_importances
array([7.33442355e-02, 6.29090705e-02, 4.11437985e-02, 1.46726854e-02,
```

```
       1.41064835e-02, 1.48742809e-02, 1.42575993e-02, 3.66158981e-01,
       5.64191792e-02, 1.08792957e-01, 5.33510773e-02, 1.03114883e-02,
       1.64780994e-01, 6.02803867e-05, 1.96041560e-03, 2.85647464e-03])
```

Let's display these importance scores next to their corresponding attribute names:

```
>>> extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
>>> cat_encoder = full_pipeline.named_transformers_["cat"]
>>> cat_one_hot_attribs = list(cat_encoder.categories_[0])
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs
>>> sorted(zip(feature_importances, attributes), reverse=True)
[(0.3661589806181342, 'median_income'),
 (0.1647809935615905, 'INLAND'),
 (0.10879295677551573, 'pop_per_hhold'),
 (0.07334423551601242, 'longitude'),
 (0.0629090704826203, 'latitude'),
 (0.05641917918195401, 'rooms_per_hhold'),
 (0.05335107734767581, 'bedrooms_per_room'),
 (0.041143798478729635, 'housing_median_age'),
 (0.014874280890402767, 'population'),
 (0.014672685420543237, 'total_rooms'),
 (0.014257599323407807, 'households'),
 (0.014106483453584102, 'total_bedrooms'),
 (0.010311488326303787, '<1H OCEAN'),
 (0.002856474637320158, 'NEAR OCEAN'),
 (0.00196041559947807, 'NEAR BAY'),
 (6.028038672736599e-05, 'ISLAND')]
```

With this information, you may want to try dropping some of the less useful features (e.g., apparently only one `ocean_proximity` category is really useful, so you could try dropping the others).

You should also look at the specific errors that your system makes, then try to understand why it makes them and what could fix the problem (adding extra features or getting rid of uninformative ones, cleaning up outliers, etc.).

Evaluate Your System on the Test Set

After tweaking your models for a while, you eventually have a system that performs sufficiently well. Now is the time to evaluate the final model on the test set. There is nothing special about this process; just get the predictors and the labels from your test set, run your `full_pipeline` to transform the data (call `transform()`, *not* `fit_transform()`—you do not want to fit the test set!), and evaluate the final model on the test set:

```
final_model = grid_search.best_estimator_

X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

X_test_prepared = full_pipeline.transform(X_test)
```

```
final_predictions = final_model.predict(X_test_prepared)

final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)   # => evaluates to 47,730.2
```

In some cases, such a point estimate of the generalization error will not be quite enough to convince you to launch: what if it is just 0.1% better than the model currently in production? You might want to have an idea of how precise this estimate is. For this, you can compute a 95% *confidence interval* for the generalization error using `scipy.stats.t.interval()`:

```
>>> from scipy import stats
>>> confidence = 0.95
>>> squared_errors = (final_predictions - y_test) ** 2
>>> np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
...                          loc=squared_errors.mean(),
...                          scale=stats.sem(squared_errors)))
...
array([45685.10470776, 49691.25001878])
```

If you did a lot of hyperparameter tuning, the performance will usually be slightly worse than what you measured using cross-validation (because your system ends up fine-tuned to perform well on the validation data and will likely not perform as well on unknown datasets). It is not the case in this example, but when this happens you must resist the temptation to tweak the hyperparameters to make the numbers look good on the test set; the improvements would be unlikely to generalize to new data.

Now comes the project prelaunch phase: you need to present your solution (highlighting what you have learned, what worked and what did not, what assumptions were made, and what your system's limitations are), document everything, and create nice presentations with clear visualizations and easy-to-remember statements (e.g., "the median income is the number one predictor of housing prices"). In this California housing example, the final performance of the system is not better than the experts' price estimates, which were often off by about 20%, but it may still be a good idea to launch it, especially if this frees up some time for the experts so they can work on more interesting and productive tasks.

Launch, Monitor, and Maintain Your System

Perfect, you got approval to launch! You now need to get your solution ready for production (e.g., polish the code, write documentation and tests, and so on). Then you can deploy your model to your production environment. One way to do this is to save the trained Scikit-Learn model (e.g., using `joblib`), including the full preprocessing and prediction pipeline, then load this trained model within your production environment and use it to make predictions by calling its `predict()` method. For example, perhaps the model will be used within a website: the user will type in some data

about a new district and click the Estimate Price button. This will send a query containing the data to the web server, which will forward it to your web application, and finally your code will simply call the model's `predict()` method (you want to load the model upon server startup, rather than every time the model is used). Alternatively, you can wrap the model within a dedicated web service that your web application can query through a REST API[23] (see Figure 2-17). This makes it easier to upgrade your model to new versions without interrupting the main application. It also simplifies scaling, since you can start as many web services as needed and load-balance the requests coming from your web application across these web services. Moreover, it allows your web application to use any language, not just Python.

Figure 2-17. A model deployed as a web service and used by a web application

Another popular strategy is to deploy your model on the cloud, for example on Google Cloud AI Platform (formerly known as Google Cloud ML Engine): just save your model using `joblib` and upload it to Google Cloud Storage (GCS), then head over to Google Cloud AI Platform and create a new model version, pointing it to the GCS file. That's it! This gives you a simple web service that takes care of load balancing and scaling for you. It take JSON requests containing the input data (e.g., of a district) and returns JSON responses containing the predictions. You can then use this web service in your website (or whatever production environment you are using). As we will see in Chapter 19, deploying TensorFlow models on AI Platform is not much different from deploying Scikit-Learn models.

But deployment is not the end of the story. You also need to write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops. This could be a steep drop, likely due to a broken component in your infrastructure, but be aware that it could also be a gentle decay that could easily go unnoticed for a long time. This is quite common because models tend to "rot" over time: indeed, the world changes, so if the model was trained with last year's data, it may not be adapted to today's data.

23 In a nutshell, a REST (or RESTful) API is an HTTP-based API that follows some conventions, such as using standard HTTP verbs to read, update, create, or delete resources (GET, POST, PUT, and DELETE) and using JSON for the inputs and outputs.

Even a model trained to classify pictures of cats and dogs may need to be retrained regularly, not because cats and dogs will mutate overnight, but because cameras keep changing, along with image formats, sharpness, brightness, and size ratios. Moreover, people may love different breeds next year, or they may decide to dress their pets with tiny hats—who knows?

So you need to monitor your model's live performance. But how do you that? Well, it depends. In some cases, the model's performance can be inferred from downstream metrics. For example, if your model is part of a recommender system and it suggests products that the users may be interested in, then it's easy to monitor the number of recommended products sold each day. If this number drops (compared to non-recommended products), then the prime suspect is the model. This may be because the data pipeline is broken, or perhaps the model needs to be retrained on fresh data (as we will discuss shortly).

However, it's not always possible to determine the model's performance without any human analysis. For example, suppose you trained an image classification model (see Chapter 3) to detect several product defects on a production line. How can you get an alert if the model's performance drops, before thousands of defective products get shipped to your clients? One solution is to send to human raters a sample of all the pictures that the model classified (especially pictures that the model wasn't so sure about). Depending on the task, the raters may need to be experts, or they could be nonspecialists, such as workers on a crowdsourcing platform (e.g., Amazon Mechanical Turk). In some applications they could even be the users themselves, responding for example via surveys or repurposed captchas.[24]

Either way, you need to put in place a monitoring system (with or without human raters to evaluate the live model), as well as all the relevant processes to define what to do in case of failures and how to prepare for them. Unfortunately, this can be a lot of work. In fact, it is often much more work than building and training a model.

If the data keeps evolving, you will need to update your datasets and retrain your model regularly. You should probably automate the whole process as much as possible. Here are a few things you can automate:

- Collect fresh data regularly and label it (e.g., using human raters).
- Write a script to train the model and fine-tune the hyperparameters automatically. This script could run automatically, for example every day or every week, depending on your needs.

24 A captcha is a test to ensure a user is not a robot. These tests have often been used as a cheap way to label training data.

- Write another script that will evaluate both the new model and the previous model on the updated test set, and deploy the model to production if the performance has not decreased (if it did, make sure you investigate why).

You should also make sure you evaluate the model's input data quality. Sometimes performance will degrade slightly because of a poor-quality signal (e.g., a malfunctioning sensor sending random values, or another team's output becoming stale), but it may take a while before your system's performance degrades enough to trigger an alert. If you monitor your model's inputs, you may catch this earlier. For example, you could trigger an alert if more and more inputs are missing a feature, or if its mean or standard deviation drifts too far from the training set, or a categorical feature starts containing new categories.

Finally, make sure you keep backups of every model you create and have the process and tools in place to roll back to a previous model quickly, in case the new model starts failing badly for some reason. Having backups also makes it possible to easily compare new models with previous ones. Similarly, you should keep backups of every version of your datasets so that you can roll back to a previous dataset if the new one ever gets corrupted (e.g., if the fresh data that gets added to it turns out to be full of outliers). Having backups of your datasets also allows you to evaluate any model against any previous dataset.

You may want to create several subsets of the test set in order to evaluate how well your model performs on specific parts of the data. For example, you may want to have a subset containing only the most recent data, or a test set for specific kinds of inputs (e.g., districts located inland versus districts located near the ocean). This will give you a deeper understanding of your model's strengths and weaknesses.

As you can see, Machine Learning involves quite a lot of infrastructure, so don't be surprised if your first ML project takes a lot of effort and time to build and deploy to production. Fortunately, once all the infrastructure is in place, going from idea to production will be much faster.

Try It Out!

Hopefully this chapter gave you a good idea of what a Machine Learning project looks like as well as showing you some of the tools you can use to train a great system. As you can see, much of the work is in the data preparation step: building monitoring tools, setting up human evaluation pipelines, and automating regular model training. The Machine Learning algorithms are important, of course, but it is probably prefera-

ble to be comfortable with the overall process and know three or four algorithms well rather than to spend all your time exploring advanced algorithms.

So, if you have not already done so, now is a good time to pick up a laptop, select a dataset that you are interested in, and try to go through the whole process from A to Z. A good place to start is on a competition website such as *http://kaggle.com/*: you will have a dataset to play with, a clear goal, and people to share the experience with. Have fun!

Exercises

The following exercises are all based on this chapter's housing dataset:

1. Try a Support Vector Machine regressor (`sklearn.svm.SVR`) with various hyperparameters, such as `kernel="linear"` (with various values for the `C` hyperparameter) or `kernel="rbf"` (with various values for the `C` and `gamma` hyperparameters). Don't worry about what these hyperparameters mean for now. How does the best `SVR` predictor perform?
2. Try replacing `GridSearchCV` with `RandomizedSearchCV`.
3. Try adding a transformer in the preparation pipeline to select only the most important attributes.
4. Try creating a single pipeline that does the full data preparation plus the final prediction.
5. Automatically explore some preparation options using `GridSearchCV`.

Solutions to these exercises can be found in the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

CHAPTER 3

Classification

In Chapter 1 I mentioned that the most common supervised learning tasks are regression (predicting values) and classification (predicting classes). In Chapter 2 we explored a regression task, predicting housing values, using various algorithms such as Linear Regression, Decision Trees, and Random Forests (which will be explained in further detail in later chapters). Now we will turn our attention to classification systems.

MNIST

In this chapter we will be using the MNIST dataset, which is a set of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau. Each image is labeled with the digit it represents. This set has been studied so much that it is often called the "hello world" of Machine Learning: whenever people come up with a new classification algorithm they are curious to see how it will perform on MNIST, and anyone who learns Machine Learning tackles this dataset sooner or later.

Scikit-Learn provides many helper functions to download popular datasets. MNIST is one of them. The following code fetches the MNIST dataset:[1]

```
>>> from sklearn.datasets import fetch_openml
>>> mnist = fetch_openml('mnist_784', version=1)
>>> mnist.keys()
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'details',
           'categories', 'url'])
```

1 By default Scikit-Learn caches downloaded datasets in a directory called *$HOME/scikit_learn_data*.

Datasets loaded by Scikit-Learn generally have a similar dictionary structure, including the following:

- A `DESCR` key describing the dataset
- A `data` key containing an array with one row per instance and one column per feature
- A `target` key containing an array with the labels

Let's look at these arrays:

```
>>> X, y = mnist["data"], mnist["target"]
>>> X.shape
(70000, 784)
>>> y.shape
(70000,)
```

There are 70,000 images, and each image has 784 features. This is because each image is 28 × 28 pixels, and each feature simply represents one pixel's intensity, from 0 (white) to 255 (black). Let's take a peek at one digit from the dataset. All you need to do is grab an instance's feature vector, reshape it to a 28 × 28 array, and display it using Matplotlib's `imshow()` function:

```
import matplotlib as mpl
import matplotlib.pyplot as plt

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)

plt.imshow(some_digit_image, cmap="binary")
plt.axis("off")
plt.show()
```

This looks like a 5, and indeed that's what the label tells us:

```
>>> y[0]
'5'
```

Note that the label is a string. Most ML algorithms expect numbers, so let's cast `y` to integer:

```
>>> y = y.astype(np.uint8)
```

To give you a feel for the complexity of the classification task, Figure 3-1 shows a few more images from the MNIST dataset.

Figure 3-1. Digits from the MNIST dataset

But wait! You should always create a test set and set it aside before inspecting the data closely. The MNIST dataset is actually already split into a training set (the first 60,000 images) and a test set (the last 10,000 images):

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

The training set is already shuffled for us, which is good because this guarantees that all cross-validation folds will be similar (you don't want one fold to be missing some digits). Moreover, some learning algorithms are sensitive to the order of the training instances, and they perform poorly if they get many similar instances in a row. Shuffling the dataset ensures that this won't happen.[2]

2 Shuffling may be a bad idea in some contexts—for example, if you are working on time series data (such as stock market prices or weather conditions). We will explore this in the next chapters.

Training a Binary Classifier

Let's simplify the problem for now and only try to identify one digit—for example, the number 5. This "5-detector" will be an example of a *binary classifier*, capable of distinguishing between just two classes, 5 and not-5. Let's create the target vectors for this classification task:

```
y_train_5 = (y_train == 5)  # True for all 5s, False for all other digits
y_test_5 = (y_test == 5)
```

Now let's pick a classifier and train it. A good place to start is with a *Stochastic Gradient Descent* (SGD) classifier, using Scikit-Learn's `SGDClassifier` class. This classifier has the advantage of being capable of handling very large datasets efficiently. This is in part because SGD deals with training instances independently, one at a time (which also makes SGD well suited for online learning), as we will see later. Let's create an `SGDClassifier` and train it on the whole training set:

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

The `SGDClassifier` relies on randomness during training (hence the name "stochastic"). If you want reproducible results, you should set the `random_state` parameter.

Now we can use it to detect images of the number 5:

```
>>> sgd_clf.predict([some_digit])
array([ True])
```

The classifier guesses that this image represents a 5 (`True`). Looks like it guessed right in this particular case! Now, let's evaluate this model's performance.

Performance Measures

Evaluating a classifier is often significantly trickier than evaluating a regressor, so we will spend a large part of this chapter on this topic. There are many performance measures available, so grab another coffee and get ready to learn many new concepts and acronyms!

Measuring Accuracy Using Cross-Validation

A good way to evaluate a model is to use cross-validation, just as you did in Chapter 2.

Implementing Cross-Validation

Occasionally you will need more control over the cross-validation process than what Scikit-Learn provides off the shelf. In these cases, you can implement cross-validation yourself. The following code does roughly the same thing as Scikit-Learn's `cross_val_score()` function, and it prints the same result:

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3, random_state=42)

for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred))  # prints 0.9502, 0.96565, and 0.96495
```

The `StratifiedKFold` class performs stratified sampling (as explained in Chapter 2) to produce folds that contain a representative ratio of each class. At each iteration the code creates a clone of the classifier, trains that clone on the training folds, and makes predictions on the test fold. Then it counts the number of correct predictions and outputs the ratio of correct predictions.

Let's use the `cross_val_score()` function to evaluate our `SGDClassifier` model, using K-fold cross-validation with three folds. Remember that K-fold cross-validation means splitting the training set into K folds (in this case, three), then making predictions and evaluating them on each fold using a model trained on the remaining folds (see Chapter 2):

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.96355, 0.93795, 0.95615])
```

Wow! Above 93% accuracy (ratio of correct predictions) on all cross-validation folds? This looks amazing, doesn't it? Well, before you get too excited, let's look at a very dumb classifier that just classifies every single image in the "not-5" class:

```
from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        return self
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

Can you guess this model's accuracy? Let's find out:

```
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.91125, 0.90855, 0.90915])
```

That's right, it has over 90% accuracy! This is simply because only about 10% of the images are 5s, so if you always guess that an image is *not* a 5, you will be right about 90% of the time. Beats Nostradamus.

This demonstrates why accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with *skewed datasets* (i.e., when some classes are much more frequent than others).

Confusion Matrix

A much better way to evaluate the performance of a classifier is to look at the *confusion matrix*. The general idea is to count the number of times instances of class A are classified as class B. For example, to know the number of times the classifier confused images of 5s with 3s, you would look in the fifth row and third column of the confusion matrix.

To compute the confusion matrix, you first need to have a set of predictions so that they can be compared to the actual targets. You could make predictions on the test set, but let's keep it untouched for now (remember that you want to use the test set only at the very end of your project, once you have a classifier that you are ready to launch). Instead, you can use the `cross_val_predict()` function:

```
from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

Just like the `cross_val_score()` function, `cross_val_predict()` performs K-fold cross-validation, but instead of returning the evaluation scores, it returns the predictions made on each test fold. This means that you get a clean prediction for each instance in the training set ("clean" meaning that the prediction is made by a model that never saw the data during training).

Now you are ready to get the confusion matrix using the `confusion_matrix()` function. Just pass it the target classes (`y_train_5`) and the predicted classes (`y_train_pred`):

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53057,  1522],
       [ 1325,  4096]])
```

Each row in a confusion matrix represents an *actual class*, while each column represents a *predicted class*. The first row of this matrix considers non-5 images (the *negative class*): 53,057 of them were correctly classified as non-5s (they are called *true negatives*), while the remaining 1,522 were wrongly classified as 5s (*false positives*). The second row considers the images of 5s (the *positive class*): 1,325 were wrongly classified as non-5s (*false negatives*), while the remaining 4,096 were correctly classified as 5s (*true positives*). A perfect classifier would have only true positives and true negatives, so its confusion matrix would have nonzero values only on its main diagonal (top left to bottom right):

```
>>> y_train_perfect_predictions = y_train_5  # pretend we reached perfection
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579,     0],
       [    0,  5421]])
```

The confusion matrix gives you a lot of information, but sometimes you may prefer a more concise metric. An interesting one to look at is the accuracy of the positive predictions; this is called the *precision* of the classifier (Equation 3-1).

Equation 3-1. Precision

$$\text{precision} = \frac{TP}{TP + FP}$$

TP is the number of true positives, and *FP* is the number of false positives.

A trivial way to have perfect precision is to make one single positive prediction and ensure it is correct (precision = 1/1 = 100%). But this would not be very useful, since the classifier would ignore all but one positive instance. So precision is typically used along with another metric named *recall*, also called *sensitivity* or the *true positive rate* (TPR): this is the ratio of positive instances that are correctly detected by the classifier (Equation 3-2).

Equation 3-2. Recall

$$\text{recall} = \frac{TP}{TP + FN}$$

FN is, of course, the number of false negatives.

If you are confused about the confusion matrix, Figure 3-2 may help.

Figure 3-2. An illustrated confusion matrix shows examples of true negatives (top left), false positives (top right), false negatives (lower left), and true positives (lower right)

Precision and Recall

Scikit-Learn provides several functions to compute classifier metrics, including precision and recall:

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1522)
0.7290850836596654
>>> recall_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1325)
0.7555801512636044
```

Now your 5-detector does not look as shiny as it did when you looked at its accuracy. When it claims an image represents a 5, it is correct only 72.9% of the time. Moreover, it only detects 75.6% of the 5s.

It is often convenient to combine precision and recall into a single metric called the F_1 *score*, in particular if you need a simple way to compare two classifiers. The F_1 score is the *harmonic mean* of precision and recall (Equation 3-3). Whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values. As a result, the classifier will only get a high F_1 score if both recall and precision are high.

Equation 3-3. F_1

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

To compute the F_1 score, simply call the `f1_score()` function:

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7420962043663375
```

The F_1 score favors classifiers that have similar precision and recall. This is not always what you want: in some contexts you mostly care about precision, and in other contexts you really care about recall. For example, if you trained a classifier to detect videos that are safe for kids, you would probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision), rather than a classifier that has a much higher recall but lets a few really bad videos show up in your product (in such cases, you may even want to add a human pipeline to check the classifier's video selection). On the other hand, suppose you train a classifier to detect shoplifters in surveillance images: it is probably fine if your classifier has only 30% precision as long as it has 99% recall (sure, the security guards will get a few false alerts, but almost all shoplifters will get caught).

Unfortunately, you can't have it both ways: increasing precision reduces recall, and vice versa. This is called the *precision/recall trade-off*.

Precision/Recall Trade-off

To understand this trade-off, let's look at how the `SGDClassifier` makes its classification decisions. For each instance, it computes a score based on a *decision function*. If that score is greater than a threshold, it assigns the instance to the positive class; otherwise it assigns it to the negative class. Figure 3-3 shows a few digits positioned from the lowest score on the left to the highest score on the right. Suppose the *decision threshold* is positioned at the central arrow (between the two 5s): you will find 4 true positives (actual 5s) on the right of that threshold, and 1 false positive (actually a 6). Therefore, with that threshold, the precision is 80% (4 out of 5). But out of 6 actual 5s, the classifier only detects 4, so the recall is 67% (4 out of 6). If you raise the threshold (move it to the arrow on the right), the false positive (the 6) becomes a true negative, thereby increasing the precision (up to 100% in this case), but one true positive becomes a false negative, decreasing recall down to 50%. Conversely, lowering the threshold increases recall and reduces precision.

Figure 3-3. In this precision/recall trade-off, images are ranked by their classifier score, and those above the chosen decision threshold are considered positive; the higher the threshold, the lower the recall, but (in general) the higher the precision

Scikit-Learn does not let you set the threshold directly, but it does give you access to the decision scores that it uses to make predictions. Instead of calling the classifier's `predict()` method, you can call its `decision_function()` method, which returns a score for each instance, and then use any threshold you want to make predictions based on those scores:

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([2412.53175101])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True])
```

The `SGDClassifier` uses a threshold equal to 0, so the previous code returns the same result as the `predict()` method (i.e., `True`). Let's raise the threshold:

```
>>> threshold = 8000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False])
```

This confirms that raising the threshold decreases recall. The image actually represents a 5, and the classifier detects it when the threshold is 0, but it misses it when the threshold is increased to 8,000.

How do you decide which threshold to use? First, use the `cross_val_predict()` function to get the scores of all instances in the training set, but this time specify that you want to return decision scores instead of predictions:

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

With these scores, use the `precision_recall_curve()` function to compute precision and recall for all possible thresholds:

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

Finally, use Matplotlib to plot precision and recall as functions of the threshold value (Figure 3-4):

```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    [...] # highlight the threshold and add the legend, axis label, and grid

plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```

Figure 3-4. Precision and recall versus the decision threshold

You may wonder why the precision curve is bumpier than the recall curve in Figure 3-4. The reason is that precision may sometimes go down when you raise the threshold (although in general it will go up). To understand why, look back at Figure 3-3 and notice what happens when you start from the central threshold and move it just one digit to the right: precision goes from 4/5 (80%) down to 3/4 (75%). On the other hand, recall can only go down when the threshold is increased, which explains why its curve looks smooth.

Another way to select a good precision/recall trade-off is to plot precision directly against recall, as shown in Figure 3-5 (the same threshold as earlier is highlighted).

Figure 3-5. Precision versus recall

You can see that precision really starts to fall sharply around 80% recall. You will probably want to select a precision/recall trade-off just before that drop—for example, at around 60% recall. But of course, the choice depends on your project.

Suppose you decide to aim for 90% precision. You look up the first plot and find that you need to use a threshold of about 8,000. To be more precise you can search for the lowest threshold that gives you at least 90% precision (`np.argmax()` will give you the first index of the maximum value, which in this case means the first `True` value):

```
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)] # ~7816
```

To make predictions (on the training set for now), instead of calling the classifier's `predict()` method, you can run this code:

```
y_train_pred_90 = (y_scores >= threshold_90_precision)
```

Let's check these predictions' precision and recall:

```
>>> precision_score(y_train_5, y_train_pred_90)
0.9000380083618396
>>> recall_score(y_train_5, y_train_pred_90)
0.4368197749492714
```

Great, you have a 90% precision classifier! As you can see, it is fairly easy to create a classifier with virtually any precision you want: just set a high enough threshold, and you're done. But wait, not so fast. A high-precision classifier is not very useful if its recall is too low!

If someone says, "Let's reach 99% precision," you should ask, "At what recall?"

The ROC Curve

The *receiver operating characteristic* (ROC) curve is another common tool used with binary classifiers. It is very similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve plots the *true positive rate* (another name for recall) against the *false positive rate* (FPR). The FPR is the ratio of negative instances that are incorrectly classified as positive. It is equal to 1 – the *true negative rate* (TNR), which is the ratio of negative instances that are correctly classified as negative. The TNR is also called *specificity*. Hence, the ROC curve plots *sensitivity* (recall) versus 1 – *specificity*.

To plot the ROC curve, you first use the `roc_curve()` function to compute the TPR and FPR for various threshold values:

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

Then you can plot the FPR against the TPR using Matplotlib. This code produces the plot in Figure 3-6:

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--') # Dashed diagonal
    [...] # Add axis labels and grid

plot_roc_curve(fpr, tpr)
plt.show()
```

Once again there is a trade-off: the higher the recall (TPR), the more false positives (FPR) the classifier produces. The dotted line represents the ROC curve of a purely random classifier; a good classifier stays as far away from that line as possible (toward the top-left corner).

Figure 3-6. This ROC curve plots the false positive rate against the true positive rate for all possible thresholds; the red circle highlights the chosen ratio (at 43.68% recall)

One way to compare classifiers is to measure the *area under the curve* (AUC). A perfect classifier will have a ROC AUC equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5. Scikit-Learn provides a function to compute the ROC AUC:

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(y_train_5, y_scores)
0.9611778893101814
```

Since the ROC curve is so similar to the precision/recall (PR) curve, you may wonder how to decide which one to use. As a rule of thumb, you should prefer the PR curve whenever the positive class is rare or when you care more about the false positives than the false negatives. Otherwise, use the ROC curve. For example, looking at the previous ROC curve (and the ROC AUC score), you may think that the classifier is really good. But this is mostly because there are few positives (5s) compared to the negatives (non-5s). In contrast, the PR curve makes it clear that the classifier has room for improvement (the curve could be closer to the top-left corner).

Let's now train a `RandomForestClassifier` and compare its ROC curve and ROC AUC score to those of the `SGDClassifier`. First, you need to get scores for each instance in the training set. But due to the way it works (see Chapter 7), the `RandomForestClassifier` class does not have a `decision_function()` method. Instead, it

has a `predict_proba()` method. Scikit-Learn classifiers generally have one or the other, or both. The `predict_proba()` method returns an array containing a row per instance and a column per class, each containing the probability that the given instance belongs to the given class (e.g., 70% chance that the image represents a 5):

```
from sklearn.ensemble import RandomForestClassifier

forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                    method="predict_proba")
```

The `roc_curve()` function expects labels and scores, but instead of scores you can give it class probabilities. Let's use the positive class's probability as the score:

```
y_scores_forest = y_probas_forest[:, 1]   # score = proba of positive class
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
```

Now you are ready to plot the ROC curve. It is useful to plot the first ROC curve as well to see how they compare (Figure 3-7):

```
plt.plot(fpr, tpr, "b:", label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.legend(loc="lower right")
plt.show()
```

Figure 3-7. Comparing ROC curves: the Random Forest classifier is superior to the SGD classifier because its ROC curve is much closer to the top-left corner, and it has a greater AUC

As you can see in Figure 3-7, the `RandomForestClassifier`'s ROC curve looks much better than the `SGDClassifier`'s: it comes much closer to the top-left corner. As a result, its ROC AUC score is also significantly better:

```
>>> roc_auc_score(y_train_5, y_scores_forest)
0.9983436731328145
```

Try measuring the precision and recall scores: you should find 99.0% precision and 86.6% recall. Not too bad!

You now know how to train binary classifiers, choose the appropriate metric for your task, evaluate your classifiers using cross-validation, select the precision/recall trade-off that fits your needs, and use ROC curves and ROC AUC scores to compare various models. Now let's try to detect more than just the 5s.

Multiclass Classification

Whereas binary classifiers distinguish between two classes, *multiclass classifiers* (also called *multinomial classifiers*) can distinguish between more than two classes.

Some algorithms (such as SGD classifiers, Random Forest classifiers, and naive Bayes classifiers) are capable of handling multiple classes natively. Others (such as Logistic Regression or Support Vector Machine classifiers) are strictly binary classifiers. However, there are various strategies that you can use to perform multiclass classification with multiple binary classifiers.

One way to create a system that can classify the digit images into 10 classes (from 0 to 9) is to train 10 binary classifiers, one for each digit (a 0-detector, a 1-detector, a 2-detector, and so on). Then when you want to classify an image, you get the decision score from each classifier for that image and you select the class whose classifier outputs the highest score. This is called the *one-versus-the-rest* (OvR) strategy (also called *one-versus-all*).

Another strategy is to train a binary classifier for every pair of digits: one to distinguish 0s and 1s, another to distinguish 0s and 2s, another for 1s and 2s, and so on. This is called the *one-versus-one* (OvO) strategy. If there are N classes, you need to train $N \times (N - 1) / 2$ classifiers. For the MNIST problem, this means training 45 binary classifiers! When you want to classify an image, you have to run the image through all 45 classifiers and see which class wins the most duels. The main advantage of OvO is that each classifier only needs to be trained on the part of the training set for the two classes that it must distinguish.

Some algorithms (such as Support Vector Machine classifiers) scale poorly with the size of the training set. For these algorithms OvO is preferred because it is faster to train many classifiers on small training sets than to train few classifiers on large training sets. For most binary classification algorithms, however, OvR is preferred.

Scikit-Learn detects when you try to use a binary classification algorithm for a multiclass classification task, and it automatically runs OvR or OvO, depending on the algorithm. Let's try this with a Support Vector Machine classifier (see Chapter 5), using the `sklearn.svm.SVC` class:

```
>>> from sklearn.svm import SVC
>>> svm_clf = SVC()
>>> svm_clf.fit(X_train, y_train) # y_train, not y_train_5
>>> svm_clf.predict([some_digit])
array([5], dtype=uint8)
```

That was easy! This code trains the `SVC` on the training set using the original target classes from 0 to 9 (`y_train`), instead of the 5-versus-the-rest target classes (`y_train_5`). Then it makes a prediction (a correct one in this case). Under the hood, Scikit-Learn actually used the OvO strategy: it trained 45 binary classifiers, got their decision scores for the image, and selected the class that won the most duels.

If you call the `decision_function()` method, you will see that it returns 10 scores per instance (instead of just 1). That's one score per class:

```
>>> some_digit_scores = svm_clf.decision_function([some_digit])
>>> some_digit_scores
array([[ 2.92492871,  7.02307409,  3.93648529,  0.90117363,  5.96945908,
         9.5       ,  1.90718593,  8.02755089, -0.13202708,  4.94216947]])
```

The highest score is indeed the one corresponding to class `5`:

```
>>> np.argmax(some_digit_scores)
5
>>> svm_clf.classes_
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
>>> svm_clf.classes_[5]
5
```

When a classifier is trained, it stores the list of target classes in its `classes_` attribute, ordered by value. In this case, the index of each class in the `classes_` array conveniently matches the class itself (e.g., the class at index 5 happens to be class `5`), but in general you won't be so lucky.

If you want to force Scikit-Learn to use one-versus-one or one-versus-the-rest, you can use the `OneVsOneClassifier` or `OneVsRestClassifier` classes. Simply create an instance and pass a classifier to its constructor (it does not even have to be a binary classifier). For example, this code creates a multiclass classifier using the OvR strategy, based on an `SVC`:

```
>>> from sklearn.multiclass import OneVsRestClassifier
>>> ovr_clf = OneVsRestClassifier(SVC())
>>> ovr_clf.fit(X_train, y_train)
```

```
>>> ovr_clf.predict([some_digit])
array([5], dtype=uint8)
>>> len(ovr_clf.estimators_)
10
```

Training an `SGDClassifier` (or a `RandomForestClassifier`) is just as easy:

```
>>> sgd_clf.fit(X_train, y_train)
>>> sgd_clf.predict([some_digit])
array([5], dtype=uint8)
```

This time Scikit-Learn did not have to run OvR or OvO because SGD classifiers can directly classify instances into multiple classes. The `decision_function()` method now returns one value per class. Let's look at the score that the SGD classifier assigned to each class:

```
>>> sgd_clf.decision_function([some_digit])
array([[-15955.22628, -38080.96296, -13326.66695,   573.52692, -17680.68466,
          2412.53175, -25526.86498, -12290.15705, -7946.05205, -10631.35889]])
```

You can see that the classifier is fairly confident about its prediction: almost all scores are largely negative, while class `5` has a score of 2412.5. The model has a slight doubt regarding class `3`, which gets a score of 573.5. Now of course you want to evaluate this classifier. As usual, you can use cross-validation. Use the `cross_val_score()` function to evaluate the `SGDClassifier`'s accuracy:

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([0.8489802 , 0.87129356, 0.86988048])
```

It gets over 84% on all test folds. If you used a random classifier, you would get 10% accuracy, so this is not such a bad score, but you can still do much better. Simply scaling the inputs (as discussed in Chapter 2) increases accuracy above 89%:

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
>>> cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
array([0.89707059, 0.8960948 , 0.90693604])
```

Error Analysis

If this were a real project, you would now follow the steps in your Machine Learning project checklist (see Appendix B). You'd explore data preparation options, try out multiple models (shortlisting the best ones and fine-tuning their hyperparameters using `GridSearchCV`), and automate as much as possible. Here, we will assume that you have found a promising model and you want to find ways to improve it. One way to do this is to analyze the types of errors it makes.

First, look at the confusion matrix. You need to make predictions using the `cross_val_predict()` function, then call the `confusion_matrix()` function, just like you did earlier:

```
>>> y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
>>> conf_mx = confusion_matrix(y_train, y_train_pred)
>>> conf_mx
array([[5578,    0,   22,    7,    8,   45,   35,    5,  222,    1],
       [   0, 6410,   35,   26,    4,   44,    4,    8,  198,   13],
       [  28,   27, 5232,  100,   74,   27,   68,   37,  354,   11],
       [  23,   18,  115, 5254,    2,  209,   26,   38,  373,   73],
       [  11,   14,   45,   12, 5219,   11,   33,   26,  299,  172],
       [  26,   16,   31,  173,   54, 4484,   76,   14,  482,   65],
       [  31,   17,   45,    2,   42,   98, 5556,    3,  123,    1],
       [  20,   10,   53,   27,   50,   13,    3, 5696,  173,  220],
       [  17,   64,   47,   91,    3,  125,   24,   11, 5421,   48],
       [  24,   18,   29,   67,  116,   39,    1,  174,  329, 5152]])
```

That's a lot of numbers. It's often more convenient to look at an image representation of the confusion matrix, using Matplotlib's `matshow()` function:

```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```

This confusion matrix looks pretty good, since most images are on the main diagonal, which means that they were classified correctly. The 5s look slightly darker than the other digits, which could mean that there are fewer images of 5s in the dataset or that the classifier does not perform as well on 5s as on other digits. In fact, you can verify that both are the case.

Let's focus the plot on the errors. First, you need to divide each value in the confusion matrix by the number of images in the corresponding class so that you can compare error rates instead of absolute numbers of errors (which would make abundant classes look unfairly bad):

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
```

Fill the diagonal with zeros to keep only the errors, and plot the result:

```
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```

You can clearly see the kinds of errors the classifier makes. Remember that rows represent actual classes, while columns represent predicted classes. The column for class 8 is quite bright, which tells you that many images get misclassified as 8s. However, the row for class 8 is not that bad, telling you that actual 8s in general get properly classified as 8s. As you can see, the confusion matrix is not necessarily symmetrical. You can also see that 3s and 5s often get confused (in both directions).

Analyzing the confusion matrix often gives you insights into ways to improve your classifier. Looking at this plot, it seems that your efforts should be spent on reducing the false 8s. For example, you could try to gather more training data for digits that look like 8s (but are not) so that the classifier can learn to distinguish them from real 8s. Or you could engineer new features that would help the classifier—for example, writing an algorithm to count the number of closed loops (e.g., 8 has two, 6 has one, 5 has none). Or you could preprocess the images (e.g., using Scikit-Image, Pillow, or OpenCV) to make some patterns, such as closed loops, stand out more.

Analyzing individual errors can also be a good way to gain insights on what your classifier is doing and why it is failing, but it is more difficult and time-consuming. For example, let's plot examples of 3s and 5s (the `plot_digits()` function just uses Matplotlib's `imshow()` function; see this chapter's Jupyter notebook for details):

```
cl_a, cl_b = 3, 5
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]
```

```
plt.figure(figsize=(8,8))
plt.subplot(221); plot_digits(X_aa[:25], images_per_row=5)
plt.subplot(222); plot_digits(X_ab[:25], images_per_row=5)
plt.subplot(223); plot_digits(X_ba[:25], images_per_row=5)
plt.subplot(224); plot_digits(X_bb[:25], images_per_row=5)
plt.show()
```

The two 5 × 5 blocks on the left show digits classified as 3s, and the two 5 × 5 blocks on the right show images classified as 5s. Some of the digits that the classifier gets wrong (i.e., in the bottom-left and top-right blocks) are so badly written that even a human would have trouble classifying them (e.g., the 5 in the first row and second column truly looks like a badly written 3). However, most misclassified images seem like obvious errors to us, and it's hard to understand why the classifier made the mistakes it did.[3] The reason is that we used a simple `SGDClassifier`, which is a linear model. All it does is assign a weight per class to each pixel, and when it sees a new image it just sums up the weighted pixel intensities to get a score for each class. So since 3s and 5s differ only by a few pixels, this model will easily confuse them.

The main difference between 3s and 5s is the position of the small line that joins the top line to the bottom arc. If you draw a 3 with the junction slightly shifted to the left, the classifier might classify it as a 5, and vice versa. In other words, this classifier is quite sensitive to image shifting and rotation. So one way to reduce the 3/5 confusion would be to preprocess the images to ensure that they are well centered and not too rotated. This will probably help reduce other errors as well.

3 But remember that our brain is a fantastic pattern recognition system, and our visual system does a lot of complex preprocessing before any information reaches our consciousness, so the fact that it feels simple does not mean that it is.

Multilabel Classification

Until now each instance has always been assigned to just one class. In some cases you may want your classifier to output multiple classes for each instance. Consider a face-recognition classifier: what should it do if it recognizes several people in the same picture? It should attach one tag per person it recognizes. Say the classifier has been trained to recognize three faces, Alice, Bob, and Charlie. Then when the classifier is shown a picture of Alice and Charlie, it should output [1, 0, 1] (meaning "Alice yes, Bob no, Charlie yes"). Such a classification system that outputs multiple binary tags is called a *multilabel classification* system.

We won't go into face recognition just yet, but let's look at a simpler example, just for illustration purposes:

```
from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

This code creates a `y_multilabel` array containing two target labels for each digit image: the first indicates whether or not the digit is large (7, 8, or 9), and the second indicates whether or not it is odd. The next lines create a `KNeighborsClassifier` instance (which supports multilabel classification, though not all classifiers do), and we train it using the multiple targets array. Now you can make a prediction, and notice that it outputs two labels:

```
>>> knn_clf.predict([some_digit])
array([[False,  True]])
```

And it gets it right! The digit 5 is indeed not large (`False`) and odd (`True`).

There are many ways to evaluate a multilabel classifier, and selecting the right metric really depends on your project. One approach is to measure the F_1 score for each individual label (or any other binary classifier metric discussed earlier), then simply compute the average score. This code computes the average F_1 score across all labels:

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel, cv=3)
>>> f1_score(y_multilabel, y_train_knn_pred, average="macro")
0.976410265560605
```

This assumes that all labels are equally important, however, which may not be the case. In particular, if you have many more pictures of Alice than of Bob or Charlie, you may want to give more weight to the classifier's score on pictures of Alice. One simple option is to give each label a weight equal to its *support* (i.e., the number of

instances with that target label). To do this, simply set `average="weighted"` in the preceding code.[4]

Multioutput Classification

The last type of classification task we are going to discuss here is called *multioutput–multiclass classification* (or simply *multioutput classification*). It is simply a generalization of multilabel classification where each label can be multiclass (i.e., it can have more than two possible values).

To illustrate this, let's build a system that removes noise from images. It will take as input a noisy digit image, and it will (hopefully) output a clean digit image, represented as an array of pixel intensities, just like the MNIST images. Notice that the classifier's output is multilabel (one label per pixel) and each label can have multiple values (pixel intensity ranges from 0 to 255). It is thus an example of a multioutput classification system.

The line between classification and regression is sometimes blurry, such as in this example. Arguably, predicting pixel intensity is more akin to regression than to classification. Moreover, multioutput systems are not limited to classification tasks; you could even have a system that outputs multiple labels per instance, including both class labels and value labels.

Let's start by creating the training and test sets by taking the MNIST images and adding noise to their pixel intensities with NumPy's `randint()` function. The target images will be the original images:

```
noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

Let's take a peek at an image from the test set (yes, we're snooping on the test data, so you should be frowning right now):

4 Scikit-Learn offers a few other averaging options and multilabel classifier metrics; see the documentation for more details.

On the left is the noisy input image, and on the right is the clean target image. Now let's train the classifier and make it clean this image:

```
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[some_index]])
plot_digit(clean_digit)
```

Looks close enough to the target! This concludes our tour of classification. You should now know how to select good metrics for classification tasks, pick the appropriate precision/recall trade-off, compare classifiers, and more generally build good classification systems for a variety of tasks.

Exercises

1. Try to build a classifier for the MNIST dataset that achieves over 97% accuracy on the test set. Hint: the `KNeighborsClassifier` works quite well for this task; you just need to find good hyperparameter values (try a grid search on the `weights` and `n_neighbors` hyperparameters).
2. Write a function that can shift an MNIST image in any direction (left, right, up, or down) by one pixel.[5] Then, for each image in the training set, create four shifted copies (one per direction) and add them to the training set. Finally, train your best model on this expanded training set and measure its accuracy on the test set. You should observe that your model performs even better now! This technique of artificially growing the training set is called *data augmentation* or *training set expansion*.

5 You can use the `shift()` function from the `scipy.ndimage.interpolation` module. For example, `shift(image, [2, 1], cval=0)` shifts the image two pixels down and one pixel to the right.

3. Tackle the Titanic dataset. A great place to start is on Kaggle.
4. Build a spam classifier (a more challenging exercise):
   - Download examples of spam and ham from Apache SpamAssassin's public datasets.
   - Unzip the datasets and familiarize yourself with the data format.
   - Split the datasets into a training set and a test set.
   - Write a data preparation pipeline to convert each email into a feature vector. Your preparation pipeline should transform an email into a (sparse) vector that indicates the presence or absence of each possible word. For example, if all emails only ever contain four words, "Hello," "how," "are," "you," then the email "Hello you Hello Hello you" would be converted into a vector [1, 0, 0, 1] (meaning ["Hello" is present, "how" is absent, "are" is absent, "you" is present]), or [3, 0, 0, 2] if you prefer to count the number of occurrences of each word.

     You may want to add hyperparameters to your preparation pipeline to control whether or not to strip off email headers, convert each email to lowercase, remove punctuation, replace all URLs with "URL," replace all numbers with "NUMBER," or even perform *stemming* (i.e., trim off word endings; there are Python libraries available to do this).

     Finally, try out several classifiers and see if you can build a great spam classifier, with both high recall and high precision.

Solutions to these exercises can be found in the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

CHAPTER 4
Training Models

So far we have treated Machine Learning models and their training algorithms mostly like black boxes. If you went through some of the exercises in the previous chapters, you may have been surprised by how much you can get done without knowing anything about what's under the hood: you optimized a regression system, you improved a digit image classifier, and you even built a spam classifier from scratch, all this without knowing how they actually work. Indeed, in many situations you don't really need to know the implementation details.

However, having a good understanding of how things work can help you quickly home in on the appropriate model, the right training algorithm to use, and a good set of hyperparameters for your task. Understanding what's under the hood will also help you debug issues and perform error analysis more efficiently. Lastly, most of the topics discussed in this chapter will be essential in understanding, building, and training neural networks (discussed in Part II of this book).

In this chapter we will start by looking at the Linear Regression model, one of the simplest models there is. We will discuss two very different ways to train it:

- Using a direct "closed-form" equation that directly computes the model parameters that best fit the model to the training set (i.e., the model parameters that minimize the cost function over the training set).
- Using an iterative optimization approach called Gradient Descent (GD) that gradually tweaks the model parameters to minimize the cost function over the training set, eventually converging to the same set of parameters as the first method. We will look at a few variants of Gradient Descent that we will use again and again when we study neural networks in Part II: Batch GD, Mini-batch GD, and Stochastic GD.

Next we will look at Polynomial Regression, a more complex model that can fit nonlinear datasets. Since this model has more parameters than Linear Regression, it is more prone to overfitting the training data, so we will look at how to detect whether or not this is the case using learning curves, and then we will look at several regularization techniques that can reduce the risk of overfitting the training set.

Finally, we will look at two more models that are commonly used for classification tasks: Logistic Regression and Softmax Regression.

There will be quite a few math equations in this chapter, using basic notions of linear algebra and calculus. To understand these equations, you will need to know what vectors and matrices are; how to transpose them, multiply them, and inverse them; and what partial derivatives are. If you are unfamiliar with these concepts, please go through the linear algebra and calculus introductory tutorials available as Jupyter notebooks in the online supplemental material. For those who are truly allergic to mathematics, you should still go through this chapter and simply skip the equations; hopefully, the text will be sufficient to help you understand most of the concepts.

Linear Regression

In Chapter 1 we looked at a simple regression model of life satisfaction: *life_satisfaction* $= \theta_0 + \theta_1 \times$ *GDP_per_capita*.

This model is just a linear function of the input feature `GDP_per_capita`. θ_0 and θ_1 are the model's parameters.

More generally, a linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the *bias term* (also called the *intercept term*), as shown in Equation 4-1.

Equation 4-1. Linear Regression model prediction

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

In this equation:

- $\hat{y}$ is the predicted value.
- n is the number of features.
- x_i is the ith feature value.
- θ_j is the jth model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \cdots, \theta_n$).

This can be written much more concisely using a vectorized form, as shown in Equation 4-2.

Equation 4-2. Linear Regression model prediction (vectorized form)

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

In this equation:

- $\boldsymbol{\theta}$ is the model's *parameter vector*, containing the bias term θ_0 and the feature weights θ_1 to θ_n.
- $\mathbf{x}$ is the instance's *feature vector*, containing x_0 to x_n, with x_0 always equal to 1.
- $\boldsymbol{\theta} \cdot \mathbf{x}$ is the dot product of the vectors $\boldsymbol{\theta}$ and $\mathbf{x}$, which is of course equal to $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$.
- $h_{\boldsymbol{\theta}}$ is the hypothesis function, using the model parameters $\boldsymbol{\theta}$.

In Machine Learning, vectors are often represented as *column vectors*, which are 2D arrays with a single column. If $\boldsymbol{\theta}$ and $\mathbf{x}$ are column vectors, then the prediction is $\hat{y} = \boldsymbol{\theta}^\top \mathbf{x}$, where $\boldsymbol{\theta}^\top$ is the *transpose* of $\boldsymbol{\theta}$ (a row vector instead of a column vector) and $\boldsymbol{\theta}^\top \mathbf{x}$ is the matrix multiplication of $\boldsymbol{\theta}^\top$ and $\mathbf{x}$. It is of course the same prediction, except that it is now represented as a single-cell matrix rather than a scalar value. In this book I will use this notation to avoid switching between dot products and matrix multiplications.

OK, that's the Linear Regression model—but how do we train it? Well, recall that training a model means setting its parameters so that the model best fits the training set. For this purpose, we first need a measure of how well (or poorly) the model fits the training data. In Chapter 2 we saw that the most common performance measure of a regression model is the Root Mean Square Error (RMSE) (Equation 2-1). Therefore, to train a Linear Regression model, we need to find the value of $\boldsymbol{\theta}$ that minimizes the RMSE. In practice, it is simpler to minimize the mean squared error (MSE) than the RMSE, and it leads to the same result (because the value that minimizes a function also minimizes its square root).[1]

1 It is often the case that a learning algorithm will try to optimize a different function than the performance measure used to evaluate the final model. This is generally because that function is easier to compute, because it has useful differentiation properties that the performance measure lacks, or because we want to constrain the model during training, as you will see when we discuss regularization.

The MSE of a Linear Regression hypothesis $h_{\boldsymbol{\theta}}$ on a training set $\mathbf{X}$ is calculated using Equation 4-3.

Equation 4-3. MSE cost function for a Linear Regression model

$$\mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^{\top} \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

Most of these notations were presented in Chapter 2 (see "Notations" on page 40). The only difference is that we write $h_{\boldsymbol{\theta}}$ instead of just h to make it clear that the model is parametrized by the vector $\boldsymbol{\theta}$. To simplify notations, we will just write $\mathrm{MSE}(\boldsymbol{\theta})$ instead of $\mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}})$.

The Normal Equation

To find the value of $\boldsymbol{\theta}$ that minimizes the cost function, there is a *closed-form solution*—in other words, a mathematical equation that gives the result directly. This is called the *Normal Equation* (Equation 4-4).

Equation 4-4. Normal Equation

$$\widehat{\boldsymbol{\theta}} = \left(\mathbf{X}^{\top}\mathbf{X}\right)^{-1} \mathbf{X}^{\top} \mathbf{y}$$

In this equation:

- $\widehat{\boldsymbol{\theta}}$ is the value of $\boldsymbol{\theta}$ that minimizes the cost function.
- $\mathbf{y}$ is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

Let's generate some linear-looking data to test this equation on (Figure 4-1):

```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

Figure 4-1. Randomly generated linear dataset

Now let's compute $\widehat{\boldsymbol{\theta}}$ using the Normal Equation. We will use the `inv()` function from NumPy's linear algebra module (`np.linalg`) to compute the inverse of a matrix, and the `dot()` method for matrix multiplication:

```
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

The function that we used to generate the data is $y = 4 + 3x_1 +$ Gaussian noise. Let's see what the equation found:

```
>>> theta_best
array([[4.21509616],
       [2.77011339]])
```

We would have hoped for $\theta_0 = 4$ and $\theta_1 = 3$ instead of $\theta_0 = 4.215$ and $\theta_1 = 2.770$. Close enough, but the noise made it impossible to recover the exact parameters of the original function.

Now we can make predictions using $\widehat{\boldsymbol{\theta}}$:

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new] # add x0 = 1 to each instance
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[4.21509616],
       [9.75532293]])
```

Let's plot this model's predictions (Figure 4-2):

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

Figure 4-2. Linear Regression model predictions

Performing Linear Regression using Scikit-Learn is simple:[2]

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([4.21509616]), array([[2.77011339]]))
>>> lin_reg.predict(X_new)
array([[4.21509616],
       [9.75532293]])
```

The `LinearRegression` class is based on the `scipy.linalg.lstsq()` function (the name stands for "least squares"), which you could call directly:

```
>>> theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
>>> theta_best_svd
array([[4.21509616],
       [2.77011339]])
```

This function computes $\widehat{\boldsymbol{\theta}} = \mathbf{X}^{+}\mathbf{y}$, where $\mathbf{X}^{+}$ is the *pseudoinverse* of $\mathbf{X}$ (specifically, the Moore-Penrose inverse). You can use `np.linalg.pinv()` to compute the pseudoinverse directly:

2 Note that Scikit-Learn separates the bias term (`intercept_`) from the feature weights (`coef_`).

```
>>> np.linalg.pinv(X_b).dot(y)
array([[4.21509616],
       [2.77011339]])
```

The pseudoinverse itself is computed using a standard matrix factorization technique called *Singular Value Decomposition* (SVD) that can decompose the training set matrix $\mathbf{X}$ into the matrix multiplication of three matrices $\mathbf{U}\ \mathbf{\Sigma}\ \mathbf{V}^{\intercal}$ (see `numpy.linalg.svd()`). The pseudoinverse is computed as $\mathbf{X}^{+} = \mathbf{V}\mathbf{\Sigma}^{+}\mathbf{U}^{\intercal}$. To compute the matrix $\mathbf{\Sigma}^{+}$, the algorithm takes $\mathbf{\Sigma}$ and sets to zero all values smaller than a tiny threshold value, then it replaces all the nonzero values with their inverse, and finally it transposes the resulting matrix. This approach is more efficient than computing the Normal Equation, plus it handles edge cases nicely: indeed, the Normal Equation may not work if the matrix $\mathbf{X}^{\intercal}\mathbf{X}$ is not invertible (i.e., singular), such as if $m < n$ or if some features are redundant, but the pseudoinverse is always defined.

Computational Complexity

The Normal Equation computes the inverse of $\mathbf{X}^{\intercal}\ \mathbf{X}$, which is an $(n + 1) \times (n + 1)$ matrix (where n is the number of features). The *computational complexity* of inverting such a matrix is typically about $O(n^{2.4})$ to $O(n^{3})$, depending on the implementation. In other words, if you double the number of features, you multiply the computation time by roughly $2^{2.4} = 5.3$ to $2^{3} = 8$.

The SVD approach used by Scikit-Learn's `LinearRegression` class is about $O(n^{2})$. If you double the number of features, you multiply the computation time by roughly 4.

Both the Normal Equation and the SVD approach get very slow when the number of features grows large (e.g., 100,000). On the positive side, both are linear with regard to the number of instances in the training set (they are $O(m)$), so they handle large training sets efficiently, provided they can fit in memory.

Also, once you have trained your Linear Regression model (using the Normal Equation or any other algorithm), predictions are very fast: the computational complexity is linear with regard to both the number of instances you want to make predictions on and the number of features. In other words, making predictions on twice as many instances (or twice as many features) will take roughly twice as much time.

Now we will look at a very different way to train a Linear Regression model, which is better suited for cases where there are a large number of features or too many training instances to fit in memory.

Gradient Descent

Gradient Descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function.

Suppose you are lost in the mountains in a dense fog, and you can only feel the slope of the ground below your feet. A good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope. This is exactly what Gradient Descent does: it measures the local gradient of the error function with regard to the parameter vector $\boldsymbol{\theta}$, and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum!

Concretely, you start by filling $\boldsymbol{\theta}$ with random values (this is called *random initialization*). Then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm *converges* to a minimum (see Figure 4-3).

Figure 4-3. In this depiction of Gradient Descent, the model parameters are initialized randomly and get tweaked repeatedly to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually get smaller as the parameters approach the minimum

An important parameter in Gradient Descent is the size of the steps, determined by the *learning rate* hyperparameter. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time (see Figure 4-4).

Figure 4-4. The learning rate is too small

On the other hand, if the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before. This might make the algorithm diverge, with larger and larger values, failing to find a good solution (see Figure 4-5).

Figure 4-5. The learning rate is too large

Finally, not all cost functions look like nice, regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum difficult. Figure 4-6 shows the two main challenges with Gradient Descent. If the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the *global minimum*. If it starts on the right, then it will take a very long time to cross the plateau. And if you stop too early, you will never reach the global minimum.

Figure 4-6. Gradient Descent pitfalls

Fortunately, the MSE cost function for a Linear Regression model happens to be a *convex function*, which means that if you pick any two points on the curve, the line segment joining them never crosses the curve. This implies that there are no local minima, just one global minimum. It is also a continuous function with a slope that never changes abruptly.[3] These two facts have a great consequence: Gradient Descent is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high).

In fact, the cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales. Figure 4-7 shows Gradient Descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right).[4]

Figure 4-7. Gradient Descent with (left) and without (right) feature scaling

3 Technically speaking, its derivative is *Lipschitz continuous*.

4 Since feature 1 is smaller, it takes a larger change in θ_1 to affect the cost function, which is why the bowl is elongated along the θ_1 axis.

As you can see, on the left the Gradient Descent algorithm goes straight toward the minimum, thereby reaching it quickly, whereas on the right it first goes in a direction almost orthogonal to the direction of the global minimum, and it ends with a long march down an almost flat valley. It will eventually reach the minimum, but it will take a long time.

When using Gradient Descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's `StandardScaler` class), or else it will take much longer to converge.

This diagram also illustrates the fact that training a model means searching for a combination of model parameters that minimizes a cost function (over the training set). It is a search in the model's *parameter space*: the more parameters a model has, the more dimensions this space has, and the harder the search is: searching for a needle in a 300-dimensional haystack is much trickier than in 3 dimensions. Fortunately, since the cost function is convex in the case of Linear Regression, the needle is simply at the bottom of the bowl.

Batch Gradient Descent

To implement Gradient Descent, you need to compute the gradient of the cost function with regard to each model parameter θ_j. In other words, you need to calculate how much the cost function will change if you change θ_j just a little bit. This is called a *partial derivative*. It is like asking "What is the slope of the mountain under my feet if I face east?" and then asking the same question facing north (and so on for all other dimensions, if you can imagine a universe with more than three dimensions). Equation 4-5 computes the partial derivative of the cost function with regard to parameter θ_j, noted $\partial \text{MSE}(\boldsymbol{\theta}) / \partial\theta_j$.

Equation 4-5. Partial derivatives of the cost function

$$\frac{\partial}{\partial\theta_j}\text{MSE}(\boldsymbol{\theta}) = \frac{2}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^\top\mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}$$

Instead of computing these partial derivatives individually, you can use Equation 4-6 to compute them all in one go. The gradient vector, noted $\nabla_{\boldsymbol{\theta}}\text{MSE}(\boldsymbol{\theta})$, contains all the partial derivatives of the cost function (one for each model parameter).

Equation 4-6. Gradient vector of the cost function

$$\nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \mathrm{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \mathrm{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \mathrm{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^{\intercal}(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

Notice that this formula involves calculations over the full training set **X**, at each Gradient Descent step! This is why the algorithm is called *Batch Gradient Descent*: it uses the whole batch of training data at every step (actually, *Full Gradient Descent* would probably be a better name). As a result it is terribly slow on very large training sets (but we will see much faster Gradient Descent algorithms shortly). However, Gradient Descent scales well with the number of features; training a Linear Regression model when there are hundreds of thousands of features is much faster using Gradient Descent than using the Normal Equation or SVD decomposition.

Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. This means subtracting $\nabla_{\boldsymbol{\theta}}\mathrm{MSE}(\boldsymbol{\theta})$ from $\boldsymbol{\theta}$. This is where the learning rate η comes into play:[5] multiply the gradient vector by η to determine the size of the downhill step (Equation 4-7).

Equation 4-7. Gradient Descent step

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta})$$

Let's look at a quick implementation of this algorithm:

```
eta = 0.1  # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)  # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

5 Eta (η) is the seventh letter of the Greek alphabet.

That wasn't too hard! Let's look at the resulting `theta`:

```
>>> theta
array([[4.21509616],
       [2.77011339]])
```

Hey, that's exactly what the Normal Equation found! Gradient Descent worked perfectly. But what if you had used a different learning rate `eta`? Figure 4-8 shows the first 10 steps of Gradient Descent using three different learning rates (the dashed line represents the starting point).

Figure 4-8. Gradient Descent with various learning rates

On the left, the learning rate is too low: the algorithm will eventually reach the solution, but it will take a long time. In the middle, the learning rate looks pretty good: in just a few iterations, it has already converged to the solution. On the right, the learning rate is too high: the algorithm diverges, jumping all over the place and actually getting further and further away from the solution at every step.

To find a good learning rate, you can use grid search (see Chapter 2). However, you may want to limit the number of iterations so that grid search can eliminate models that take too long to converge.

You may wonder how to set the number of iterations. If it is too low, you will still be far away from the optimal solution when the algorithm stops; but if it is too high, you will waste time while the model parameters do not change anymore. A simple solution is to set a very large number of iterations but to interrupt the algorithm when the gradient vector becomes tiny—that is, when its norm becomes smaller than a tiny number ϵ (called the *tolerance*)—because this happens when Gradient Descent has (almost) reached the minimum.

Convergence Rate

When the cost function is convex and its slope does not change abruptly (as is the case for the MSE cost function), Batch Gradient Descent with a fixed learning rate will eventually converge to the optimal solution, but you may have to wait a while: it can take $O(1/\epsilon)$ iterations to reach the optimum within a range of ϵ, depending on the shape of the cost function. If you divide the tolerance by 10 to have a more precise solution, then the algorithm may have to run about 10 times longer.

Stochastic Gradient Descent

The main problem with Batch Gradient Descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large. At the opposite extreme, *Stochastic Gradient Descent* picks a random instance in the training set at every step and computes the gradients based only on that single instance. Obviously, working on a single instance at a time makes the algorithm much faster because it has very little data to manipulate at every iteration. It also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration (Stochastic GD can be implemented as an out-of-core algorithm; see Chapter 1).

On the other hand, due to its stochastic (i.e., random) nature, this algorithm is much less regular than Batch Gradient Descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down (see Figure 4-9). So once the algorithm stops, the final parameter values are good, but not optimal.

Figure 4-9. With Stochastic Gradient Descent, each training step is much faster but also much more stochastic than when using Batch Gradient Descent

When the cost function is very irregular (as in Figure 4-6), this can actually help the algorithm jump out of local minima, so Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does.

Therefore, randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum. One solution to this dilemma is to gradually reduce the learning rate. The steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum. This process is akin to *simulated annealing*, an algorithm inspired from the process in metallurgy of annealing, where molten metal is slowly cooled down. The function that determines the learning rate at each iteration is called the *learning schedule*. If the learning rate is reduced too quickly, you may get stuck in a local minimum, or even end up frozen halfway to the minimum. If the learning rate is reduced too slowly, you may jump around the minimum for a long time and end up with a suboptimal solution if you halt training too early.

This code implements Stochastic Gradient Descent using a simple learning schedule:

```
n_epochs = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1)  # random initialization

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

By convention we iterate by rounds of m iterations; each round is called an *epoch*. While the Batch Gradient Descent code iterated 1,000 times through the whole training set, this code goes through the training set only 50 times and reaches a pretty good solution:

```
>>> theta
array([[4.21076011],
       [2.74856079]])
```

Figure 4-10 shows the first 20 steps of training (notice how irregular the steps are).

Figure 4-10. The first 20 steps of Stochastic Gradient Descent

Note that since instances are picked randomly, some instances may be picked several times per epoch, while others may not be picked at all. If you want to be sure that the algorithm goes through every instance at each epoch, another approach is to shuffle the training set (making sure to shuffle the input features and the labels jointly), then go through it instance by instance, then shuffle it again, and so on. However, this approach generally converges more slowly.

When using Stochastic Gradient Descent, the training instances must be independent and identically distributed (IID) to ensure that the parameters get pulled toward the global optimum, on average. A simple way to ensure this is to shuffle the instances during training (e.g., pick each instance randomly, or shuffle the training set at the beginning of each epoch). If you do not shuffle the instances—for example, if the instances are sorted by label—then SGD will start by optimizing for one label, then the next, and so on, and it will not settle close to the global minimum.

To perform Linear Regression using Stochastic GD with Scikit-Learn, you can use the `SGDRegressor` class, which defaults to optimizing the squared error cost function. The following code runs for maximum 1,000 epochs or until the loss drops by less than 0.001 during one epoch (`max_iter=1000`, `tol=1e-3`). It starts with a learning rate of 0.1 (`eta0=0.1`), using the default learning schedule (different from the preceding one). Lastly, it does not use any regularization (`penalty=None`; more details on this shortly):

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

Once again, you find a solution quite close to the one returned by the Normal Equation:

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))
```

Mini-batch Gradient Descent

The last Gradient Descent algorithm we will look at is called *Mini-batch Gradient Descent*. It is simple to understand once you know Batch and Stochastic Gradient Descent: at each step, instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in Stochastic GD), Mini-batch GD computes the gradients on small random sets of instances called *mini-batches*. The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.

The algorithm's progress in parameter space is less erratic than with Stochastic GD, especially with fairly large mini-batches. As a result, Mini-batch GD will end up walking around a bit closer to the minimum than Stochastic GD—but it may be harder for it to escape from local minima (in the case of problems that suffer from local minima, unlike Linear Regression). Figure 4-11 shows the paths taken by the three Gradient Descent algorithms in parameter space during training. They all end up near the minimum, but Batch GD's path actually stops at the minimum, while both Stochastic GD and Mini-batch GD continue to walk around. However, don't forget that Batch GD takes a lot of time to take each step, and Stochastic GD and Mini-batch GD would also reach the minimum if you used a good learning schedule.

Figure 4-11. Gradient Descent paths in parameter space

Let's compare the algorithms we've discussed so far for Linear Regression[6] (recall that m is the number of training instances and n is the number of features); see Table 4-1.

Table 4-1. Comparison of algorithms for Linear Regression

Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	`SGDRegressor`
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`

There is almost no difference after training: all these algorithms end up with very similar models and make predictions in exactly the same way.

Polynomial Regression

What if your data is more complex than a straight line? Surprisingly, you can use a linear model to fit nonlinear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called *Polynomial Regression*.

Let's look at an example. First, let's generate some nonlinear data, based on a simple *quadratic equation*[7] (plus some noise; see Figure 4-12):

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

6 While the Normal Equation can only perform Linear Regression, the Gradient Descent algorithms can be used to train many other models, as we will see.

7 A quadratic equation is of the form $y = ax^2 + bx + c$.

Figure 4-12. Generated nonlinear and noisy dataset

Clearly, a straight line will never fit this data properly. So let's use Scikit-Learn's `PolynomialFeatures` class to transform our training data, adding the square (second-degree polynomial) of each feature in the training set as a new feature (in this case there is just one feature):

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])
```

`X_poly` now contains the original feature of `X` plus the square of this feature. Now you can fit a `LinearRegression` model to this extended training data (Figure 4-13):

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

Figure 4-13. Polynomial Regression model predictions

Not bad: the model estimates $\hat{y} = 0.56{x_1}^2 + 0.93x_1 + 1.78$ when in fact the original function was $y = 0.5{x_1}^2 + 1.0x_1 + 2.0 + \text{Gaussian noise}$.

Note that when there are multiple features, Polynomial Regression is capable of finding relationships between features (which is something a plain Linear Regression model cannot do). This is made possible by the fact that `PolynomialFeatures` also adds all combinations of features up to the given degree. For example, if there were two features a and b, `PolynomialFeatures` with `degree=3` would not only add the features a^2, a^3, b^2, and b^3, but also the combinations ab, a^2b, and ab^2.

`PolynomialFeatures(degree=d)` transforms an array containing n features into an array containing $(n + d)! / d!n!$ features, where $n!$ is the *factorial* of n, equal to $1 \times 2 \times 3 \times \cdots \times n$. Beware of the combinatorial explosion of the number of features!

Learning Curves

If you perform high-degree Polynomial Regression, you will likely fit the training data much better than with plain Linear Regression. For example, Figure 4-14 applies a 300-degree polynomial model to the preceding training data, and compares the result with a pure linear model and a quadratic model (second-degree polynomial). Notice how the 300-degree polynomial model wiggles around to get as close as possible to the training instances.

Figure 4-14. High-degree Polynomial Regression

This high-degree Polynomial Regression model is severely overfitting the training data, while the linear model is underfitting it. The model that will generalize best in this case is the quadratic model, which makes sense because the data was generated using a quadratic model. But in general you won't know what function generated the data, so how can you decide how complex your model should be? How can you tell that your model is overfitting or underfitting the data?

In Chapter 2 you used cross-validation to get an estimate of a model's generalization performance. If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then your model is overfitting. If it performs poorly on both, then it is underfitting. This is one way to tell when a model is too simple or too complex.

Another way to tell is to look at the *learning curves*: these are plots of the model's performance on the training set and the validation set as a function of the training set size (or the training iteration). To generate the plots, train the model several times on different sized subsets of the training set. The following code defines a function that, given some training data, plots the learning curves of a model:

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

Let's look at the learning curves of the plain Linear Regression model (a straight line; see Figure 4-15):

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```

Figure 4-15. Learning curves

This model that's underfitting deserves a bit of explanation. First, let's look at the performance on the training data: when there are just one or two instances in the training set, the model can fit them perfectly, which is why the curve starts at zero. But as new instances are added to the training set, it becomes impossible for the model to fit the training data perfectly, both because the data is noisy and because it is not linear at all. So the error on the training data goes up until it reaches a plateau, at which point adding new instances to the training set doesn't make the average error much better or worse. Now let's look at the performance of the model on the validation data. When the model is trained on very few training instances, it is incapable of generalizing properly, which is why the validation error is initially quite big. Then, as the

model is shown more training examples, it learns, and thus the validation error slowly goes down. However, once again a straight line cannot do a good job modeling the data, so the error ends up at a plateau, very close to the other curve.

These learning curves are typical of a model that's underfitting. Both curves have reached a plateau; they are close and fairly high.

If your model is underfitting the training data, adding more training examples will not help. You need to use a more complex model or come up with better features.

Now let's look at the learning curves of a 10th-degree polynomial model on the same data (Figure 4-16):

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
```

Figure 4-16. Learning curves for the 10th-degree polynomial model

These learning curves look a bit like the previous ones, but there are two very important differences:

- The error on the training data is much lower than with the Linear Regression model.

- There is a gap between the curves. This means that the model performs significantly better on the training data than on the validation data, which is the hallmark of an overfitting model. If you used a much larger training set, however, the two curves would continue to get closer.

One way to improve an overfitting model is to feed it more training data until the validation error reaches the training error.

The Bias/Variance Trade-off

An important theoretical result of statistics and Machine Learning is the fact that a model's generalization error can be expressed as the sum of three very different errors:

Bias
: This part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.[8]

Variance
: This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance and thus overfit the training data.

Irreducible error
: This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).

Increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, reducing a model's complexity increases its bias and reduces its variance. This is why it is called a trade-off.

Regularized Linear Models

As we saw in Chapters 1 and 2, a good way to reduce overfitting is to regularize the model (i.e., to constrain it): the fewer degrees of freedom it has, the harder it will be

8 This notion of bias is not to be confused with the bias term of linear models.

for it to overfit the data. A simple way to regularize a polynomial model is to reduce the number of polynomial degrees.

For a linear model, regularization is typically achieved by constraining the weights of the model. We will now look at Ridge Regression, Lasso Regression, and Elastic Net, which implement three different ways to constrain the weights.

Ridge Regression

Ridge Regression (also called *Tikhonov regularization*) is a regularized version of Linear Regression: a *regularization term* equal to $\alpha\sum_{i=1}^{n}\theta_i^2$ is added to the cost function. This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible. Note that the regularization term should only be added to the cost function during training. Once the model is trained, you want to use the unregularized performance measure to evaluate the model's performance.

It is quite common for the cost function used during training to be different from the performance measure used for testing. Apart from regularization, another reason they might be different is that a good training cost function should have optimization-friendly derivatives, while the performance measure used for testing should be as close as possible to the final objective. For example, classifiers are often trained using a cost function such as the log loss (discussed in a moment) but evaluated using precision/recall.

The hyperparameter α controls how much you want to regularize the model. If $\alpha = 0$, then Ridge Regression is just Linear Regression. If α is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean. Equation 4-8 presents the Ridge Regression cost function.[9]

Equation 4-8. Ridge Regression cost function

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha\frac{1}{2}\sum_{i=1}^{n}\theta_i^2$$

Note that the bias term θ_0 is not regularized (the sum starts at $i = 1$, not 0). If we define $\mathbf{w}$ as the vector of feature weights (θ_1 to θ_n), then the regularization term is

9 It is common to use the notation $J(\boldsymbol{\theta})$ for cost functions that don't have a short name; we will often use this notation throughout the rest of this book. The context will make it clear which cost function is being discussed.

equal to ½($\| \mathbf{w} \|_2$)^2, where $\| \mathbf{w} \|_2$ represents the ℓ_2 norm of the weight vector.[10] For Gradient Descent, just add $\alpha\mathbf{w}$ to the MSE gradient vector (Equation 4-6).

It is important to scale the data (e.g., using a `StandardScaler`) before performing Ridge Regression, as it is sensitive to the scale of the input features. This is true of most regularized models.

Figure 4-17 shows several Ridge models trained on some linear data using different α values. On the left, plain Ridge models are used, leading to linear predictions. On the right, the data is first expanded using `PolynomialFeatures(degree=10)`, then it is scaled using a `StandardScaler`, and finally the Ridge models are applied to the resulting features: this is Polynomial Regression with Ridge regularization. Note how increasing α leads to flatter (i.e., less extreme, more reasonable) predictions, thus reducing the model's variance but increasing its bias.

Figure 4-17. A linear model (left) and a polynomial model (right), both with various levels of Ridge regularization

As with Linear Regression, we can perform Ridge Regression either by computing a closed-form equation or by performing Gradient Descent. The pros and cons are the

10 Norms are discussed in Chapter 2.

same. Equation 4-9 shows the closed-form solution, where $\mathbf{A}$ is the $(n + 1) \times (n + 1)$ *identity matrix*,[11] except with a 0 in the top-left cell, corresponding to the bias term.

Equation 4-9. Ridge Regression closed-form solution

$$\widehat{\boldsymbol{\theta}} = \left(\mathbf{X}^{\intercal}\mathbf{X} + \alpha\mathbf{A}\right)^{-1} \quad \mathbf{X}^{\intercal} \quad \mathbf{y}$$

Here is how to perform Ridge Regression with Scikit-Learn using a closed-form solution (a variant of Equation 4-9 that uses a matrix factorization technique by André-Louis Cholesky):

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[1.55071465]])
```

And using Stochastic Gradient Descent:[12]

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.47012588])
```

The `penalty` hyperparameter sets the type of regularization term to use. Specifying `"l2"` indicates that you want SGD to add a regularization term to the cost function equal to half the square of the ℓ_2 norm of the weight vector: this is simply Ridge Regression.

Lasso Regression

Least Absolute Shrinkage and Selection Operator Regression (usually simply called *Lasso Regression*) is another regularized version of Linear Regression: just like Ridge Regression, it adds a regularization term to the cost function, but it uses the ℓ_1 norm of the weight vector instead of half the square of the ℓ_2 norm (see Equation 4-10).

Equation 4-10. Lasso Regression cost function

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha\Sigma_{i=1}^{n}\left|\theta_i\right|$$

11 A square matrix full of 0s except for 1s on the main diagonal (top left to bottom right).

12 Alternatively you can use the `Ridge` class with the `"sag"` solver. Stochastic Average GD is a variant of Stochastic GD. For more details, see the presentation "Minimizing Finite Sums with the Stochastic Average Gradient Algorithm" by Mark Schmidt et al. from the University of British Columbia.

Figure 4-18 shows the same thing as Figure 4-17 but replaces Ridge models with Lasso models and uses smaller α values.

Figure 4-18. A linear model (left) and a polynomial model (right), both using various levels of Lasso regularization

An important characteristic of Lasso Regression is that it tends to eliminate the weights of the least important features (i.e., set them to zero). For example, the dashed line in the righthand plot in Figure 4-18 (with $\alpha = 10^{-7}$) looks quadratic, almost linear: all the weights for the high-degree polynomial features are equal to zero. In other words, Lasso Regression automatically performs feature selection and outputs a *sparse model* (i.e., with few nonzero feature weights).

You can get a sense of why this is the case by looking at Figure 4-19: the axes represent two model parameters, and the background contours represent different loss functions. In the top-left plot, the contours represent the ℓ_1 loss ($|\theta_1| + |\theta_2|$), which drops linearly as you get closer to any axis. For example, if you initialize the model parameters to $\theta_1 = 2$ and $\theta_2 = 0.5$, running Gradient Descent will decrement both parameters equally (as represented by the dashed yellow line); therefore θ_2 will reach 0 first (since it was closer to 0 to begin with). After that, Gradient Descent will roll down the gutter until it reaches $\theta_1 = 0$ (with a bit of bouncing around, since the gradients of ℓ_1 never get close to 0: they are either –1 or 1 for each parameter). In the top-right plot, the contours represent Lasso's cost function (i.e., an MSE cost function plus an ℓ_1 loss). The small white circles show the path that Gradient Descent takes to optimize some model parameters that were initialized around $\theta_1 = 0.25$ and $\theta_2 = -1$: notice once again how the path quickly reaches $\theta_2 = 0$, then rolls down the gutter and ends up bouncing around the global optimum (represented by the red square). If we increased α, the global optimum would move left along the dashed yellow line, while

if we decreased α, the global optimum would move right (in this example, the optimal parameters for the unregularized MSE are $\theta_1 = 2$ and $\theta_2 = 0.5$).

Figure 4-19. Lasso versus Ridge regularization

The two bottom plots show the same thing but with an ℓ_2 penalty instead. In the bottom-left plot, you can see that the ℓ_2 loss decreases with the distance to the origin, so Gradient Descent just takes a straight path toward that point. In the bottom-right plot, the contours represent Ridge Regression's cost function (i.e., an MSE cost function plus an ℓ_2 loss). There are two main differences with Lasso. First, the gradients get smaller as the parameters approach the global optimum, so Gradient Descent naturally slows down, which helps convergence (as there is no bouncing around). Second, the optimal parameters (represented by the red square) get closer and closer to the origin when you increase α, but they never get eliminated entirely.

To avoid Gradient Descent from bouncing around the optimum at the end when using Lasso, you need to gradually reduce the learning rate during training (it will still bounce around the optimum, but the steps will get smaller and smaller, so it will converge).

The Lasso cost function is not differentiable at $\theta_i = 0$ (for $i = 1, 2, \cdots, n$), but Gradient Descent still works fine if you use a *subgradient vector* $\mathbf{g}$[13] instead when any $\theta_i = 0$. Equation 4-11 shows a subgradient vector equation you can use for Gradient Descent with the Lasso cost function.

Equation 4-11. Lasso Regression subgradient vector

$$g(\boldsymbol{\theta}, J) = \nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta}) + \alpha \begin{pmatrix} \operatorname{sign}(\theta_1) \\ \operatorname{sign}(\theta_2) \\ \vdots \\ \operatorname{sign}(\theta_n) \end{pmatrix} \quad \text{where } \operatorname{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

Here is a small Scikit-Learn example using the `Lasso` class:

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([1.53788174])
```

Note that you could instead use `SGDRegressor(penalty="l1")`.

Elastic Net

Elastic Net is a middle ground between Ridge Regression and Lasso Regression. The regularization term is a simple mix of both Ridge and Lasso's regularization terms, and you can control the mix ratio r. When $r = 0$, Elastic Net is equivalent to Ridge Regression, and when $r = 1$, it is equivalent to Lasso Regression (see Equation 4-12).

Equation 4-12. Elastic Net cost function

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + r\alpha \Sigma_{i=1}^{n} \left|\theta_i\right| + \frac{1-r}{2}\alpha \Sigma_{i=1}^{n} \theta_i^2$$

So when should you use plain Linear Regression (i.e., without any regularization), Ridge, Lasso, or Elastic Net? It is almost always preferable to have at least a little bit of regularization, so generally you should avoid plain Linear Regression. Ridge is a good default, but if you suspect that only a few features are useful, you should prefer Lasso or Elastic Net because they tend to reduce the useless features' weights down to zero, as we have discussed. In general, Elastic Net is preferred over Lasso because Lasso

13 You can think of a subgradient vector at a nondifferentiable point as an intermediate vector between the gradient vectors around that point.

may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated.

Here is a short example that uses Scikit-Learn's `ElasticNet` (`l1_ratio` corresponds to the mix ratio *r*):

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([1.54333232])
```

Early Stopping

A very different way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum. This is called *early stopping*. Figure 4-20 shows a complex model (in this case, a high-degree Polynomial Regression model) being trained with Batch Gradient Descent. As the epochs go by the algorithm learns, and its prediction error (RMSE) on the training set goes down, along with its prediction error on the validation set. After a while though, the validation error stops decreasing and starts to go back up. This indicates that the model has started to overfit the training data. With early stopping you just stop training as soon as the validation error reaches the minimum. It is such a simple and efficient regularization technique that Geoffrey Hinton called it a "beautiful free lunch."

Figure 4-20. Early stopping regularization

With Stochastic and Mini-batch Gradient Descent, the curves are not so smooth, and it may be hard to know whether you have reached the minimum or not. One solution is to stop only after the validation error has been above the minimum for some time (when you are confident that the model will not do any better), then roll back the model parameters to the point where the validation error was at a minimum.

Here is a basic implementation of early stopping:

```
from sklearn.base import clone

# prepare the data
poly_scaler = Pipeline([
        ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
        ("std_scaler", StandardScaler())
    ])
X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)  # continues where it left off
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

Note that with `warm_start=True`, when the `fit()` method is called it continues training where it left off, instead of restarting from scratch.

Logistic Regression

As we discussed in Chapter 1, some regression algorithms can be used for classification (and vice versa). *Logistic Regression* (also called *Logit Regression*) is commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?). If the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the *positive class*, labeled "1"), and otherwise it predicts that it does not (i.e., it belongs to the *negative class*, labeled "0"). This makes it a binary classifier.

Estimating Probabilities

So how does Logistic Regression work? Just like a Linear Regression model, a Logistic Regression model computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly like the Linear Regression model does, it outputs the *logistic* of this result (see Equation 4-13).

Equation 4-13. Logistic Regression model estimated probability (vectorized form)

$$\hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma(\mathbf{x}^{\intercal}\boldsymbol{\theta})$$

The logistic—noted $\sigma(\cdot)$—is a *sigmoid function* (i.e., *S*-shaped) that outputs a number between 0 and 1. It is defined as shown in Equation 4-14 and Figure 4-21.

Equation 4-14. Logistic function

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

Figure 4-21. Logistic function

Once the Logistic Regression model has estimated the probability $\hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x})$ that an instance $\mathbf{x}$ belongs to the positive class, it can make its prediction $\hat{y}$ easily (see Equation 4-15).

Equation 4-15. Logistic Regression model prediction

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases}$$

Notice that $\sigma(t) < 0.5$ when $t < 0$, and $\sigma(t) \geq 0.5$ when $t \geq 0$, so a Logistic Regression model predicts 1 if $\mathbf{x}^{\intercal}\boldsymbol{\theta}$ is positive and 0 if it is negative.

The score t is often called the *logit*. The name comes from the fact that the logit function, defined as $\text{logit}(p) = \log(p / (1 - p))$, is the inverse of the logistic function. Indeed, if you compute the logit of the estimated probability p, you will find that the result is t. The logit is also called the *log-odds*, since it is the log of the ratio between the estimated probability for the positive class and the estimated probability for the negative class.

Training and Cost Function

Now you know how a Logistic Regression model estimates probabilities and makes predictions. But how is it trained? The objective of training is to set the parameter vector $\boldsymbol{\theta}$ so that the model estimates high probabilities for positive instances ($y = 1$) and low probabilities for negative instances ($y = 0$). This idea is captured by the cost function shown in Equation 4-16 for a single training instance $\mathbf{x}$.

Equation 4-16. Cost function of a single training instance

$$c(\boldsymbol{\theta}) = \begin{cases} -\log\left(\hat{p}\right) & \text{if } y = 1 \\ -\log\left(1 - \hat{p}\right) & \text{if } y = 0 \end{cases}$$

This cost function makes sense because $-\log(t)$ grows very large when t approaches 0, so the cost will be large if the model estimates a probability close to 0 for a positive instance, and it will also be very large if the model estimates a probability close to 1 for a negative instance. On the other hand, $-\log(t)$ is close to 0 when t is close to 1, so the cost will be close to 0 if the estimated probability is close to 0 for a negative instance or close to 1 for a positive instance, which is precisely what we want.

The cost function over the whole training set is the average cost over all training instances. It can be written in a single expression called the *log loss*, shown in Equation 4-17.

Equation 4-17. Logistic Regression cost function (log loss)

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}log\left(\hat{p}^{(i)}\right) + \left(1 - y^{(i)}\right)log\left(1 - \hat{p}^{(i)}\right)\right]$$

The bad news is that there is no known closed-form equation to compute the value of $\boldsymbol{\theta}$ that minimizes this cost function (there is no equivalent of the Normal Equation). The good news is that this cost function is convex, so Gradient Descent (or any other optimization algorithm) is guaranteed to find the global minimum (if the learning

rate is not too large and you wait long enough). The partial derivatives of the cost function with regard to the jth model parameter θ_j are given by Equation 4-18.

Equation 4-18. Logistic cost function partial derivatives

$$\frac{\partial}{\partial \theta_j} \mathrm{J}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(\sigma\left(\boldsymbol{\theta}^{\top} \mathbf{x}^{(i)}\right) - y^{(i)} \right) x_j^{(i)}$$

This equation looks very much like Equation 4-5: for each instance it computes the prediction error and multiplies it by the jth feature value, and then it computes the average over all training instances. Once you have the gradient vector containing all the partial derivatives, you can use it in the Batch Gradient Descent algorithm. That's it: you now know how to train a Logistic Regression model. For Stochastic GD you would take one instance at a time, and for Mini-batch GD you would use a mini-batch at a time.

Decision Boundaries

Let's use the iris dataset to illustrate Logistic Regression. This is a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: *Iris setosa*, *Iris versicolor*, and *Iris virginica* (see Figure 4-22).

Figure 4-22. Flowers of three iris plant species[14]

14 Photos reproduced from the corresponding Wikipedia pages. *Iris virginica* photo by Frank Mayfield (Creative Commons BY-SA 2.0), *Iris versicolor* photo by D. Gordon E. Robertson (Creative Commons BY-SA 3.0), *Iris setosa* photo public domain.

Let's try to build a classifier to detect the *Iris virginica* type based only on the petal width feature. First let's load the data:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
>>> X = iris["data"][:, 3:]  # petal width
>>> y = (iris["target"] == 2).astype(np.int)  # 1 if Iris virginica, else 0
```

Now let's train a Logistic Regression model:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)
```

Let's look at the model's estimated probabilities for flowers with petal widths varying from 0 cm to 3 cm (Figure 4-23):[15]

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris virginica")
# + more Matplotlib code to make the image look pretty
```

Figure 4-23. Estimated probabilities and decision boundary

The petal width of *Iris virginica* flowers (represented by triangles) ranges from 1.4 cm to 2.5 cm, while the other iris flowers (represented by squares) generally have a smaller petal width, ranging from 0.1 cm to 1.8 cm. Notice that there is a bit of overlap. Above about 2 cm the classifier is highly confident that the flower is an *Iris virginica* (it outputs a high probability for that class), while below 1 cm it is highly confident that it is not an *Iris virginica* (high probability for the "Not Iris virginica"

15 NumPy's `reshape()` function allows one dimension to be –1, which means "unspecified": the value is inferred from the length of the array and the remaining dimensions.

class). In between these extremes, the classifier is unsure. However, if you ask it to predict the class (using the `predict()` method rather than the `predict_proba()` method), it will return whichever class is the most likely. Therefore, there is a *decision boundary* at around 1.6 cm where both probabilities are equal to 50%: if the petal width is higher than 1.6 cm, the classifier will predict that the flower is an *Iris virginica*, and otherwise it will predict that it is not (even if it is not very confident):

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

Figure 4-24 shows the same dataset, but this time displaying two features: petal width and length. Once trained, the Logistic Regression classifier can, based on these two features, estimate the probability that a new flower is an *Iris virginica*. The dashed line represents the points where the model estimates a 50% probability: this is the model's decision boundary. Note that it is a linear boundary.[16] Each parallel line represents the points where the model outputs a specific probability, from 15% (bottom left) to 90% (top right). All the flowers beyond the top-right line have an over 90% chance of being *Iris virginica*, according to the model.

Figure 4-24. Linear decision boundary

Just like the other linear models, Logistic Regression models can be regularized using ℓ_1 or ℓ_2 penalties. Scikit-Learn actually adds an ℓ_2 penalty by default.

The hyperparameter controlling the regularization strength of a Scikit-Learn `LogisticRegression` model is not `alpha` (as in other linear models), but its inverse: `C`. The higher the value of `C`, the *less* the model is regularized.

16 It is the the set of points **x** such that $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$, which defines a straight line.

Softmax Regression

The Logistic Regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers (as discussed in Chapter 3). This is called *Softmax Regression*, or *Multinomial Logistic Regression*.

The idea is simple: when given an instance $\mathbf{x}$, the Softmax Regression model first computes a score $s_k(\mathbf{x})$ for each class k, then estimates the probability of each class by applying the *softmax function* (also called the *normalized exponential*) to the scores. The equation to compute $s_k(\mathbf{x})$ should look familiar, as it is just like the equation for Linear Regression prediction (see Equation 4-19).

Equation 4-19. Softmax score for class k

$$s_k(\mathbf{x}) = \mathbf{x}^\top \boldsymbol{\theta}^{(k)}$$

Note that each class has its own dedicated parameter vector $\boldsymbol{\theta}^{(k)}$. All these vectors are typically stored as rows in a *parameter matrix* $\boldsymbol{\Theta}$.

Once you have computed the score of every class for the instance $\mathbf{x}$, you can estimate the probability $\hat{p}_k$ that the instance belongs to class k by running the scores through the softmax function (Equation 4-20). The function computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials). The scores are generally called logits or log-odds (although they are actually unnormalized log-odds).

Equation 4-20. Softmax function

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

In this equation:

- K is the number of classes.
- $\mathbf{s}(\mathbf{x})$ is a vector containing the scores of each class for the instance $\mathbf{x}$.
- $\sigma(\mathbf{s}(\mathbf{x}))_k$ is the estimated probability that the instance $\mathbf{x}$ belongs to class k, given the scores of each class for that instance.

Just like the Logistic Regression classifier, the Softmax Regression classifier predicts the class with the highest estimated probability (which is simply the class with the highest score), as shown in Equation 4-21.

Equation 4-21. Softmax Regression classifier prediction

$$\hat{y} = \underset{k}{\operatorname{argmax}}\ \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}}\ s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}}\ \left(\left(\boldsymbol{\theta}^{(k)}\right)^{\mathsf{T}}\mathbf{x}\right)$$

The *argmax* operator returns the value of a variable that maximizes a function. In this equation, it returns the value of k that maximizes the estimated probability $\sigma(\mathbf{s}(\mathbf{x}))_k$.

The Softmax Regression classifier predicts only one class at a time (i.e., it is multiclass, not multioutput), so it should be used only with mutually exclusive classes, such as different types of plants. You cannot use it to recognize multiple people in one picture.

Now that you know how the model estimates probabilities and makes predictions, let's take a look at training. The objective is to have a model that estimates a high probability for the target class (and consequently a low probability for the other classes). Minimizing the cost function shown in Equation 4-22, called the *cross entropy*, should lead to this objective because it penalizes the model when it estimates a low probability for a target class. Cross entropy is frequently used to measure how well a set of estimated class probabilities matches the target classes.

Equation 4-22. Cross entropy cost function

$$J(\boldsymbol{\Theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)}\log\left(\hat{p}_k^{(i)}\right)$$

In this equation:

- $y_k^{(i)}$ is the target probability that the ith instance belongs to class k. In general, it is either equal to 1 or 0, depending on whether the instance belongs to the class or not.

Notice that when there are just two classes ($K = 2$), this cost function is equivalent to the Logistic Regression's cost function (log loss; see Equation 4-17).

Cross Entropy

Cross entropy originated from information theory. Suppose you want to efficiently transmit information about the weather every day. If there are eight options (sunny, rainy, etc.), you could encode each option using three bits because $2^3 = 8$. However, if you think it will be sunny almost every day, it would be much more efficient to code "sunny" on just one bit (0) and the other seven options on four bits (starting with a 1). Cross entropy measures the average number of bits you actually send per option. If your assumption about the weather is perfect, cross entropy will be equal to the entropy of the weather itself (i.e., its intrinsic unpredictability). But if your assumptions are wrong (e.g., if it rains often), cross entropy will be greater by an amount called the *Kullback–Leibler (KL) divergence*.

The cross entropy between two probability distributions p and q is defined as $H(p,q) = -\Sigma_x p(x) \log q(x)$ (at least when the distributions are discrete). For more details, check out my video on the subject.

The gradient vector of this cost function with regard to $\boldsymbol{\theta}^{(k)}$ is given by Equation 4-23.

Equation 4-23. Cross entropy gradient vector for class k

$$\nabla_{\boldsymbol{\theta}^{(k)}} J(\boldsymbol{\Theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{p}_k^{(i)} - y_k^{(i)}\right) \mathbf{x}^{(i)}$$

Now you can compute the gradient vector for every class, then use Gradient Descent (or any other optimization algorithm) to find the parameter matrix $\boldsymbol{\Theta}$ that minimizes the cost function.

Let's use Softmax Regression to classify the iris flowers into all three classes. Scikit-Learn's `LogisticRegression` uses one-versus-the-rest by default when you train it on more than two classes, but you can set the `multi_class` hyperparameter to `"multinomial"` to switch it to Softmax Regression. You must also specify a solver that supports Softmax Regression, such as the `"lbfgs"` solver (see Scikit-Learn's documentation for more details). It also applies ℓ_2 regularization by default, which you can control using the hyperparameter `C`:

```
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"]

softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10)
softmax_reg.fit(X, y)
```

So the next time you find an iris with petals that are 5 cm long and 2 cm wide, you can ask your model to tell you what type of iris it is, and it will answer *Iris virginica* (class 2) with 94.2% probability (or *Iris versicolor* with 5.8% probability):

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```

Figure 4-25 shows the resulting decision boundaries, represented by the background colors. Notice that the decision boundaries between any two classes are linear. The figure also shows the probabilities for the *Iris versicolor* class, represented by the curved lines (e.g., the line labeled with 0.450 represents the 45% probability boundary). Notice that the model can predict a class that has an estimated probability below 50%. For example, at the point where all decision boundaries meet, all classes have an equal estimated probability of 33%.

Figure 4-25. Softmax Regression decision boundaries

Exercises

1. Which Linear Regression training algorithm can you use if you have a training set with millions of features?
2. Suppose the features in your training set have very different scales. Which algorithms might suffer from this, and how? What can you do about it?
3. Can Gradient Descent get stuck in a local minimum when training a Logistic Regression model?
4. Do all Gradient Descent algorithms lead to the same model, provided you let them run long enough?
5. Suppose you use Batch Gradient Descent and you plot the validation error at every epoch. If you notice that the validation error consistently goes up, what is likely going on? How can you fix this?
6. Is it a good idea to stop Mini-batch Gradient Descent immediately when the validation error goes up?

7. Which Gradient Descent algorithm (among those we discussed) will reach the vicinity of the optimal solution the fastest? Which will actually converge? How can you make the others converge as well?
8. Suppose you are using Polynomial Regression. You plot the learning curves and you notice that there is a large gap between the training error and the validation error. What is happening? What are three ways to solve this?
9. Suppose you are using Ridge Regression and you notice that the training error and the validation error are almost equal and fairly high. Would you say that the model suffers from high bias or high variance? Should you increase the regularization hyperparameter α or reduce it?
10. Why would you want to use:
    a. Ridge Regression instead of plain Linear Regression (i.e., without any regularization)?
    b. Lasso instead of Ridge Regression?
    c. Elastic Net instead of Lasso?
11. Suppose you want to classify pictures as outdoor/indoor and daytime/nighttime. Should you implement two Logistic Regression classifiers or one Softmax Regression classifier?
12. Implement Batch Gradient Descent with early stopping for Softmax Regression (without using Scikit-Learn).

Solutions to these exercises are available in Appendix A.

CHAPTER 5

Support Vector Machines

A *Support Vector Machine* (SVM) is a powerful and versatile Machine Learning model, capable of performing linear or nonlinear classification, regression, and even outlier detection. It is one of the most popular models in Machine Learning, and anyone interested in Machine Learning should have it in their toolbox. SVMs are particularly well suited for classification of complex small- or medium-sized datasets.

This chapter will explain the core concepts of SVMs, how to use them, and how they work.

Linear SVM Classification

The fundamental idea behind SVMs is best explained with some pictures. Figure 5-1 shows part of the iris dataset that was introduced at the end of Chapter 4. The two classes can clearly be separated easily with a straight line (they are *linearly separable*). The left plot shows the decision boundaries of three possible linear classifiers. The model whose decision boundary is represented by the dashed line is so bad that it does not even separate the classes properly. The other two models work perfectly on this training set, but their decision boundaries come so close to the instances that these models will probably not perform as well on new instances. In contrast, the solid line in the plot on the right represents the decision boundary of an SVM classifier; this line not only separates the two classes but also stays as far away from the closest training instances as possible. You can think of an SVM classifier as fitting the widest possible street (represented by the parallel dashed lines) between the classes. This is called *large margin classification*.

Figure 5-1. Large margin classification

Notice that adding more training instances "off the street" will not affect the decision boundary at all: it is fully determined (or "supported") by the instances located on the edge of the street. These instances are called the *support vectors* (they are circled in Figure 5-1).

Figure 5-2. Sensitivity to feature scales

SVMs are sensitive to the feature scales, as you can see in Figure 5-2: in the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible street is close to horizontal. After feature scaling (e.g., using Scikit-Learn's `StandardScaler`), the decision boundary in the right plot looks much better.

Soft Margin Classification

If we strictly impose that all instances must be off the street and on the right side, this is called *hard margin classification*. There are two main issues with hard margin classification. First, it only works if the data is linearly separable. Second, it is sensitive to outliers. Figure 5-3 shows the iris dataset with just one additional outlier: on the left, it is impossible to find a hard margin; on the right, the decision boundary ends up very different from the one we saw in Figure 5-1 without the outlier, and it will probably not generalize as well.

Figure 5-3. Hard margin sensitivity to outliers

To avoid these issues, use a more flexible model. The objective is to find a good balance between keeping the street as large as possible and limiting the *margin violations* (i.e., instances that end up in the middle of the street or even on the wrong side). This is called *soft margin classification*.

When creating an SVM model using Scikit-Learn, we can specify a number of hyperparameters. `C` is one of those hyperparameters. If we set it to a low value, then we end up with the model on the left of Figure 5-4. With a high value, we get the model on the right. Margin violations are bad. It's usually better to have few of them. However, in this case the model on the left has a lot of margin violations but will probably generalize better.

Figure 5-4. Large margin (left) versus fewer margin violations (right)

If your SVM model is overfitting, you can try regularizing it by reducing `C`.

The following Scikit-Learn code loads the iris dataset, scales the features, and then trains a linear SVM model (using the `LinearSVC` class with `C=1` and the *hinge loss* function, described shortly) to detect *Iris virginica* flowers:

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.float64)  # Iris virginica

svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=1, loss="hinge")),
    ])

svm_clf.fit(X, y)
```

The resulting model is represented on the left in Figure 5-4.

Then, as usual, you can use the model to make predictions:

```
>>> svm_clf.predict([[5.5, 1.7]])
array([1.])
```

Unlike Logistic Regression classifiers, SVM classifiers do not output probabilities for each class.

Instead of using the `LinearSVC` class, we could use the `SVC` class with a linear kernel. When creating the SVC model, we would write `SVC(kernel="linear", C=1)`. Or we could use the `SGDClassifier` class, with `SGDClassifier(loss="hinge", alpha=1/(m*C))`. This applies regular Stochastic Gradient Descent (see Chapter 4) to train a linear SVM classifier. It does not converge as fast as the `LinearSVC` class, but it can be useful to handle online classification tasks or huge datasets that do not fit in memory (out-of-core training).

The `LinearSVC` class regularizes the bias term, so you should center the training set first by subtracting its mean. This is automatic if you scale the data using the `StandardScaler`. Also make sure you set the `loss` hyperparameter to `"hinge"`, as it is not the default value. Finally, for better performance, you should set the `dual` hyperparameter to `False`, unless there are more features than training instances (we will discuss duality later in the chapter).

Nonlinear SVM Classification

Although linear SVM classifiers are efficient and work surprisingly well in many cases, many datasets are not even close to being linearly separable. One approach to handling nonlinear datasets is to add more features, such as polynomial features (as you did in Chapter 4); in some cases this can result in a linearly separable dataset. Consider the left plot in Figure 5-5: it represents a simple dataset with just one feature, x_1. This dataset is not linearly separable, as you can see. But if you add a second feature $x_2 = (x_1)^2$, the resulting 2D dataset is perfectly linearly separable.

Figure 5-5. Adding features to make a dataset linearly separable

To implement this idea using Scikit-Learn, create a `Pipeline` containing a `PolynomialFeatures` transformer (discussed in "Polynomial Regression" on page 128), followed by a `StandardScaler` and a `LinearSVC`. Let's test this on the moons dataset: this is a toy dataset for binary classification in which the data points are shaped as two interleaving half circles (see Figure 5-6). You can generate this dataset using the `make_moons()` function:

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

X, y = make_moons(n_samples=100, noise=0.15)
polynomial_svm_clf = Pipeline([
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge"))
    ])

polynomial_svm_clf.fit(X, y)
```

Figure 5-6. Linear SVM classifier using polynomial features

Polynomial Kernel

Adding polynomial features is simple to implement and can work great with all sorts of Machine Learning algorithms (not just SVMs). That said, at a low polynomial degree, this method cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow.

Fortunately, when using SVMs you can apply an almost miraculous mathematical technique called the *kernel trick* (explained in a moment). The kernel trick makes it possible to get the same result as if you had added many polynomial features, even with very high-degree polynomials, without actually having to add them. So there is no combinatorial explosion of the number of features because you don't actually add any features. This trick is implemented by the `SVC` class. Let's test it on the moons dataset:

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)
```

This code trains an SVM classifier using a third-degree polynomial kernel. It is represented on the left in Figure 5-7. On the right is another SVM classifier using a 10th-degree polynomial kernel. Obviously, if your model is overfitting, you might want to reduce the polynomial degree. Conversely, if it is underfitting, you can try increasing it. The hyperparameter `coef0` controls how much the model is influenced by high-degree polynomials versus low-degree polynomials.

Figure 5-7. SVM classifiers with a polynomial kernel

A common approach to finding the right hyperparameter values is to use grid search (see Chapter 2). It is often faster to first do a very coarse grid search, then a finer grid search around the best values found. Having a good sense of what each hyperparameter actually does can also help you search in the right part of the hyperparameter space.

Similarity Features

Another technique to tackle nonlinear problems is to add features computed using a *similarity function*, which measures how much each instance resembles a particular *landmark*. For example, let's take the 1D dataset discussed earlier and add two landmarks to it at $x_1 = -2$ and $x_1 = 1$ (see the left plot in Figure 5-8). Next, let's define the similarity function to be the Gaussian *Radial Basis Function* (RBF) with $\gamma = 0.3$ (see Equation 5-1).

Equation 5-1. Gaussian RBF

$$\phi_\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma \| \mathbf{x} - \ell \|^2\right)$$

This is a bell-shaped function varying from 0 (very far away from the landmark) to 1 (at the landmark). Now we are ready to compute the new features. For example, let's look at the instance $x_1 = -1$: it is located at a distance of 1 from the first landmark and 2 from the second landmark. Therefore its new features are $x_2 = \exp(-0.3 \times 1^2) \approx 0.74$ and $x_3 = \exp(-0.3 \times 2^2) \approx 0.30$. The plot on the right in Figure 5-8 shows the transformed dataset (dropping the original features). As you can see, it is now linearly separable.

Figure 5-8. Similarity features using the Gaussian RBF

You may wonder how to select the landmarks. The simplest approach is to create a landmark at the location of each and every instance in the dataset. Doing that creates many dimensions and thus increases the chances that the transformed training set will be linearly separable. The downside is that a training set with m instances and n features gets transformed into a training set with m instances and m features (assuming you drop the original features). If your training set is very large, you end up with an equally large number of features.

Gaussian RBF Kernel

Just like the polynomial features method, the similarity features method can be useful with any Machine Learning algorithm, but it may be computationally expensive to compute all the additional features, especially on large training sets. Once again the kernel trick does its SVM magic, making it possible to obtain a similar result as if you had added many similarity features. Let's try the `SVC` class with the Gaussian RBF kernel:

```
rbf_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ])
rbf_kernel_svm_clf.fit(X, y)
```

This model is represented at the bottom left in Figure 5-9. The other plots show models trained with different values of hyperparameters `gamma` (γ) and `C`. Increasing `gamma` makes the bell-shaped curve narrower (see the lefthand plots in Figure 5-8). As a result, each instance's range of influence is smaller: the decision boundary ends up being more irregular, wiggling around individual instances. Conversely, a small `gamma` value makes the bell-shaped curve wider: instances have a larger range of influence, and the decision boundary ends up smoother. So γ acts like a regularization

hyperparameter: if your model is overfitting, you should reduce it; if it is underfitting, you should increase it (similar to the `C` hyperparameter).

Figure 5-9. SVM classifiers using an RBF kernel

Other kernels exist but are used much more rarely. Some kernels are specialized for specific data structures. *String kernels* are sometimes used when classifying text documents or DNA sequences (e.g., using the *string subsequence kernel* or kernels based on the *Levenshtein distance*).

With so many kernels to choose from, how can you decide which one to use? As a rule of thumb, you should always try the linear kernel first (remember that `LinearSVC` is much faster than `SVC(kernel="linear")`), especially if the training set is very large or if it has plenty of features. If the training set is not too large, you should also try the Gaussian RBF kernel; it works well in most cases. Then if you have spare time and computing power, you can experiment with a few other kernels, using cross-validation and grid search. You'd want to experiment like that especially if there are kernels specialized for your training set's data structure.

Computational Complexity

The `LinearSVC` class is based on the `liblinear` library, which implements an optimized algorithm for linear SVMs.[1] It does not support the kernel trick, but it scales almost linearly with the number of training instances and the number of features. Its training time complexity is roughly $O(m \times n)$.

The algorithm takes longer if you require very high precision. This is controlled by the tolerance hyperparameter ϵ (called `tol` in Scikit-Learn). In most classification tasks, the default tolerance is fine.

The `SVC` class is based on the `libsvm` library, which implements an algorithm that supports the kernel trick.[2] The training time complexity is usually between $O(m^2 \times n)$ and $O(m^3 \times n)$. Unfortunately, this means that it gets dreadfully slow when the number of training instances gets large (e.g., hundreds of thousands of instances). This algorithm is perfect for complex small or medium-sized training sets. It scales well with the number of features, especially with *sparse features* (i.e., when each instance has few nonzero features). In this case, the algorithm scales roughly with the average number of nonzero features per instance. Table 5-1 compares Scikit-Learn's SVM classification classes.

Table 5-1. Comparison of Scikit-Learn classes for SVM classification

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
`LinearSVC`	$O(m \times n)$	No	Yes	No
`SGDClassifier`	$O(m \times n)$	Yes	Yes	No
`SVC`	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

SVM Regression

As mentioned earlier, the SVM algorithm is versatile: not only does it support linear and nonlinear classification, but it also supports linear and nonlinear regression. To use SVMs for regression instead of classification, the trick is to reverse the objective: instead of trying to fit the largest possible street between two classes while limiting margin violations, SVM Regression tries to fit as many instances as possible *on* the street while limiting margin violations (i.e., instances *off* the street). The width of the street is controlled by a hyperparameter, ϵ. Figure 5-10 shows two linear SVM

1 Chih-Jen Lin et al., "A Dual Coordinate Descent Method for Large-Scale Linear SVM," *Proceedings of the 25th International Conference on Machine Learning* (2008): 408–415.

2 John Platt, "Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines" (Microsoft Research technical report, April 21, 1998), *https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-98-14.pdf*.

Regression models trained on some random linear data, one with a large margin ($\epsilon = 1.5$) and the other with a small margin ($\epsilon = 0.5$).

Figure 5-10. SVM Regression

Adding more training instances within the margin does not affect the model's predictions; thus, the model is said to be *ϵ-insensitive*.

You can use Scikit-Learn's `LinearSVR` class to perform linear SVM Regression. The following code produces the model represented on the left in Figure 5-10 (the training data should be scaled and centered first):

```
from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

To tackle nonlinear regression tasks, you can use a kernelized SVM model. Figure 5-11 shows SVM Regression on a random quadratic training set, using a second-degree polynomial kernel. There is little regularization in the left plot (i.e., a large `C` value), and much more regularization in the right plot (i.e., a small `C` value).

Figure 5-11. SVM Regression using a second-degree polynomial kernel

The following code uses Scikit-Learn's `SVR` class (which supports the kernel trick) to produce the model represented on the left in Figure 5-11:

```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```

The `SVR` class is the regression equivalent of the `SVC` class, and the `LinearSVR` class is the regression equivalent of the `LinearSVC` class. The `LinearSVR` class scales linearly with the size of the training set (just like the `LinearSVC` class), while the `SVR` class gets much too slow when the training set grows large (just like the `SVC` class).

SVMs can also be used for outlier detection; see Scikit-Learn's documentation for more details.

Under the Hood

This section explains how SVMs make predictions and how their training algorithms work, starting with linear SVM classifiers. If you are just getting started with Machine Learning, you can safely skip it and go straight to the exercises at the end of this chapter, and come back later when you want to get a deeper understanding of SVMs.

First, a word about notations. In Chapter 4 we used the convention of putting all the model parameters in one vector $\boldsymbol{\theta}$, including the bias term θ_0 and the input feature weights θ_1 to θ_n, and adding a bias input $x_0 = 1$ to all instances. In this chapter we will use a convention that is more convenient (and more common) when dealing with

SVMs: the bias term will be called b, and the feature weights vector will be called $\mathbf{w}$. No bias feature will be added to the input feature vectors.

Decision Function and Predictions

The linear SVM classifier model predicts the class of a new instance $\mathbf{x}$ by simply computing the decision function $\mathbf{w}^\top \mathbf{x} + b = w_1 x_1 + \cdots + w_n x_n + b$. If the result is positive, the predicted class $\hat{y}$ is the positive class (1), and otherwise it is the negative class (0); see Equation 5-2.

Equation 5-2. Linear SVM classifier prediction

$$\hat{y} = \begin{cases} 0 & \text{if } \mathbf{w}^\top \mathbf{x} + b < 0, \\ 1 & \text{if } \mathbf{w}^\top \mathbf{x} + b \geq 0 \end{cases}$$

Figure 5-12 shows the decision function that corresponds to the model in the left in Figure 5-4: it is a 2D plane because this dataset has two features (petal width and petal length). The decision boundary is the set of points where the decision function is equal to 0: it is the intersection of two planes, which is a straight line (represented by the thick solid line).[3]

Figure 5-12. Decision function for the iris dataset

3 More generally, when there are n features, the decision function is an n-dimensional *hyperplane*, and the decision boundary is an $(n - 1)$-dimensional hyperplane.

The dashed lines represent the points where the decision function is equal to 1 or –1: they are parallel and at equal distance to the decision boundary, and they form a margin around it. Training a linear SVM classifier means finding the values of **w** and b that make this margin as wide as possible while avoiding margin violations (hard margin) or limiting them (soft margin).

Training Objective

Consider the slope of the decision function: it is equal to the norm of the weight vector, $\| \mathbf{w} \|$. If we divide this slope by 2, the points where the decision function is equal to ±1 are going to be twice as far away from the decision boundary. In other words, dividing the slope by 2 will multiply the margin by 2. This may be easier to visualize in 2D, as shown in Figure 5-13. The smaller the weight vector **w**, the larger the margin.

Figure 5-13. A smaller weight vector results in a larger margin

So we want to minimize $\| \mathbf{w} \|$ to get a large margin. If we also want to avoid any margin violations (hard margin), then we need the decision function to be greater than 1 for all positive training instances and lower than –1 for negative training instances. If we define $t^{(i)} = -1$ for negative instances (if $y^{(i)} = 0$) and $t^{(i)} = 1$ for positive instances (if $y^{(i)} = 1$), then we can express this constraint as $t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1$ for all instances.

We can therefore express the hard margin linear SVM classifier objective as the constrained optimization problem in Equation 5-3.

Equation 5-3. Hard margin linear SVM classifier objective

$$\underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^\top \mathbf{w}$$

$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b\right) \geq 1 \quad \text{for } i = 1, 2, \cdots, m$$

We are minimizing ½ $\mathbf{w}^\top \mathbf{w}$, which is equal to ½$\|\mathbf{w}\|^2$, rather than minimizing $\|\mathbf{w}\|$. Indeed, ½$\|\mathbf{w}\|^2$ has a nice, simple derivative (it is just $\mathbf{w}$), while $\|\mathbf{w}\|$ is not differentiable at $\mathbf{w} = 0$. Optimization algorithms work much better on differentiable functions.

To get the soft margin objective, we need to introduce a *slack variable* $\zeta^{(i)} \geq 0$ for each instance:[4] $\zeta^{(i)}$ measures how much the ith instance is allowed to violate the margin. We now have two conflicting objectives: make the slack variables as small as possible to reduce the margin violations, and make ½ $\mathbf{w}^\top \mathbf{w}$ as small as possible to increase the margin. This is where the `C` hyperparameter comes in: it allows us to define the trade-off between these two objectives. This gives us the constrained optimization problem in Equation 5-4.

Equation 5-4. Soft margin linear SVM classifier objective

$$\underset{\mathbf{w}, b, \zeta}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^\top \mathbf{w} + C \sum_{i=1}^{m} \zeta^{(i)}$$

$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b\right) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

Quadratic Programming

The hard margin and soft margin problems are both convex quadratic optimization problems with linear constraints. Such problems are known as *Quadratic Programming* (QP) problems. Many off-the-shelf solvers are available to solve QP problems by using a variety of techniques that are outside the scope of this book.[5]

4 Zeta (ζ) is the sixth letter of the Greek alphabet.

5 To learn more about Quadratic Programming, you can start by reading Stephen Boyd and Lieven Vandenberghe's book *Convex Optimization* (Cambridge University Press, 2004) or watch Richard Brown's series of video lectures.

The general problem formulation is given by Equation 5-5.

Equation 5-5. Quadratic Programming problem

$$\begin{aligned}
&\underset{\mathbf{p}}{\text{Minimize}} \quad \frac{1}{2}\mathbf{p}^\intercal\mathbf{H}\mathbf{p} \quad + \quad \mathbf{f}^\intercal\mathbf{p} \\
&\text{subject to} \quad \mathbf{A}\mathbf{p} \le \mathbf{b} \\
&\qquad\text{where} \begin{cases} \mathbf{p} & \text{is an } n_p\text{-dimensional vector } (n_p = \text{number of parameters}), \\ \mathbf{H} & \text{is an } n_p \times n_p \text{ matrix}, \\ \mathbf{f} & \text{is an } n_p\text{-dimensional vector}, \\ \mathbf{A} & \text{is an } n_c \times n_p \text{ matrix } (n_c = \text{number of constraints}), \\ \mathbf{b} & \text{is an } n_c\text{-dimensional vector}. \end{cases}
\end{aligned}$$

Note that the expression $\mathbf{A}\ \mathbf{p} \le \mathbf{b}$ defines n_c constraints: $\mathbf{p}^\intercal\ \mathbf{a}^{(i)} \le b^{(i)}$ for $i = 1, 2, \cdots, n_c$, where $\mathbf{a}^{(i)}$ is the vector containing the elements of the ith row of $\mathbf{A}$ and $b^{(i)}$ is the ith element of $\mathbf{b}$.

You can easily verify that if you set the QP parameters in the following way, you get the hard margin linear SVM classifier objective:

- $n_p = n + 1$, where n is the number of features (the +1 is for the bias term).
- $n_c = m$, where m is the number of training instances.
- $\mathbf{H}$ is the $n_p \times n_p$ identity matrix, except with a zero in the top-left cell (to ignore the bias term).
- $\mathbf{f} = 0$, an n_p-dimensional vector full of 0s.
- $\mathbf{b} = -1$, an n_c-dimensional vector full of −1s.
- $\mathbf{a}^{(i)} = -t^{(i)}\ \dot{\mathbf{x}}^{(i)}$, where $\dot{\mathbf{x}}^{(i)}$ is equal to $\mathbf{x}^{(i)}$ with an extra bias feature $\dot{\mathbf{x}}_0 = 1$.

One way to train a hard margin linear SVM classifier is to use an off-the-shelf QP solver and pass it the preceding parameters. The resulting vector $\mathbf{p}$ will contain the bias term $b = p_0$ and the feature weights $w_i = p_i$ for $i = 1, 2, \cdots, n$. Similarly, you can use a QP solver to solve the soft margin problem (see the exercises at the end of the chapter).

To use the kernel trick, we are going to look at a different constrained optimization problem.

The Dual Problem

Given a constrained optimization problem, known as the *primal problem*, it is possible to express a different but closely related problem, called its *dual problem*. The

solution to the dual problem typically gives a lower bound to the solution of the primal problem, but under some conditions it can have the same solution as the primal problem. Luckily, the SVM problem happens to meet these conditions,[6] so you can choose to solve the primal problem or the dual problem; both will have the same solution. Equation 5-6 shows the dual form of the linear SVM objective (if you are interested in knowing how to derive the dual problem from the primal problem, see Appendix C).

Equation 5-6. Dual form of the linear SVM objective

$$\underset{\alpha}{\text{minimize}} \; \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)\top} \mathbf{x}^{(j)} - \sum_{i=1}^{m} \alpha^{(i)}$$

$$\text{subject to} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

Once you find the vector $\widehat{\boldsymbol{\alpha}}$ that minimizes this equation (using a QP solver), use Equation 5-7 to compute $\widehat{\mathbf{w}}$ and $\hat{b}$ that minimize the primal problem.

Equation 5-7. From the dual solution to the primal solution

$$\widehat{\mathbf{w}} = \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left(t^{(i)} - \widehat{\mathbf{w}}^\top \mathbf{x}^{(i)} \right)$$

The dual problem is faster to solve than the primal one when the number of training instances is smaller than the number of features. More importantly, the dual problem makes the kernel trick possible, while the primal does not. So what is this kernel trick, anyway?

Kernelized SVMs

Suppose you want to apply a second-degree polynomial transformation to a two-dimensional training set (such as the moons training set), then train a linear SVM classifier on the transformed training set. Equation 5-8 shows the second-degree polynomial mapping function ϕ that you want to apply.

6 The objective function is convex, and the inequality constraints are continuously differentiable and convex functions.

Equation 5-8. Second-degree polynomial mapping

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} {x_1}^2 \\ \sqrt{2}\, x_1 x_2 \\ {x_2}^2 \end{pmatrix}$$

Notice that the transformed vector is 3D instead of 2D. Now let's look at what happens to a couple of 2D vectors, **a** and **b**, if we apply this second-degree polynomial mapping and then compute the dot product[7] of the transformed vectors (See Equation 5-9).

Equation 5-9. Kernel trick for a second-degree polynomial mapping

$$\begin{aligned} \phi(\mathbf{a})^\top \phi(\mathbf{b}) &= \begin{pmatrix} {a_1}^2 \\ \sqrt{2}\, a_1 a_2 \\ {a_2}^2 \end{pmatrix}^\top \begin{pmatrix} {b_1}^2 \\ \sqrt{2}\, b_1 b_2 \\ {b_2}^2 \end{pmatrix} = {a_1}^2 {b_1}^2 + 2 a_1 b_1 a_2 b_2 + {a_2}^2 {b_2}^2 \\ &= (a_1 b_1 + a_2 b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^\top \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right)^2 = (\mathbf{a}^\top \mathbf{b})^2 \end{aligned}$$

How about that? The dot product of the transformed vectors is equal to the square of the dot product of the original vectors: $\phi(\mathbf{a})^\top \phi(\mathbf{b}) = (\mathbf{a}^\top \mathbf{b})^2$.

Here is the key insight: if you apply the transformation ϕ to all training instances, then the dual problem (see Equation 5-6) will contain the dot product $\phi(\mathbf{x}^{(i)})^\top \phi(\mathbf{x}^{(j)})$. But if ϕ is the second-degree polynomial transformation defined in Equation 5-8, then you can replace this dot product of transformed vectors simply by $\left(\mathbf{x}^{(i)\top}\mathbf{x}^{(j)}\right)^2$. So, you don't need to transform the training instances at all; just replace the dot product by its square in Equation 5-6. The result will be strictly the same as if you had gone through the trouble of transforming the training set then fitting a linear SVM algorithm, but this trick makes the whole process much more computationally efficient.

The function $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^\top \mathbf{b})^2$ is a second-degree polynomial kernel. In Machine Learning, a *kernel* is a function capable of computing the dot product $\phi(\mathbf{a})^\top \phi(\mathbf{b})$,

7 As explained in Chapter 4, the dot product of two vectors **a** and **b** is normally noted $\mathbf{a} \cdot \mathbf{b}$. However, in Machine Learning, vectors are frequently represented as column vectors (i.e., single-column matrices), so the dot product is achieved by computing $\mathbf{a}^\top\mathbf{b}$. To remain consistent with the rest of the book, we will use this notation here, ignoring the fact that this technically results in a single-cell matrix rather than a scalar value.

based only on the original vectors **a** and **b**, without having to compute (or even to know about) the transformation ϕ. Equation 5-10 lists some of the most commonly used kernels.

Equation 5-10. Common kernels

$$\begin{aligned}
\text{Linear:} \quad & K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^\top \mathbf{b} \\
\text{Polynomial:} \quad & K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^\top \mathbf{b} + r)^d \\
\text{Gaussian RBF:} \quad & K(\mathbf{a}, \mathbf{b}) = \exp\left(-\gamma \| \mathbf{a} - \mathbf{b} \|^2\right) \\
\text{Sigmoid:} \quad & K(\mathbf{a}, \mathbf{b}) = \tanh\left(\gamma \mathbf{a}^\top \mathbf{b} + r\right)
\end{aligned}$$

Mercer's Theorem

According to *Mercer's theorem*, if a function $K(\mathbf{a}, \mathbf{b})$ respects a few mathematical conditions called *Mercer's conditions* (e.g., K must be continuous and symmetric in its arguments so that $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$, etc.), then there exists a function ϕ that maps **a** and **b** into another space (possibly with much higher dimensions) such that $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^\top \phi(\mathbf{b})$. You can use K as a kernel because you know ϕ exists, even if you don't know what ϕ is. In the case of the Gaussian RBF kernel, it can be shown that ϕ maps each training instance to an infinite-dimensional space, so it's a good thing you don't need to actually perform the mapping!

Note that some frequently used kernels (such as the sigmoid kernel) don't respect all of Mercer's conditions, yet they generally work well in practice.

There is still one loose end we must tie up. Equation 5-7 shows how to go from the dual solution to the primal solution in the case of a linear SVM classifier. But if you apply the kernel trick, you end up with equations that include $\phi(x^{(i)})$. In fact, $\widehat{\mathbf{w}}$ must have the same number of dimensions as $\phi(x^{(i)})$, which may be huge or even infinite, so you can't compute it. But how can you make predictions without knowing $\widehat{\mathbf{w}}$? Well, the good news is that you can plug the formula for $\widehat{\mathbf{w}}$ from Equation 5-7 into the decision function for a new instance $\mathbf{x}^{(n)}$, and you get an equation with only dot products between input vectors. This makes it possible to use the kernel trick (Equation 5-11).

Equation 5-11. Making predictions with a kernelized SVM

$$
\begin{aligned}
h_{\widehat{\mathbf{w}}, \hat{b}}\left(\phi\left(\mathbf{x}^{(n)}\right)\right) &= \widehat{\mathbf{w}}^{\mathsf{T}}\phi\left(\mathbf{x}^{(n)}\right) + \hat{b} = \left(\sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \phi\left(\mathbf{x}^{(i)}\right)\right)^{\mathsf{T}} \phi\left(\mathbf{x}^{(n)}\right) + \hat{b} \\
&= \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \left(\phi\left(\mathbf{x}^{(i)}\right)^{\mathsf{T}} \phi\left(\mathbf{x}^{(n)}\right)\right) + \hat{b} \\
&= \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \hat{\alpha}^{(i)} t^{(i)} K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}\right) + \hat{b}
\end{aligned}
$$

Note that since $\alpha^{(i)} \neq 0$ only for support vectors, making predictions involves computing the dot product of the new input vector $\mathbf{x}^{(n)}$ with only the support vectors, not all the training instances. Of course, you need to use the same trick to compute the bias term $\hat{b}$ (Equation 5-12).

Equation 5-12. Using the kernel trick to compute the bias term

$$
\begin{aligned}
\hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left(t^{(i)} - \widehat{\mathbf{w}}^{\mathsf{T}}\phi\left(\mathbf{x}^{(i)}\right)\right) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left(t^{(i)} - \left(\sum_{j=1}^{m} \hat{\alpha}^{(j)} t^{(j)} \phi\left(\mathbf{x}^{(j)}\right)\right)^{\mathsf{T}} \phi\left(\mathbf{x}^{(i)}\right)\right) \\
&= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left(t^{(i)} - \sum_{\substack{j=1 \\ \hat{\alpha}^{(j)} > 0}}^{m} \hat{\alpha}^{(j)} t^{(j)} K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right)\right)
\end{aligned}
$$

If you are starting to get a headache, it's perfectly normal: it's an unfortunate side effect of the kernel trick.

Online SVMs

Before concluding this chapter, let's take a quick look at online SVM classifiers (recall that online learning means learning incrementally, typically as new instances arrive).

For linear SVM classifiers, one method for implementing an online SVM classifier is to use Gradient Descent (e.g., using `SGDClassifier`) to minimize the cost function in Equation 5-13, which is derived from the primal problem. Unfortunately, Gradient Descent converges much more slowly than the methods based on QP.

Equation 5-13. Linear SVM classifier cost function

$$J(\mathbf{w}, b) = \frac{1}{2}\mathbf{w}^\top\mathbf{w} \quad + \quad C\sum_{i=1}^{m} max\left(0, 1 - t^{(i)}\left(\mathbf{w}^\top\mathbf{x}^{(i)} + b\right)\right)$$

The first sum in the cost function will push the model to have a small weight vector **w**, leading to a larger margin. The second sum computes the total of all margin violations. An instance's margin violation is equal to 0 if it is located off the street and on the correct side, or else it is proportional to the distance to the correct side of the street. Minimizing this term ensures that the model makes the margin violations as small and as few as possible.

Hinge Loss

The function $max(0, 1 - t)$ is called the *hinge loss* function (see the following image). It is equal to 0 when $t \geq 1$. Its derivative (slope) is equal to -1 if $t < 1$ and 0 if $t > 1$. It is not differentiable at $t = 1$, but just like for Lasso Regression (see "Lasso Regression" on page 137), you can still use Gradient Descent using any *subderivative* at $t = 1$ (i.e., any value between -1 and 0).

It is also possible to implement online kernelized SVMs, as described in the papers "Incremental and Decremental Support Vector Machine Learning"[8] and "Fast Kernel Classifiers with Online and Active Learning".[9] These kernelized SVMs are imple-

8 Gert Cauwenberghs and Tomaso Poggio, "Incremental and Decremental Support Vector Machine Learning," *Proceedings of the 13th International Conference on Neural Information Processing Systems* (2000): 388–394.

9 Antoine Bordes et al., "Fast Kernel Classifiers with Online and Active Learning," *Journal of Machine Learning Research* 6 (2005): 1579–1619.

mented in Matlab and C++. For large-scale nonlinear problems, you may want to consider using neural networks instead (see Part II).

Exercises

1. What is the fundamental idea behind Support Vector Machines?
2. What is a support vector?
3. Why is it important to scale the inputs when using SVMs?
4. Can an SVM classifier output a confidence score when it classifies an instance? What about a probability?
5. Should you use the primal or the dual form of the SVM problem to train a model on a training set with millions of instances and hundreds of features?
6. Say you've trained an SVM classifier with an RBF kernel, but it seems to underfit the training set. Should you increase or decrease γ (`gamma`)? What about `C`?
7. How should you set the QP parameters ($\mathbf{H}$, $\mathbf{f}$, $\mathbf{A}$, and $\mathbf{b}$) to solve the soft margin linear SVM classifier problem using an off-the-shelf QP solver?
8. Train a `LinearSVC` on a linearly separable dataset. Then train an `SVC` and a `SGDClassifier` on the same dataset. See if you can get them to produce roughly the same model.
9. Train an SVM classifier on the MNIST dataset. Since SVM classifiers are binary classifiers, you will need to use one-versus-the-rest to classify all 10 digits. You may want to tune the hyperparameters using small validation sets to speed up the process. What accuracy can you reach?
10. Train an SVM regressor on the California housing dataset.

Solutions to these exercises are available in Appendix A.

CHAPTER 6

Decision Trees

Like SVMs, *Decision Trees* are versatile Machine Learning algorithms that can perform both classification and regression tasks, and even multioutput tasks. They are powerful algorithms, capable of fitting complex datasets. For example, in Chapter 2 you trained a `DecisionTreeRegressor` model on the California housing dataset, fitting it perfectly (actually, overfitting it).

Decision Trees are also the fundamental components of Random Forests (see Chapter 7), which are among the most powerful Machine Learning algorithms available today.

In this chapter we will start by discussing how to train, visualize, and make predictions with Decision Trees. Then we will go through the CART training algorithm used by Scikit-Learn, and we will discuss how to regularize trees and use them for regression tasks. Finally, we will discuss some of the limitations of Decision Trees.

Training and Visualizing a Decision Tree

To understand Decision Trees, let's build one and take a look at how it makes predictions. The following code trains a `DecisionTreeClassifier` on the iris dataset (see Chapter 4):

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

You can visualize the trained Decision Tree by first using the `export_graphviz()` method to output a graph definition file called *iris_tree.dot*:

```
from sklearn.tree import export_graphviz

export_graphviz(
        tree_clf,
        out_file=image_path("iris_tree.dot"),
        feature_names=iris.feature_names[2:],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
```

Then you can use the `dot` command-line tool from the Graphviz package to convert this *.dot* file to a variety of formats, such as PDF or PNG.[1] This command line converts the *.dot* file to a *.png* image file:

```
$ dot -Tpng iris_tree.dot -o iris_tree.png
```

Your first Decision Tree looks like Figure 6-1.

Figure 6-1. Iris Decision Tree

Making Predictions

Let's see how the tree represented in Figure 6-1 makes predictions. Suppose you find an iris flower and you want to classify it. You start at the *root node* (depth 0, at the top): this node asks whether the flower's petal length is smaller than 2.45 cm. If it is, then you move down to the root's left child node (depth 1, left). In this case, it is a *leaf*

1 Graphviz is an open source graph visualization software package, available at *http://www.graphviz.org/*.

node (i.e., it does not have any child nodes), so it does not ask any questions: simply look at the predicted class for that node, and the Decision Tree predicts that your flower is an *Iris setosa* (`class=setosa`).

Now suppose you find another flower, and this time the petal length is greater than 2.45 cm. You must move down to the root's right child node (depth 1, right), which is not a leaf node, so the node asks another question: is the petal width smaller than 1.75 cm? If it is, then your flower is most likely an *Iris versicolor* (depth 2, left). If not, it is likely an *Iris virginica* (depth 2, right). It's really that simple.

One of the many qualities of Decision Trees is that they require very little data preparation. In fact, they don't require feature scaling or centering at all.

A node's `samples` attribute counts how many training instances it applies to. For example, 100 training instances have a petal length greater than 2.45 cm (depth 1, right), and of those 100, 54 have a petal width smaller than 1.75 cm (depth 2, left). A node's `value` attribute tells you how many training instances of each class this node applies to: for example, the bottom-right node applies to 0 *Iris setosa*, 1 *Iris versicolor*, and 45 *Iris virginica*. Finally, a node's `gini` attribute measures its *impurity*: a node is "pure" (`gini=0`) if all training instances it applies to belong to the same class. For example, since the depth-1 left node applies only to *Iris setosa* training instances, it is pure and its `gini` score is 0. Equation 6-1 shows how the training algorithm computes the `gini` score G_i of the ith node. The depth-2 left node has a `gini` score equal to $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$.

Equation 6-1. Gini impurity

$$G_i = 1 - \sum_{k=1}^{n} {p_{i,k}}^2$$

In this equation:

- $p_{i,k}$ is the ratio of class k instances among the training instances in the ith node.

Scikit-Learn uses the CART algorithm, which produces only *binary trees*: nonleaf nodes always have two children (i.e., questions only have yes/no answers). However, other algorithms such as ID3 can produce Decision Trees with nodes that have more than two children.

Figure 6-2 shows this Decision Tree's decision boundaries. The thick vertical line represents the decision boundary of the root node (depth 0): petal length = 2.45 cm. Since the lefthand area is pure (only *Iris setosa*), it cannot be split any further. However, the righthand area is impure, so the depth-1 right node splits it at petal width = 1.75 cm (represented by the dashed line). Since `max_depth` was set to 2, the Decision Tree stops right there. If you set `max_depth` to 3, then the two depth-2 nodes would each add another decision boundary (represented by the dotted lines).

Figure 6-2. Decision Tree decision boundaries

Model Interpretation: White Box Versus Black Box

Decision Trees are intuitive, and their decisions are easy to interpret. Such models are often called *white box models*. In contrast, as we will see, Random Forests or neural networks are generally considered *black box models*. They make great predictions, and you can easily check the calculations that they performed to make these predictions; nevertheless, it is usually hard to explain in simple terms why the predictions were made. For example, if a neural network says that a particular person appears on a picture, it is hard to know what contributed to this prediction: did the model recognize that person's eyes? Their mouth? Their nose? Their shoes? Or even the couch that they were sitting on? Conversely, Decision Trees provide nice, simple classification rules that can even be applied manually if need be (e.g., for flower classification).

Estimating Class Probabilities

A Decision Tree can also estimate the probability that an instance belongs to a particular class k. First it traverses the tree to find the leaf node for this instance, and then it returns the ratio of training instances of class k in this node. For example, suppose you have found a flower whose petals are 5 cm long and 1.5 cm wide. The

corresponding leaf node is the depth-2 left node, so the Decision Tree should output the following probabilities: 0% for *Iris setosa* (0/54), 90.7% for *Iris versicolor* (49/54), and 9.3% for *Iris virginica* (5/54). And if you ask it to predict the class, it should output *Iris versicolor* (class 1) because it has the highest probability. Let's check this:

```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[0.        , 0.90740741, 0.09259259]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

Perfect! Notice that the estimated probabilities would be identical anywhere else in the bottom-right rectangle of Figure 6-2—for example, if the petals were 6 cm long and 1.5 cm wide (even though it seems obvious that it would most likely be an *Iris virginica* in this case).

The CART Training Algorithm

Scikit-Learn uses the *Classification and Regression Tree* (CART) algorithm to train Decision Trees (also called "growing" trees). The algorithm works by first splitting the training set into two subsets using a single feature k and a threshold t_k (e.g., "petal length ≤ 2.45 cm"). How does it choose k and t_k? It searches for the pair (k, t_k) that produces the purest subsets (weighted by their size). Equation 6-2 gives the cost function that the algorithm tries to minimize.

Equation 6-2. CART cost function for classification

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

$$\text{where } \begin{cases} G_{\text{left/right}} \text{ measures the impurity of the left/right subset,} \\ m_{\text{left/right}} \text{ is the number of instances in the left/right subset.} \end{cases}$$

Once the CART algorithm has successfully split the training set in two, it splits the subsets using the same logic, then the sub-subsets, and so on, recursively. It stops recursing once it reaches the maximum depth (defined by the `max_depth` hyperparameter), or if it cannot find a split that will reduce impurity. A few other hyperparameters (described in a moment) control additional stopping conditions (`min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf`, and `max_leaf_nodes`).

As you can see, the CART algorithm is a *greedy algorithm*: it greedily searches for an optimum split at the top level, then repeats the process at each subsequent level. It does not check whether or not the split will lead to the lowest possible impurity several levels down. A greedy algorithm often produces a solution that's reasonably good but not guaranteed to be optimal.

Unfortunately, finding the optimal tree is known to be an *NP-Complete* problem:[2] it requires $O(\exp(m))$ time, making the problem intractable even for small training sets. This is why we must settle for a "reasonably good" solution.

Computational Complexity

Making predictions requires traversing the Decision Tree from the root to a leaf. Decision Trees generally are approximately balanced, so traversing the Decision Tree requires going through roughly $O(\log_2(m))$ nodes.[3] Since each node only requires checking the value of one feature, the overall prediction complexity is $O(\log_2(m))$, independent of the number of features. So predictions are very fast, even when dealing with large training sets.

The training algorithm compares all features (or less if `max_features` is set) on all samples at each node. Comparing all features on all samples at each node results in a training complexity of $O(n \times m \log_2(m))$. For small training sets (less than a few thousand instances), Scikit-Learn can speed up training by presorting the data (set `presort=True`), but doing that slows down training considerably for larger training sets.

Gini Impurity or Entropy?

By default, the Gini impurity measure is used, but you can select the *entropy* impurity measure instead by setting the `criterion` hyperparameter to `"entropy"`. The concept of entropy originated in thermodynamics as a measure of molecular disorder: entropy approaches zero when molecules are still and well ordered. Entropy later spread to a wide variety of domains, including Shannon's *information theory*, where it measures the average information content of a message:[4] entropy is zero when all messages are identical. In Machine Learning, entropy is frequently used as an

2 P is the set of problems that can be solved in polynomial time. NP is the set of problems whose solutions can be verified in polynomial time. An NP-Hard problem is a problem to which any NP problem can be reduced in polynomial time. An NP-Complete problem is both NP and NP-Hard. A major open mathematical question is whether or not P = NP. If P ≠ NP (which seems likely), then no polynomial algorithm will ever be found for any NP-Complete problem (except perhaps on a quantum computer).

3 $\log_2$ is the binary logarithm. It is equal to $\log_2(m) = \log(m) / \log(2)$.

4 A reduction of entropy is often called an *information gain*.

impurity measure: a set's entropy is zero when it contains instances of only one class. Equation 6-3 shows the definition of the entropy of the ith node. For example, the depth-2 left node in Figure 6-1 has an entropy equal to $-(49/54)\log_2(49/54) - (5/54)\log_2(5/54) \approx 0.445$.

Equation 6-3. Entropy

$$H_i = -\sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^{n} p_{i,k} \log_2\left(p_{i,k}\right)$$

So, should you use Gini impurity or entropy? The truth is, most of the time it does not make a big difference: they lead to similar trees. Gini impurity is slightly faster to compute, so it is a good default. However, when they differ, Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees.[5]

Regularization Hyperparameters

Decision Trees make very few assumptions about the training data (as opposed to linear models, which assume that the data is linear, for example). If left unconstrained, the tree structure will adapt itself to the training data, fitting it very closely—indeed, most likely overfitting it. Such a model is often called a *nonparametric model*, not because it does not have any parameters (it often has a lot) but because the number of parameters is not determined prior to training, so the model structure is free to stick closely to the data. In contrast, a *parametric model*, such as a linear model, has a predetermined number of parameters, so its degree of freedom is limited, reducing the risk of overfitting (but increasing the risk of underfitting).

To avoid overfitting the training data, you need to restrict the Decision Tree's freedom during training. As you know by now, this is called regularization. The regularization hyperparameters depend on the algorithm used, but generally you can at least restrict the maximum depth of the Decision Tree. In Scikit-Learn, this is controlled by the `max_depth` hyperparameter (the default value is `None`, which means unlimited). Reducing `max_depth` will regularize the model and thus reduce the risk of overfitting.

The `DecisionTreeClassifier` class has a few other parameters that similarly restrict the shape of the Decision Tree: `min_samples_split` (the minimum number of samples a node must have before it can be split), `min_samples_leaf` (the minimum number of samples a leaf node must have), `min_weight_fraction_leaf` (same as

5 See Sebastian Raschka's interesting analysis for more details.

`min_samples_leaf` but expressed as a fraction of the total number of weighted instances), `max_leaf_nodes` (the maximum number of leaf nodes), and `max_features` (the maximum number of features that are evaluated for splitting at each node). Increasing `min_*` hyperparameters or reducing `max_*` hyperparameters will regularize the model.

Other algorithms work by first training the Decision Tree without restrictions, then *pruning* (deleting) unnecessary nodes. A node whose children are all leaf nodes is considered unnecessary if the purity improvement it provides is not statistically significant. Standard statistical tests, such as the χ^2 *test* (chi-squared test), are used to estimate the probability that the improvement is purely the result of chance (which is called the *null hypothesis*). If this probability, called the *p-value*, is higher than a given threshold (typically 5%, controlled by a hyperparameter), then the node is considered unnecessary and its children are deleted. The pruning continues until all unnecessary nodes have been pruned.

Figure 6-3 shows two Decision Trees trained on the moons dataset (introduced in Chapter 5). On the left the Decision Tree is trained with the default hyperparameters (i.e., no restrictions), and on the right it's trained with `min_samples_leaf=4`. It is quite obvious that the model on the left is overfitting, and the model on the right will probably generalize better.

Figure 6-3. Regularization using `min_samples_leaf`

Regression

Decision Trees are also capable of performing regression tasks. Let's build a regression tree using Scikit-Learn's `DecisionTreeRegressor` class, training it on a noisy quadratic dataset with `max_depth=2`:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```

The resulting tree is represented in Figure 6-4.

Figure 6-4. A Decision Tree for regression

This tree looks very similar to the classification tree you built earlier. The main difference is that instead of predicting a class in each node, it predicts a value. For example, suppose you want to make a prediction for a new instance with $x_1 = 0.6$. You traverse the tree starting at the root, and you eventually reach the leaf node that predicts `value=0.111`. This prediction is the average target value of the 110 training instances associated with this leaf node, and it results in a mean squared error equal to 0.015 over these 110 instances.

This model's predictions are represented on the left in Figure 6-5. If you set `max_depth=3`, you get the predictions represented on the right. Notice how the predicted value for each region is always the average target value of the instances in that region. The algorithm splits each region in a way that makes most training instances as close as possible to that predicted value.

Figure 6-5. Predictions of two Decision Tree regression models

The CART algorithm works mostly the same way as earlier, except that instead of trying to split the training set in a way that minimizes impurity, it now tries to split the training set in a way that minimizes the MSE. Equation 6-4 shows the cost function that the algorithm tries to minimize.

Equation 6-4. CART cost function for regression

$$J(k, t_k) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{MSE}_{\text{right}} \quad \text{where} \begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in \text{node}} \left(\hat{y}_{\text{node}} - y^{(i)}\right)^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)} \end{cases}$$

Just like for classification tasks, Decision Trees are prone to overfitting when dealing with regression tasks. Without any regularization (i.e., using the default hyperparameters), you get the predictions on the left in Figure 6-6. These predictions are obviously overfitting the training set very badly. Just setting `min_samples_leaf=10` results in a much more reasonable model, represented on the right in Figure 6-6.

Figure 6-6. Regularizing a Decision Tree regressor

Instability

Hopefully by now you are convinced that Decision Trees have a lot going for them: they are simple to understand and interpret, easy to use, versatile, and powerful. However, they do have a few limitations. First, as you may have noticed, Decision Trees love orthogonal decision boundaries (all splits are perpendicular to an axis), which makes them sensitive to training set rotation. For example, Figure 6-7 shows a simple linearly separable dataset: on the left, a Decision Tree can split it easily, while on the right, after the dataset is rotated by 45°, the decision boundary looks unnecessarily convoluted. Although both Decision Trees fit the training set perfectly, it is very likely that the model on the right will not generalize well. One way to limit this problem is to use Principal Component Analysis (see Chapter 8), which often results in a better orientation of the training data.

Figure 6-7. Sensitivity to training set rotation

More generally, the main issue with Decision Trees is that they are very sensitive to small variations in the training data. For example, if you just remove the widest *Iris versicolor* from the iris training set (the one with petals 4.8 cm long and 1.8 cm wide) and train a new Decision Tree, you may get the model represented in Figure 6-8. As you can see, it looks very different from the previous Decision Tree (Figure 6-2). Actually, since the training algorithm used by Scikit-Learn is stochastic,[6] you may get very different models even on the same training data (unless you set the `random_state` hyperparameter).

6 It randomly selects the set of features to evaluate at each node.

Figure 6-8. Sensitivity to training set details

Random Forests can limit this instability by averaging predictions over many trees, as we will see in the next chapter.

Exercises

1. What is the approximate depth of a Decision Tree trained (without restrictions) on a training set with one million instances?
2. Is a node's Gini impurity generally lower or greater than its parent's? Is it *generally* lower/greater, or *always* lower/greater?
3. If a Decision Tree is overfitting the training set, is it a good idea to try decreasing `max_depth`?
4. If a Decision Tree is underfitting the training set, is it a good idea to try scaling the input features?
5. If it takes one hour to train a Decision Tree on a training set containing 1 million instances, roughly how much time will it take to train another Decision Tree on a training set containing 10 million instances?
6. If your training set contains 100,000 instances, will setting `presort=True` speed up training?
7. Train and fine-tune a Decision Tree for the moons dataset by following these steps:
   a. Use `make_moons(n_samples=10000, noise=0.4)` to generate a moons dataset.
   b. Use `train_test_split()` to split the dataset into a training set and a test set.

c. Use grid search with cross-validation (with the help of the `GridSearchCV` class) to find good hyperparameter values for a `DecisionTreeClassifier`. Hint: try various values for `max_leaf_nodes`.

d. Train it on the full training set using these hyperparameters, and measure your model's performance on the test set. You should get roughly 85% to 87% accuracy.

8. Grow a forest by following these steps:

   a. Continuing the previous exercise, generate 1,000 subsets of the training set, each containing 100 instances selected randomly. Hint: you can use Scikit-Learn's `ShuffleSplit` class for this.

   b. Train one Decision Tree on each subset, using the best hyperparameter values found in the previous exercise. Evaluate these 1,000 Decision Trees on the test set. Since they were trained on smaller sets, these Decision Trees will likely perform worse than the first Decision Tree, achieving only about 80% accuracy.

   c. Now comes the magic. For each test set instance, generate the predictions of the 1,000 Decision Trees, and keep only the most frequent prediction (you can use SciPy's `mode()` function for this). This approach gives you *majority-vote predictions* over the test set.

   d. Evaluate these predictions on the test set: you should obtain a slightly higher accuracy than your first model (about 0.5 to 1.5% higher). Congratulations, you have trained a Random Forest classifier!

Solutions to these exercises are available in Appendix A.

CHAPTER 7

Ensemble Learning and Random Forests

Suppose you pose a complex question to thousands of random people, then aggregate their answers. In many cases you will find that this aggregated answer is better than an expert's answer. This is called the *wisdom of the crowd*. Similarly, if you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor. A group of predictors is called an *ensemble*; thus, this technique is called *Ensemble Learning*, and an Ensemble Learning algorithm is called an *Ensemble method*.

As an example of an Ensemble method, you can train a group of Decision Tree classifiers, each on a different random subset of the training set. To make predictions, you obtain the predictions of all the individual trees, then predict the class that gets the most votes (see the last exercise in Chapter 6). Such an ensemble of Decision Trees is called a *Random Forest*, and despite its simplicity, this is one of the most powerful Machine Learning algorithms available today.

As discussed in Chapter 2, you will often use Ensemble methods near the end of a project, once you have already built a few good predictors, to combine them into an even better predictor. In fact, the winning solutions in Machine Learning competitions often involve several Ensemble methods (most famously in the Netflix Prize competition).

In this chapter we will discuss the most popular Ensemble methods, including *bagging*, *boosting*, and *stacking*. We will also explore Random Forests.

Voting Classifiers

Suppose you have trained a few classifiers, each one achieving about 80% accuracy. You may have a Logistic Regression classifier, an SVM classifier, a Random Forest classifier, a K-Nearest Neighbors classifier, and perhaps a few more (see Figure 7-1).

Figure 7-1. Training diverse classifiers

A very simple way to create an even better classifier is to aggregate the predictions of each classifier and predict the class that gets the most votes. This majority-vote classifier is called a *hard voting* classifier (see Figure 7-2).

Figure 7-2. Hard voting classifier predictions

Somewhat surprisingly, this voting classifier often achieves a higher accuracy than the best classifier in the ensemble. In fact, even if each classifier is a *weak learner* (meaning it does only slightly better than random guessing), the ensemble can still be a *strong learner* (achieving high accuracy), provided there are a sufficient number of weak learners and they are sufficiently diverse.

How is this possible? The following analogy can help shed some light on this mystery. Suppose you have a slightly biased coin that has a 51% chance of coming up heads and 49% chance of coming up tails. If you toss it 1,000 times, you will generally get more or less 510 heads and 490 tails, and hence a majority of heads. If you do the math, you will find that the probability of obtaining a majority of heads after 1,000 tosses is close to 75%. The more you toss the coin, the higher the probability (e.g., with 10,000 tosses, the probability climbs over 97%). This is due to the *law of large numbers*: as you keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (51%). Figure 7-3 shows 10 series of biased coin tosses. You can see that as the number of tosses increases, the ratio of heads approaches 51%. Eventually all 10 series end up so close to 51% that they are consistently above 50%.

Figure 7-3. The law of large numbers

Similarly, suppose you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time (barely better than random guessing). If you predict the majority voted class, you can hope for up to 75% accuracy! However, this is only true if all classifiers are perfectly independent, making uncorrelated errors, which is clearly not the case because they are trained on the same data. They are likely to make the same types of errors, so there will be many majority votes for the wrong class, reducing the ensemble's accuracy.

Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is to train them using very different algorithms. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy.

The following code creates and trains a voting classifier in Scikit-Learn, composed of three diverse classifiers (the training set is the moons dataset, introduced in Chapter 5):

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

Let's look at each classifier's accuracy on the test set:

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.888
VotingClassifier 0.904
```

There you have it! The voting classifier slightly outperforms all the individual classifiers.

If all classifiers are able to estimate class probabilities (i.e., they all have a `predict_proba()` method), then you can tell Scikit-Learn to predict the class with the highest class probability, averaged over all the individual classifiers. This is called *soft voting*. It often achieves higher performance than hard voting because it gives more weight to highly confident votes. All you need to do is replace `voting="hard"` with `voting="soft"` and ensure that all classifiers can estimate class probabilities. This is not the case for the `SVC` class by default, so you need to set its `probability` hyperparameter to `True` (this will make the `SVC` class use cross-validation to estimate class probabilities, slowing down training, and it will add a `predict_proba()` method). If you modify the preceding code to use soft voting, you will find that the voting classifier achieves over 91.2% accuracy!

Bagging and Pasting

One way to get a diverse set of classifiers is to use very different training algorithms, as just discussed. Another approach is to use the same training algorithm for every predictor and train them on different random subsets of the training set. When sam-

pling is performed *with* replacement, this method is called *bagging*[1] (short for *bootstrap aggregating*[2]). When sampling is performed *without* replacement, it is called *pasting*.[3]

In other words, both bagging and pasting allow training instances to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor. This sampling and training process is represented in Figure 7-4.

Figure 7-4. Bagging and pasting involves training several predictors on different random samples of the training set

Once all predictors are trained, the ensemble can make a prediction for a new instance by simply aggregating the predictions of all predictors. The aggregation function is typically the *statistical mode* (i.e., the most frequent prediction, just like a hard voting classifier) for classification, or the average for regression. Each individual predictor has a higher bias than if it were trained on the original training set, but aggregation reduces both bias and variance.[4] Generally, the net result is that the ensemble has a similar bias but a lower variance than a single predictor trained on the original training set.

1 Leo Breiman, "Bagging Predictors," *Machine Learning* 24, no. 2 (1996): 123–140.

2 In statistics, resampling with replacement is called *bootstrapping*.

3 Leo Breiman, "Pasting Small Votes for Classification in Large Databases and On-Line," *Machine Learning* 36, no. 1–2 (1999): 85–103.

4 Bias and variance were introduced in Chapter 4.

As you can see in Figure 7-4, predictors can all be trained in parallel, via different CPU cores or even different servers. Similarly, predictions can be made in parallel. This is one of the reasons bagging and pasting are such popular methods: they scale very well.

Bagging and Pasting in Scikit-Learn

Scikit-Learn offers a simple API for both bagging and pasting with the `BaggingClassifier` class (or `BaggingRegressor` for regression). The following code trains an ensemble of 500 Decision Tree classifiers:[5] each is trained on 100 training instances randomly sampled from the training set with replacement (this is an example of bagging, but if you want to use pasting instead, just set `bootstrap=False`). The `n_jobs` parameter tells Scikit-Learn the number of CPU cores to use for training and predictions (`-1` tells Scikit-Learn to use all available cores):

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

The `BaggingClassifier` automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities (i.e., if it has a `predict_proba()` method), which is the case with Decision Tree classifiers.

Figure 7-5 compares the decision boundary of a single Decision Tree with the decision boundary of a bagging ensemble of 500 trees (from the preceding code), both trained on the moons dataset. As you can see, the ensemble's predictions will likely generalize much better than the single Decision Tree's predictions: the ensemble has a comparable bias but a smaller variance (it makes roughly the same number of errors on the training set, but the decision boundary is less irregular).

5 `max_samples` can alternatively be set to a float between 0.0 and 1.0, in which case the max number of instances to sample is equal to the size of the training set times `max_samples`.

Figure 7-5. A single Decision Tree (left) versus a bagging ensemble of 500 trees (right)

Bootstrapping introduces a bit more diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias than pasting; but the extra diversity also means that the predictors end up being less correlated, so the ensemble's variance is reduced. Overall, bagging often results in better models, which explains why it is generally preferred. However, if you have spare time and CPU power, you can use cross-validation to evaluate both bagging and pasting and select the one that works best.

Out-of-Bag Evaluation

With bagging, some instances may be sampled several times for any given predictor, while others may not be sampled at all. By default a `BaggingClassifier` samples *m* training instances with replacement (`bootstrap=True`), where *m* is the size of the training set. This means that only about 63% of the training instances are sampled on average for each predictor.[6] The remaining 37% of the training instances that are not sampled are called *out-of-bag* (oob) instances. Note that they are not the same 37% for all predictors.

Since a predictor never sees the oob instances during training, it can be evaluated on these instances, without the need for a separate validation set. You can evaluate the ensemble itself by averaging out the oob evaluations of each predictor.

In Scikit-Learn, you can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic oob evaluation after training. The following code demonstrates this. The resulting evaluation score is available through the `oob_score_` variable:

6 As *m* grows, this ratio approaches $1 - \exp(-1) \approx 63.212\%$.

```
>>> bag_clf = BaggingClassifier(
...     DecisionTreeClassifier(), n_estimators=500,
...     bootstrap=True, n_jobs=-1, oob_score=True)
...
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.90133333333333332
```

According to this oob evaluation, this `BaggingClassifier` is likely to achieve about 90.1% accuracy on the test set. Let's verify this:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.91200000000000003
```

We get 91.2% accuracy on the test set—close enough!

The oob decision function for each training instance is also available through the `oob_decision_function_` variable. In this case (since the base estimator has a `predict_proba()` method), the decision function returns the class probabilities for each training instance. For example, the oob evaluation estimates that the first training instance has a 68.25% probability of belonging to the positive class (and 31.75% of belonging to the negative class):

```
>>> bag_clf.oob_decision_function_
array([[0.31746032, 0.68253968],
       [0.34117647, 0.65882353],
       [1.        , 0.        ],
       ...
       [1.        , 0.        ],
       [0.03108808, 0.96891192],
       [0.57291667, 0.42708333]])
```

Random Patches and Random Subspaces

The `BaggingClassifier` class supports sampling the features as well. Sampling is controlled by two hyperparameters: `max_features` and `bootstrap_features`. They work the same way as `max_samples` and `bootstrap`, but for feature sampling instead of instance sampling. Thus, each predictor will be trained on a random subset of the input features.

This technique is particularly useful when you are dealing with high-dimensional inputs (such as images). Sampling both training instances and features is called the *Random Patches* method.[7] Keeping all training instances (by setting `bootstrap=False`

7 Gilles Louppe and Pierre Geurts, "Ensembles on Random Patches," *Lecture Notes in Computer Science* 7523 (2012): 346–361.

and `max_samples=1.0`) but sampling features (by setting `bootstrap_features` to `True` and/or `max_features` to a value smaller than `1.0`) is called the *Random Subspaces* method.[8]

Sampling features results in even more predictor diversity, trading a bit more bias for a lower variance.

Random Forests

As we have discussed, a Random Forest[9] is an ensemble of Decision Trees, generally trained via the bagging method (or sometimes pasting), typically with `max_samples` set to the size of the training set. Instead of building a `BaggingClassifier` and passing it a `DecisionTreeClassifier`, you can instead use the `RandomForestClassifier` class, which is more convenient and optimized for Decision Trees[10] (similarly, there is a `RandomForestRegressor` class for regression tasks). The following code uses all available CPU cores to train a Random Forest classifier with 500 trees (each limited to maximum 16 nodes):

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

With a few exceptions, a `RandomForestClassifier` has all the hyperparameters of a `DecisionTreeClassifier` (to control how trees are grown), plus all the hyperparameters of a `BaggingClassifier` to control the ensemble itself.[11]

The Random Forest algorithm introduces extra randomness when growing trees; instead of searching for the very best feature when splitting a node (see Chapter 6), it searches for the best feature among a random subset of features. The algorithm results in greater tree diversity, which (again) trades a higher bias for a lower variance, generally yielding an overall better model. The following `BaggingClassifier` is roughly equivalent to the previous `RandomForestClassifier`:

8 Tin Kam Ho, "The Random Subspace Method for Constructing Decision Forests," *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, no. 8 (1998): 832–844.

9 Tin Kam Ho, "Random Decision Forests," *Proceedings of the Third International Conference on Document Analysis and Recognition* 1 (1995): 278.

10 The `BaggingClassifier` class remains useful if you want a bag of something other than Decision Trees.

11 There are a few notable exceptions: `splitter` is absent (forced to `"random"`), `presort` is absent (forced to `False`), `max_samples` is absent (forced to `1.0`), and `base_estimator` is absent (forced to `DecisionTreeClassifier` with the provided hyperparameters).

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

Extra-Trees

When you are growing a tree in a Random Forest, at each node only a random subset of the features is considered for splitting (as discussed earlier). It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds (like regular Decision Trees do).

A forest of such extremely random trees is called an *Extremely Randomized Trees* ensemble[12] (or *Extra-Trees* for short). Once again, this technique trades more bias for a lower variance. It also makes Extra-Trees much faster to train than regular Random Forests, because finding the best possible threshold for each feature at every node is one of the most time-consuming tasks of growing a tree.

You can create an Extra-Trees classifier using Scikit-Learn's `ExtraTreesClassifier` class. Its API is identical to the `RandomForestClassifier` class. Similarly, the `ExtraTreesRegressor` class has the same API as the `RandomForestRegressor` class.

It is hard to tell in advance whether a `RandomForestClassifier` will perform better or worse than an `ExtraTreesClassifier`. Generally, the only way to know is to try both and compare them using cross-validation (tuning the hyperparameters using grid search).

Feature Importance

Yet another great quality of Random Forests is that they make it easy to measure the relative importance of each feature. Scikit-Learn measures a feature's importance by looking at how much the tree nodes that use that feature reduce impurity on average (across all trees in the forest). More precisely, it is a weighted average, where each node's weight is equal to the number of training samples that are associated with it (see Chapter 6).

Scikit-Learn computes this score automatically for each feature after training, then it scales the results so that the sum of all importances is equal to 1. You can access the result using the `feature_importances_` variable. For example, the following code trains a `RandomForestClassifier` on the iris dataset (introduced in Chapter 4) and outputs each feature's importance. It seems that the most important features are the petal length (44%) and width (42%), while sepal length and width are rather unimportant in comparison (11% and 2%, respectively):

12 Pierre Geurts et al., "Extremely Randomized Trees," *Machine Learning* 63, no. 1 (2006): 3–42.

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
...     print(name, score)
...
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

Similarly, if you train a Random Forest classifier on the MNIST dataset (introduced in Chapter 3) and plot each pixel's importance, you get the image represented in Figure 7-6.

Figure 7-6. MNIST pixel importance (according to a Random Forest classifier)

Random Forests are very handy to get a quick understanding of what features actually matter, in particular if you need to perform feature selection.

Boosting

Boosting (originally called *hypothesis boosting*) refers to any Ensemble method that can combine several weak learners into a strong learner. The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor. There are many boosting methods available, but by far the most popular are

AdaBoost[13] (short for *Adaptive Boosting*) and *Gradient Boosting*. Let's start with AdaBoost.

AdaBoost

One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted. This results in new predictors focusing more and more on the hard cases. This is the technique used by AdaBoost.

For example, when training an AdaBoost classifier, the algorithm first trains a base classifier (such as a Decision Tree) and uses it to make predictions on the training set. The algorithm then increases the relative weight of misclassified training instances. Then it trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on (see Figure 7-7).

Figure 7-7. AdaBoost sequential training with instance weight updates

Figure 7-8 shows the decision boundaries of five consecutive predictors on the moons dataset (in this example, each predictor is a highly regularized SVM classifier with an RBF kernel[14]). The first classifier gets many instances wrong, so their weights

13 Yoav Freund and Robert E. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting," *Journal of Computer and System Sciences* 55, no. 1 (1997): 119–139.

14 This is just for illustrative purposes. SVMs are generally not good base predictors for AdaBoost; they are slow and tend to be unstable with it.

get boosted. The second classifier therefore does a better job on these instances, and so on. The plot on the right represents the same sequence of predictors, except that the learning rate is halved (i.e., the misclassified instance weights are boosted half as much at every iteration). As you can see, this sequential learning technique has some similarities with Gradient Descent, except that instead of tweaking a single predictor's parameters to minimize a cost function, AdaBoost adds predictors to the ensemble, gradually making it better.

Figure 7-8. Decision boundaries of consecutive predictors

Once all predictors are trained, the ensemble makes predictions very much like bagging or pasting, except that predictors have different weights depending on their overall accuracy on the weighted training set.

There is one important drawback to this sequential learning technique: it cannot be parallelized (or only partially), since each predictor can only be trained after the previous predictor has been trained and evaluated. As a result, it does not scale as well as bagging or pasting.

Let's take a closer look at the AdaBoost algorithm. Each instance weight $w^{(i)}$ is initially set to $1/m$. A first predictor is trained, and its weighted error rate r_1 is computed on the training set; see Equation 7-1.

Equation 7-1. Weighted error rate of the jth predictor

$$r_j = \frac{\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}}$$

where $\hat{y}_j^{(i)}$ is the jth predictor's prediction for the ith instance.

The predictor's weight α_j is then computed using Equation 7-2, where η is the learning rate hyperparameter (defaults to 1).[15] The more accurate the predictor is, the higher its weight will be. If it is just guessing randomly, then its weight will be close to zero. However, if it is most often wrong (i.e., less accurate than random guessing), then its weight will be negative.

Equation 7-2. Predictor weight

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

Next, the AdaBoost algorithm updates the instance weights, using Equation 7-3, which boosts the weights of the misclassified instances.

Equation 7-3. Weight update rule

$$\text{for } i = 1, 2, \cdots, m$$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \widehat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp\left(\alpha_j\right) & \text{if } \widehat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

Then all the instance weights are normalized (i.e., divided by $\sum_{i=1}^{m} w^{(i)}$).

Finally, a new predictor is trained using the updated weights, and the whole process is repeated (the new predictor's weight is computed, the instance weights are updated, then another predictor is trained, and so on). The algorithm stops when the desired number of predictors is reached, or when a perfect predictor is found.

To make predictions, AdaBoost simply computes the predictions of all the predictors and weighs them using the predictor weights α_j. The predicted class is the one that receives the majority of weighted votes (see Equation 7-4).

Equation 7-4. AdaBoost predictions

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x}) = k}}^{N} \alpha_j \quad \text{where } N \text{ is the number of predictors.}$$

15 The original AdaBoost algorithm does not use a learning rate hyperparameter.

Scikit-Learn uses a multiclass version of AdaBoost called *SAMME*[16] (which stands for *Stagewise Additive Modeling using a Multiclass Exponential loss function*). When there are just two classes, SAMME is equivalent to AdaBoost. If the predictors can estimate class probabilities (i.e., if they have a `predict_proba()` method), Scikit-Learn can use a variant of SAMME called *SAMME.R* (the *R* stands for "Real"), which relies on class probabilities rather than predictions and generally performs better.

The following code trains an AdaBoost classifier based on 200 *Decision Stumps* using Scikit-Learn's `AdaBoostClassifier` class (as you might expect, there is also an `AdaBoostRegressor` class). A Decision Stump is a Decision Tree with `max_depth=1`—in other words, a tree composed of a single decision node plus two leaf nodes. This is the default base estimator for the `AdaBoostClassifier` class:

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5)
ada_clf.fit(X_train, y_train)
```

If your AdaBoost ensemble is overfitting the training set, you can try reducing the number of estimators or more strongly regularizing the base estimator.

Gradient Boosting

Another very popular boosting algorithm is *Gradient Boosting*.[17] Just like AdaBoost, Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the *residual errors* made by the previous predictor.

Let's go through a simple regression example, using Decision Trees as the base predictors (of course, Gradient Boosting also works great with regression tasks). This is called *Gradient Tree Boosting*, or *Gradient Boosted Regression Trees* (GBRT). First, let's fit a `DecisionTreeRegressor` to the training set (for example, a noisy quadratic training set):

16 For more details, see Ji Zhu et al., "Multi-Class AdaBoost," *Statistics and Its Interface* 2, no. 3 (2009): 349–360.

17 Gradient Boosting was first introduced in Leo Breiman's 1997 paper "Arcing the Edge" and was further developed in the 1999 paper "Greedy Function Approximation: A Gradient Boosting Machine" by Jerome H. Friedman.

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

Next, we'll train a second `DecisionTreeRegressor` on the residual errors made by the first predictor:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

Then we train a third regressor on the residual errors made by the second predictor:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

Now we have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

Figure 7-9 represents the predictions of these three trees in the left column, and the ensemble's predictions in the right column. In the first row, the ensemble has just one tree, so its predictions are exactly the same as the first tree's predictions. In the second row, a new tree is trained on the residual errors of the first tree. On the right you can see that the ensemble's predictions are equal to the sum of the predictions of the first two trees. Similarly, in the third row another tree is trained on the residual errors of the second tree. You can see that the ensemble's predictions gradually get better as trees are added to the ensemble.

A simpler way to train GBRT ensembles is to use Scikit-Learn's `GradientBoostingRegressor` class. Much like the `RandomForestRegressor` class, it has hyperparameters to control the growth of Decision Trees (e.g., `max_depth`, `min_samples_leaf`), as well as hyperparameters to control the ensemble training, such as the number of trees (`n_estimators`). The following code creates the same ensemble as the previous one:

```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
gbrt.fit(X, y)
```

Figure 7-9. In this depiction of Gradient Boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions

The `learning_rate` hyperparameter scales the contribution of each tree. If you set it to a low value, such as `0.1`, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called *shrinkage*. Figure 7-10 shows two GBRT ensembles trained with a low learning rate: the one on the left does not have enough trees to fit the training set, while the one on the right has too many trees and overfits the training set.

Figure 7-10. GBRT ensembles with not enough predictors (left) and too many (right)

In order to find the optimal number of trees, you can use early stopping (see Chapter 4). A simple way to implement this is to use the `staged_predict()` method: it returns an iterator over the predictions made by the ensemble at each stage of training (with one tree, two trees, etc.). The following code trains a GBRT ensemble with 120 trees, then measures the validation error at each stage of training to find the optimal number of trees, and finally trains another GBRT ensemble using the optimal number of trees:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split(X, y)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors) + 1

gbrt_best = GradientBoostingRegressor(max_depth=2,n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```

The validation errors are represented on the left of Figure 7-11, and the best model's predictions are represented on the right.

Figure 7-11. Tuning the number of trees using early stopping

It is also possible to implement early stopping by actually stopping training early (instead of training a large number of trees first and then looking back to find the optimal number). You can do so by setting `warm_start=True`, which makes Scikit-Learn keep existing trees when the `fit()` method is called, allowing incremental training. The following code stops training when the validation error does not improve for five iterations in a row:

```
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)

min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
        if error_going_up == 5:
            break  # early stopping
```

The `GradientBoostingRegressor` class also supports a `subsample` hyperparameter, which specifies the fraction of training instances to be used for training each tree. For example, if `subsample=0.25`, then each tree is trained on 25% of the training instances, selected randomly. As you can probably guess by now, this technique trades a higher bias for a lower variance. It also speeds up training considerably. This is called *Stochastic Gradient Boosting*.

It is possible to use Gradient Boosting with other cost functions. This is controlled by the `loss` hyperparameter (see Scikit-Learn's documentation for more details).

It is worth noting that an optimized implementation of Gradient Boosting is available in the popular Python library XGBoost, which stands for Extreme Gradient Boosting. This package was initially developed by Tianqi Chen as part of the Distributed (Deep) Machine Learning Community (DMLC), and it aims to be extremely fast, scalable, and portable. In fact, XGBoost is often an important component of the winning entries in ML competitions. XGBoost's API is quite similar to Scikit-Learn's:

```
import xgboost

xgb_reg = xgboost.XGBRegressor()
xgb_reg.fit(X_train, y_train)
y_pred = xgb_reg.predict(X_val)
```

XGBoost also offers several nice features, such as automatically taking care of early stopping:

```
xgb_reg.fit(X_train, y_train,
            eval_set=[(X_val, y_val)], early_stopping_rounds=2)
y_pred = xgb_reg.predict(X_val)
```

You should definitely check it out!

Stacking

The last Ensemble method we will discuss in this chapter is called *stacking* (short for *stacked generalization*).[18] It is based on a simple idea: instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation? Figure 7-12 shows such an ensemble performing a regression task on a new instance. Each of the bottom three predictors predicts a different value (3.1, 2.7, and 2.9), and then the final predictor (called a *blender*, or a *meta learner*) takes these predictions as inputs and makes the final prediction (3.0).

18 David H. Wolpert, "Stacked Generalization," *Neural Networks* 5, no. 2 (1992): 241–259.

Figure 7-12. Aggregating predictions using a blending predictor

To train the blender, a common approach is to use a hold-out set.[19] Let's see how it works. First, the training set is split into two subsets. The first subset is used to train the predictors in the first layer (see Figure 7-13).

Figure 7-13. Training the first layer

Next, the first layer's predictors are used to make predictions on the second (held-out) set (see Figure 7-14). This ensures that the predictions are "clean," since the predictors never saw these instances during training. For each instance in the hold-out

19 Alternatively, it is possible to use out-of-fold predictions. In some contexts this is called *stacking*, while using a hold-out set is called *blending*. For many people these terms are synonymous.

set, there are three predicted values. We can create a new training set using these predicted values as input features (which makes this new training set 3D), and keeping the target values. The blender is trained on this new training set, so it learns to predict the target value, given the first layer's predictions.

Figure 7-14. Training the blender

It is actually possible to train several different blenders this way (e.g., one using Linear Regression, another using Random Forest Regression), to get a whole layer of blenders. The trick is to split the training set into three subsets: the first one is used to train the first layer, the second one is used to create the training set used to train the second layer (using predictions made by the predictors of the first layer), and the third one is used to create the training set to train the third layer (using predictions made by the predictors of the second layer). Once this is done, we can make a prediction for a new instance by going through each layer sequentially, as shown in Figure 7-15.

Figure 7-15. Predictions in a multilayer stacking ensemble

Unfortunately, Scikit-Learn does not support stacking directly, but it is not too hard to roll out your own implementation (see the following exercises). Alternatively, you can use an open source implementation such as `DESlib`.

Exercises

1. If you have trained five different models on the exact same training data, and they all achieve 95% precision, is there any chance that you can combine these models to get better results? If so, how? If not, why?
2. What is the difference between hard and soft voting classifiers?
3. Is it possible to speed up training of a bagging ensemble by distributing it across multiple servers? What about pasting ensembles, boosting ensembles, Random Forests, or stacking ensembles?
4. What is the benefit of out-of-bag evaluation?
5. What makes Extra-Trees more random than regular Random Forests? How can this extra randomness help? Are Extra-Trees slower or faster than regular Random Forests?
6. If your AdaBoost ensemble underfits the training data, which hyperparameters should you tweak and how?

7. If your Gradient Boosting ensemble overfits the training set, should you increase or decrease the learning rate?
8. Load the MNIST data (introduced in Chapter 3), and split it into a training set, a validation set, and a test set (e.g., use 50,000 instances for training, 10,000 for validation, and 10,000 for testing). Then train various classifiers, such as a Random Forest classifier, an Extra-Trees classifier, and an SVM classifier. Next, try to combine them into an ensemble that outperforms each individual classifier on the validation set, using soft or hard voting. Once you have found one, try it on the test set. How much better does it perform compared to the individual classifiers?
9. Run the individual classifiers from the previous exercise to make predictions on the validation set, and create a new training set with the resulting predictions: each training instance is a vector containing the set of predictions from all your classifiers for an image, and the target is the image's class. Train a classifier on this new training set. Congratulations, you have just trained a blender, and together with the classifiers it forms a stacking ensemble! Now evaluate the ensemble on the test set. For each image in the test set, make predictions with all your classifiers, then feed the predictions to the blender to get the ensemble's predictions. How does it compare to the voting classifier you trained earlier?

Solutions to these exercises are available in Appendix A.

CHAPTER 8

Dimensionality Reduction

Many Machine Learning problems involve thousands or even millions of features for each training instance. Not only do all these features make training extremely slow, but they can also make it much harder to find a good solution, as we will see. This problem is often referred to as the *curse of dimensionality*.

Fortunately, in real-world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one. For example, consider the MNIST images (introduced in Chapter 3): the pixels on the image borders are almost always white, so you could completely drop these pixels from the training set without losing much information. Figure 7-6 confirms that these pixels are utterly unimportant for the classification task. Additionally, two neighboring pixels are often highly correlated: if you merge them into a single pixel (e.g., by taking the mean of the two pixel intensities), you will not lose much information.

Reducing dimensionality does cause some information loss (just like compressing an image to JPEG can degrade its quality), so even though it will speed up training, it may make your system perform slightly worse. It also makes your pipelines a bit more complex and thus harder to maintain. So, if training is too slow, you should first try to train your system with the original data before considering using dimensionality reduction. In some cases, reducing the dimensionality of the training data may filter out some noise and unnecessary details and thus result in higher performance, but in general it won't; it will just speed up training.

Apart from speeding up training, dimensionality reduction is also extremely useful for data visualization (or *DataViz*). Reducing the number of dimensions down to two (or three) makes it possible to plot a condensed view of a high-dimensional training

set on a graph and often gain some important insights by visually detecting patterns, such as clusters. Moreover, DataViz is essential to communicate your conclusions to people who are not data scientists—in particular, decision makers who will use your results.

In this chapter we will discuss the curse of dimensionality and get a sense of what goes on in high-dimensional space. Then, we will consider the two main approaches to dimensionality reduction (projection and Manifold Learning), and we will go through three of the most popular dimensionality reduction techniques: PCA, Kernel PCA, and LLE.

The Curse of Dimensionality

We are so used to living in three dimensions[1] that our intuition fails us when we try to imagine a high-dimensional space. Even a basic 4D hypercube is incredibly hard to picture in our minds (see Figure 8-1), let alone a 200-dimensional ellipsoid bent in a 1,000-dimensional space.

Figure 8-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)[2]

It turns out that many things behave very differently in high-dimensional space. For example, if you pick a random point in a unit square (a 1 × 1 square), it will have only about a 0.4% chance of being located less than 0.001 from a border (in other words, it is very unlikely that a random point will be "extreme" along any dimension). But in a 10,000-dimensional unit hypercube, this probability is greater than 99.999999%. Most points in a high-dimensional hypercube are very close to the border.[3]

1 Well, four dimensions if you count time, and a few more if you are a string theorist.

2 Watch a rotating tesseract projected into 3D space at *https://homl.info/30*. Image by Wikipedia user Nerd-Boy1392 (Creative Commons BY-SA 3.0). Reproduced from *https://en.wikipedia.org/wiki/Tesseract*.

3 Fun fact: anyone you know is probably an extremist in at least one dimension (e.g., how much sugar they put in their coffee), if you consider enough dimensions.

Here is a more troublesome difference: if you pick two points randomly in a unit square, the distance between these two points will be, on average, roughly 0.52. If you pick two random points in a unit 3D cube, the average distance will be roughly 0.66. But what about two points picked randomly in a 1,000,000-dimensional hypercube? The average distance, believe it or not, will be about 408.25 (roughly $\sqrt{1,000,000/6}$)! This is counterintuitive: how can two points be so far apart when they both lie within the same unit hypercube? Well, there's just plenty of space in high dimensions. As a result, high-dimensional datasets are at risk of being very sparse: most training instances are likely to be far away from each other. This also means that a new instance will likely be far away from any training instance, making predictions much less reliable than in lower dimensions, since they will be based on much larger extrapolations. In short, the more dimensions the training set has, the greater the risk of overfitting it.

In theory, one solution to the curse of dimensionality could be to increase the size of the training set to reach a sufficient density of training instances. Unfortunately, in practice, the number of training instances required to reach a given density grows exponentially with the number of dimensions. With just 100 features (significantly fewer than in the MNIST problem), you would need more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other on average, assuming they were spread out uniformly across all dimensions.

Main Approaches for Dimensionality Reduction

Before we dive into specific dimensionality reduction algorithms, let's take a look at the two main approaches to reducing dimensionality: projection and Manifold Learning.

Projection

In most real-world problems, training instances are *not* spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated (as discussed earlier for MNIST). As a result, all training instances lie within (or close to) a much lower-dimensional *subspace* of the high-dimensional space. This sounds very abstract, so let's look at an example. In Figure 8-2 you can see a 3D dataset represented by circles.

Figure 8-2. A 3D dataset lying close to a 2D subspace

Notice that all training instances lie close to a plane: this is a lower-dimensional (2D) subspace of the high-dimensional (3D) space. If we project every training instance perpendicularly onto this subspace (as represented by the short lines connecting the instances to the plane), we get the new 2D dataset shown in Figure 8-3. Ta-da! We have just reduced the dataset's dimensionality from 3D to 2D. Note that the axes correspond to new features z_1 and z_2 (the coordinates of the projections on the plane).

Figure 8-3. The new 2D dataset after projection

However, projection is not always the best approach to dimensionality reduction. In many cases the subspace may twist and turn, such as in the famous *Swiss roll* toy dataset represented in Figure 8-4.

Figure 8-4. Swiss roll dataset

Simply projecting onto a plane (e.g., by dropping x_3) would squash different layers of the Swiss roll together, as shown on the left side of Figure 8-5. What you really want is to unroll the Swiss roll to obtain the 2D dataset on the right side of Figure 8-5.

Figure 8-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)

Manifold Learning

The Swiss roll is an example of a 2D *manifold*. Put simply, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space. More generally, a d-dimensional manifold is a part of an n-dimensional space (where $d < n$) that locally resembles a d-dimensional hyperplane. In the case of the Swiss roll, $d = 2$ and $n = 3$: it locally resembles a 2D plane, but it is rolled in the third dimension.

Many dimensionality reduction algorithms work by modeling the manifold on which the training instances lie; this is called *Manifold Learning*. It relies on the *manifold assumption*, also called the *manifold hypothesis*, which holds that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold. This assumption is very often empirically observed.

Once again, think about the MNIST dataset: all handwritten digit images have some similarities. They are made of connected lines, the borders are white, and they are more or less centered. If you randomly generated images, only a ridiculously tiny fraction of them would look like handwritten digits. In other words, the degrees of freedom available to you if you try to create a digit image are dramatically lower than the degrees of freedom you would have if you were allowed to generate any image you wanted. These constraints tend to squeeze the dataset into a lower-dimensional manifold.

The manifold assumption is often accompanied by another implicit assumption: that the task at hand (e.g., classification or regression) will be simpler if expressed in the lower-dimensional space of the manifold. For example, in the top row of Figure 8-6 the Swiss roll is split into two classes: in the 3D space (on the left), the decision boundary would be fairly complex, but in the 2D unrolled manifold space (on the right), the decision boundary is a straight line.

However, this implicit assumption does not always hold. For example, in the bottom row of Figure 8-6, the decision boundary is located at $x_1 = 5$. This decision boundary looks very simple in the original 3D space (a vertical plane), but it looks more complex in the unrolled manifold (a collection of four independent line segments).

In short, reducing the dimensionality of your training set before training a model will usually speed up training, but it may not always lead to a better or simpler solution; it all depends on the dataset.

Hopefully you now have a good sense of what the curse of dimensionality is and how dimensionality reduction algorithms can fight it, especially when the manifold assumption holds. The rest of this chapter will go through some of the most popular algorithms.

Figure 8-6. The decision boundary may not always be simpler with lower dimensions

PCA

Principal Component Analysis (PCA) is by far the most popular dimensionality reduction algorithm. First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it, just like in Figure 8-2.

Preserving the Variance

Before you can project the training set onto a lower-dimensional hyperplane, you first need to choose the right hyperplane. For example, a simple 2D dataset is represented on the left in Figure 8-7, along with three different axes (i.e., 1D hyperplanes). On the right is the result of the projection of the dataset onto each of these axes. As you can see, the projection onto the solid line preserves the maximum variance, while the projection onto the dotted line preserves very little variance and the projection onto the dashed line preserves an intermediate amount of variance.

Figure 8-7. Selecting the subspace to project on

It seems reasonable to select the axis that preserves the maximum amount of variance, as it will most likely lose less information than the other projections. Another way to justify this choice is that it is the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis. This is the rather simple idea behind PCA.[4]

Principal Components

PCA identifies the axis that accounts for the largest amount of variance in the training set. In Figure 8-7, it is the solid line. It also finds a second axis, orthogonal to the first one, that accounts for the largest amount of remaining variance. In this 2D example there is no choice: it is the dotted line. If it were a higher-dimensional dataset, PCA would also find a third axis, orthogonal to both previous axes, and a fourth, a fifth, and so on—as many axes as the number of dimensions in the dataset.

The ith axis is called the ith *principal component* (PC) of the data. In Figure 8-7, the first PC is the axis on which vector $\mathbf{c}_1$ lies, and the second PC is the axis on which vector $\mathbf{c}_2$ lies. In Figure 8-2 the first two PCs are the orthogonal axes on which the two arrows lie, on the plane, and the third PC is the axis orthogonal to that plane.

4 Karl Pearson, "On Lines and Planes of Closest Fit to Systems of Points in Space," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2, no. 11 (1901): 559-572, *https://homl.info/pca*.

For each principal component, PCA finds a zero-centered unit vector pointing in the direction of the PC. Since two opposing unit vectors lie on the same axis, the direction of the unit vectors returned by PCA is not stable: if you perturb the training set slightly and run PCA again, the unit vectors may point in the opposite direction as the original vectors. However, they will generally still lie on the same axes. In some cases, a pair of unit vectors may even rotate or swap (if the variances along these two axes are close), but the plane they define will generally remain the same.

So how can you find the principal components of a training set? Luckily, there is a standard matrix factorization technique called *Singular Value Decomposition* (SVD) that can decompose the training set matrix $\mathbf{X}$ into the matrix multiplication of three matrices $\mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^{\intercal}$, where $\mathbf{V}$ contains the unit vectors that define all the principal components that we are looking for, as shown in Equation 8-1.

Equation 8-1. Principal components matrix

$$\mathbf{V} = \begin{pmatrix} | & | & & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_n \\ | & | & & | \end{pmatrix}$$

The following Python code uses NumPy's `svd()` function to obtain all the principal components of the training set, then extracts the two unit vectors that define the first two PCs:

```
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```

PCA assumes that the dataset is centered around the origin. As we will see, Scikit-Learn's PCA classes take care of centering the data for you. If you implement PCA yourself (as in the preceding example), or if you use other libraries, don't forget to center the data first.

Projecting Down to d Dimensions

Once you have identified all the principal components, you can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components. Selecting this hyperplane ensures that the projection will preserve as much variance as possible. For example, in Figure 8-2 the 3D dataset is projected down to the 2D plane defined by the first two principal

components, preserving a large part of the dataset's variance. As a result, the 2D projection looks very much like the original 3D dataset.

To project the training set onto the hyperplane and obtain a reduced dataset $\mathbf{X}_{d\text{-proj}}$ of dimensionality d, compute the matrix multiplication of the training set matrix $\mathbf{X}$ by the matrix $\mathbf{W}_d$, defined as the matrix containing the first d columns of $\mathbf{V}$, as shown in Equation 8-2.

Equation 8-2. Projecting the training set down to d dimensions

$$\mathbf{X}_{d\text{-proj}} = \mathbf{X}\mathbf{W}_d$$

The following Python code projects the training set onto the plane defined by the first two principal components:

```
W2 = Vt.T[:, :2]
X2D = X_centered.dot(W2)
```

There you have it! You now know how to reduce the dimensionality of any dataset down to any number of dimensions, while preserving as much variance as possible.

Using Scikit-Learn

Scikit-Learn's `PCA` class uses SVD decomposition to implement PCA, just like we did earlier in this chapter. The following code applies PCA to reduce the dimensionality of the dataset down to two dimensions (note that it automatically takes care of centering the data):

```
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
```

After fitting the `PCA` transformer to the dataset, its `components_` attribute holds the transpose of $\mathbf{W}_d$ (e.g., the unit vector that defines the first principal component is equal to `pca.components_.T[:, 0]`).

Explained Variance Ratio

Another useful piece of information is the *explained variance ratio* of each principal component, available via the `explained_variance_ratio_` variable. The ratio indicates the proportion of the dataset's variance that lies along each principal component. For example, let's look at the explained variance ratios of the first two components of the 3D dataset represented in Figure 8-2:

```
>>> pca.explained_variance_ratio_
array([0.84248607, 0.14631839])
```

This output tells you that 84.2% of the dataset's variance lies along the first PC, and 14.6% lies along the second PC. This leaves less than 1.2% for the third PC, so it is reasonable to assume that the third PC probably carries little information.

Choosing the Right Number of Dimensions

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is simpler to choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95%). Unless, of course, you are reducing dimensionality for data visualization—in that case you will want to reduce the dimensionality down to 2 or 3.

The following code performs PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 95% of the training set's variance:

```
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
```

You could then set `n_components=d` and run PCA again. But there is a much better option: instead of specifying the number of principal components you want to preserve, you can set `n_components` to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve:

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

Yet another option is to plot the explained variance as a function of the number of dimensions (simply plot `cumsum`; see Figure 8-8). There will usually be an elbow in the curve, where the explained variance stops growing fast. In this case, you can see that reducing the dimensionality down to about 100 dimensions wouldn't lose too much explained variance.

Figure 8-8. Explained variance as a function of the number of dimensions

PCA for Compression

After dimensionality reduction, the training set takes up much less space. As an example, try applying PCA to the MNIST dataset while preserving 95% of its variance. You should find that each instance will have just over 150 features, instead of the original 784 features. So, while most of the variance is preserved, the dataset is now less than 20% of its original size! This is a reasonable compression ratio, and you can see how this size reduction can speed up a classification algorithm (such as an SVM classifier) tremendously.

It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. This won't give you back the original data, since the projection lost a bit of information (within the 5% variance that was dropped), but it will likely be close to the original data. The mean squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the *reconstruction error*.

The following code compresses the MNIST dataset down to 154 dimensions, then uses the `inverse_transform()` method to decompress it back to 784 dimensions:

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

Figure 8-9 shows a few digits from the original training set (on the left), and the corresponding digits after compression and decompression. You can see that there is a slight image quality loss, but the digits are still mostly intact.

Figure 8-9. MNIST compression that preserves 95% of the variance

The equation of the inverse transformation is shown in Equation 8-3.

Equation 8-3. PCA inverse transformation, back to the original number of dimensions

$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d\text{-proj}}\mathbf{W}_d^{\mathsf{T}}$$

Randomized PCA

If you set the `svd_solver` hyperparameter to `"randomized"`, Scikit-Learn uses a stochastic algorithm called *Randomized PCA* that quickly finds an approximation of the first d principal components. Its computational complexity is $O(m \times d^2) + O(d^3)$, instead of $O(m \times n^2) + O(n^3)$ for the full SVD approach, so it is dramatically faster than full SVD when d is much smaller than n:

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

By default, `svd_solver` is actually set to `"auto"`: Scikit-Learn automatically uses the randomized PCA algorithm if m or n is greater than 500 and d is less than 80% of m or n, or else it uses the full SVD approach. If you want to force Scikit-Learn to use full SVD, you can set the `svd_solver` hyperparameter to `"full"`.

Incremental PCA

One problem with the preceding implementations of PCA is that they require the whole training set to fit in memory in order for the algorithm to run. Fortunately, *Incremental PCA* (IPCA) algorithms have been developed. They allow you to split the training set into mini-batches and feed an IPCA algorithm one mini-batch at a time.

This is useful for large training sets and for applying PCA online (i.e., on the fly, as new instances arrive).

The following code splits the MNIST dataset into 100 mini-batches (using NumPy's `array_split()` function) and feeds them to Scikit-Learn's `IncrementalPCA` class[5] to reduce the dimensionality of the MNIST dataset down to 154 dimensions (just like before). Note that you must call the `partial_fit()` method with each mini-batch, rather than the `fit()` method with the whole training set:

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

Alternatively, you can use NumPy's `memmap` class, which allows you to manipulate a large array stored in a binary file on disk as if it were entirely in memory; the class loads only the data it needs in memory, when it needs it. Since the `IncrementalPCA` class uses only a small part of the array at any given time, the memory usage remains under control. This makes it possible to call the usual `fit()` method, as you can see in the following code:

```
X_mm = np.memmap(filename, dtype="float32", mode="readonly", shape=(m, n))

batch_size = m // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mm)
```

Kernel PCA

In Chapter 5 we discussed the kernel trick, a mathematical technique that implicitly maps instances into a very high-dimensional space (called the *feature space*), enabling nonlinear classification and regression with Support Vector Machines. Recall that a linear decision boundary in the high-dimensional feature space corresponds to a complex nonlinear decision boundary in the *original space*.

It turns out that the same trick can be applied to PCA, making it possible to perform complex nonlinear projections for dimensionality reduction. This is called *Kernel*

5 Scikit-Learn uses the algorithm described in David A. Ross et al., "Incremental Learning for Robust Visual Tracking," *International Journal of Computer Vision* 77, no. 1–3 (2008): 125–141.

PCA (kPCA).[6] It is often good at preserving clusters of instances after projection, or sometimes even unrolling datasets that lie close to a twisted manifold.

The following code uses Scikit-Learn's `KernelPCA` class to perform kPCA with an RBF kernel (see Chapter 5 for more details about the RBF kernel and other kernels):

```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

Figure 8-10 shows the Swiss roll, reduced to two dimensions using a linear kernel (equivalent to simply using the `PCA` class), an RBF kernel, and a sigmoid kernel.

Figure 8-10. Swiss roll reduced to 2D using kPCA with various kernels

Selecting a Kernel and Tuning Hyperparameters

As kPCA is an unsupervised learning algorithm, there is no obvious performance measure to help you select the best kernel and hyperparameter values. That said, dimensionality reduction is often a preparation step for a supervised learning task (e.g., classification), so you can use grid search to select the kernel and hyperparameters that lead to the best performance on that task. The following code creates a two-step pipeline, first reducing dimensionality to two dimensions using kPCA, then applying Logistic Regression for classification. Then it uses `GridSearchCV` to find the best kernel and `gamma` value for kPCA in order to get the best classification accuracy at the end of the pipeline:

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
```

6 Bernhard Schölkopf et al., "Kernel Principal Component Analysis," in *Lecture Notes in Computer Science* 1327 (Berlin: Springer, 1997): 583–588.

```
clf = Pipeline([
        ("kpca", KernelPCA(n_components=2)),
        ("log_reg", LogisticRegression())
    ])

param_grid = [{
        "kpca__gamma": np.linspace(0.03, 0.05, 10),
        "kpca__kernel": ["rbf", "sigmoid"]
    }]

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

The best kernel and hyperparameters are then available through the `best_params_` variable:

```
>>> print(grid_search.best_params_)
{'kpca__gamma': 0.043333333333333335, 'kpca__kernel': 'rbf'}
```

Another approach, this time entirely unsupervised, is to select the kernel and hyperparameters that yield the lowest reconstruction error. Note that reconstruction is not as easy as with linear PCA. Here's why. Figure 8-11 shows the original Swiss roll 3D dataset (top left) and the resulting 2D dataset after kPCA is applied using an RBF kernel (top right). Thanks to the kernel trick, this transformation is mathematically equivalent to using the *feature map* φ to map the training set to an infinite-dimensional feature space (bottom right), then projecting the transformed training set down to 2D using linear PCA.

Notice that if we could invert the linear PCA step for a given instance in the reduced space, the reconstructed point would lie in feature space, not in the original space (e.g., like the one represented by an X in the diagram). Since the feature space is infinite-dimensional, we cannot compute the reconstructed point, and therefore we cannot compute the true reconstruction error. Fortunately, it is possible to find a point in the original space that would map close to the reconstructed point. This point is called the reconstruction *pre-image*. Once you have this pre-image, you can measure its squared distance to the original instance. You can then select the kernel and hyperparameters that minimize this reconstruction pre-image error.

Figure 8-11. Kernel PCA and the reconstruction pre-image error

You may be wondering how to perform this reconstruction. One solution is to train a supervised regression model, with the projected instances as the training set and the original instances as the targets. Scikit-Learn will do this automatically if you set `fit_inverse_transform=True`, as shown in the following code:[7]

```
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433,
                    fit_inverse_transform=True)
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)
```

By default, `fit_inverse_transform=False` and `KernelPCA` has no `inverse_transform()` method. This method only gets created when you set `fit_inverse_transform=True`.

7 If you set `fit_inverse_transform=True`, Scikit-Learn will use the algorithm (based on Kernel Ridge Regression) described in Gokhan H. Bakır et al., "Learning to Find Pre-Images", *Proceedings of the 16th International Conference on Neural Information Processing Systems* (2004): 449–456.

You can then compute the reconstruction pre-image error:

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(X, X_preimage)
32.786308795766132
```

Now you can use grid search with cross-validation to find the kernel and hyperparameters that minimize this error.

LLE

Locally Linear Embedding (LLE)[8] is another powerful *nonlinear dimensionality reduction* (NLDR) technique. It is a Manifold Learning technique that does not rely on projections, like the previous algorithms do. In a nutshell, LLE works by first measuring how each training instance linearly relates to its closest neighbors (c.n.), and then looking for a low-dimensional representation of the training set where these local relationships are best preserved (more details shortly). This approach makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise.

The following code uses Scikit-Learn's `LocallyLinearEmbedding` class to unroll the Swiss roll:

```
from sklearn.manifold import LocallyLinearEmbedding

lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
X_reduced = lle.fit_transform(X)
```

The resulting 2D dataset is shown in Figure 8-12. As you can see, the Swiss roll is completely unrolled, and the distances between instances are locally well preserved. However, distances are not preserved on a larger scale: the left part of the unrolled Swiss roll is stretched, while the right part is squeezed. Nevertheless, LLE did a pretty good job at modeling the manifold.

8 Sam T. Roweis and Lawrence K. Saul, "Nonlinear Dimensionality Reduction by Locally Linear Embedding," *Science* 290, no. 5500 (2000): 2323–2326.

Figure 8-12. Unrolled Swiss roll using LLE

Here's how LLE works: for each training instance $\mathbf{x}^{(i)}$, the algorithm identifies its k closest neighbors (in the preceding code $k = 10$), then tries to reconstruct $\mathbf{x}^{(i)}$ as a linear function of these neighbors. More specifically, it finds the weights $w_{i,j}$ such that the squared distance between $\mathbf{x}^{(i)}$ and $\sum_{j=1}^{m} w_{i,j}\mathbf{x}^{(j)}$ is as small as possible, assuming $w_{i,j} = 0$ if $\mathbf{x}^{(j)}$ is not one of the k closest neighbors of $\mathbf{x}^{(i)}$. Thus the first step of LLE is the constrained optimization problem described in Equation 8-4, where $\mathbf{W}$ is the weight matrix containing all the weights $w_{i,j}$. The second constraint simply normalizes the weights for each training instance $\mathbf{x}^{(i)}$.

Equation 8-4. LLE step one: linearly modeling local relationships

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\mathbf{x}^{(i)} - \sum_{j=1}^{m} w_{i,j}\mathbf{x}^{(j)} \right)^2$$

$$\text{subject to} \begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ c.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^{m} w_{i,j} = 1 & \text{for } i = 1, 2, \cdots, m \end{cases}$$

After this step, the weight matrix $\widehat{\mathbf{W}}$ (containing the weights $\widehat{w}_{i,j}$) encodes the local linear relationships between the training instances. The second step is to map the training instances into a d-dimensional space (where $d < n$) while preserving these local relationships as much as possible. If $\mathbf{z}^{(i)}$ is the image of $\mathbf{x}^{(i)}$ in this d-dimensional

space, then we want the squared distance between $\mathbf{z}^{(i)}$ and $\sum_{j=1}^{m} \widehat{w}_{i,j}\mathbf{z}^{(j)}$ to be as small as possible. This idea leads to the unconstrained optimization problem described in Equation 8-5. It looks very similar to the first step, but instead of keeping the instances fixed and finding the optimal weights, we are doing the reverse: keeping the weights fixed and finding the optimal position of the instances' images in the low-dimensional space. Note that $\mathbf{Z}$ is the matrix containing all $\mathbf{z}^{(i)}$.

Equation 8-5. LLE step two: reducing dimensionality while preserving relationships

$$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(\mathbf{z}^{(i)} - \sum_{j=1}^{m} \widehat{w}_{i,j}\mathbf{z}^{(j)} \right)^2$$

Scikit-Learn's LLE implementation has the following computational complexity: $O(m \log(m)n \log(k))$ for finding the k nearest neighbors, $O(mnk^3)$ for optimizing the weights, and $O(dm^2)$ for constructing the low-dimensional representations. Unfortunately, the m^2 in the last term makes this algorithm scale poorly to very large datasets.

Other Dimensionality Reduction Techniques

There are many other dimensionality reduction techniques, several of which are available in Scikit-Learn. Here are some of the most popular ones:

Random Projections

As its name suggests, projects the data to a lower-dimensional space using a random linear projection. This may sound crazy, but it turns out that such a random projection is actually very likely to preserve distances well, as was demonstrated mathematically by William B. Johnson and Joram Lindenstrauss in a famous lemma. The quality of the dimensionality reduction depends on the number of instances and the target dimensionality, but surprisingly not on the initial dimensionality. Check out the documentation for the `sklearn.random_projection` package for more details.

Multidimensional Scaling (MDS)

Reduces dimensionality while trying to preserve the distances between the instances.

Isomap

Creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the *geodesic distances*[9] between the instances.

t-Distributed Stochastic Neighbor Embedding (t-SNE)

Reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space (e.g., to visualize the MNIST images in 2D).

Linear Discriminant Analysis (LDA)

Is a classification algorithm, but during training it learns the most discriminative axes between the classes, and these axes can then be used to define a hyperplane onto which to project the data. The benefit of this approach is that the projection will keep classes as far apart as possible, so LDA is a good technique to reduce dimensionality before running another classification algorithm such as an SVM classifier.

Figure 8-13 shows the results of a few of these techniques.

Figure 8-13. Using various techniques to reduce the Swill roll to 2D

Exercises

1. What are the main motivations for reducing a dataset's dimensionality? What are the main drawbacks?
2. What is the curse of dimensionality?

9 The geodesic distance between two nodes in a graph is the number of nodes on the shortest path between these nodes.

3. Once a dataset's dimensionality has been reduced, is it possible to reverse the operation? If so, how? If not, why?
4. Can PCA be used to reduce the dimensionality of a highly nonlinear dataset?
5. Suppose you perform PCA on a 1,000-dimensional dataset, setting the explained variance ratio to 95%. How many dimensions will the resulting dataset have?
6. In what cases would you use vanilla PCA, Incremental PCA, Randomized PCA, or Kernel PCA?
7. How can you evaluate the performance of a dimensionality reduction algorithm on your dataset?
8. Does it make any sense to chain two different dimensionality reduction algorithms?
9. Load the MNIST dataset (introduced in Chapter 3) and split it into a training set and a test set (take the first 60,000 instances for training, and the remaining 10,000 for testing). Train a Random Forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set. Next, use PCA to reduce the dataset's dimensionality, with an explained variance ratio of 95%. Train a new Random Forest classifier on the reduced dataset and see how long it takes. Was training much faster? Next, evaluate the classifier on the test set. How does it compare to the previous classifier?
10. Use t-SNE to reduce the MNIST dataset down to two dimensions and plot the result using Matplotlib. You can use a scatterplot using 10 different colors to represent each image's target class. Alternatively, you can replace each dot in the scatterplot with the corresponding instance's class (a digit from 0 to 9), or even plot scaled-down versions of the digit images themselves (if you plot all digits, the visualization will be too cluttered, so you should either draw a random sample or plot an instance only if no other instance has already been plotted at a close distance). You should get a nice visualization with well-separated clusters of digits. Try using other dimensionality reduction algorithms such as PCA, LLE, or MDS and compare the resulting visualizations.

Solutions to these exercises are available in Appendix A.

CHAPTER 9
Unsupervised Learning Techniques

Although most of the applications of Machine Learning today are based on supervised learning (and as a result, this is where most of the investments go to), the vast majority of the available data is unlabeled: we have the input features **X**, but we do not have the labels **y**. The computer scientist Yann LeCun famously said that "if intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake." In other words, there is a huge potential in unsupervised learning that we have only barely started to sink our teeth into.

Say you want to create a system that will take a few pictures of each item on a manufacturing production line and detect which items are defective. You can fairly easily create a system that will take pictures automatically, and this might give you thousands of pictures every day. You can then build a reasonably large dataset in just a few weeks. But wait, there are no labels! If you want to train a regular binary classifier that will predict whether an item is defective or not, you will need to label every single picture as "defective" or "normal." This will generally require human experts to sit down and manually go through all the pictures. This is a long, costly, and tedious task, so it will usually only be done on a small subset of the available pictures. As a result, the labeled dataset will be quite small, and the classifier's performance will be disappointing. Moreover, every time the company makes any change to its products, the whole process will need to be started over from scratch. Wouldn't it be great if the algorithm could just exploit the unlabeled data without needing humans to label every picture? Enter unsupervised learning.

In Chapter 8 we looked at the most common unsupervised learning task: dimensionality reduction. In this chapter we will look at a few more unsupervised learning tasks and algorithms:

Clustering
: The goal is to group similar instances together into *clusters*. Clustering is a great tool for data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction, and more.

Anomaly detection
: The objective is to learn what "normal" data looks like, and then use that to detect abnormal instances, such as defective items on a production line or a new trend in a time series.

Density estimation
: This is the task of estimating the *probability density function* (PDF) of the random process that generated the dataset. Density estimation is commonly used for anomaly detection: instances located in very low-density regions are likely to be anomalies. It is also useful for data analysis and visualization.

Ready for some cake? We will start with clustering, using K-Means and DBSCAN, and then we will discuss Gaussian mixture models and see how they can be used for density estimation, clustering, and anomaly detection.

Clustering

As you enjoy a hike in the mountains, you stumble upon a plant you have never seen before. You look around and you notice a few more. They are not identical, yet they are sufficiently similar for you to know that they most likely belong to the same species (or at least the same genus). You may need a botanist to tell you what species that is, but you certainly don't need an expert to identify groups of similar-looking objects. This is called *clustering*: it is the task of identifying similar instances and assigning them to *clusters*, or groups of similar instances.

Just like in classification, each instance gets assigned to a group. However, unlike classification, clustering is an unsupervised task. Consider Figure 9-1: on the left is the iris dataset (introduced in Chapter 4), where each instance's species (i.e., its class) is represented with a different marker. It is a labeled dataset, for which classification algorithms such as Logistic Regression, SVMs, or Random Forest classifiers are well suited. On the right is the same dataset, but without the labels, so you cannot use a classification algorithm anymore. This is where clustering algorithms step in: many of them can easily detect the lower-left cluster. It is also quite easy to see with our own eyes, but it is not so obvious that the upper-right cluster is composed of two distinct sub-clusters. That said, the dataset has two additional features (sepal length and width), not represented here, and clustering algorithms can make good use of all features, so in fact they identify the three clusters fairly well (e.g., using a Gaussian mixture model, only 5 instances out of 150 are assigned to the wrong cluster).

Figure 9-1. Classification (left) versus clustering (right)

Clustering is used in a wide variety of applications, including these:

For customer segmentation
: You can cluster your customers based on their purchases and their activity on your website. This is useful to understand who your customers are and what they need, so you can adapt your products and marketing campaigns to each segment. For example, customer segmentation can be useful in *recommender systems* to suggest content that other users in the same cluster enjoyed.

For data analysis
: When you analyze a new dataset, it can be helpful to run a clustering algorithm, and then analyze each cluster separately.

As a dimensionality reduction technique
: Once a dataset has been clustered, it is usually possible to measure each instance's *affinity* with each cluster (affinity is any measure of how well an instance fits into a cluster). Each instance's feature vector $\mathbf{x}$ can then be replaced with the vector of its cluster affinities. If there are k clusters, then this vector is k-dimensional. This vector is typically much lower-dimensional than the original feature vector, but it can preserve enough information for further processing.

For anomaly detection (also called outlier detection)
: Any instance that has a low affinity to all the clusters is likely to be an anomaly. For example, if you have clustered the users of your website based on their behavior, you can detect users with unusual behavior, such as an unusual number of requests per second. Anomaly detection is particularly useful in detecting defects in manufacturing, or for *fraud detection*.

For semi-supervised learning
: If you only have a few labels, you could perform clustering and propagate the labels to all the instances in the same cluster. This technique can greatly increase

the number of labels available for a subsequent supervised learning algorithm, and thus improve its performance.

For search engines
: Some search engines let you search for images that are similar to a reference image. To build such a system, you would first apply a clustering algorithm to all the images in your database; similar images would end up in the same cluster. Then when a user provides a reference image, all you need to do is use the trained clustering model to find this image's cluster, and you can then simply return all the images from this cluster.

To segment an image
: By clustering pixels according to their color, then replacing each pixel's color with the mean color of its cluster, it is possible to considerably reduce the number of different colors in the image. Image segmentation is used in many object detection and tracking systems, as it makes it easier to detect the contour of each object.

There is no universal definition of what a cluster is: it really depends on the context, and different algorithms will capture different kinds of clusters. Some algorithms look for instances centered around a particular point, called a *centroid*. Others look for continuous regions of densely packed instances: these clusters can take on any shape. Some algorithms are hierarchical, looking for clusters of clusters. And the list goes on.

In this section, we will look at two popular clustering algorithms, K-Means and DBSCAN, and explore some of their applications, such as nonlinear dimensionality reduction, semi-supervised learning, and anomaly detection.

K-Means

Consider the unlabeled dataset represented in Figure 9-2: you can clearly see five blobs of instances. The K-Means algorithm is a simple algorithm capable of clustering this kind of dataset very quickly and efficiently, often in just a few iterations. It was proposed by Stuart Lloyd at Bell Labs in 1957 as a technique for pulse-code modulation, but it was only published outside of the company in 1982.[1] In 1965, Edward W. Forgy had published virtually the same algorithm, so K-Means is sometimes referred to as Lloyd–Forgy.

1 Stuart P. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory* 28, no. 2 (1982): 129–137.

Figure 9-2. An unlabeled dataset composed of five blobs of instances

Let's train a K-Means clusterer on this dataset. It will try to find each blob's center and assign each instance to the closest blob:

```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k)
y_pred = kmeans.fit_predict(X)
```

Note that you have to specify the number of clusters *k* that the algorithm must find. In this example, it is pretty obvious from looking at the data that *k* should be set to 5, but in general it is not that easy. We will discuss this shortly.

Each instance was assigned to one of the five clusters. In the context of clustering, an instance's *label* is the index of the cluster that this instance gets assigned to by the algorithm: this is not to be confused with the class labels in classification (remember that clustering is an unsupervised learning task). The `KMeans` instance preserves a copy of the labels of the instances it was trained on, available via the `labels_` instance variable:

```
>>> y_pred
array([4, 0, 1, ..., 2, 1, 0], dtype=int32)
>>> y_pred is kmeans.labels_
True
```

We can also take a look at the five centroids that the algorithm found:

```
>>> kmeans.cluster_centers_
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

You can easily assign new instances to the cluster whose centroid is closest:

```
>>> X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
>>> kmeans.predict(X_new)
array([1, 1, 2, 2], dtype=int32)
```

If you plot the cluster's decision boundaries, you get a Voronoi tessellation (see Figure 9-3, where each centroid is represented with an X).

Figure 9-3. K-Means decision boundaries (Voronoi tessellation)

The vast majority of the instances were clearly assigned to the appropriate cluster, but a few instances were probably mislabeled (especially near the boundary between the top-left cluster and the central cluster). Indeed, the K-Means algorithm does not behave very well when the blobs have very different diameters because all it cares about when assigning an instance to a cluster is the distance to the centroid.

Instead of assigning each instance to a single cluster, which is called *hard clustering*, it can be useful to give each instance a score per cluster, which is called *soft clustering*. The score can be the distance between the instance and the centroid; conversely, it can be a similarity score (or affinity), such as the Gaussian Radial Basis Function (introduced in Chapter 5). In the `KMeans` class, the `transform()` method measures the distance from each instance to every centroid:

```
>>> kmeans.transform(X_new)
array([[2.81093633, 0.32995317, 2.9042344 , 1.49439034, 2.88633901],
       [5.80730058, 2.80290755, 5.84739223, 4.4759332 , 5.84236351],
       [1.21475352, 3.29399768, 0.29040966, 1.69136631, 1.71086031],
       [0.72581411, 3.21806371, 0.36159148, 1.54808703, 1.21567622]])
```

In this example, the first instance in `X_new` is located at a distance of 2.81 from the first centroid, 0.33 from the second centroid, 2.90 from the third centroid, 1.49 from the fourth centroid, and 2.89 from the fifth centroid. If you have a high-dimensional dataset and you transform it this way, you end up with a *k*-dimensional dataset: this transformation can be a very efficient nonlinear dimensionality reduction technique.

The K-Means algorithm

So, how does the algorithm work? Well, suppose you were given the centroids. You could easily label all the instances in the dataset by assigning each of them to the cluster whose centroid is closest. Conversely, if you were given all the instance labels, you could easily locate all the centroids by computing the mean of the instances for each cluster. But you are given neither the labels nor the centroids, so how can you proceed? Well, just start by placing the centroids randomly (e.g., by picking *k* instances at random and using their locations as centroids). Then label the instances, update the centroids, label the instances, update the centroids, and so on until the centroids stop moving. The algorithm is guaranteed to converge in a finite number of steps (usually quite small); it will not oscillate forever.[2]

You can see the algorithm in action in Figure 9-4: the centroids are initialized randomly (top left), then the instances are labeled (top right), then the centroids are updated (center left), the instances are relabeled (center right), and so on. As you can see, in just three iterations, the algorithm has reached a clustering that seems close to optimal.

The computational complexity of the algorithm is generally linear with regard to the number of instances *m*, the number of clusters *k*, and the number of dimensions *n*. However, this is only true when the data has a clustering structure. If it does not, then in the worst-case scenario the complexity can increase exponentially with the number of instances. In practice, this rarely happens, and K-Means is generally one of the fastest clustering algorithms.

2 That's because the mean squared distance between the instances and their closest centroid can only go down at each step.

Figure 9-4. The K-Means algorithm

Although the algorithm is guaranteed to converge, it may not converge to the right solution (i.e., it may converge to a local optimum): whether it does or not depends on the centroid initialization. Figure 9-5 shows two suboptimal solutions that the algorithm can converge to if you are not lucky with the random initialization step.

Figure 9-5. Suboptimal solutions due to unlucky centroid initializations

Let's look at a few ways you can mitigate this risk by improving the centroid initialization.

Centroid initialization methods

If you happen to know approximately where the centroids should be (e.g., if you ran another clustering algorithm earlier), then you can set the `init` hyperparameter to a NumPy array containing the list of centroids, and set `n_init` to `1`:

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1)
```

Another solution is to run the algorithm multiple times with different random initializations and keep the best solution. The number of random initializations is controlled by the `n_init` hyperparameter: by default, it is equal to `10`, which means that the whole algorithm described earlier runs 10 times when you call `fit()`, and Scikit-Learn keeps the best solution. But how exactly does it know which solution is the best? It uses a performance metric! That metric is called the model's *inertia*, which is the mean squared distance between each instance and its closest centroid. It is roughly equal to 223.3 for the model on the left in Figure 9-5, 237.5 for the model on the right in Figure 9-5, and 211.6 for the model in Figure 9-3. The `KMeans` class runs the algorithm `n_init` times and keeps the model with the lowest inertia. In this example, the model in Figure 9-3 will be selected (unless we are very unlucky with `n_init` consecutive random initializations). If you are curious, a model's inertia is accessible via the `inertia_` instance variable:

```
>>> kmeans.inertia_
211.59853725816856
```

The `score()` method returns the negative inertia. Why negative? Because a predictor's `score()` method must always respect Scikit-Learn's "greater is better" rule: if a predictor is better than another, its `score()` method should return a greater score.

```
>>> kmeans.score(X)
-211.59853725816856
```

An important improvement to the K-Means algorithm, *K-Means++*, was proposed in a 2006 paper by David Arthur and Sergei Vassilvitskii.[3] They introduced a smarter initialization step that tends to select centroids that are distant from one another, and this improvement makes the K-Means algorithm much less likely to converge to a suboptimal solution. They showed that the additional computation required for the smarter initialization step is well worth it because it makes it possible to drastically reduce the number of times the algorithm needs to be run to find the optimal solution. Here is the K-Means++ initialization algorithm:

1. Take one centroid $\mathbf{c}^{(1)}$, chosen uniformly at random from the dataset.

3 David Arthur and Sergei Vassilvitskii, "k-Means++: The Advantages of Careful Seeding," *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms* (2007): 1027–1035.

2. Take a new centroid $\mathbf{c}^{(i)}$, choosing an instance $\mathbf{x}^{(i)}$ with probability $D\left(\mathbf{x}^{(i)}\right)^2 / \sum_{j=1}^{m} D\left(\mathbf{x}^{(j)}\right)^2$, where $D(\mathbf{x}^{(i)})$ is the distance between the instance $\mathbf{x}^{(i)}$ and the closest centroid that was already chosen. This probability distribution ensures that instances farther away from already chosen centroids are much more likely be selected as centroids.
3. Repeat the previous step until all k centroids have been chosen.

The `KMeans` class uses this initialization method by default. If you want to force it to use the original method (i.e., picking k instances randomly to define the initial centroids), then you can set the `init` hyperparameter to `"random"`. You will rarely need to do this.

Accelerated K-Means and mini-batch K-Means

Another important improvement to the K-Means algorithm was proposed in a 2003 paper by Charles Elkan.[4] It considerably accelerates the algorithm by avoiding many unnecessary distance calculations. Elkan achieved this by exploiting the triangle inequality (i.e., that a straight line is always the shortest distance between two points[5]) and by keeping track of lower and upper bounds for distances between instances and centroids. This is the algorithm the `KMeans` class uses by default (you can force it to use the original algorithm by setting the `algorithm` hyperparameter to `"full"`, although you probably will never need to).

Yet another important variant of the K-Means algorithm was proposed in a 2010 paper by David Sculley.[6] Instead of using the full dataset at each iteration, the algorithm is capable of using mini-batches, moving the centroids just slightly at each iteration. This speeds up the algorithm typically by a factor of three or four and makes it possible to cluster huge datasets that do not fit in memory. Scikit-Learn implements this algorithm in the `MiniBatchKMeans` class. You can just use this class like the `KMeans` class:

```
from sklearn.cluster import MiniBatchKMeans

minibatch_kmeans = MiniBatchKMeans(n_clusters=5)
minibatch_kmeans.fit(X)
```

4 Charles Elkan, "Using the Triangle Inequality to Accelerate k-Means," *Proceedings of the 20th International Conference on Machine Learning* (2003): 147–153.

5 The triangle inequality is AC ≤ AB + BC where A, B and C are three points and AB, AC, and BC are the distances between these points.

6 David Sculley, "Web-Scale K-Means Clustering," *Proceedings of the 19th International Conference on World Wide Web* (2010): 1177–1178.

If the dataset does not fit in memory, the simplest option is to use the `memmap` class, as we did for incremental PCA in Chapter 8. Alternatively, you can pass one mini-batch at a time to the `partial_fit()` method, but this will require much more work, since you will need to perform multiple initializations and select the best one yourself (see the mini-batch K-Means section of the notebook for an example).

Although the Mini-batch K-Means algorithm is much faster than the regular K-Means algorithm, its inertia is generally slightly worse, especially as the number of clusters increases. You can see this in Figure 9-6: the plot on the left compares the inertias of Mini-batch K-Means and regular K-Means models trained on the previous dataset using various numbers of clusters *k*. The difference between the two curves remains fairly constant, but this difference becomes more and more significant as *k* increases, since the inertia becomes smaller and smaller. In the plot on the right, you can see that Mini-batch K-Means is much faster than regular K-Means, and this difference increases with *k*.

Figure 9-6. Mini-batch K-Means has a higher inertia than K-Means (left) but it is much faster (right), especially as k increases

Finding the optimal number of clusters

So far, we have set the number of clusters *k* to 5 because it was obvious by looking at the data that this was the correct number of clusters. But in general, it will not be so easy to know how to set *k*, and the result might be quite bad if you set it to the wrong value. As you can see in Figure 9-7, setting *k* to 3 or 8 results in fairly bad models.

Figure 9-7. Bad choices for the number of clusters: when k is too small, separate clusters get merged (left), and when k is too large, some clusters get chopped into multiple pieces (right)

You might be thinking that we could just pick the model with the lowest inertia, right? Unfortunately, it is not that simple. The inertia for k=3 is 653.2, which is much higher than for k=5 (which was 211.6). But with k=8, the inertia is just 119.1. The inertia is not a good performance metric when trying to choose k because it keeps getting lower as we increase k. Indeed, the more clusters there are, the closer each instance will be to its closest centroid, and therefore the lower the inertia will be. Let's plot the inertia as a function of k (see Figure 9-8).

Figure 9-8. When plotting the inertia as a function of the number of clusters k, the curve often contains an inflexion point called the "elbow"

As you can see, the inertia drops very quickly as we increase k up to 4, but then it decreases much more slowly as we keep increasing k. This curve has roughly the shape of an arm, and there is an "elbow" at k = 4. So, if we did not know better, 4 would be a good choice: any lower value would be dramatic, while any higher value would not help much, and we might just be splitting perfectly good clusters in half for no good reason.

This technique for choosing the best value for the number of clusters is rather coarse. A more precise approach (but also more computationally expensive) is to use the *silhouette score*, which is the mean *silhouette coefficient* over all the instances. An

instance's silhouette coefficient is equal to $(b - a) / \max(a, b)$, where a is the mean distance to the other instances in the same cluster (i.e., the mean intra-cluster distance) and b is the mean nearest-cluster distance (i.e., the mean distance to the instances of the next closest cluster, defined as the one that minimizes b, excluding the instance's own cluster). The silhouette coefficient can vary between −1 and +1. A coefficient close to +1 means that the instance is well inside its own cluster and far from other clusters, while a coefficient close to 0 means that it is close to a cluster boundary, and finally a coefficient close to −1 means that the instance may have been assigned to the wrong cluster.

To compute the silhouette score, you can use Scikit-Learn's `silhouette_score()` function, giving it all the instances in the dataset and the labels they were assigned:

```
>>> from sklearn.metrics import silhouette_score
>>> silhouette_score(X, kmeans.labels_)
0.655517642572828
```

Let's compare the silhouette scores for different numbers of clusters (see Figure 9-9).

Figure 9-9. Selecting the number of clusters k using the silhouette score

As you can see, this visualization is much richer than the previous one: although it confirms that $k = 4$ is a very good choice, it also underlines the fact that $k = 5$ is quite good as well, and much better than $k = 6$ or 7. This was not visible when comparing inertias.

An even more informative visualization is obtained when you plot every instance's silhouette coefficient, sorted by the cluster they are assigned to and by the value of the coefficient. This is called a *silhouette diagram* (see Figure 9-10). Each diagram contains one knife shape per cluster. The shape's height indicates the number of instances the cluster contains, and its width represents the sorted silhouette coefficients of the instances in the cluster (wider is better). The dashed line indicates the mean silhouette coefficient.

Figure 9-10. Analyzing the silhouette diagrams for various values of k

The vertical dashed lines represent the silhouette score for each number of clusters. When most of the instances in a cluster have a lower coefficient than this score (i.e., if many of the instances stop short of the dashed line, ending to the left of it), then the cluster is rather bad since this means its instances are much too close to other clusters. We can see that when $k = 3$ and when $k = 6$, we get bad clusters. But when $k = 4$ or $k = 5$, the clusters look pretty good: most instances extend beyond the dashed line, to the right and closer to 1.0. When $k = 4$, the cluster at index 1 (the third from the top) is rather big. When $k = 5$, all clusters have similar sizes. So, even though the overall silhouette score from $k = 4$ is slightly greater than for $k = 5$, it seems like a good idea to use $k = 5$ to get clusters of similar sizes.

Limits of K-Means

Despite its many merits, most notably being fast and scalable, K-Means is not perfect. As we saw, it is necessary to run the algorithm several times to avoid suboptimal solutions, plus you need to specify the number of clusters, which can be quite a hassle. Moreover, K-Means does not behave very well when the clusters have varying sizes,

different densities, or nonspherical shapes. For example, Figure 9-11 shows how K-Means clusters a dataset containing three ellipsoidal clusters of different dimensions, densities, and orientations.

Figure 9-11. K-Means fails to cluster these ellipsoidal blobs properly

As you can see, neither of these solutions is any good. The solution on the left is better, but it still chops off 25% of the middle cluster and assigns it to the cluster on the right. The solution on the right is just terrible, even though its inertia is lower. So, depending on the data, different clustering algorithms may perform better. On these types of elliptical clusters, Gaussian mixture models work great.

It is important to scale the input features before you run K-Means, or the clusters may be very stretched and K-Means will perform poorly. Scaling the features does not guarantee that all the clusters will be nice and spherical, but it generally improves things.

Now let's look at a few ways we can benefit from clustering. We will use K-Means, but feel free to experiment with other clustering algorithms.

Using Clustering for Image Segmentation

Image segmentation is the task of partitioning an image into multiple segments. In *semantic segmentation*, all pixels that are part of the same object type get assigned to the same segment. For example, in a self-driving car's vision system, all pixels that are part of a pedestrian's image might be assigned to the "pedestrian" segment (there would be one segment containing all the pedestrians). In *instance segmentation*, all pixels that are part of the same individual object are assigned to the same segment. In this case there would be a different segment for each pedestrian. The state of the art in semantic or instance segmentation today is achieved using complex architectures based on convolutional neural networks (see Chapter 14). Here, we are going to do something much simpler: *color segmentation*. We will simply assign pixels to the same segment if they have a similar color. In some applications, this may be sufficient. For

example, if you want to analyze satellite images to measure how much total forest area there is in a region, color segmentation may be just fine.

First, use Matplotlib's `imread()` function to load the image (see the upper-left image in Figure 9-12):

```
>>> from matplotlib.image import imread  # or `from imageio import imread`
>>> image = imread(os.path.join("images","unsupervised_learning","ladybug.png"))
>>> image.shape
(533, 800, 3)
```

The image is represented as a 3D array. The first dimension's size is the height; the second is the width; and the third is the number of color channels, in this case red, green, and blue (RGB). In other words, for each pixel there is a 3D vector containing the intensities of red, green, and blue, each between 0.0 and 1.0 (or between 0 and 255, if you use `imageio.imread()`). Some images may have fewer channels, such as grayscale images (one channel). And some images may have more channels, such as images with an additional *alpha channel* for transparency or satellite images, which often contain channels for many light frequencies (e.g., infrared). The following code reshapes the array to get a long list of RGB colors, then it clusters these colors using K-Means:

```
X = image.reshape(-1, 3)
kmeans = KMeans(n_clusters=8).fit(X)
segmented_img = kmeans.cluster_centers_[kmeans.labels_]
segmented_img = segmented_img.reshape(image.shape)
```

For example, it may identify a color cluster for all shades of green. Next, for each color (e.g., dark green), it looks for the mean color of the pixel's color cluster. For example, all shades of green may be replaced with the same light green color (assuming the mean color of the green cluster is light green). Finally, it reshapes this long list of colors to get the same shape as the original image. And we're done!

This outputs the image shown in the upper right of Figure 9-12. You can experiment with various numbers of clusters, as shown in the figure. When you use fewer than eight clusters, notice that the ladybug's flashy red color fails to get a cluster of its own: it gets merged with colors from the environment. This is because K-Means prefers clusters of similar sizes. The ladybug is small—much smaller than the rest of the image—so even though its color is flashy, K-Means fails to dedicate a cluster to it.

Figure 9-12. Image segmentation using K-Means with various numbers of color clusters

That wasn't too hard, was it? Now let's look at another application of clustering: preprocessing.

Using Clustering for Preprocessing

Clustering can be an efficient approach to dimensionality reduction, in particular as a preprocessing step before a supervised learning algorithm. As an example of using clustering for dimensionality reduction, let's tackle the digits dataset, which is a simple MNIST-like dataset containing 1,797 grayscale 8 × 8 images representing the digits 0 to 9. First, load the dataset:

```
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)
```

Now, split it into a training set and a test set:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits)
```

Next, fit a Logistic Regression model:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
```

Let's evaluate its accuracy on the test set:

```
>>> log_reg.score(X_test, y_test)
0.9688888888888889
```

OK, that's our baseline: 96.9% accuracy. Let's see if we can do better by using K-Means as a preprocessing step. We will create a pipeline that will first cluster the training set into 50 clusters and replace the images with their distances to these 50 clusters, then apply a Logistic Regression model:

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=50)),
    ("log_reg", LogisticRegression()),
])
pipeline.fit(X_train, y_train)
```

Since there are 10 different digits, it is tempting to set the number of clusters to 10. However, each digit can be written several different ways, so it is preferable to use a larger number of clusters, such as 50.

Now let's evaluate this classification pipeline:

```
>>> pipeline.score(X_test, y_test)
0.9777777777777777
```

How about that? We reduced the error rate by almost 30% (from about 3.1% to about 2.2%)!

But we chose the number of clusters k arbitrarily; we can surely do better. Since K-Means is just a preprocessing step in a classification pipeline, finding a good value for k is much simpler than earlier. There's no need to perform silhouette analysis or minimize the inertia; the best value of k is simply the one that results in the best classification performance during cross-validation. We can use `GridSearchCV` to find the optimal number of clusters:

```
from sklearn.model_selection import GridSearchCV

param_grid = dict(kmeans__n_clusters=range(2, 100))
grid_clf = GridSearchCV(pipeline, param_grid, cv=3, verbose=2)
grid_clf.fit(X_train, y_train)
```

Let's look at the best value for k and the performance of the resulting pipeline:

```
>>> grid_clf.best_params_
{'kmeans__n_clusters': 99}
>>> grid_clf.score(X_test, y_test)
0.9822222222222222
```

With $k = 99$ clusters, we get a significant accuracy boost, reaching 98.22% accuracy on the test set. Cool! You may want to keep exploring higher values for k, since 99 was the largest value in the range we explored.

Using Clustering for Semi-Supervised Learning

Another use case for clustering is in semi-supervised learning, when we have plenty of unlabeled instances and very few labeled instances. Let's train a Logistic Regression model on a sample of 50 labeled instances from the digits dataset:

```
n_labeled = 50
log_reg = LogisticRegression()
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

What is the performance of this model on the test set?

```
>>> log_reg.score(X_test, y_test)
0.8333333333333334
```

The accuracy is just 83.3%. It should come as no surprise that this is much lower than earlier, when we trained the model on the full training set. Let's see how we can do better. First, let's cluster the training set into 50 clusters. Then for each cluster, let's find the image closest to the centroid. We will call these images the *representative images*:

```
k = 50
kmeans = KMeans(n_clusters=k)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = np.argmin(X_digits_dist, axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

Figure 9-13 shows these 50 representative images.

Figure 9-13. Fifty representative digit images (one per cluster)

Let's look at each image and manually label it:

```
y_representative_digits = np.array([4, 8, 0, 6, 8, 3, ..., 7, 6, 2, 3, 1, 1])
```

Now we have a dataset with just 50 labeled instances, but instead of being random instances, each of them is a representative image of its cluster. Let's see if the performance is any better:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_representative_digits, y_representative_digits)
>>> log_reg.score(X_test, y_test)
0.9222222222222223
```

Wow! We jumped from 83.3% accuracy to 92.2%, although we are still only training the model on 50 instances. Since it is often costly and painful to label instances, especially when it has to be done manually by experts, it is a good idea to label representative instances rather than just random instances.

But perhaps we can go one step further: what if we propagated the labels to all the other instances in the same cluster? This is called *label propagation*:

```
y_train_propagated = np.empty(len(X_train), dtype=np.int32)
for i in range(k):
    y_train_propagated[kmeans.labels_==i] = y_representative_digits[i]
```

Now let's train the model again and look at its performance:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train, y_train_propagated)
>>> log_reg.score(X_test, y_test)
0.9333333333333333
```

We got a reasonable accuracy boost, but nothing absolutely astounding. The problem is that we propagated each representative instance's label to all the instances in the same cluster, including the instances located close to the cluster boundaries, which are more likely to be mislabeled. Let's see what happens if we only propagate the labels to the 20% of the instances that are closest to the centroids:

```
percentile_closest = 20

X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]
for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1

partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]
```

Now let's train the model again on this partially propagated dataset:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
>>> log_reg.score(X_test, y_test)
0.94
```

Nice! With just 50 labeled instances (only 5 examples per class on average!), we got 94.0% accuracy, which is pretty close to the performance of Logistic Regression on the fully labeled digits dataset (which was 96.9%). This good performance is due to the fact that the propagated labels are actually pretty good—their accuracy is very close to 99%, as the following code shows:

```
>>> np.mean(y_train_partially_propagated == y_train[partially_propagated])
0.9896907216494846
```

Active Learning

To continue improving your model and your training set, the next step could be to do a few rounds of *active learning*, which is when a human expert interacts with the learning algorithm, providing labels for specific instances when the algorithm requests them. There are many different strategies for active learning, but one of the most common ones is called *uncertainty sampling*. Here is how it works:

1. The model is trained on the labeled instances gathered so far, and this model is used to make predictions on all the unlabeled instances.
2. The instances for which the model is most uncertain (i.e., when its estimated probability is lowest) are given to the expert to be labeled.
3. You iterate this process until the performance improvement stops being worth the labeling effort.

Other strategies include labeling the instances that would result in the largest model change, or the largest drop in the model's validation error, or the instances that different models disagree on (e.g., an SVM or a Random Forest).

Before we move on to Gaussian mixture models, let's take a look at DBSCAN, another popular clustering algorithm that illustrates a very different approach based on local density estimation. This approach allows the algorithm to identify clusters of arbitrary shapes.

DBSCAN

This algorithm defines clusters as continuous regions of high density. Here is how it works:

- For each instance, the algorithm counts how many instances are located within a small distance ε (epsilon) from it. This region is called the instance's *ε-neighborhood*.
- If an instance has at least `min_samples` instances in its ε-neighborhood (including itself), then it is considered a *core instance*. In other words, core instances are those that are located in dense regions.
- All instances in the neighborhood of a core instance belong to the same cluster. This neighborhood may include other core instances; therefore, a long sequence of neighboring core instances forms a single cluster.

- Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly.

This algorithm works well if all the clusters are dense enough and if they are well separated by low-density regions. The `DBSCAN` class in Scikit-Learn is as simple to use as you might expect. Let's test it on the moons dataset, introduced in Chapter 5:

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05)
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

The labels of all the instances are now available in the `labels_` instance variable:

```
>>> dbscan.labels_
array([ 0,  2, -1, -1,  1,  0,  0,  0, ...,  3,  2,  3,  3,  4,  2,  6,  3])
```

Notice that some instances have a cluster index equal to −1, which means that they are considered as anomalies by the algorithm. The indices of the core instances are available in the `core_sample_indices_` instance variable, and the core instances themselves are available in the `components_` instance variable:

```
>>> len(dbscan.core_sample_indices_)
808
>>> dbscan.core_sample_indices_
array([ 0,  4,  5,  6,  7,  8, 10, 11, ..., 992, 993, 995, 997, 998, 999])
>>> dbscan.components_
array([[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
                  ...
       [-0.94355873,  0.3278936 ],
       [ 0.79419406,  0.60777171]])
```

This clustering is represented in the lefthand plot of Figure 9-14. As you can see, it identified quite a lot of anomalies, plus seven different clusters. How disappointing! Fortunately, if we widen each instance's neighborhood by increasing `eps` to 0.2, we get the clustering on the right, which looks perfect. Let's continue with this model.

Figure 9-14. DBSCAN clustering using two different neighborhood radiuses

Somewhat surprisingly, the `DBSCAN` class does not have a `predict()` method, although it has a `fit_predict()` method. In other words, it cannot predict which cluster a new instance belongs to. This implementation decision was made because different classification algorithms can be better for different tasks, so the authors decided to let the user choose which one to use. Moreover, it's not hard to implement. For example, let's train a `KNeighborsClassifier`:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_])
```

Now, given a few new instances, we can predict which cluster they most likely belong to and even estimate a probability for each cluster:

```
>>> X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
>>> knn.predict(X_new)
array([1, 0, 1, 0])
>>> knn.predict_proba(X_new)
array([[0.18, 0.82],
       [1.  , 0.  ],
       [0.12, 0.88],
       [1.  , 0.  ]])
```

Note that we only trained the classifier on the core instances, but we could also have chosen to train it on all the instances, or all but the anomalies: this choice depends on the final task.

The decision boundary is represented in Figure 9-15 (the crosses represent the four instances in `X_new`). Notice that since there is no anomaly in the training set, the classifier always chooses a cluster, even when that cluster is far away. It is fairly straightforward to introduce a maximum distance, in which case the two instances that are far away from both clusters are classified as anomalies. To do this, use the `kneighbors()` method of the `KNeighborsClassifier`. Given a set of instances, it returns the

distances and the indices of the k nearest neighbors in the training set (two matrices, each with k columns):

```
>>> y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
>>> y_pred = dbscan.labels_[dbscan.core_sample_indices_][y_pred_idx]
>>> y_pred[y_dist > 0.2] = -1
>>> y_pred.ravel()
array([-1,  0,  1, -1])
```

Figure 9-15. Decision boundary between two clusters

In short, DBSCAN is a very simple yet powerful algorithm capable of identifying any number of clusters of any shape. It is robust to outliers, and it has just two hyperparameters (`eps` and `min_samples`). If the density varies significantly across the clusters, however, it can be impossible for it to capture all the clusters properly. Its computational complexity is roughly $O(m \log m)$, making it pretty close to linear with regard to the number of instances, but Scikit-Learn's implementation can require up to $O(m^2)$ memory if `eps` is large.

You may also want to try *Hierarchical DBSCAN* (HDBSCAN), which is implemented in the scikit-learn-contrib project.

Other Clustering Algorithms

Scikit-Learn implements several more clustering algorithms that you should take a look at. We cannot cover them all in detail here, but here is a brief overview:

Agglomerative clustering

A hierarchy of clusters is built from the bottom up. Think of many tiny bubbles floating on water and gradually attaching to each other until there's one big group of bubbles. Similarly, at each iteration, agglomerative clustering connects the nearest pair of clusters (starting with individual instances). If you drew a tree

with a branch for every pair of clusters that merged, you would get a binary tree of clusters, where the leaves are the individual instances. This approach scales very well to large numbers of instances or clusters. It can capture clusters of various shapes, it produces a flexible and informative cluster tree instead of forcing you to choose a particular cluster scale, and it can be used with any pairwise distance. It can scale nicely to large numbers of instances if you provide a connectivity matrix, which is a sparse $m \times m$ matrix that indicates which pairs of instances are neighbors (e.g., returned by `sklearn.neighbors.kneighbors_graph()`). Without a connectivity matrix, the algorithm does not scale well to large datasets.

BIRCH

The BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) algorithm was designed specifically for very large datasets, and it can be faster than batch K-Means, with similar results, as long as the number of features is not too large (<20). During training, it builds a tree structure containing just enough information to quickly assign each new instance to a cluster, without having to store all the instances in the tree: this approach allows it to use limited memory, while handling huge datasets.

Mean-Shift

This algorithm starts by placing a circle centered on each instance; then for each circle it computes the mean of all the instances located within it, and it shifts the circle so that it is centered on the mean. Next, it iterates this mean-shifting step until all the circles stop moving (i.e., until each of them is centered on the mean of the instances it contains). Mean-Shift shifts the circles in the direction of higher density, until each of them has found a local density maximum. Finally, all the instances whose circles have settled in the same place (or close enough) are assigned to the same cluster. Mean-Shift has some of the same features as DBSCAN, like how it can find any number of clusters of any shape, it has very few hyperparameters (just one—the radius of the circles, called the *bandwidth*), and it relies on local density estimation. But unlike DBSCAN, Mean-Shift tends to chop clusters into pieces when they have internal density variations. Unfortunately, its computational complexity is $O(m^2)$, so it is not suited for large datasets.

Affinity propagation

This algorithm uses a voting system, where instances vote for similar instances to be their representatives, and once the algorithm converges, each representative and its voters form a cluster. Affinity propagation can detect any number of clusters of different sizes. Unfortunately, this algorithm has a computational complexity of $O(m^2)$, so it too is not suited for large datasets.

Spectral clustering

This algorithm takes a similarity matrix between the instances and creates a low-dimensional embedding from it (i.e., it reduces its dimensionality), then it uses

another clustering algorithm in this low-dimensional space (Scikit-Learn's implementation uses K-Means.) Spectral clustering can capture complex cluster structures, and it can also be used to cut graphs (e.g., to identify clusters of friends on a social network). It does not scale well to large numbers of instances, and it does not behave well when the clusters have very different sizes.

Now let's dive into Gaussian mixture models, which can be used for density estimation, clustering, and anomaly detection.

Gaussian Mixtures

A *Gaussian mixture model* (GMM) is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions whose parameters are unknown. All the instances generated from a single Gaussian distribution form a cluster that typically looks like an ellipsoid. Each cluster can have a different ellipsoidal shape, size, density, and orientation, just like in Figure 9-11. When you observe an instance, you know it was generated from one of the Gaussian distributions, but you are not told which one, and you do not know what the parameters of these distributions are.

There are several GMM variants. In the simplest variant, implemented in the `GaussianMixture` class, you must know in advance the number k of Gaussian distributions. The dataset $\mathbf{X}$ is assumed to have been generated through the following probabilistic process:

- For each instance, a cluster is picked randomly from among k clusters. The probability of choosing the jth cluster is defined by the cluster's weight, $\phi^{(j)}$.[7] The index of the cluster chosen for the ith instance is noted $z^{(i)}$.
- If $z^{(i)}=j$, meaning the ith instance has been assigned to the jth cluster, the location $\mathbf{x}^{(i)}$ of this instance is sampled randomly from the Gaussian distribution with mean $\boldsymbol{\mu}^{(j)}$ and covariance matrix $\boldsymbol{\Sigma}^{(j)}$. This is noted $\mathbf{x}^{(i)} \sim \mathcal{N}\left(\boldsymbol{\mu}^{(j)}, \boldsymbol{\Sigma}^{(j)}\right)$.

This generative process can be represented as a graphical model. Figure 9-16 represents the structure of the conditional dependencies between random variables.

7 Phi (ϕ or φ) is the 21st letter of the Greek alphabet.

Figure 9-16. A graphical representation of a Gaussian mixture model, including its parameters (squares), random variables (circles), and their conditional dependencies (solid arrows)

Here is how to interpret the figure:[8]

- The circles represent random variables.
- The squares represent fixed values (i.e., parameters of the model).
- The large rectangles are called *plates*. They indicate that their content is repeated several times.
- The number at the bottom right of each plate indicates how many times its content is repeated. So, there are m random variables $z^{(i)}$ (from $z^{(1)}$ to $z^{(m)}$) and m random variables $\mathbf{x}^{(i)}$. There are also k means $\boldsymbol{\mu}^{(j)}$ and k covariance matrices $\boldsymbol{\Sigma}^{(j)}$. Lastly, there is just one weight vector $\boldsymbol{\phi}$ (containing all the weights $\phi^{(1)}$ to $\phi^{(k)}$).
- Each variable $z^{(i)}$ is drawn from the *categorical distribution* with weights $\boldsymbol{\phi}$. Each variable $\mathbf{x}^{(i)}$ is drawn from the normal distribution, with the mean and covariance matrix defined by its cluster $z^{(i)}$.
- The solid arrows represent conditional dependencies. For example, the probability distribution for each random variable $z^{(i)}$ depends on the weight vector $\boldsymbol{\phi}$. Note that when an arrow crosses a plate boundary, it means that it applies to all the repetitions of that plate. For example, the weight vector $\boldsymbol{\phi}$ conditions the probability distributions of all the random variables $\mathbf{x}^{(1)}$ to $\mathbf{x}^{(m)}$.
- The squiggly arrow from $z^{(i)}$ to $\mathbf{x}^{(i)}$ represents a switch: depending on the value of $z^{(i)}$, the instance $\mathbf{x}^{(i)}$ will be sampled from a different Gaussian distribution. For example, if $z^{(i)}=j$, then $\mathbf{x}^{(i)} \sim \mathcal{N}\left(\boldsymbol{\mu}^{(j)}, \boldsymbol{\Sigma}^{(j)}\right)$.

8 Most of these notations are standard, but a few additional notations were taken from the Wikipedia article on plate notation.

- Shaded nodes indicate that the value is known. So, in this case, only the random variables $\mathbf{x}^{(i)}$ have known values: they are called *observed variables*. The unknown random variables $z^{(i)}$ are called *latent variables*.

So, what can you do with such a model? Well, given the dataset $\mathbf{X}$, you typically want to start by estimating the weights $\boldsymbol{\phi}$ and all the distribution parameters $\boldsymbol{\mu}^{(1)}$ to $\boldsymbol{\mu}^{(k)}$ and $\boldsymbol{\Sigma}^{(1)}$ to $\boldsymbol{\Sigma}^{(k)}$. Scikit-Learn's `GaussianMixture` class makes this super easy:

```
from sklearn.mixture import GaussianMixture

gm = GaussianMixture(n_components=3, n_init=10)
gm.fit(X)
```

Let's look at the parameters that the algorithm estimated:

```
>>> gm.weights_
array([0.20965228, 0.4000662 , 0.39028152])
>>> gm.means_
array([[ 3.39909717,  1.05933727],
       [-1.40763984,  1.42710194],
       [ 0.05135313,  0.07524095]])
>>> gm.covariances_
array([[[ 1.14807234, -0.03270354],
        [-0.03270354,  0.95496237]],

       [[ 0.63478101,  0.72969804],
        [ 0.72969804,  1.1609872 ]],

       [[ 0.68809572,  0.79608475],
        [ 0.79608475,  1.21234145]]])
```

Great, it worked fine! Indeed, the weights that were used to generate the data were 0.2, 0.4, and 0.4; and similarly, the means and covariance matrices were very close to those found by the algorithm. But how? This class relies on the *Expectation-Maximization* (EM) algorithm, which has many similarities with the K-Means algorithm: it also initializes the cluster parameters randomly, then it repeats two steps until convergence, first assigning instances to clusters (this is called the *expectation step*) and then updating the clusters (this is called the *maximization step*). Sounds familiar, right? In the context of clustering, you can think of EM as a generalization of K-Means that not only finds the cluster centers ($\boldsymbol{\mu}^{(1)}$ to $\boldsymbol{\mu}^{(k)}$), but also their size, shape, and orientation ($\boldsymbol{\Sigma}^{(1)}$ to $\boldsymbol{\Sigma}^{(k)}$), as well as their relative weights ($\phi^{(1)}$ to $\phi^{(k)}$). Unlike K-Means, though, EM uses soft cluster assignments, not hard assignments. For each instance, during the expectation step, the algorithm estimates the probability that it belongs to each cluster (based on the current cluster parameters). Then, during the maximization step, each cluster is updated using *all* the instances in the dataset, with each instance weighted by the estimated probability that it belongs to that cluster. These probabilities are called the *responsibilities* of the clusters for the instances.

During the maximization step, each cluster's update will mostly be impacted by the instances it is most responsible for.

Unfortunately, just like K-Means, EM can end up converging to poor solutions, so it needs to be run several times, keeping only the best solution. This is why we set `n_init` to `10`. Be careful: by default `n_init` is set to `1`.

You can check whether or not the algorithm converged and how many iterations it took:

```
>>> gm.converged_
True
>>> gm.n_iter_
3
```

Now that you have an estimate of the location, size, shape, orientation, and relative weight of each cluster, the model can easily assign each instance to the most likely cluster (hard clustering) or estimate the probability that it belongs to a particular cluster (soft clustering). Just use the `predict()` method for hard clustering, or the `predict_proba()` method for soft clustering:

```
>>> gm.predict(X)
array([2, 2, 1, ..., 0, 0, 0])
>>> gm.predict_proba(X)
array([[2.32389467e-02, 6.77397850e-07, 9.76760376e-01],
       [1.64685609e-02, 6.75361303e-04, 9.82856078e-01],
       [2.01535333e-06, 9.99923053e-01, 7.49319577e-05],
       ...,
       [9.99999571e-01, 2.13946075e-26, 4.28788333e-07],
       [1.00000000e+00, 1.46454409e-41, 5.12459171e-16],
       [1.00000000e+00, 8.02006365e-41, 2.27626238e-15]])
```

A Gaussian mixture model is a *generative model*, meaning you can sample new instances from it (note that they are ordered by cluster index):

```
>>> X_new, y_new = gm.sample(6)
>>> X_new
array([[ 2.95400315,  2.63680992],
       [-1.16654575,  1.62792705],
       [-1.39477712, -1.48511338],
       [ 0.27221525,  0.690366  ],
       [ 0.54095936,  0.48591934],
       [ 0.38064009, -0.56240465]])

>>> y_new
array([0, 1, 2, 2, 2, 2])
```

It is also possible to estimate the density of the model at any given location. This is achieved using the `score_samples()` method: for each instance it is given, this

method estimates the log of the *probability density function* (PDF) at that location. The greater the score, the higher the density:

```
>>> gm.score_samples(X)
array([-2.60782346, -3.57106041, -3.33003479, ..., -3.51352783,
       -4.39802535, -3.80743859])
```

If you compute the exponential of these scores, you get the value of the PDF at the location of the given instances. These are not probabilities, but probability *densities*: they can take on any positive value, not just a value between 0 and 1. To estimate the probability that an instance will fall within a particular region, you would have to integrate the PDF over that region (if you do so over the entire space of possible instance locations, the result will be 1).

Figure 9-17 shows the cluster means, the decision boundaries (dashed lines), and the density contours of this model.

Figure 9-17. Cluster means, decision boundaries, and density contours of a trained Gaussian mixture model

Nice! The algorithm clearly found an excellent solution. Of course, we made its task easy by generating the data using a set of 2D Gaussian distributions (unfortunately, real-life data is not always so Gaussian and low-dimensional). We also gave the algorithm the correct number of clusters. When there are many dimensions, or many clusters, or few instances, EM can struggle to converge to the optimal solution. You might need to reduce the difficulty of the task by limiting the number of parameters that the algorithm has to learn. One way to do this is to limit the range of shapes and orientations that the clusters can have. This can be achieved by imposing constraints on the covariance matrices. To do this, set the `covariance_type` hyperparameter to one of the following values:

`"spherical"`
: All clusters must be spherical, but they can have different diameters (i.e., different variances).

`"diag"`
: Clusters can take on any ellipsoidal shape of any size, but the ellipsoid's axes must be parallel to the coordinate axes (i.e., the covariance matrices must be diagonal).

`"tied"`
: All clusters must have the same ellipsoidal shape, size, and orientation (i.e., all clusters share the same covariance matrix).

By default, `covariance_type` is equal to `"full"`, which means that each cluster can take on any shape, size, and orientation (it has its own unconstrained covariance matrix). Figure 9-18 plots the solutions found by the EM algorithm when `covariance_type` is set to `"tied"` or `"spherical."`

Figure 9-18. Gaussian mixtures for tied clusters (left) and spherical clusters (right)

The computational complexity of training a `GaussianMixture` model depends on the number of instances m, the number of dimensions n, the number of clusters k, and the constraints on the covariance matrices. If `covariance_type` is `"spherical"` or `"diag"`, it is $O(kmn)$, assuming the data has a clustering structure. If `covariance_type` is `"tied"` or `"full"`, it is $O(kmn^2 + kn^3)$, so it will not scale to large numbers of features.

Gaussian mixture models can also be used for anomaly detection. Let's see how.

Anomaly Detection Using Gaussian Mixtures

Anomaly detection (also called *outlier detection*) is the task of detecting instances that deviate strongly from the norm. These instances are called *anomalies*, or *outliers*, while the normal instances are called *inliers*. Anomaly detection is useful in a wide variety of applications, such as fraud detection, detecting defective products in manufacturing, or removing outliers from a dataset before training another model (which can significantly improve the performance of the resulting model).

Using a Gaussian mixture model for anomaly detection is quite simple: any instance located in a low-density region can be considered an anomaly. You must define what density threshold you want to use. For example, in a manufacturing company that tries to detect defective products, the ratio of defective products is usually well known. Say it is equal to 4%. You then set the density threshold to be the value that results in having 4% of the instances located in areas below that threshold density. If you notice that you get too many false positives (i.e., perfectly good products that are flagged as defective), you can lower the threshold. Conversely, if you have too many false negatives (i.e., defective products that the system does not flag as defective), you can increase the threshold. This is the usual precision/recall trade-off (see Chapter 3). Here is how you would identify the outliers using the fourth percentile lowest density as the threshold (i.e., approximately 4% of the instances will be flagged as anomalies):

```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 4)
anomalies = X[densities < density_threshold]
```

Figure 9-19 represents these anomalies as stars.

Figure 9-19. Anomaly detection using a Gaussian mixture model

A closely related task is *novelty detection*: it differs from anomaly detection in that the algorithm is assumed to be trained on a "clean" dataset, uncontaminated by outliers, whereas anomaly detection does not make this assumption. Indeed, outlier detection is often used to clean up a dataset.

Gaussian mixture models try to fit all the data, including the outliers, so if you have too many of them, this will bias the model's view of "normality," and some outliers may wrongly be considered as normal. If this happens, you can try to fit the model once, use it to detect and remove the most extreme outliers, then fit the model again on the cleaned-up dataset. Another approach is to use robust covariance estimation methods (see the `EllipticEnvelope` class).

Just like K-Means, the `GaussianMixture` algorithm requires you to specify the number of clusters. So, how can you find it?

Selecting the Number of Clusters

With K-Means, you could use the inertia or the silhouette score to select the appropriate number of clusters. But with Gaussian mixtures, it is not possible to use these metrics because they are not reliable when the clusters are not spherical or have different sizes. Instead, you can try to find the model that minimizes a *theoretical information criterion*, such as the *Bayesian information criterion* (BIC) or the *Akaike information criterion* (AIC), defined in Equation 9-1.

Equation 9-1. Bayesian information criterion (BIC) and Akaike information criterion (AIC)

$$BIC = \log(m)p - 2\log\left(\hat{L}\right)$$

$$AIC = 2p - 2\log\left(\hat{L}\right)$$

In these equations:

- m is the number of instances, as always.
- p is the number of parameters learned by the model.
- $\hat{L}$ is the maximized value of the *likelihood function* of the model.

Both the BIC and the AIC penalize models that have more parameters to learn (e.g., more clusters) and reward models that fit the data well. They often end up selecting the same model. When they differ, the model selected by the BIC tends to be simpler

(fewer parameters) than the one selected by the AIC, but tends to not fit the data quite as well (this is especially true for larger datasets).

Likelihood Function

The terms "probability" and "likelihood" are often used interchangeably in the English language, but they have very different meanings in statistics. Given a statistical model with some parameters $\boldsymbol{\theta}$, the word "probability" is used to describe how plausible a future outcome $\mathbf{x}$ is (knowing the parameter values $\boldsymbol{\theta}$), while the word "likelihood" is used to describe how plausible a particular set of parameter values $\boldsymbol{\theta}$ are, after the outcome $\mathbf{x}$ is known.

Consider a 1D mixture model of two Gaussian distributions centered at −4 and +1. For simplicity, this toy model has a single parameter θ that controls the standard deviations of both distributions. The top-left contour plot in Figure 9-20 shows the entire model $f(x; \theta)$ as a function of both x and θ. To estimate the probability distribution of a future outcome x, you need to set the model parameter θ. For example, if you set θ to 1.3 (the horizontal line), you get the probability density function $f(x; \theta{=}1.3)$ shown in the lower-left plot. Say you want to estimate the probability that x will fall between −2 and +2. You must calculate the integral of the PDF on this range (i.e., the surface of the shaded region). But what if you don't know θ, and instead if you have observed a single instance $x{=}2.5$ (the vertical line in the upper-left plot)? In this case, you get the likelihood function $\mathscr{L}(\theta|x{=}2.5){=}f(x{=}2.5; \theta)$, represented in the upper-right plot.

Figure 9-20. A model's parametric function (top left), and some derived functions: a PDF (lower left), a likelihood function (top right), and a log likelihood function (lower right)

In short, the PDF is a function of x (with θ fixed), while the likelihood function is a function of θ (with x fixed). It is important to understand that the likelihood function is *not* a probability distribution: if you integrate a probability distribution over all possible values of x, you always get 1; but if you integrate the likelihood function over all possible values of θ, the result can be any positive value.

Given a dataset $\mathbf{X}$, a common task is to try to estimate the most likely values for the model parameters. To do this, you must find the values that maximize the likelihood function, given $\mathbf{X}$. In this example, if you have observed a single instance x=2.5, the *maximum likelihood estimate* (MLE) of θ is $\hat{\theta}$=1.5. If a prior probability distribution g over θ exists, it is possible to take it into account by maximizing $\mathscr{L}(\theta|x)g(\theta)$ rather than just maximizing $\mathscr{L}(\theta|x)$. This is called *maximum a-posteriori* (MAP) estimation. Since MAP constrains the parameter values, you can think of it as a regularized version of MLE.

Notice that maximizing the likelihood function is equivalent to maximizing its logarithm (represented in the lower-righthand plot in Figure 9-20). Indeed the logarithm is a strictly increasing function, so if θ maximizes the log likelihood, it also maximizes the likelihood. It turns out that it is generally easier to maximize the log likelihood. For example, if you observed several independent instances $x^{(1)}$ to $x^{(m)}$, you would need to find the value of θ that maximizes the product of the individual likelihood functions. But it is equivalent, and much simpler, to maximize the sum (not the product) of the log likelihood functions, thanks to the magic of the logarithm which converts products into sums: $\log(ab)=\log(a)+\log(b)$.

Once you have estimated $\hat{\theta}$, the value of θ that maximizes the likelihood function, then you are ready to compute $\hat{L} = \mathscr{L}\left(\hat{\theta}, \mathbf{X}\right)$, which is the value used to compute the AIC and BIC; you can think of it as a measure of how well the model fits the data.

To compute the BIC and AIC, call the `bic()` and `aic()` methods:

```
>>> gm.bic(X)
8189.74345832983
>>> gm.aic(X)
8102.518178214792
```

Figure 9-21 shows the BIC for different numbers of clusters k. As you can see, both the BIC and the AIC are lowest when k=3, so it is most likely the best choice. Note that we could also search for the best value for the `covariance_type` hyperparameter. For example, if it is `"spherical"` rather than `"full"`, then the model has significantly fewer parameters to learn, but it does not fit the data as well.

Figure 9-21. AIC and BIC for different numbers of clusters k

Bayesian Gaussian Mixture Models

Rather than manually searching for the optimal number of clusters, you can use the `BayesianGaussianMixture` class, which is capable of giving weights equal (or close) to zero to unnecessary clusters. Set the number of clusters `n_components` to a value that you have good reason to believe is greater than the optimal number of clusters (this assumes some minimal knowledge about the problem at hand), and the algorithm will eliminate the unnecessary clusters automatically. For example, let's set the number of clusters to 10 and see what happens:

```
>>> from sklearn.mixture import BayesianGaussianMixture
>>> bgm = BayesianGaussianMixture(n_components=10, n_init=10)
>>> bgm.fit(X)
>>> np.round(bgm.weights_, 2)
array([0.4 , 0.21, 0.4 , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ])
```

Perfect: the algorithm automatically detected that only three clusters are needed, and the resulting clusters are almost identical to the ones in Figure 9-17.

In this model, the cluster parameters (including the weights, means, and covariance matrices) are not treated as fixed model parameters anymore, but as latent random variables, like the cluster assignments (see Figure 9-22). So **z** now includes both the cluster parameters and the cluster assignments.

The Beta distribution is commonly used to model random variables whose values lie within a fixed range. In this case, the range is from 0 to 1. The Stick-Breaking Process (SBP) is best explained through an example: suppose Φ=[0.3, 0.6, 0.5,…], then 30% of the instances will be assigned to cluster 0, then 60% of the remaining instances will be assigned to cluster 1, then 50% of the remaining instances will be assigned to cluster 2, and so on. This process is a good model for datasets where new instances are more likely to join large clusters than small clusters (e.g., people are more likely to move to larger cities). If the concentration α is high, then Φ values will likely be close to 0, and the SBP generate many clusters. Conversely, if the concentration is low, then Φ values

will likely be close to 1, and there will be few clusters. Finally, the Wishart distribution is used to sample covariance matrices: the parameters d and V control the distribution of cluster shapes.

Figure 9-22. Bayesian Gaussian mixture model

Prior knowledge about the latent variables **z** can be encoded in a probability distribution $p(\mathbf{z})$ called the *prior*. For example, we may have a prior belief that the clusters are likely to be few (low concentration), or conversely, that they are likely to be plentiful (high concentration). This prior belief about the number of clusters can be adjusted using the `weight_concentration_prior` hyperparameter. Setting it to 0.01 or 10,000 gives very different clusterings (see Figure 9-23). The more data we have, however, the less the priors matter. In fact, to plot diagrams with such large differences, you must use very strong priors and little data.

Figure 9-23. Using different concentration priors on the same data results in different numbers of clusters

Bayes' theorem (Equation 9-2) tells us how to update the probability distribution over the latent variables after we observe some data **X**. It computes the *posterior* distribution $p(\mathbf{z}|\mathbf{X})$, which is the conditional probability of **z** given **X**.

Equation 9-2. Bayes' theorem

$$p(\mathbf{z}|\mathbf{X}) = \text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}} = \frac{p(\mathbf{X}|\mathbf{z})\, p(\mathbf{z})}{p(\mathbf{X})}$$

Unfortunately, in a Gaussian mixture model (and many other problems), the denominator $p(\mathbf{x})$ is intractable, as it requires integrating over all the possible values of **z** (Equation 9-3), which would require considering all possible combinations of cluster parameters and cluster assignments.

Equation 9-3. The evidence $p(\mathbf{X})$ is often intractable

$$p\left(\mathbf{X}\right) = \int p(\mathbf{X}|\mathbf{z})p(\mathbf{z})d\mathbf{z}$$

This intractability is one of the central problems in Bayesian statistics, and there are several approaches to solving it. One of them is *variational inference*, which picks a family of distributions $q(\mathbf{z}; \boldsymbol{\lambda})$ with its own *variational parameters* $\boldsymbol{\lambda}$ (lambda), then optimizes these parameters to make $q(\mathbf{z})$ a good approximation of $p(\mathbf{z}|\mathbf{X})$. This is achieved by finding the value of $\boldsymbol{\lambda}$ that minimizes the KL divergence from $q(\mathbf{z})$ to $p(\mathbf{z}|\mathbf{X})$, noted $D_{KL}(q\|p)$. The KL divergence equation is shown in Equation 9-4, and it can be rewritten as the log of the evidence ($\log p(\mathbf{X})$) minus the *evidence lower bound* (ELBO). Since the log of the evidence does not depend on q, it is a constant term, so minimizing the KL divergence just requires maximizing the ELBO.

Equation 9-4. KL divergence from $q(\mathbf{z})$ to $p(\mathbf{z}|\mathbf{X})$

$$\begin{aligned}
D_{KL}(q \parallel p) &= \mathbb{E}_q\left[\log \frac{q(\mathbf{z})}{p(\mathbf{z} \mid \mathbf{X})}\right] \\
&= \mathbb{E}_q[\log q(\mathbf{z}) - \log p(\mathbf{z} \mid \mathbf{X})] \\
&= \mathbb{E}_q\left[\log q(\mathbf{z}) - \log \frac{p(\mathbf{z}, \mathbf{X})}{p(\mathbf{X})}\right] \\
&= \mathbb{E}_q[\log q(\mathbf{z}) - \log p(\mathbf{z}, \mathbf{X}) + \log p(\mathbf{X})] \\
&= \mathbb{E}_q[\log q(\mathbf{z})] - \mathbb{E}_q[\log p(\mathbf{z}, \mathbf{X})] + \mathbb{E}_q[\log p(\mathbf{X})] \\
&= \mathbb{E}_q[\log p(\mathbf{X})] - \left(\mathbb{E}_q[\log p(\mathbf{z}, \mathbf{X})] - \mathbb{E}_q[\log q(\mathbf{z})]\right) \\
&= \log p(\mathbf{X}) - \text{ELBO} \\
&\quad \text{where } \text{ELBO} = \mathbb{E}_q[\log p(\mathbf{z}, \mathbf{X})] - \mathbb{E}_q[\log q(\mathbf{z})]
\end{aligned}$$

In practice, there are different techniques to maximize the ELBO. In *mean field variational inference*, it is necessary to pick the family of distributions $q(\mathbf{z}; \boldsymbol{\lambda})$ and the prior $p(z)$ very carefully to ensure that the equation for the ELBO simplifies to a form that can be computed. Unfortunately, there is no general way to do this. Picking the right family of distributions and the right prior depends on the task and requires some mathematical skills. For example, the distributions and lower-bound equations used in Scikit-Learn's `BayesianGaussianMixture` class are presented in the documentation. From these equations it is possible to derive update equations for the cluster parameters and assignment variables: these are then used very much like in the Expectation-Maximization algorithm. In fact, the computational complexity of the `BayesianGaussianMixture` class is similar to that of the `GaussianMixture` class (but generally significantly slower). A simpler approach to maximizing the ELBO is called *black box stochastic variational inference* (BBSVI): at each iteration, a few samples are drawn from q, and they are used to estimate the gradients of the ELBO with regard to the variational parameters $\boldsymbol{\lambda}$, which are then used in a gradient ascent step. This approach makes it possible to use Bayesian inference with any kind of model (provided it is differentiable), even deep neural networks; using Bayesian inference with deep neural networks is called Bayesian Deep Learning.

If you want to dive deeper into Bayesian statistics, check out the book *Bayesian Data Analysis* by Andrew Gelman et al. (Chapman & Hall).

Gaussian mixture models work great on clusters with ellipsoidal shapes, but if you try to fit a dataset with different shapes, you may have bad surprises. For example, let's see what happens if we use a Bayesian Gaussian mixture model to cluster the moons dataset (see Figure 9-24).

Figure 9-24. Fitting a Gaussian mixture to nonellipsoidal clusters

Oops! The algorithm desperately searched for ellipsoids, so it found eight different clusters instead of two. The density estimation is not too bad, so this model could perhaps be used for anomaly detection, but it failed to identify the two moons. Let's now look at a few clustering algorithms capable of dealing with arbitrarily shaped clusters.

Other Algorithms for Anomaly and Novelty Detection

Scikit-Learn implements other algorithms dedicated to anomaly detection or novelty detection:

PCA (and other dimensionality reduction techniques with an `inverse_transform()` method)

If you compare the reconstruction error of a normal instance with the reconstruction error of an anomaly, the latter will usually be much larger. This is a simple and often quite efficient anomaly detection approach (see this chapter's exercises for an application of this approach).

Fast-MCD (minimum covariance determinant)

Implemented by the `EllipticEnvelope` class, this algorithm is useful for outlier detection, in particular to clean up a dataset. It assumes that the normal instances (inliers) are generated from a single Gaussian distribution (not a mixture). It also assumes that the dataset is contaminated with outliers that were not generated from this Gaussian distribution. When the algorithm estimates the parameters of the Gaussian distribution (i.e., the shape of the elliptic envelope around the inliers), it is careful to ignore the instances that are most likely outliers. This technique gives a better estimation of the elliptic envelope and thus makes the algorithm better at identifying the outliers.

Isolation Forest

This is an efficient algorithm for outlier detection, especially in high-dimensional datasets. The algorithm builds a Random Forest in which each Decision Tree is grown randomly: at each node, it picks a feature randomly, then it picks a random threshold value (between the min and max values) to split the dataset in two. The dataset gradually gets chopped into pieces this way, until all instances end up isolated from the other instances. Anomalies are usually far from other instances, so on average (across all the Decision Trees) they tend to get isolated in fewer steps than normal instances.

Local Outlier Factor (LOF)

This algorithm is also good for outlier detection. It compares the density of instances around a given instance to the density around its neighbors. An anomaly is often more isolated than its k nearest neighbors.

One-class SVM

This algorithm is better suited for novelty detection. Recall that a kernelized SVM classifier separates two classes by first (implicitly) mapping all the instances to a high-dimensional space, then separating the two classes using a linear SVM classifier within this high-dimensional space (see Chapter 5). Since we just have one class of instances, the one-class SVM algorithm instead tries to separate the instances in high-dimensional space from the origin. In the original space, this will correspond to finding a small region that encompasses all the instances. If a new instance does not fall within this region, it is an anomaly. There are a few hyperparameters to tweak: the usual ones for a kernelized SVM, plus a margin hyperparameter that corresponds to the probability of a new instance being mistakenly considered as novel when it is in fact normal. It works great, especially with high-dimensional datasets, but like all SVMs it does not scale to large datasets.

Exercises

1. How would you define clustering? Can you name a few clustering algorithms?
2. What are some of the main applications of clustering algorithms?
3. Describe two techniques to select the right number of clusters when using K-Means.
4. What is label propagation? Why would you implement it, and how?
5. Can you name two clustering algorithms that can scale to large datasets? And two that look for regions of high density?
6. Can you think of a use case where active learning would be useful? How would you implement it?
7. What is the difference between anomaly detection and novelty detection?
8. What is a Gaussian mixture? What tasks can you use it for?
9. Can you name two techniques to find the right number of clusters when using a Gaussian mixture model?
10. The classic Olivetti faces dataset contains 400 grayscale 64 × 64–pixel images of faces. Each image is flattened to a 1D vector of size 4,096. 40 different people were photographed (10 times each), and the usual task is to train a model that can predict which person is represented in each picture. Load the dataset using the `sklearn.datasets.fetch_olivetti_faces()` function, then split it into a training set, a validation set, and a test set (note that the dataset is already scaled between 0 and 1). Since the dataset is quite small, you probably want to use stratified sampling to ensure that there are the same number of images per person in each set. Next, cluster the images using K-Means, and ensure that you have a

good number of clusters (using one of the techniques discussed in this chapter). Visualize the clusters: do you see similar faces in each cluster?

11. Continuing with the Olivetti faces dataset, train a classifier to predict which person is represented in each picture, and evaluate it on the validation set. Next, use K-Means as a dimensionality reduction tool, and train a classifier on the reduced set. Search for the number of clusters that allows the classifier to get the best performance: what performance can you reach? What if you append the features from the reduced set to the original features (again, searching for the best number of clusters)?
12. Train a Gaussian mixture model on the Olivetti faces dataset. To speed up the algorithm, you should probably reduce the dataset's dimensionality (e.g., use PCA, preserving 99% of the variance). Use the model to generate some new faces (using the `sample()` method), and visualize them (if you used PCA, you will need to use its `inverse_transform()` method). Try to modify some images (e.g., rotate, flip, darken) and see if the model can detect the anomalies (i.e., compare the output of the `score_samples()` method for normal images and for anomalies).
13. Some dimensionality reduction techniques can also be used for anomaly detection. For example, take the Olivetti faces dataset and reduce it with PCA, preserving 99% of the variance. Then compute the reconstruction error for each image. Next, take some of the modified images you built in the previous exercise, and look at their reconstruction error: notice how much larger the reconstruction error is. If you plot a reconstructed image, you will see why: it tries to reconstruct a normal face.

Solutions to these exercises are available in Appendix A.

PART II

Neural Networks and Deep Learning

CHAPTER 10

Introduction to Artificial Neural Networks with Keras

Birds inspired us to fly, burdock plants inspired Velcro, and nature has inspired countless more inventions. It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine. This is the logic that sparked *artificial neural networks* (ANNs): an ANN is a Machine Learning model inspired by the networks of biological neurons found in our brains. However, although planes were inspired by birds, they don't have to flap their wings. Similarly, ANNs have gradually become quite different from their biological cousins. Some researchers even argue that we should drop the biological analogy altogether (e.g., by saying "units" rather than "neurons"), lest we restrict our creativity to biologically plausible systems.[1]

ANNs are at the very core of Deep Learning. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex Machine Learning tasks such as classifying billions of images (e.g., Google Images), powering speech recognition services (e.g., Apple's Siri), recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or learning to beat the world champion at the game of Go (DeepMind's AlphaGo).

The first part of this chapter introduces artificial neural networks, starting with a quick tour of the very first ANN architectures and leading up to *Multilayer Perceptrons* (MLPs), which are heavily used today (other architectures will be explored in the next chapters). In the second part, we will look at how to implement neural networks using the popular Keras API. This is a beautifully designed and simple high-

1 You can get the best of both worlds by being open to biological inspirations without being afraid to create biologically unrealistic models, as long as they work well.

level API for building, training, evaluating, and running neural networks. But don't be fooled by its simplicity: it is expressive and flexible enough to let you build a wide variety of neural network architectures. In fact, it will probably be sufficient for most of your use cases. And should you ever need extra flexibility, you can always write custom Keras components using its lower-level API, as we will see in Chapter 12.

But first, let's go back in time to see how artificial neural networks came to be!

From Biological to Artificial Neurons

Surprisingly, ANNs have been around for quite a while: they were first introduced back in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts. In their landmark paper[2] "A Logical Calculus of Ideas Immanent in Nervous Activity," McCulloch and Pitts presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using *propositional logic*. This was the first artificial neural network architecture. Since then many other architectures have been invented, as we will see.

The early successes of ANNs led to the widespread belief that we would soon be conversing with truly intelligent machines. When it became clear in the 1960s that this promise would go unfulfilled (at least for quite a while), funding flew elsewhere, and ANNs entered a long winter. In the early 1980s, new architectures were invented and better training techniques were developed, sparking a revival of interest in *connectionism* (the study of neural networks). But progress was slow, and by the 1990s other powerful Machine Learning techniques were invented, such as Support Vector Machines (see Chapter 5). These techniques seemed to offer better results and stronger theoretical foundations than ANNs, so once again the study of neural networks was put on hold.

We are now witnessing yet another wave of interest in ANNs. Will this wave die out like the previous ones did? Well, here are a few good reasons to believe that this time is different and that the renewed interest in ANNs will have a much more profound impact on our lives:

- There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
- The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to Moore's law (the number of components in integrated circuits has

2 Warren S. McCulloch and Walter Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," *The Bulletin of Mathematical Biology* 5, no. 4 (1943): 115–113.

doubled about every 2 years over the last 50 years), but also thanks to the gaming industry, which has stimulated the production of powerful GPU cards by the millions. Moreover, cloud platforms have made this power accessible to everyone.

- The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have had a huge positive impact.
- Some theoretical limitations of ANNs have turned out to be benign in practice. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima, but it turns out that this is rather rare in practice (and when it is the case, they are usually fairly close to the global optimum).
- ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the headline news, which pulls more and more attention and funding toward them, resulting in more and more progress and even more amazing products.

Biological Neurons

Before we discuss artificial neurons, let's take a quick look at a biological neuron (represented in Figure 10-1). It is an unusual-looking cell mostly found in animal brains. It's composed of a *cell body* containing the nucleus and most of the cell's complex components, many branching extensions called *dendrites*, plus one very long extension called the *axon*. The axon's length may be just a few times longer than the cell body, or up to tens of thousands of times longer. Near its extremity the axon splits off into many branches called *telodendria*, and at the tip of these branches are minuscule structures called *synaptic terminals* (or simply *synapses*), which are connected to the dendrites or cell bodies of other neurons.[3] Biological neurons produce short electrical impulses called *action potentials* (APs, or just *signals*) which travel along the axons and make the synapses release chemical signals called *neurotransmitters*. When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses (actually, it depends on the neurotransmitters, as some of them inhibit the neuron from firing).

3 They are not actually attached, just so close that they can very quickly exchange chemical signals.

Figure 10-1. Biological neuron[4]

Thus, individual biological neurons seem to behave in a rather simple way, but they are organized in a vast network of billions, with each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a network of fairly simple neurons, much like a complex anthill can emerge from the combined efforts of simple ants. The architecture of biological neural networks (BNNs)[5] is still the subject of active research, but some parts of the brain have been mapped, and it seems that neurons are often organized in consecutive layers, especially in the cerebral cortex (i.e., the outer layer of your brain), as shown in Figure 10-2.

4 Image by Bruce Blaus (Creative Commons 3.0). Reproduced from *https://en.wikipedia.org/wiki/Neuron*.

5 In the context of Machine Learning, the phrase "neural networks" generally refers to ANNs, not BNNs.

Figure 10-2. Multiple layers in a biological neural network (human cortex)[6]

Logical Computations with Neurons

McCulloch and Pitts proposed a very simple model of the biological neuron, which later became known as an *artificial neuron*: it has one or more binary (on/off) inputs and one binary output. The artificial neuron activates its output when more than a certain number of its inputs are active. In their paper, they showed that even with such a simplified model it is possible to build a network of artificial neurons that computes any logical proposition you want. To see how such a network works, let's build a few ANNs that perform various logical computations (see Figure 10-3), assuming that a neuron is activated when at least two of its inputs are active.

Figure 10-3. ANNs performing simple logical computations

6 Drawing of a cortical lamination by S. Ramon y Cajal (public domain). Reproduced from *https://en.wikipedia.org/wiki/Cerebral_cortex*.

Let's see what these networks do:

- The first network on the left is the identity function: if neuron A is activated, then neuron C gets activated as well (since it receives two input signals from neuron A); but if neuron A is off, then neuron C is off as well.
- The second network performs a logical AND: neuron C is activated only when both neurons A and B are activated (a single input signal is not enough to activate neuron C).
- The third network performs a logical OR: neuron C gets activated if either neuron A or neuron B is activated (or both).
- Finally, if we suppose that an input connection can inhibit the neuron's activity (which is the case with biological neurons), then the fourth network computes a slightly more complex logical proposition: neuron C is activated only if neuron A is active and neuron B is off. If neuron A is active all the time, then you get a logical NOT: neuron C is active when neuron B is off, and vice versa.

You can imagine how these networks can be combined to compute complex logical expressions (see the exercises at the end of the chapter for an example).

The Perceptron

The *Perceptron* is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt. It is based on a slightly different artificial neuron (see Figure 10-4) called a *threshold logic unit* (TLU), or sometimes a *linear threshold unit* (LTU). The inputs and output are numbers (instead of binary on/off values), and each input connection is associated with a weight. The TLU computes a weighted sum of its inputs ($z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{x}^\intercal \mathbf{w}$), then applies a *step function* to that sum and outputs the result: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z)$, where $z = \mathbf{x}^\intercal \mathbf{w}$.

Figure 10-4. Threshold logic unit: an artificial neuron which computes a weighted sum of its inputs then applies a step function

The most common step function used in Perceptrons is the *Heaviside step function* (see Equation 10-1). Sometimes the sign function is used instead.

Equation 10-1. Common step functions used in Perceptrons (assuming threshold = 0)

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

A single TLU can be used for simple linear binary classification. It computes a linear combination of the inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise it outputs the negative class (just like a Logistic Regression or linear SVM classifier). You could, for example, use a single TLU to classify iris flowers based on petal length and width (also adding an extra bias feature $x_0 = 1$, just like we did in previous chapters). Training a TLU in this case means finding the right values for w_0, w_1, and w_2 (the training algorithm is discussed shortly).

A Perceptron is simply composed of a single layer of TLUs,[7] with each TLU connected to all the inputs. When all the neurons in a layer are connected to every neuron in the previous layer (i.e., its input neurons), the layer is called a *fully connected layer*, or a *dense layer*. The inputs of the Perceptron are fed to special passthrough neurons called *input neurons*: they output whatever input they are fed. All the input neurons form the *input layer*. Moreover, an extra bias feature is generally added ($x_0 = 1$): it is typically represented using a special type of neuron called a *bias neuron*, which outputs 1 all the time. A Perceptron with two inputs and three outputs is represented in Figure 10-5. This Perceptron can classify instances simultaneously into three different binary classes, which makes it a multioutput classifier.

7 The name *Perceptron* is sometimes used to mean a tiny network with a single TLU.

Figure 10-5. Architecture of a Perceptron with two input neurons, one bias neuron, and three output neurons

Thanks to the magic of linear algebra, Equation 10-2 makes it possible to efficiently compute the outputs of a layer of artificial neurons for several instances at once.

Equation 10-2. Computing the outputs of a fully connected layer

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

In this equation:

- As always, **X** represents the matrix of input features. It has one row per instance and one column per feature.
- The weight matrix **W** contains all the connection weights except for the ones from the bias neuron. It has one row per input neuron and one column per artificial neuron in the layer.
- The bias vector **b** contains all the connection weights between the bias neuron and the artificial neurons. It has one bias term per artificial neuron.
- The function ϕ is called the *activation function*: when the artificial neurons are TLUs, it is a step function (but we will discuss other activation functions shortly).

So, how is a Perceptron trained? The Perceptron training algorithm proposed by Rosenblatt was largely inspired by *Hebb's rule*. In his 1949 book *The Organization of Behavior* (Wiley), Donald Hebb suggested that when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger. Siegrid Löwel later summarized Hebb's idea in the catchy phrase, "Cells that fire together, wire together"; that is, the connection weight between two neurons tends to increase when they fire simultaneously. This rule later became known as Hebb's rule (or *Hebbian learning*). Perceptrons are trained using a variant of this rule that takes into account the error made by the network when it makes a prediction; the

Perceptron learning rule reinforces connections that help reduce the error. More specifically, the Perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction. The rule is shown in Equation 10-3.

Equation 10-3. Perceptron learning rule (weight update)

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i$$

In this equation:

- $w_{i,j}$ is the connection weight between the ith input neuron and the jth output neuron.
- x_i is the ith input value of the current training instance.
- $\hat{y}_j$ is the output of the jth output neuron for the current training instance.
- y_j is the target output of the jth output neuron for the current training instance.
- η is the learning rate.

The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex patterns (just like Logistic Regression classifiers). However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution.[8] This is called the *Perceptron convergence theorem*.

Scikit-Learn provides a `Perceptron` class that implements a single-TLU network. It can be used pretty much as you would expect—for example, on the iris dataset (introduced in Chapter 4):

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)  # Iris setosa?

per_clf = Perceptron()
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

8 Note that this solution is not unique: when data points are linearly separable, there is an infinity of hyperplanes that can separate them.

You may have noticed that the Perceptron learning algorithm strongly resembles Stochastic Gradient Descent. In fact, Scikit-Learn's `Perceptron` class is equivalent to using an `SGDClassifier` with the following hyperparameters: `loss="perceptron"`, `learning_rate="constant"`, `eta0=1` (the learning rate), and `penalty=None` (no regularization).

Note that contrary to Logistic Regression classifiers, Perceptrons do not output a class probability; rather, they make predictions based on a hard threshold. This is one reason to prefer Logistic Regression over Perceptrons.

In their 1969 monograph *Perceptrons*, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of Perceptrons—in particular, the fact that they are incapable of solving some trivial problems (e.g., the *Exclusive OR* (XOR) classification problem; see the left side of Figure 10-6). This is true of any other linear classification model (such as Logistic Regression classifiers), but researchers had expected much more from Perceptrons, and some were so disappointed that they dropped neural networks altogether in favor of higher-level problems such as logic, problem solving, and search.

It turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons. The resulting ANN is called a *Multilayer Perceptron* (MLP). An MLP can solve the XOR problem, as you can verify by computing the output of the MLP represented on the right side of Figure 10-6: with inputs (0, 0) or (1, 1), the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1. All connections have a weight equal to 1, except the four connections where the weight is shown. Try verifying that this network indeed solves the XOR problem!

Figure 10-6. XOR classification problem and an MLP that solves it

The Multilayer Perceptron and Backpropagation

An MLP is composed of one (passthrough) *input layer*, one or more layers of TLUs, called *hidden layers*, and one final layer of TLUs called the *output layer* (see Figure 10-7). The layers close to the input layer are usually called the *lower layers*, and the ones close to the outputs are usually called the *upper layers*. Every layer except the output layer includes a bias neuron and is fully connected to the next layer.

Figure 10-7. Architecture of a Multilayer Perceptron with two inputs, one hidden layer of four neurons, and three output neurons (the bias neurons are shown here, but usually they are implicit)

The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a *feedforward neural network* (FNN).

When an ANN contains a deep stack of hidden layers,[9] it is called a *deep neural network* (DNN). The field of Deep Learning studies DNNs, and more generally models containing deep stacks of computations. Even so, many people talk about Deep Learning whenever neural networks are involved (even shallow ones).

For many years researchers struggled to find a way to train MLPs, without success. But in 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a

9 In the 1990s, an ANN with more than two hidden layers was considered deep. Nowadays, it is common to see ANNs with dozens of layers, or even hundreds, so the definition of "deep" is quite fuzzy.

groundbreaking paper[10] that introduced the *backpropagation* training algorithm, which is still used today. In short, it is Gradient Descent (introduced in Chapter 4) using an efficient technique for computing the gradients automatically:[11] in just two passes through the network (one forward, one backward), the backpropagation algorithm is able to compute the gradient of the network's error with regard to every single model parameter. In other words, it can find out how each connection weight and each bias term should be tweaked in order to reduce the error. Once it has these gradients, it just performs a regular Gradient Descent step, and the whole process is repeated until the network converges to the solution.

Automatically computing gradients is called *automatic differentiation*, or *autodiff*. There are various autodiff techniques, with different pros and cons. The one used by backpropagation is called *reverse-mode autodiff*. It is fast and precise, and is well suited when the function to differentiate has many variables (e.g., connection weights) and few outputs (e.g., one loss). If you want to learn more about autodiff, check out Appendix D.

Let's run through this algorithm in a bit more detail:

- It handles one mini-batch at a time (for example, containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an *epoch*.
- Each mini-batch is passed to the network's input layer, which sends it to the first hidden layer. The algorithm then computes the output of all the neurons in this layer (for every instance in the mini-batch). The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer. This is the *forward pass*: it is exactly like making predictions, except all intermediate results are preserved since they are needed for the backward pass.
- Next, the algorithm measures the network's output error (i.e., it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error).
- Then it computes how much each output connection contributed to the error. This is done analytically by applying the *chain rule* (perhaps the most fundamental rule in calculus), which makes this step fast and precise.

10 David Rumelhart et al. "Learning Internal Representations by Error Propagation," (Defense Technical Information Center technical report, September 1985).

11 This technique was actually independently invented several times by various researchers in different fields, starting with Paul Werbos in 1974.

- The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until the algorithm reaches the input layer. As explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward through the network (hence the name of the algorithm).
- Finally, the algorithm performs a Gradient Descent step to tweak all the connection weights in the network, using the error gradients it just computed.

This algorithm is so important that it's worth summarizing it again: for each training instance, the backpropagation algorithm first makes a prediction (forward pass) and measures the error, then goes through each layer in reverse to measure the error contribution from each connection (reverse pass), and finally tweaks the connection weights to reduce the error (Gradient Descent step).

It is important to initialize all the hidden layers' connection weights randomly, or else training will fail. For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and thus backpropagation will affect them in exactly the same way, so they will remain identical. In other words, despite having hundreds of neurons per layer, your model will act as if it had only one neuron per layer: it won't be too smart. If instead you randomly initialize the weights, you *break the symmetry* and allow backpropagation to train a diverse team of neurons.

In order for this algorithm to work properly, its authors made a key change to the MLP's architecture: they replaced the step function with the logistic (sigmoid) function, $\sigma(z) = 1 / (1 + \exp(-z))$. This was essential because the step function contains only flat segments, so there is no gradient to work with (Gradient Descent cannot move on a flat surface), while the logistic function has a well-defined nonzero derivative everywhere, allowing Gradient Descent to make some progress at every step. In fact, the backpropagation algorithm works well with many other activation functions, not just the logistic function. Here are two other popular choices:

The hyperbolic tangent function: $\tanh(z) = 2\sigma(2z) - 1$

Just like the logistic function, this activation function is *S*-shaped, continuous, and differentiable, but its output value ranges from –1 to 1 (instead of 0 to 1 in the case of the logistic function). That range tends to make each layer's output more or less centered around 0 at the beginning of training, which often helps speed up convergence.

The Rectified Linear Unit function: ReLU(z) = max(0, z)

The ReLU function is continuous but unfortunately not differentiable at $z = 0$ (the slope changes abruptly, which can make Gradient Descent bounce around), and its derivative is 0 for $z < 0$. In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default.[12] Most importantly, the fact that it does not have a maximum output value helps reduce some issues during Gradient Descent (we will come back to this in Chapter 11).

These popular activation functions and their derivatives are represented in Figure 10-8. But wait! Why do we need activation functions in the first place? Well, if you chain several linear transformations, all you get is a linear transformation. For example, if $f(x) = 2x + 3$ and $g(x) = 5x - 1$, then chaining these two linear functions gives you another linear function: $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$. So if you don't have some nonlinearity between layers, then even a deep stack of layers is equivalent to a single layer, and you can't solve very complex problems with that. Conversely, a large enough DNN with nonlinear activations can theoretically approximate any continuous function.

Figure 10-8. Activation functions and their derivatives

OK! You know where neural nets came from, what their architecture is, and how to compute their outputs. You've also learned about the backpropagation algorithm. But what exactly can you do with them?

Regression MLPs

First, MLPs can be used for regression tasks. If you want to predict a single value (e.g., the price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value. For multivariate regression (i.e., to predict

12 Biological neurons seem to implement a roughly sigmoid (*S*-shaped) activation function, so researchers stuck to sigmoid functions for a very long time. But it turns out that ReLU generally works better in ANNs. This is one of the cases where the biological analogy was misleading.

multiple values at once), you need one output neuron per output dimension. For example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two output neurons. If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object. So, you end up with four output neurons.

In general, when building an MLP for regression, you do not want to use any activation function for the output neurons, so they are free to output any range of values. If you want to guarantee that the output will always be positive, then you can use the ReLU activation function in the output layer. Alternatively, you can use the *softplus* activation function, which is a smooth variant of ReLU: $\text{softplus}(z) = \log(1 + \exp(z))$. It is close to 0 when z is negative, and close to z when z is positive. Finally, if you want to guarantee that the predictions will fall within a given range of values, then you can use the logistic function or the hyperbolic tangent, and then scale the labels to the appropriate range: 0 to 1 for the logistic function and –1 to 1 for the hyperbolic tangent.

The loss function to use during training is typically the mean squared error, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you can use the Huber loss, which is a combination of both.

The Huber loss is quadratic when the error is smaller than a threshold δ (typically 1) but linear when the error is larger than δ. The linear part makes it less sensitive to outliers than the mean squared error, and the quadratic part allows it to converge faster and be more precise than the mean absolute error.

Table 10-1 summarizes the typical architecture of a regression MLP.

Table 10-1. Typical regression MLP architecture

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Classification MLPs

MLPs can also be used for classification tasks. For a binary classification problem, you just need a single output neuron using the logistic activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class. The estimated probability of the negative class is equal to one minus that number.

MLPs can also easily handle multilabel binary classification tasks (see Chapter 3). For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or nonurgent email. In this case, you would need two output neurons, both using the logistic activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent. More generally, you would dedicate one output neuron for each positive class. Note that the output probabilities do not necessarily add up to 1. This lets the model output any combination of labels: you can have nonurgent ham, urgent ham, nonurgent spam, and perhaps even urgent spam (although that would probably be an error).

If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the softmax activation function for the whole output layer (see Figure 10-9). The softmax function (introduced in Chapter 4) will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1 (which is required if the classes are exclusive). This is called multiclass classification.

Figure 10-9. A modern MLP (including ReLU and softmax) for classification

Regarding the loss function, since we are predicting probability distributions, the cross-entropy loss (also called the log loss, see Chapter 4) is generally a good choice.

Table 10-2 summarizes the typical architecture of a classification MLP.

Table 10-2. Typical classification MLP architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

Before we go on, I recommend you go through exercise 1 at the end of this chapter. You will play with various neural network architectures and visualize their outputs using the *TensorFlow Playground*. This will be very useful to better understand MLPs, including the effects of all the hyperparameters (number of layers and neurons, activation functions, and more).

Now you have all the concepts you need to start implementing MLPs with Keras!

Implementing MLPs with Keras

Keras is a high-level Deep Learning API that allows you to easily build, train, evaluate, and execute all sorts of neural networks. Its documentation (or specification) is available at *https://keras.io/*. The reference implementation, also called Keras, was developed by François Chollet as part of a research project[13] and was released as an open source project in March 2015. It quickly gained popularity, owing to its ease of use, flexibility, and beautiful design. To perform the heavy computations required by neural networks, this reference implementation relies on a computation backend. At present, you can choose from three popular open source Deep Learning libraries: TensorFlow, Microsoft Cognitive Toolkit (CNTK), and Theano. Therefore, to avoid any confusion, we will refer to this reference implementation as *multibackend Keras*.

Since late 2016, other implementations have been released. You can now run Keras on Apache MXNet, Apple's Core ML, JavaScript or TypeScript (to run Keras code in a web browser), and PlaidML (which can run on all sorts of GPU devices, not just Nvidia). Moreover, TensorFlow itself now comes bundled with its own Keras implementation, tf.keras. It only supports TensorFlow as the backend, but it has the advantage of offering some very useful extra features (see Figure 10-10): for example, it supports

13 Project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System).

TensorFlow's Data API, which makes it easy to load and preprocess data efficiently. For this reason, we will use tf.keras in this book. However, in this chapter we will not use any of the TensorFlow-specific features, so the code should run fine on other Keras implementations as well (at least in Python), with only minor modifications, such as changing the imports.

Figure 10-10. Two implementations of the Keras API: multibackend Keras (left) and tf.keras (right)

The most popular Deep Learning library, after Keras and TensorFlow, is Facebook's PyTorch library. The good news is that its API is quite similar to Keras's (in part because both APIs were inspired by Scikit-Learn and Chainer), so once you know Keras, it is not difficult to switch to PyTorch, if you ever want to. PyTorch's popularity grew exponentially in 2018, largely thanks to its simplicity and excellent documentation, which were not TensorFlow 1.x's main strengths. However, TensorFlow 2 is arguably just as simple as PyTorch, as it has adopted Keras as its official high-level API and its developers have greatly simplified and cleaned up the rest of the API. The documentation has also been completely reorganized, and it is much easier to find what you need now. Similarly, PyTorch's main weaknesses (e.g., limited portability and no computation graph analysis) have been largely addressed in PyTorch 1.0. Healthy competition is beneficial to everyone.

All right, it's time to code! As tf.keras is bundled with TensorFlow, let's start by installing TensorFlow.

Installing TensorFlow 2

Assuming you installed Jupyter and Scikit-Learn by following the installation instructions in Chapter 2, use pip to install TensorFlow. If you created an isolated environment using virtualenv, you first need to activate it:

```
$ cd $ML_PATH                 # Your ML working directory (e.g., $HOME/ml)
$ source my_env/bin/activate  # on Linux or macOS
$ .\my_env\Scripts\activate   # on Windows
```

Next, install TensorFlow 2 (if you are not using a virtualenv, you will need administrator rights, or to add the `--user` option):

```
$ python3 -m pip install -U tensorflow
```

For GPU support, at the time of this writing you need to install `tensorflow-gpu` instead of `tensorflow`, but the TensorFlow team is working on having a single library that will support both CPU-only and GPU-equipped systems. You will still need to install extra libraries for GPU support (see *https://tensorflow.org/install* for more details). We will look at GPUs in more depth in Chapter 19.

To test your installation, open a Python shell or a Jupyter notebook, then import TensorFlow and tf.keras and print their versions:

```
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> tf.__version__
'2.0.0'
>>> keras.__version__
'2.2.4-tf'
```

The second version is the version of the Keras API implemented by tf.keras. Note that it ends with `-tf`, highlighting the fact that tf.keras implements the Keras API, plus some extra TensorFlow-specific features.

Now let's use tf.keras! We'll start by building a simple image classifier.

Building an Image Classifier Using the Sequential API

First, we need to load a dataset. In this chapter we will tackle Fashion MNIST, which is a drop-in replacement of MNIST (introduced in Chapter 3). It has the exact same format as MNIST (70,000 grayscale images of 28 × 28 pixels each, with 10 classes), but the images represent fashion items rather than handwritten digits, so each class is more diverse, and the problem turns out to be significantly more challenging than MNIST. For example, a simple linear model reaches about 92% accuracy on MNIST, but only about 83% on Fashion MNIST.

Using Keras to load the dataset

Keras provides some utility functions to fetch and load common datasets, including MNIST, Fashion MNIST, and the California housing dataset we used in Chapter 2. Let's load Fashion MNIST:

```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

When loading MNIST or Fashion MNIST using Keras rather than Scikit-Learn, one important difference is that every image is represented as a 28 × 28 array rather than a 1D array of size 784. Moreover, the pixel intensities are represented as integers (from 0 to 255) rather than floats (from 0.0 to 255.0). Let's take a look at the shape and data type of the training set:

```
>>> X_train_full.shape
(60000, 28, 28)
>>> X_train_full.dtype
dtype('uint8')
```

Note that the dataset is already split into a training set and a test set, but there is no validation set, so we'll create one now. Additionally, since we are going to train the neural network using Gradient Descent, we must scale the input features. For simplicity, we'll scale the pixel intensities down to the 0–1 range by dividing them by 255.0 (this also converts them to floats):

```
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

With MNIST, when the label is equal to 5, it means that the image represents the handwritten digit 5. Easy. For Fashion MNIST, however, we need the list of class names to know what we are dealing with:

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

For example, the first image in the training set represents a coat:

```
>>> class_names[y_train[0]]
'Coat'
```

Figure 10-11 shows some samples from the Fashion MNIST dataset.

Figure 10-11. Samples from Fashion MNIST

Creating the model using the Sequential API

Now let's build the neural network! Here is a classification MLP with two hidden layers:

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

Let's go through this code line by line:

- The first line creates a `Sequential` model. This is the simplest kind of Keras model for neural networks that are just composed of a single stack of layers connected sequentially. This is called the Sequential API.
- Next, we build the first layer and add it to the model. It is a `Flatten` layer whose role is to convert each input image into a 1D array: if it receives input data `X`, it computes `X.reshape(-1, 1)`. This layer does not have any parameters; it is just there to do some simple preprocessing. Since it is the first layer in the model, you should specify the `input_shape`, which doesn't include the batch size, only the shape of the instances. Alternatively, you could add a `keras.layers.InputLayer` as the first layer, setting `input_shape=[28,28]`.
- Next we add a `Dense` hidden layer with 300 neurons. It will use the ReLU activation function. Each `Dense` layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs. It also manages a vector of bias terms (one per neuron). When it receives some input data, it computes Equation 10-2.
- Then we add a second `Dense` hidden layer with 100 neurons, also using the ReLU activation function.
- Finally, we add a `Dense` output layer with 10 neurons (one per class), using the softmax activation function (because the classes are exclusive).

Specifying `activation="relu"` is equivalent to specifying `activation=keras.activations.relu`. Other activation functions are available in the `keras.activations` package, we will use many of them in this book. See *https://keras.io/activations/* for the full list.

Instead of adding the layers one by one as we just did, you can pass a list of layers when creating the `Sequential` model:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Using Code Examples from keras.io

Code examples documented on keras.io will work fine with tf.keras, but you need to change the imports. For example, consider this keras.io code:

```
from keras.layers import Dense
output_layer = Dense(10)
```

You must change the imports like this:

```
from tensorflow.keras.layers import Dense
output_layer = Dense(10)
```

Or simply use full paths, if you prefer:

```
from tensorflow import keras
output_layer = keras.layers.Dense(10)
```

This approach is more verbose, but I use it in this book so you can easily see which packages to use, and to avoid confusion between standard classes and custom classes. In production code, I prefer the previous approach. Many people also use `from tensorflow.keras import layers` followed by `layers.Dense(10)`.

The model's `summary()` method displays all the model's layers,[14] including each layer's name (which is automatically generated unless you set it when creating the layer), its output shape (`None` means the batch size can be anything), and its number of parameters. The summary ends with the total number of parameters, including trainable and non-trainable parameters. Here we only have trainable parameters (we will see examples of non-trainable parameters in Chapter 11):

```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 300)               235500
_________________________________________________________________
```

14 You can use `keras.utils.plot_model()` to generate an image of your model.

```
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

Note that `Dense` layers often have a *lot* of parameters. For example, the first hidden layer has 784 × 300 connection weights, plus 300 bias terms, which adds up to 235,500 parameters! This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the risk of overfitting, especially when you do not have a lot of training data. We will come back to this later.

You can easily get a model's list of layers, to fetch a layer by its index, or you can fetch it by name:

```
>>> model.layers
[<tensorflow.python.keras.layers.core.Flatten at 0x132414e48>,
 <tensorflow.python.keras.layers.core.Dense at 0x1324149b0>,
 <tensorflow.python.keras.layers.core.Dense at 0x1356ba8d0>,
 <tensorflow.python.keras.layers.core.Dense at 0x13240d240>]
>>> hidden1 = model.layers[1]
>>> hidden1.name
'dense'
>>> model.get_layer('dense') is hidden1
True
```

All the parameters of a layer can be accessed using its `get_weights()` and `set_weights()` methods. For a `Dense` layer, this includes both the connection weights and the bias terms:

```
>>> weights, biases = hidden1.get_weights()
>>> weights
array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
         0.03859074, -0.06889391],
       ...,
       [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,
         0.00272203, -0.06793761]], dtype=float32)
>>> weights.shape
(784, 300)
>>> biases
array([0., 0., 0., 0., 0., 0., 0., 0., 0., ...,  0., 0., 0.], dtype=float32)
>>> biases.shape
(300,)
```

Notice that the `Dense` layer initialized the connection weights randomly (which is needed to break symmetry, as we discussed earlier), and the biases were initialized to zeros, which is fine. If you ever want to use a different initialization method, you can set `kernel_initializer` (*kernel* is another name for the matrix of connection

weights) or `bias_initializer` when creating the layer. We will discuss initializers further in Chapter 11, but if you want the full list, see *https://keras.io/initializers/*.

The shape of the weight matrix depends on the number of inputs. This is why it is recommended to specify the `input_shape` when creating the first layer in a `Sequential` model. However, if you do not specify the input shape, it's OK: Keras will simply wait until it knows the input shape before it actually builds the model. This will happen either when you feed it actual data (e.g., during training), or when you call its `build()` method. Until the model is really built, the layers will not have any weights, and you will not be able to do certain things (such as print the model summary or save the model). So, if you know the input shape when creating the model, it is best to specify it.

Compiling the model

After a model is created, you must call its `compile()` method to specify the loss function and the optimizer to use. Optionally, you can specify a list of extra metrics to compute during training and evaluation:

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

Using `loss="sparse_categorical_crossentropy"` is equivalent to using `loss=keras.losses.sparse_categorical_crossentropy`. Similarly, specifying `optimizer="sgd"` is equivalent to specifying `optimizer=keras.optimizers.SGD()`, and `metrics=["accuracy"]` is equivalent to `metrics=[keras.metrics.sparse_categorical_accuracy]` (when using this loss). We will use many other losses, optimizers, and metrics in this book; for the full lists, see *https://keras.io/losses*, *https://keras.io/optimizers*, and *https://keras.io/metrics*.

This code requires some explanation. First, we use the `"sparse_categorical_crossentropy"` loss because we have sparse labels (i.e., for each instance, there is just a target class index, from 0 to 9 in this case), and the classes are exclusive. If instead we had one target probability per class for each instance (such as one-hot vectors, e.g. `[0., 0., 0., 1., 0., 0., 0., 0., 0., 0.]` to represent class 3), then we would need to use the `"categorical_crossentropy"` loss instead. If we were doing binary classification (with one or more binary labels), then we would use the `"sigmoid"` (i.e., logistic) activation function in the output layer instead of the `"softmax"` activation function, and we would use the `"binary_crossentropy"` loss.

If you want to convert sparse labels (i.e., class indices) to one-hot vector labels, use the `keras.utils.to_categorical()` function. To go the other way round, use the `np.argmax()` function with `axis=1`.

Regarding the optimizer, `"sgd"` means that we will train the model using simple Stochastic Gradient Descent. In other words, Keras will perform the backpropagation algorithm described earlier (i.e., reverse-mode autodiff plus Gradient Descent). We will discuss more efficient optimizers in Chapter 11 (they improve the Gradient Descent part, not the autodiff).

When using the `SGD` optimizer, it is important to tune the learning rate. So, you will generally want to use `optimizer=keras.optimizers.SGD(lr=???)` to set the learning rate, rather than `optimizer="sgd"`, which defaults to `lr=0.01`.

Finally, since this is a classifier, it's useful to measure its `"accuracy"` during training and evaluation.

Training and evaluating the model

Now the model is ready to be trained. For this we simply need to call its `fit()` method:

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [======] - 3s 49us/sample - loss: 0.7218     - accuracy: 0.7660
                                     - val_loss: 0.4973 - val_accuracy: 0.8366
Epoch 2/30
55000/55000 [======] - 2s 45us/sample - loss: 0.4840     - accuracy: 0.8327
                                     - val_loss: 0.4456 - val_accuracy: 0.8480
[...]
Epoch 30/30
55000/55000 [======] - 3s 53us/sample - loss: 0.2252     - accuracy: 0.9192
                                     - val_loss: 0.2999 - val_accuracy: 0.8926
```

We pass it the input features (`X_train`) and the target classes (`y_train`), as well as the number of epochs to train (or else it would default to just 1, which would definitely not be enough to converge to a good solution). We also pass a validation set (this is optional). Keras will measure the loss and the extra metrics on this set at the end of each epoch, which is very useful to see how well the model really performs. If the performance on the training set is much better than on the validation set, your model is

probably overfitting the training set (or there is a bug, such as a data mismatch between the training set and the validation set).

And that's it! The neural network is trained.[15] At each epoch during training, Keras displays the number of instances processed so far (along with a progress bar), the mean training time per sample, and the loss and accuracy (or any other extra metrics you asked for) on both the training set and the validation set. You can see that the training loss went down, which is a good sign, and the validation accuracy reached 89.26% after 30 epochs. That's not too far from the training accuracy, so there does not seem to be much overfitting going on.

Instead of passing a validation set using the `validation_data` argument, you could set `validation_split` to the ratio of the training set that you want Keras to use for validation. For example, `validation_split=0.1` tells Keras to use the last 10% of the data (before shuffling) for validation.

If the training set was very skewed, with some classes being overrepresented and others underrepresented, it would be useful to set the `class_weight` argument when calling the `fit()` method, which would give a larger weight to underrepresented classes and a lower weight to overrepresented classes. These weights would be used by Keras when computing the loss. If you need per-instance weights, set the `sample_weight` argument (if both `class_weight` and `sample_weight` are provided, Keras multiplies them). Per-instance weights could be useful if some instances were labeled by experts while others were labeled using a crowdsourcing platform: you might want to give more weight to the former. You can also provide sample weights (but not class weights) for the validation set by adding them as a third item in the `validation_data` tuple.

The `fit()` method returns a `History` object containing the training parameters (`history.params`), the list of epochs it went through (`history.epoch`), and most importantly a dictionary (`history.history`) containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if any). If you use this dictionary to create a pandas DataFrame and call its `plot()` method, you get the learning curves shown in Figure 10-12:

15 If your training or validation data does not match the expected shape, you will get an exception. This is perhaps the most common error, so you should get familiar with the error message. The message is actually quite clear: for example, if you try to train this model with an array containing flattened images (`X_train.reshape(-1, 784)`), then you will get the following exception: "ValueError: Error when checking input: expected flatten_input to have 3 dimensions, but got array with shape (60000, 784)."

```
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # set the vertical range to [0-1]
plt.show()
```

Figure 10-12. Learning curves: the mean training loss and accuracy measured over each epoch, and the mean validation loss and accuracy measured at the end of each epoch

You can see that both the training accuracy and the validation accuracy steadily increase during training, while the training loss and the validation loss decrease. Good! Moreover, the validation curves are close to the training curves, which means that there is not too much overfitting. In this particular case, the model looks like it performed better on the validation set than on the training set at the beginning of training. But that's not the case: indeed, the validation error is computed at the *end* of each epoch, while the training error is computed using a running mean *during* each epoch. So the training curve should be shifted by half an epoch to the left. If you do that, you will see that the training and validation curves overlap almost perfectly at the beginning of training.

When plotting the training curve, it should be shifted by half an epoch to the left.

The training set performance ends up beating the validation performance, as is generally the case when you train for long enough. You can tell that the model has not quite converged yet, as the validation loss is still going down, so you should probably continue training. It's as simple as calling the `fit()` method again, since Keras just continues training where it left off (you should be able to reach close to 89% validation accuracy).

If you are not satisfied with the performance of your model, you should go back and tune the hyperparameters. The first one to check is the learning rate. If that doesn't help, try another optimizer (and always retune the learning rate after changing any hyperparameter). If the performance is still not great, then try tuning model hyperparameters such as the number of layers, the number of neurons per layer, and the types of activation functions to use for each hidden layer. You can also try tuning other hyperparameters, such as the batch size (it can be set in the `fit()` method using the `batch_size` argument, which defaults to 32). We will get back to hyperparameter tuning at the end of this chapter. Once you are satisfied with your model's validation accuracy, you should evaluate it on the test set to estimate the generalization error before you deploy the model to production. You can easily do this using the `evaluate()` method (it also supports several other arguments, such as `batch_size` and `sample_weight`; please check the documentation for more details):

```
>>> model.evaluate(X_test, y_test)
10000/10000 [==========] - 0s 29us/sample - loss: 0.3340 - accuracy: 0.8851
[0.3339798209667206, 0.8851]
```

As we saw in Chapter 2, it is common to get slightly lower performance on the test set than on the validation set, because the hyperparameters are tuned on the validation set, not the test set (however, in this example, we did not do any hyperparameter tuning, so the lower accuracy is just bad luck). Remember to resist the temptation to tweak the hyperparameters on the test set, or else your estimate of the generalization error will be too optimistic.

Using the model to make predictions

Next, we can use the model's `predict()` method to make predictions on new instances. Since we don't have actual new instances, we will just use the first three instances of the test set:

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.03, 0.  , 0.01, 0.  , 0.96],
       [0.  , 0.  , 0.98, 0.  , 0.02, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

As you can see, for each instance the model estimates one probability per class, from class 0 to class 9. For example, for the first image it estimates that the probability of class 9 (ankle boot) is 96%, the probability of class 5 (sandal) is 3%, the probability of class 7 (sneaker) is 1%, and the probabilities of the other classes are negligible. In other words, it "believes" the first image is footwear, most likely ankle boots but possibly sandals or sneakers. If you only care about the class with the highest estimated probability (even if that probability is quite low), then you can use the `predict_classes()` method instead:

```
>>> y_pred = model.predict_classes(X_new)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

Here, the classifier actually classified all three images correctly (these images are shown in Figure 10-13):

```
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1])
```

Figure 10-13. Correctly classified Fashion MNIST images

Now you know how to use the Sequential API to build, train, evaluate, and use a classification MLP. But what about regression?

Building a Regression MLP Using the Sequential API

Let's switch to the California housing problem and tackle it using a regression neural network. For simplicity, we will use Scikit-Learn's `fetch_california_housing()` function to load the data. This dataset is simpler than the one we used in Chapter 2, since it contains only numerical features (there is no `ocean_proximity` feature), and there is no missing value. After loading the data, we split it into a training set, a validation set, and a test set, and we scale all the features:

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

```
housing = fetch_california_housing()

X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

Using the Sequential API to build, train, evaluate, and use a regression MLP to make predictions is quite similar to what we did for classification. The main differences are the fact that the output layer has a single neuron (since we only want to predict a single value) and uses no activation function, and the loss function is the mean squared error. Since the dataset is quite noisy, we just use a single hidden layer with fewer neurons than before, to avoid overfitting:

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])
model.compile(loss="mean_squared_error", optimizer="sgd")
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3] # pretend these are new instances
y_pred = model.predict(X_new)
```

As you can see, the Sequential API is quite easy to use. However, although `Sequential` models are extremely common, it is sometimes useful to build neural networks with more complex topologies, or with multiple inputs or outputs. For this purpose, Keras offers the Functional API.

Building Complex Models Using the Functional API

One example of a nonsequential neural network is a *Wide & Deep* neural network. This neural network architecture was introduced in a 2016 paper by Heng-Tze Cheng et al.[16] It connects all or part of the inputs directly to the output layer, as shown in Figure 10-14. This architecture makes it possible for the neural network to learn both deep patterns (using the deep path) and simple rules (through the short path).[17] In contrast, a regular MLP forces all the data to flow through the full stack of layers;

16 Heng-Tze Cheng et al., "Wide & Deep Learning for Recommender Systems," *Proceedings of the First Workshop on Deep Learning for Recommender Systems* (2016): 7–10.

17 The short path can also be used to provide manually engineered features to the neural network.

thus, simple patterns in the data may end up being distorted by this sequence of transformations.

Figure 10-14. Wide & Deep neural network

Let's build such a neural network to tackle the California housing problem:

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.Concatenate()([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.Model(inputs=[input_], outputs=[output])
```

Let's go through each line of this code:

- First, we need to create an `Input` object.[18] This is a specification of the kind of input the model will get, including its `shape` and `dtype`. A model may actually have multiple inputs, as we will see shortly.
- Next, we create a `Dense` layer with 30 neurons, using the ReLU activation function. As soon as it is created, notice that we call it like a function, passing it the input. This is why this is called the Functional API. Note that we are just telling Keras how it should connect the layers together; no actual data is being processed yet.
- We then create a second hidden layer, and again we use it as a function. Note that we pass it the output of the first hidden layer.

18 The name `input_` is used to avoid overshadowing Python's built-in `input()` function.

- Next, we create a `Concatenate` layer, and once again we immediately use it like a function, to concatenate the input and the output of the second hidden layer. You may prefer the `keras.layers.concatenate()` function, which creates a `Concatenate` layer and immediately calls it with the given inputs.
- Then we create the output layer, with a single neuron and no activation function, and we call it like a function, passing it the result of the concatenation.
- Lastly, we create a Keras `Model`, specifying which inputs and outputs to use.

Once you have built the Keras model, everything is exactly like earlier, so there's no need to repeat it here: you must compile the model, train it, evaluate it, and use it to make predictions.

But what if you want to send a subset of the features through the wide path and a different subset (possibly overlapping) through the deep path (see Figure 10-15)? In this case, one solution is to use multiple inputs. For example, suppose we want to send five features through the wide path (features 0 to 4), and six features through the deep path (features 2 to 7):

```
input_A = keras.layers.Input(shape=[5], name="wide_input")
input_B = keras.layers.Input(shape=[6], name="deep_input")
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name="output")(concat)
model = keras.Model(inputs=[input_A, input_B], outputs=[output])
```

Figure 10-15. Handling multiple inputs

The code is self-explanatory. You should name at least the most important layers, especially when the model gets a bit complex like this. Note that we specified `inputs=[input_A, input_B]` when creating the model. Now we can compile the model as usual, but when we call the `fit()` method, instead of passing a single input matrix `X_train`, we must pass a pair of matrices `(X_train_A, X_train_B)`: one per input.[19] The same is true for `X_valid`, and also for `X_test` and `X_new` when you call `evaluate()` or `predict()`:

```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))

X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]

history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
                    validation_data=((X_valid_A, X_valid_B), y_valid))
mse_test = model.evaluate((X_test_A, X_test_B), y_test)
y_pred = model.predict((X_new_A, X_new_B))
```

There are many use cases in which you may want to have multiple outputs:

- The task may demand it. For instance, you may want to locate and classify the main object in a picture. This is both a regression task (finding the coordinates of the object's center, as well as its width and height) and a classification task.
- Similarly, you may have multiple independent tasks based on the same data. Sure, you could train one neural network per task, but in many cases you will get better results on all tasks by training a single neural network with one output per task. This is because the neural network can learn features in the data that are useful across tasks. For example, you could perform *multitask classification* on pictures of faces, using one output to classify the person's facial expression (smiling, surprised, etc.) and another output to identify whether they are wearing glasses or not.
- Another use case is as a regularization technique (i.e., a training constraint whose objective is to reduce overfitting and thus improve the model's ability to generalize). For example, you may want to add some auxiliary outputs in a neural network architecture (see Figure 10-16) to ensure that the underlying part of the network learns something useful on its own, without relying on the rest of the network.

19 Alternatively, you can pass a dictionary mapping the input names to the input values, like `{"wide_input": X_train_A, "deep_input": X_train_B}`. This is especially useful when there are many inputs, to avoid getting the order wrong.

Figure 10-16. Handling multiple outputs, in this example to add an auxiliary output for regularization

Adding extra outputs is quite easy: just connect them to the appropriate layers and add them to your model's list of outputs. For example, the following code builds the network represented in Figure 10-16:

```
[...] # Same as above, up to the main output layer
output = keras.layers.Dense(1, name="main_output")(concat)
aux_output = keras.layers.Dense(1, name="aux_output")(hidden2)
model = keras.Model(inputs=[input_A, input_B], outputs=[output, aux_output])
```

Each output will need its own loss function. Therefore, when we compile the model, we should pass a list of losses[20] (if we pass a single loss, Keras will assume that the same loss must be used for all outputs). By default, Keras will compute all these losses and simply add them up to get the final loss used for training. We care much more about the main output than about the auxiliary output (as it is just used for regularization), so we want to give the main output's loss a much greater weight. Fortunately, it is possible to set all the loss weights when compiling the model:

```
model.compile(loss=["mse", "mse"], loss_weights=[0.9, 0.1], optimizer="sgd")
```

Now when we train the model, we need to provide labels for each output. In this example, the main output and the auxiliary output should try to predict the same thing, so they should use the same labels. So instead of passing `y_train`, we need to pass `(y_train, y_train)` (and the same goes for `y_valid` and `y_test`):

```
history = model.fit(
    [X_train_A, X_train_B], [y_train, y_train], epochs=20,
    validation_data=([X_valid_A, X_valid_B], [y_valid, y_valid]))
```

20 Alternatively, you can pass a dictionary that maps each output name to the corresponding loss. Just like for the inputs, this is useful when there are multiple outputs, to avoid getting the order wrong. The loss weights and metrics (discussed shortly) can also be set using dictionaries.

When we evaluate the model, Keras will return the total loss, as well as all the individual losses:

```
total_loss, main_loss, aux_loss = model.evaluate(
    [X_test_A, X_test_B], [y_test, y_test])
```

Similarly, the `predict()` method will return predictions for each output:

```
y_pred_main, y_pred_aux = model.predict([X_new_A, X_new_B])
```

As you can see, you can build any sort of architecture you want quite easily with the Functional API. Let's look at one last way you can build Keras models.

Using the Subclassing API to Build Dynamic Models

Both the Sequential API and the Functional API are declarative: you start by declaring which layers you want to use and how they should be connected, and only then can you start feeding the model some data for training or inference. This has many advantages: the model can easily be saved, cloned, and shared; its structure can be displayed and analyzed; the framework can infer shapes and check types, so errors can be caught early (i.e., before any data ever goes through the model). It's also fairly easy to debug, since the whole model is a static graph of layers. But the flip side is just that: it's static. Some models involve loops, varying shapes, conditional branching, and other dynamic behaviors. For such cases, or simply if you prefer a more imperative programming style, the Subclassing API is for you.

Simply subclass the `Model` class, create the layers you need in the constructor, and use them to perform the computations you want in the `call()` method. For example, creating an instance of the following `WideAndDeepModel` class gives us an equivalent model to the one we just built with the Functional API. You can then compile it, evaluate it, and use it to make predictions, exactly like we just did:

```
class WideAndDeepModel(keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs) # handles standard args (e.g., name)
        self.hidden1 = keras.layers.Dense(units, activation=activation)
        self.hidden2 = keras.layers.Dense(units, activation=activation)
        self.main_output = keras.layers.Dense(1)
        self.aux_output = keras.layers.Dense(1)

    def call(self, inputs):
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output

model = WideAndDeepModel()
```

This example looks very much like the Functional API, except we do not need to create the inputs; we just use the `input` argument to the `call()` method, and we separate the creation of the layers[21] in the constructor from their usage in the `call()` method. The big difference is that you can do pretty much anything you want in the `call()` method: `for` loops, `if` statements, low-level TensorFlow operations—your imagination is the limit (see Chapter 12)! This makes it a great API for researchers experimenting with new ideas.

This extra flexibility does come at a cost: your model's architecture is hidden within the `call()` method, so Keras cannot easily inspect it; it cannot save or clone it; and when you call the `summary()` method, you only get a list of layers, without any information on how they are connected to each other. Moreover, Keras cannot check types and shapes ahead of time, and it is easier to make mistakes. So unless you really need that extra flexibility, you should probably stick to the Sequential API or the Functional API.

Keras models can be used just like regular layers, so you can easily combine them to build complex architectures.

Now that you know how to build and train neural nets using Keras, you will want to save them!

Saving and Restoring a Model

When using the Sequential API or the Functional API, saving a trained Keras model is as simple as it gets:

```
model = keras.models.Sequential([...]) # or keras.Model([...])
model.compile([...])
model.fit([...])
model.save("my_keras_model.h5")
```

Keras will use the HDF5 format to save both the model's architecture (including every layer's hyperparameters) and the values of all the model parameters for every layer (e.g., connection weights and biases). It also saves the optimizer (including its hyperparameters and any state it may have). In Chapter 19, we will see how to save a tf.keras model using TensorFlow's SavedModel format instead.

21 Keras models have an `output` attribute, so we cannot use that name for the main output layer, which is why we renamed it to `main_output`.

You will typically have a script that trains a model and saves it, and one or more scripts (or web services) that load the model and use it to make predictions. Loading the model is just as easy:

```
model = keras.models.load_model("my_keras_model.h5")
```

This will work when using the Sequential API or the Functional API, but unfortunately not when using model subclassing. You can use `save_weights()` and `load_weights()` to at least save and restore the model parameters, but you will need to save and restore everything else yourself.

But what if training lasts several hours? This is quite common, especially when training on large datasets. In this case, you should not only save your model at the end of training, but also save checkpoints at regular intervals during training, to avoid losing everything if your computer crashes. But how can you tell the `fit()` method to save checkpoints? Use callbacks.

Using Callbacks

The `fit()` method accepts a `callbacks` argument that lets you specify a list of objects that Keras will call at the start and end of training, at the start and end of each epoch, and even before and after processing each batch. For example, the `ModelCheckpoint` callback saves checkpoints of your model at regular intervals during training, by default at the end of each epoch:

```
[...] # build and compile the model
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5")
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

Moreover, if you use a validation set during training, you can set `save_best_only=True` when creating the `ModelCheckpoint`. In this case, it will only save your model when its performance on the validation set is the best so far. This way, you do not need to worry about training for too long and overfitting the training set: simply restore the last model saved after training, and this will be the best model on the validation set. The following code is a simple way to implement early stopping (introduced in Chapter 4):

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb])
model = keras.models.load_model("my_keras_model.h5") # roll back to best model
```

Another way to implement early stopping is to simply use the `EarlyStopping` callback. It will interrupt training when it measures no progress on the validation set for

a number of epochs (defined by the `patience` argument), and it will optionally roll back to the best model. You can combine both callbacks to save checkpoints of your model (in case your computer crashes) and interrupt training early when there is no more progress (to avoid wasting time and resources):

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                  restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb])
```

The number of epochs can be set to a large value since training will stop automatically when there is no more progress. In this case, there is no need to restore the best model saved because the `EarlyStopping` callback will keep track of the best weights and restore them for you at the end of training.

There are many other callbacks available in the `keras.callbacks` package.

If you need extra control, you can easily write your own custom callbacks. As an example of how to do that, the following custom callback will display the ratio between the validation loss and the training loss during training (e.g., to detect overfitting):

```
class PrintValTrainRatioCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print("\nval/train: {:.2f}".format(logs["val_loss"] / logs["loss"]))
```

As you might expect, you can implement `on_train_begin()`, `on_train_end()`, `on_epoch_begin()`, `on_epoch_end()`, `on_batch_begin()`, and `on_batch_end()`. Callbacks can also be used during evaluation and predictions, should you ever need them (e.g., for debugging). For evaluation, you should implement `on_test_begin()`, `on_test_end()`, `on_test_batch_begin()`, or `on_test_batch_end()` (called by `evaluate()`), and for prediction you should implement `on_predict_begin()`, `on_predict_end()`, `on_predict_batch_begin()`, or `on_predict_batch_end()` (called by `predict()`).

Now let's take a look at one more tool you should definitely have in your toolbox when using tf.keras: TensorBoard.

Using TensorBoard for Visualization

TensorBoard is a great interactive visualization tool that you can use to view the learning curves during training, compare learning curves between multiple runs, visualize the computation graph, analyze training statistics, view images generated by your model, visualize complex multidimensional data projected down to 3D and automatically clustered for you, and more! This tool is installed automatically when you install TensorFlow, so you already have it.

To use it, you must modify your program so that it outputs the data you want to visualize to special binary log files called *event files*. Each binary data record is called a *summary*. The TensorBoard server will monitor the log directory, and it will automatically pick up the changes and update the visualizations: this allows you to visualize live data (with a short delay), such as the learning curves during training. In general, you want to point the TensorBoard server to a root log directory and configure your program so that it writes to a different subdirectory every time it runs. This way, the same TensorBoard server instance will allow you to visualize and compare data from multiple runs of your program, without getting everything mixed up.

Let's start by defining the root log directory we will use for our TensorBoard logs, plus a small function that will generate a subdirectory path based on the current date and time so that it's different at every run. You may want to include extra information in the log directory name, such as hyperparameter values that you are testing, to make it easier to know what you are looking at in TensorBoard:

```
import os
root_logdir = os.path.join(os.curdir, "my_logs")

def get_run_logdir():
    import time
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir() # e.g., './my_logs/run_2019_06_07-15_15_22'
```

The good news is that Keras provides a nice `TensorBoard()` callback:

```
[...] # Build and compile your model
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid),
                    callbacks=[tensorboard_cb])
```

And that's all there is to it! It could hardly be easier to use. If you run this code, the `TensorBoard()` callback will take care of creating the log directory for you (along with its parent directories if needed), and during training it will create event files and write summaries to them. After running the program a second time (perhaps

changing some hyperparameter value), you will end up with a directory structure similar to this one:

```
my_logs/
├── run_2019_06_07-15_15_22
│   ├── train
│   │   ├── events.out.tfevents.1559891732.mycomputer.local.38511.694049.v2
│   │   ├── events.out.tfevents.1559891732.mycomputer.local.profile-empty
│   │   └── plugins/profile/2019-06-07_15-15-32
│   │       └── local.trace
│   └── validation
│       └── events.out.tfevents.1559891733.mycomputer.local.38511.696430.v2
└── run_2019_06_07-15_15_49
    └── [...]
```

There's one directory per run, each containing one subdirectory for training logs and one for validation logs. Both contain event files, but the training logs also include profiling traces: this allows TensorBoard to show you exactly how much time the model spent on each part of your model, across all your devices, which is great for locating performance bottlenecks.

Next you need to start the TensorBoard server. One way to do this is by running a command in a terminal. If you installed TensorFlow within a virtualenv, you should activate it. Next, run the following command at the root of the project (or from anywhere else, as long as you point to the appropriate log directory):

```
$ tensorboard --logdir=./my_logs --port=6006
TensorBoard 2.0.0 at http://mycomputer.local:6006/ (Press CTRL+C to quit)
```

If your shell cannot find the *tensorboard* script, then you must update your `PATH` environment variable so that it contains the directory in which the script was installed (alternatively, you can just replace `tensorboard` in the command line with `python3 -m tensorboard.main`). Once the server is up, you can open a web browser and go to *http://localhost:6006*.

Alternatively, you can use TensorBoard directly within Jupyter, by running the following commands. The first line loads the TensorBoard extension, and the second line starts a TensorBoard server on port 6006 (unless it is already started) and connects to it:

```
%load_ext tensorboard
%tensorboard --logdir=./my_logs --port=6006
```

Either way, you should see TensorBoard's web interface. Click the SCALARS tab to view the learning curves (see Figure 10-17). At the bottom left, select the logs you want to visualize (e.g., the training logs from the first and second run), and click the `epoch_loss` scalar. Notice that the training loss went down nicely during both runs, but the second run went down much faster. Indeed, we used a learning rate of 0.05 (`optimizer=keras.optimizers.SGD(lr=0.05)`) instead of 0.001.

Figure 10-17. Visualizing learning curves with TensorBoard

You can also visualize the whole graph, the learned weights (projected to 3D), or the profiling traces. The `TensorBoard()` callback has options to log extra data too, such as embeddings (see Chapter 13).

Additionally, TensorFlow offers a lower-level API in the `tf.summary` package. The following code creates a `SummaryWriter` using the `create_file_writer()` function, and it uses this writer as a context to log scalars, histograms, images, audio, and text, all of which can then be visualized using TensorBoard (give it a try!):

```
test_logdir = get_run_logdir()
writer = tf.summary.create_file_writer(test_logdir)
with writer.as_default():
    for step in range(1, 1000 + 1):
        tf.summary.scalar("my_scalar", np.sin(step / 10), step=step)
        data = (np.random.randn(100) + 2) * step / 100 # some random data
        tf.summary.histogram("my_hist", data, buckets=50, step=step)
        images = np.random.rand(2, 32, 32, 3) # random 32×32 RGB images
        tf.summary.image("my_images", images * step / 1000, step=step)
        texts = ["The step is " + str(step), "Its square is " + str(step**2)]
        tf.summary.text("my_text", texts, step=step)
        sine_wave = tf.math.sin(tf.range(12000) / 48000 * 2 * np.pi * step)
        audio = tf.reshape(tf.cast(sine_wave, tf.float32), [1, -1, 1])
        tf.summary.audio("my_audio", audio, sample_rate=48000, step=step)
```

This is actually a useful visualization tool to have, even beyond TensorFlow or Deep Learning.

Let's summarize what you've learned so far in this chapter: we saw where neural nets came from, what an MLP is and how you can use it for classification and regression, how to use tf.keras's Sequential API to build MLPs, and how to use the Functional API or the Subclassing API to build more complex model architectures. You learned how to save and restore a model and how to use callbacks for checkpointing, early stopping, and more. Finally, you learned how to use TensorBoard for visualization. You can already go ahead and use neural networks to tackle many problems! However, you may wonder how to choose the number of hidden layers, the number of neurons in the network, and all the other hyperparameters. Let's look at this now.

Fine-Tuning Neural Network Hyperparameters

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Not only can you use any imaginable network architecture, but even in a simple MLP you can change the number of layers, the number of neurons per layer, the type of activation function to use in each layer, the weight initialization logic, and much more. How do you know what combination of hyperparameters is the best for your task?

One option is to simply try many combinations of hyperparameters and see which one works best on the validation set (or use K-fold cross-validation). For example, we can use `GridSearchCV` or `RandomizedSearchCV` to explore the hyperparameter space, as we did in Chapter 2. To do this, we need to wrap our Keras models in objects that mimic regular Scikit-Learn regressors. The first step is to create a function that will build and compile a Keras model, given a set of hyperparameters:

```
def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3, input_shape=[8]):
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons, activation="relu"))
    model.add(keras.layers.Dense(1))
    optimizer = keras.optimizers.SGD(lr=learning_rate)
    model.compile(loss="mse", optimizer=optimizer)
    return model
```

This function creates a simple `Sequential` model for univariate regression (only one output neuron), with the given input shape and the given number of hidden layers and neurons, and it compiles it using an `SGD` optimizer configured with the specified learning rate. It is good practice to provide reasonable defaults to as many hyperparameters as you can, as Scikit-Learn does.

Next, let's create a `KerasRegressor` based on this `build_model()` function:

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

The `KerasRegressor` object is a thin wrapper around the Keras model built using `build_model()`. Since we did not specify any hyperparameters when creating it, it will use the default hyperparameters we defined in `build_model()`. Now we can use this object like a regular Scikit-Learn regressor: we can train it using its `fit()` method, then evaluate it using its `score()` method, and use it to make predictions using its `predict()` method, as you can see in the following code:

```
keras_reg.fit(X_train, y_train, epochs=100,
              validation_data=(X_valid, y_valid),
              callbacks=[keras.callbacks.EarlyStopping(patience=10)])
mse_test = keras_reg.score(X_test, y_test)
y_pred = keras_reg.predict(X_new)
```

Note that any extra parameter you pass to the `fit()` method will get passed to the underlying Keras model. Also note that the score will be the opposite of the MSE because Scikit-Learn wants scores, not losses (i.e., higher should be better).

We don't want to train and evaluate a single model like this, though we want to train hundreds of variants and see which one performs best on the validation set. Since there are many hyperparameters, it is preferable to use a randomized search rather than grid search (as we discussed in Chapter 2). Let's try to explore the number of hidden layers, the number of neurons, and the learning rate:

```
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

param_distribs = {
    "n_hidden": [0, 1, 2, 3],
    "n_neurons": np.arange(1, 100),
    "learning_rate": reciprocal(3e-4, 3e-2),
}

rnd_search_cv = RandomizedSearchCV(keras_reg, param_distribs, n_iter=10, cv=3)
rnd_search_cv.fit(X_train, y_train, epochs=100,
                  validation_data=(X_valid, y_valid),
                  callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

This is identical to what we did in Chapter 2, except here we pass extra parameters to the `fit()` method, and they get relayed to the underlying Keras models. Note that `RandomizedSearchCV` uses K-fold cross-validation, so it does not use `X_valid` and `y_valid`, which are only used for early stopping.

The exploration may last many hours, depending on the hardware, the size of the dataset, the complexity of the model, and the values of `n_iter` and `cv`. When it's over, you can access the best parameters found, the best score, and the trained Keras model like this:

```
>>> rnd_search_cv.best_params_
{'learning_rate': 0.0033625641252688094, 'n_hidden': 2, 'n_neurons': 42}
>>> rnd_search_cv.best_score_
-0.3189529188278931
>>> model = rnd_search_cv.best_estimator_.model
```

You can now save this model, evaluate it on the test set, and, if you are satisfied with its performance, deploy it to production. Using randomized search is not too hard, and it works well for many fairly simple problems. When training is slow, however (e.g., for more complex problems with larger datasets), this approach will only explore a tiny portion of the hyperparameter space. You can partially alleviate this problem by assisting the search process manually: first run a quick random search using wide ranges of hyperparameter values, then run another search using smaller ranges of values centered on the best ones found during the first run, and so on. This approach will hopefully zoom in on a good set of hyperparameters. However, it's very time consuming, and probably not the best use of your time.

Fortunately, there are many techniques to explore a search space much more efficiently than randomly. Their core idea is simple: when a region of the space turns out to be good, it should be explored more. Such techniques take care of the "zooming" process for you and lead to much better solutions in much less time. Here are some Python libraries you can use to optimize hyperparameters:

Hyperopt
: A popular library for optimizing over all sorts of complex search spaces (including real values, such as the learning rate, and discrete values, such as the number of layers).

Hyperas, kopt, or Talos
: Useful libraries for optimizing hyperparameters for Keras models (the first two are based on Hyperopt).

Keras Tuner
: An easy-to-use hyperparameter optimization library by Google for Keras models, with a hosted service for visualization and analysis.

Scikit-Optimize (`skopt`)
: A general-purpose optimization library. The `BayesSearchCV` class performs Bayesian optimization using an interface similar to `GridSearchCV`.

Spearmint
: A Bayesian optimization library.

Hyperband
: A fast hyperparameter tuning library based on the recent Hyperband paper[22] by Lisha Li et al.

Sklearn-Deap
: A hyperparameter optimization library based on evolutionary algorithms, with a `GridSearchCV`-like interface.

Moreover, many companies offer services for hyperparameter optimization. We'll discuss Google Cloud AI Platform's hyperparameter tuning service in Chapter 19. Other options include services by Arimo and SigOpt, and CallDesk's Oscar.

Hyperparameter tuning is still an active area of research, and evolutionary algorithms are making a comeback. For example, check out DeepMind's excellent 2017 paper,[23] where the authors jointly optimize a population of models and their hyperparameters. Google has also used an evolutionary approach, not just to search for hyperparameters but also to look for the best neural network architecture for the problem; their AutoML suite is already available as a cloud service. Perhaps the days of building neural networks manually will soon be over? Check out Google's post on this topic. In fact, evolutionary algorithms have been used successfully to train individual neural networks, replacing the ubiquitous Gradient Descent! For an example, see the 2017 post by Uber where the authors introduce their *Deep Neuroevolution* technique.

But despite all this exciting progress and all these tools and services, it still helps to have an idea of what values are reasonable for each hyperparameter so that you can build a quick prototype and restrict the search space. The following sections provide guidelines for choosing the number of hidden layers and neurons in an MLP and for selecting good values for some of the main hyperparameters.

Number of Hidden Layers

For many problems, you can begin with a single hidden layer and get reasonable results. An MLP with just one hidden layer can theoretically model even the most complex functions, provided it has enough neurons. But for complex problems, deep networks have a much higher *parameter efficiency* than shallow ones: they can model complex functions using exponentially fewer neurons than shallow nets, allowing them to reach much better performance with the same amount of training data.

To understand why, suppose you are asked to draw a forest using some drawing software, but you are forbidden to copy and paste anything. It would take an enormous

22 Lisha Li et al., "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization," *Journal of Machine Learning Research* 18 (April 2018): 1–52.

23 Max Jaderberg et al., "Population Based Training of Neural Networks," arXiv preprint arXiv:1711.09846 (2017).

amount of time: you would have to draw each tree individually, branch by branch, leaf by leaf. If you could instead draw one leaf, copy and paste it to draw a branch, then copy and paste that branch to create a tree, and finally copy and paste this tree to make a forest, you would be finished in no time. Real-world data is often structured in such a hierarchical way, and deep neural networks automatically take advantage of this fact: lower hidden layers model low-level structures (e.g., line segments of various shapes and orientations), intermediate hidden layers combine these low-level structures to model intermediate-level structures (e.g., squares, circles), and the highest hidden layers and the output layer combine these intermediate structures to model high-level structures (e.g., faces).

Not only does this hierarchical architecture help DNNs converge faster to a good solution, but it also improves their ability to generalize to new datasets. For example, if you have already trained a model to recognize faces in pictures and you now want to train a new neural network to recognize hairstyles, you can kickstart the training by reusing the lower layers of the first network. Instead of randomly initializing the weights and biases of the first few layers of the new neural network, you can initialize them to the values of the weights and biases of the lower layers of the first network. This way the network will not have to learn from scratch all the low-level structures that occur in most pictures; it will only have to learn the higher-level structures (e.g., hairstyles). This is called *transfer learning*.

In summary, for many problems you can start with just one or two hidden layers and the neural network will work just fine. For instance, you can easily reach above 97% accuracy on the MNIST dataset using just one hidden layer with a few hundred neurons, and above 98% accuracy using two hidden layers with the same total number of neurons, in roughly the same amount of training time. For more complex problems, you can ramp up the number of hidden layers until you start overfitting the training set. Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers (or even hundreds, but not fully connected ones, as we will see in Chapter 14), and they need a huge amount of training data. You will rarely have to train such networks from scratch: it is much more common to reuse parts of a pretrained state-of-the-art network that performs a similar task. Training will then be a lot faster and require much less data (we will discuss this in Chapter 11).

Number of Neurons per Hidden Layer

The number of neurons in the input and output layers is determined by the type of input and output your task requires. For example, the MNIST task requires $28 \times 28 = 784$ input neurons and 10 output neurons.

As for the hidden layers, it used to be common to size them to form a pyramid, with fewer and fewer neurons at each layer—the rationale being that many low-level fea-

tures can coalesce into far fewer high-level features. A typical neural network for MNIST might have 3 hidden layers, the first with 300 neurons, the second with 200, and the third with 100. However, this practice has been largely abandoned because it seems that using the same number of neurons in all hidden layers performs just as well in most cases, or even better; plus, there is only one hyperparameter to tune, instead of one per layer. That said, depending on the dataset, it can sometimes help to make the first hidden layer bigger than the others.

Just like the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting. But in practice, it's often simpler and more efficient to pick a model with more layers and neurons than you actually need, then use early stopping and other regularization techniques to prevent it from overfitting. Vincent Vanhoucke, a scientist at Google, has dubbed this the "stretch pants" approach: instead of wasting time looking for pants that perfectly match your size, just use large stretch pants that will shrink down to the right size. With this approach, you avoid bottleneck layers that could ruin your model. On the flip side, if a layer has too few neurons, it will not have enough representational power to preserve all the useful information from the inputs (e.g., a layer with two neurons can only output 2D data, so if it processes 3D data, some information will be lost). No matter how big and powerful the rest of the network is, that information will never be recovered.

In general you will get more bang for your buck by increasing the number of layers instead of the number of neurons per layer.

Learning Rate, Batch Size, and Other Hyperparameters

The numbers of hidden layers and neurons are not the only hyperparameters you can tweak in an MLP. Here are some of the most important ones, as well as tips on how to set them:

Learning rate

The learning rate is arguably the most important hyperparameter. In general, the optimal learning rate is about half of the maximum learning rate (i.e., the learning rate above which the training algorithm diverges, as we saw in Chapter 4). One way to find a good learning rate is to train the model for a few hundred iterations, starting with a very low learning rate (e.g., 10^{-5}) and gradually increasing it up to a very large value (e.g., 10). This is done by multiplying the learning rate by a constant factor at each iteration (e.g., by $\exp(\log(10^6)/500)$ to go from 10^{-5} to 10 in 500 iterations). If you plot the loss as a function of the learning rate (using a log scale for the learning rate), you should see it dropping at first. But after a while, the learning rate will be too large, so the loss will shoot back up: the opti-

mal learning rate will be a bit lower than the point at which the loss starts to climb (typically about 10 times lower than the turning point). You can then reinitialize your model and train it normally using this good learning rate. We will look at more learning rate techniques in Chapter 11.

Optimizer

Choosing a better optimizer than plain old Mini-batch Gradient Descent (and tuning its hyperparameters) is also quite important. We will see several advanced optimizers in Chapter 11.

Batch size

The batch size can have a significant impact on your model's performance and training time. The main benefit of using large batch sizes is that hardware accelerators like GPUs can process them efficiently (see Chapter 19), so the training algorithm will see more instances per second. Therefore, many researchers and practitioners recommend using the largest batch size that can fit in GPU RAM. There's a catch, though: in practice, large batch sizes often lead to training instabilities, especially at the beginning of training, and the resulting model may not generalize as well as a model trained with a small batch size. In April 2018, Yann LeCun even tweeted "Friends don't let friends use mini-batches larger than 32," citing a 2018 paper[24] by Dominic Masters and Carlo Luschi which concluded that using small batches (from 2 to 32) was preferable because small batches led to better models in less training time. Other papers point in the opposite direction, however; in 2017, papers by Elad Hoffer et al.[25] and Priya Goyal et al.[26] showed that it was possible to use very large batch sizes (up to 8,192) using various techniques such as warming up the learning rate (i.e., starting training with a small learning rate, then ramping it up, as we will see in Chapter 11). This led to a very short training time, without any generalization gap. So, one strategy is to try to use a large batch size, using learning rate warmup, and if training is unstable or the final performance is disappointing, then try using a small batch size instead.

Activation function

We discussed how to choose the activation function earlier in this chapter: in general, the ReLU activation function will be a good default for all hidden layers. For the output layer, it really depends on your task.

24 Dominic Masters and Carlo Luschi, "Revisiting Small Batch Training for Deep Neural Networks," arXiv preprint arXiv:1804.07612 (2018).

25 Elad Hoffer et al., "Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks," *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 1729–1739.

26 Priya Goyal et al., "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour," arXiv preprint arXiv: 1706.02677 (2017).

Number of iterations

In most cases, the number of training iterations does not actually need to be tweaked: just use early stopping instead.

The optimal learning rate depends on the other hyperparameters—especially the batch size—so if you modify any hyperparameter, make sure to update the learning rate as well.

For more best practices regarding tuning neural network hyperparameters, check out the excellent 2018 paper[27] by Leslie Smith.

This concludes our introduction to artificial neural networks and their implementation with Keras. In the next few chapters, we will discuss techniques to train very deep nets. We will also explore how to customize models using TensorFlow's lower-level API and how to load and preprocess data efficiently using the Data API. And we will dive into other popular neural network architectures: convolutional neural networks for image processing, recurrent neural networks for sequential data, autoencoders for representation learning, and generative adversarial networks to model and generate data.[28]

Exercises

1. The TensorFlow Playground is a handy neural network simulator built by the TensorFlow team. In this exercise, you will train several binary classifiers in just a few clicks, and tweak the model's architecture and its hyperparameters to gain some intuition on how neural networks work and what their hyperparameters do. Take some time to explore the following:
   a. The patterns learned by a neural net. Try training the default neural network by clicking the Run button (top left). Notice how it quickly finds a good solution for the classification task. The neurons in the first hidden layer have learned simple patterns, while the neurons in the second hidden layer have learned to combine the simple patterns of the first hidden layer into more complex patterns. In general, the more layers there are, the more complex the patterns can be.
   b. Activation functions. Try replacing the tanh activation function with a ReLU activation function, and train the network again. Notice that it finds a solution

27 Leslie N. Smith, "A Disciplined Approach to Neural Network Hyper-Parameters: Part 1—Learning Rate, Batch Size, Momentum, and Weight Decay," arXiv preprint arXiv:1803.09820 (2018).

28 A few extra ANN architectures are presented in Appendix E.

even faster, but this time the boundaries are linear. This is due to the shape of the ReLU function.

c. The risk of local minima. Modify the network architecture to have just one hidden layer with three neurons. Train it multiple times (to reset the network weights, click the Reset button next to the Play button). Notice that the training time varies a lot, and sometimes it even gets stuck in a local minimum.

d. What happens when neural nets are too small. Remove one neuron to keep just two. Notice that the neural network is now incapable of finding a good solution, even if you try multiple times. The model has too few parameters and systematically underfits the training set.

e. What happens when neural nets are large enough. Set the number of neurons to eight, and train the network several times. Notice that it is now consistently fast and never gets stuck. This highlights an important finding in neural network theory: large neural networks almost never get stuck in local minima, and even when they do these local optima are almost as good as the global optimum. However, they can still get stuck on long plateaus for a long time.

f. The risk of vanishing gradients in deep networks. Select the spiral dataset (the bottom-right dataset under "DATA"), and change the network architecture to have four hidden layers with eight neurons each. Notice that training takes much longer and often gets stuck on plateaus for long periods of time. Also notice that the neurons in the highest layers (on the right) tend to evolve faster than the neurons in the lowest layers (on the left). This problem, called the "vanishing gradients" problem, can be alleviated with better weight initialization and other techniques, better optimizers (such as AdaGrad or Adam), or Batch Normalization (discussed in Chapter 11).

g. Go further. Take an hour or so to play around with other parameters and get a feel for what they do, to build an intuitive understanding about neural networks.

2. Draw an ANN using the original artificial neurons (like the ones in Figure 10-3) that computes $A \oplus B$ (where $\oplus$ represents the XOR operation). Hint: $A \oplus B = (A \wedge \neg B \vee (\neg A \wedge B)$.

3. Why is it generally preferable to use a Logistic Regression classifier rather than a classical Perceptron (i.e., a single layer of threshold logic units trained using the Perceptron training algorithm)? How can you tweak a Perceptron to make it equivalent to a Logistic Regression classifier?

4. Why was the logistic activation function a key ingredient in training the first MLPs?

5. Name three popular activation functions. Can you draw them?

6. Suppose you have an MLP composed of one input layer with 10 passthrough neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons. All artificial neurons use the ReLU activation function.

   - What is the shape of the input matrix $\mathbf{X}$?
   - What are the shapes of the hidden layer's weight vector $\mathbf{W}_h$ and its bias vector $\mathbf{b}_h$?
   - What are the shapes of the output layer's weight vector $\mathbf{W}_o$ and its bias vector $\mathbf{b}_o$?
   - What is the shape of the network's output matrix $\mathbf{Y}$?
   - Write the equation that computes the network's output matrix $\mathbf{Y}$ as a function of $\mathbf{X}$, $\mathbf{W}_h$, $\mathbf{b}_h$, $\mathbf{W}_o$, and $\mathbf{b}_o$.

7. How many neurons do you need in the output layer if you want to classify email into spam or ham? What activation function should you use in the output layer? If instead you want to tackle MNIST, how many neurons do you need in the output layer, and which activation function should you use? What about for getting your network to predict housing prices, as in Chapter 2?
8. What is backpropagation and how does it work? What is the difference between backpropagation and reverse-mode autodiff?
9. Can you list all the hyperparameters you can tweak in a basic MLP? If the MLP overfits the training data, how could you tweak these hyperparameters to try to solve the problem?
10. Train a deep MLP on the MNIST dataset (you can load it using `keras.datasets.mnist.load_data()`. See if you can get over 98% precision. Try searching for the optimal learning rate by using the approach presented in this chapter (i.e., by growing the learning rate exponentially, plotting the loss, and finding the point where the loss shoots up). Try adding all the bells and whistles—save checkpoints, use early stopping, and plot learning curves using TensorBoard.

Solutions to these exercises are available in Appendix A.

CHAPTER 11
Training Deep Neural Networks

In Chapter 10 we introduced artificial neural networks and trained our first deep neural networks. But they were shallow nets, with just a few hidden layers. What if you need to tackle a complex problem, such as detecting hundreds of types of objects in high-resolution images? You may need to train a much deeper DNN, perhaps with 10 layers or many more, each containing hundreds of neurons, linked by hundreds of thousands of connections. Training a deep DNN isn't a walk in the park. Here are some of the problems you could run into:

- You may be faced with the tricky *vanishing gradients* problem or the related *exploding gradients* problem. This is when the gradients grow smaller and smaller, or larger and larger, when flowing backward through the DNN during training. Both of these problems make lower layers very hard to train.
- You might not have enough training data for such a large network, or it might be too costly to label.
- Training may be extremely slow.
- A model with millions of parameters would severely risk overfitting the training set, especially if there are not enough training instances or if they are too noisy.

In this chapter we will go through each of these problems and present techniques to solve them. We will start by exploring the vanishing and exploding gradients problems and some of their most popular solutions. Next, we will look at transfer learning and unsupervised pretraining, which can help you tackle complex tasks even when you have little labeled data. Then we will discuss various optimizers that can speed up training large models tremendously. Finally, we will go through a few popular regularization techniques for large neural networks.

With these tools, you will be able to train very deep nets. Welcome to Deep Learning!

The Vanishing/Exploding Gradients Problems

As we discussed in Chapter 10, the backpropagation algorithm works by going from the output layer to the input layer, propagating the error gradient along the way. Once the algorithm has computed the gradient of the cost function with regard to each parameter in the network, it uses these gradients to update each parameter with a Gradient Descent step.

Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the Gradient Descent update leaves the lower layers' connection weights virtually unchanged, and training never converges to a good solution. We call this the *vanishing gradients* problem. In some cases, the opposite can happen: the gradients can grow bigger and bigger until layers get insanely large weight updates and the algorithm diverges. This is the *exploding gradients* problem, which surfaces in recurrent neural networks (see Chapter 15). More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

This unfortunate behavior was empirically observed long ago, and it was one of the reasons deep neural networks were mostly abandoned in the early 2000s. It wasn't clear what caused the gradients to be so unstable when training a DNN, but some light was shed in a 2010 paper by Xavier Glorot and Yoshua Bengio.[1] The authors found a few suspects, including the combination of the popular logistic sigmoid activation function and the weight initialization technique that was most popular at the time (i.e., a normal distribution with a mean of 0 and a standard deviation of 1). In short, they showed that with this activation function and this initialization scheme, the variance of the outputs of each layer is much greater than the variance of its inputs. Going forward in the network, the variance keeps increasing after each layer until the activation function saturates at the top layers. This saturation is actually made worse by the fact that the logistic function has a mean of 0.5, not 0 (the hyperbolic tangent function has a mean of 0 and behaves slightly better than the logistic function in deep networks).

Looking at the logistic activation function (see Figure 11-1), you can see that when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0. Thus, when backpropagation kicks in it has virtually no gradient to propagate back through the network; and what little gradient exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.

1 Xavier Glorot and Yoshua Bengio, "Understanding the Difficulty of Training Deep Feedforward Neural Networks," *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (2010): 249–256.

Figure 11-1. Logistic activation function saturation

Glorot and He Initialization

In their paper, Glorot and Bengio propose a way to significantly alleviate the unstable gradients problem. They point out that we need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients. We don't want the signal to die out, nor do we want it to explode and saturate. For the signal to flow properly, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs,[2] and we need the gradients to have equal variance before and after flowing through a layer in the reverse direction (please check out the paper if you are interested in the mathematical details). It is actually not possible to guarantee both unless the layer has an equal number of inputs and neurons (these numbers are called the *fan-in* and *fan-out* of the layer), but Glorot and Bengio proposed a good compromise that has proven to work very well in practice: the connection weights of each layer must be initialized randomly as described in Equation 11-1, where $fan_{\text{avg}} = (fan_{\text{in}} + fan_{\text{out}})/2$. This initialization strategy is called *Xavier initialization* or *Glorot initialization*, after the paper's first author.

2 Here's an analogy: if you set a microphone amplifier's knob too close to zero, people won't hear your voice, but if you set it too close to the max, your voice will be saturated and people won't understand what you are saying. Now imagine a chain of such amplifiers: they all need to be set properly in order for your voice to come out loud and clear at the end of the chain. Your voice has to come out of each amplifier at the same amplitude as it came in.

Equation 11-1. Glorot initialization (when using the logistic activation function)

Normal distribution with mean 0 and variance $\sigma^2 = \frac{1}{fan_{\text{avg}}}$

Or a uniform distribution between $-r$ and $+r$, with $r = \sqrt{\frac{3}{fan_{\text{avg}}}}$

If you replace fan_{avg} with fan_{in} in Equation 11-1, you get an initialization strategy that Yann LeCun proposed in the 1990s. He called it *LeCun initialization*. Genevieve Orr and Klaus-Robert Müller even recommended it in their 1998 book *Neural Networks: Tricks of the Trade* (Springer). LeCun initialization is equivalent to Glorot initialization when $fan_{\text{in}} = fan_{\text{out}}$. It took over a decade for researchers to realize how important this trick is. Using Glorot initialization can speed up training considerably, and it is one of the tricks that led to the success of Deep Learning.

Some papers[3] have provided similar strategies for different activation functions. These strategies differ only by the scale of the variance and whether they use fan_{avg} or fan_{in}, as shown in Table 11-1 (for the uniform distribution, just compute $r = \sqrt{3\sigma^2}$). The initialization strategy for the ReLU activation function (and its variants, including the ELU activation described shortly) is sometimes called *He initialization*, after the paper's first author. The SELU activation function will be explained later in this chapter. It should be used with LeCun initialization (preferably with a normal distribution, as we will see).

Table 11-1. Initialization parameters for each type of activation function

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, logistic, softmax	$1 / fan_{\text{avg}}$
He	ReLU and variants	$2 / fan_{\text{in}}$
LeCun	SELU	$1 / fan_{\text{in}}$

By default, Keras uses Glorot initialization with a uniform distribution. When creating a layer, you can change this to He initialization by setting `kernel_initializer="he_uniform"` or `kernel_initializer="he_normal"` like this:

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

If you want He initialization with a uniform distribution but based on fan_{avg} rather than fan_{in}, you can use the `VarianceScaling` initializer like this:

3 E.g., Kaiming He et al., "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," *Proceedings of the 2015 IEEE International Conference on Computer Vision* (2015): 1026–1034.

```
he_avg_init = keras.initializers.VarianceScaling(scale=2., mode='fan_avg',
                                                 distribution='uniform')
keras.layers.Dense(10, activation="sigmoid", kernel_initializer=he_avg_init)
```

Nonsaturating Activation Functions

One of the insights in the 2010 paper by Glorot and Bengio was that the problems with unstable gradients were in part due to a poor choice of activation function. Until then most people had assumed that if Mother Nature had chosen to use roughly sigmoid activation functions in biological neurons, they must be an excellent choice. But it turns out that other activation functions behave much better in deep neural networks—in particular, the ReLU activation function, mostly because it does not saturate for positive values (and because it is fast to compute).

Unfortunately, the ReLU activation function is not perfect. It suffers from a problem known as the *dying ReLUs*: during training, some neurons effectively "die," meaning they stop outputting anything other than 0. In some cases, you may find that half of your network's neurons are dead, especially if you used a large learning rate. A neuron dies when its weights get tweaked in such a way that the weighted sum of its inputs are negative for all instances in the training set. When this happens, it just keeps outputting zeros, and Gradient Descent does not affect it anymore because the gradient of the ReLU function is zero when its input is negative.[4]

To solve this problem, you may want to use a variant of the ReLU function, such as the *leaky ReLU*. This function is defined as $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$ (see Figure 11-2). The hyperparameter α defines how much the function "leaks": it is the slope of the function for $z < 0$ and is typically set to 0.01. This small slope ensures that leaky ReLUs never die; they can go into a long coma, but they have a chance to eventually wake up. A 2015 paper[5] compared several variants of the ReLU activation function, and one of its conclusions was that the leaky variants always outperformed the strict ReLU activation function. In fact, setting $\alpha = 0.2$ (a huge leak) seemed to result in better performance than $\alpha = 0.01$ (a small leak). The paper also evaluated the *randomized leaky ReLU* (RReLU), where α is picked randomly in a given range during training and is fixed to an average value during testing. RReLU also performed fairly well and seemed to act as a regularizer (reducing the risk of overfitting the training set). Finally, the paper evaluated the *parametric leaky ReLU* (PReLU), where α is authorized to be learned during training (instead of being a hyperparameter, it becomes a parameter that can be modified by backpropagation like any other param-

4 Unless it is part of the first hidden layer, a dead neuron may sometimes come back to life: Gradient Descent may indeed tweak neurons in the layers below in such a way that the weighted sum of the dead neuron's inputs is positive again.

5 Bing Xu et al., "Empirical Evaluation of Rectified Activations in Convolutional Network," arXiv preprint arXiv:1505.00853 (2015).

eter). PReLU was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

Figure 11-2. Leaky ReLU: like ReLU, but with a small slope for negative values

Last but not least, a 2015 paper by Djork-Arné Clevert et al.[6] proposed a new activation function called the *exponential linear unit* (ELU) that outperformed all the ReLU variants in the authors' experiments: training time was reduced, and the neural network performed better on the test set. Figure 11-3 graphs the function, and Equation 11-2 shows its definition.

Equation 11-2. ELU activation function

$$\mathrm{ELU}_{\alpha}(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

Figure 11-3. ELU activation function

6 Djork-Arné Clevert et al., "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)," *Proceedings of the International Conference on Learning Representations* (2016).

The ELU activation function looks a lot like the ReLU function, with a few major differences:

- It takes on negative values when $z < 0$, which allows the unit to have an average output closer to 0 and helps alleviate the vanishing gradients problem. The hyperparameter α defines the value that the ELU function approaches when z is a large negative number. It is usually set to 1, but you can tweak it like any other hyperparameter.
- It has a nonzero gradient for $z < 0$, which avoids the dead neurons problem.
- If α is equal to 1 then the function is smooth everywhere, including around $z = 0$, which helps speed up Gradient Descent since it does not bounce as much to the left and right of $z = 0$.

The main drawback of the ELU activation function is that it is slower to compute than the ReLU function and its variants (due to the use of the exponential function). Its faster convergence rate during training compensates for that slow computation, but still, at test time an ELU network will be slower than a ReLU network.

Then, a 2017 paper[7] by Günter Klambauer et al. introduced the Scaled ELU (SELU) activation function: as its name suggests, it is a scaled variant of the ELU activation function. The authors showed that if you build a neural network composed exclusively of a stack of dense layers, and if all hidden layers use the SELU activation function, then the network will *self-normalize*: the output of each layer will tend to preserve a mean of 0 and standard deviation of 1 during training, which solves the vanishing/exploding gradients problem. As a result, the SELU activation function often significantly outperforms other activation functions for such neural nets (especially deep ones). There are, however, a few conditions for self-normalization to happen (see the paper for the mathematical justification):

- The input features must be standardized (mean 0 and standard deviation 1).
- Every hidden layer's weights must be initialized with LeCun normal initialization. In Keras, this means setting `kernel_initializer="lecun_normal"`.
- The network's architecture must be sequential. Unfortunately, if you try to use SELU in nonsequential architectures, such as recurrent networks (see Chapter 15) or networks with *skip connections* (i.e., connections that skip layers, such as in Wide & Deep nets), self-normalization will not be guaranteed, so SELU will not necessarily outperform other activation functions.

7 Günter Klambauer et al., "Self-Normalizing Neural Networks," *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 972–981.

- The paper only guarantees self-normalization if all layers are dense, but some researchers have noted that the SELU activation function can improve performance in convolutional neural nets as well (see Chapter 14).

So, which activation function should you use for the hidden layers of your deep neural networks? Although your mileage will vary, in general SELU > ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic. If the network's architecture prevents it from self-normalizing, then ELU may perform better than SELU (since SELU is not smooth at $z = 0$). If you care a lot about runtime latency, then you may prefer leaky ReLU. If you don't want to tweak yet another hyperparameter, you may use the default α values used by Keras (e.g., 0.3 for leaky ReLU). If you have spare time and computing power, you can use cross-validation to evaluate other activation functions, such as RReLU if your network is overfitting or PReLU if you have a huge training set. That said, because ReLU is the most used activation function (by far), many libraries and hardware accelerators provide ReLU-specific optimizations; therefore, if speed is your priority, ReLU might still be the best choice.

To use the leaky ReLU activation function, create a `LeakyReLU` layer and add it to your model just after the layer you want to apply it to:

```
model = keras.models.Sequential([
    [...]
    keras.layers.Dense(10, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(alpha=0.2),
    [...]
])
```

For PReLU, replace `LeakyRelu(alpha=0.2)` with `PReLU()`. There is currently no official implementation of RReLU in Keras, but you can fairly easily implement your own (to learn how to do that, see the exercises at the end of Chapter 12).

For SELU activation, set `activation="selu"` and `kernel_initializer="lecun_normal"` when creating a layer:

```
layer = keras.layers.Dense(10, activation="selu",
                           kernel_initializer="lecun_normal")
```

Batch Normalization

Although using He initialization along with ELU (or any variant of ReLU) can significantly reduce the danger of the vanishing/exploding gradients problems at the beginning of training, it doesn't guarantee that they won't come back during training.

In a 2015 paper,[8] Sergey Ioffe and Christian Szegedy proposed a technique called *Batch Normalization* (BN) that addresses these problems. The technique consists of adding an operation in the model just before or after the activation function of each hidden layer. This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting. In other words, the operation lets the model learn the optimal scale and mean of each of the layer's inputs. In many cases, if you add a BN layer as the very first layer of your neural network, you do not need to standardize your training set (e.g., using a `StandardScaler`); the BN layer will do it for you (well, approximately, since it only looks at one batch at a time, and it can also rescale and shift each input feature).

In order to zero-center and normalize the inputs, the algorithm needs to estimate each input's mean and standard deviation. It does so by evaluating the mean and standard deviation of the input over the current mini-batch (hence the name "Batch Normalization"). The whole operation is summarized step by step in Equation 11-3.

Equation 11-3. Batch Normalization algorithm

1. $\boldsymbol{\mu}_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$

2. $\boldsymbol{\sigma}_B{}^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \boldsymbol{\mu}_B\right)^2$

3. $\widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B{}^2 + \varepsilon}}$

4. $\mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \widehat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$

In this algorithm:

- $\boldsymbol{\mu}_B$ is the vector of input means, evaluated over the whole mini-batch B (it contains one mean per input).
- $\boldsymbol{\sigma}_B$ is the vector of input standard deviations, also evaluated over the whole mini-batch (it contains one standard deviation per input).
- m_B is the number of instances in the mini-batch.
- $\widehat{\mathbf{x}}^{(i)}$ is the vector of zero-centered and normalized inputs for instance i.

8 Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *Proceedings of the 32nd International Conference on Machine Learning* (2015): 448–456.

- $\boldsymbol{\gamma}$ is the output scale parameter vector for the layer (it contains one scale parameter per input).
- $\otimes$ represents element-wise multiplication (each input is multiplied by its corresponding output scale parameter).
- $\boldsymbol{\beta}$ is the output shift (offset) parameter vector for the layer (it contains one offset parameter per input). Each input is offset by its corresponding shift parameter.
- ε is a tiny number that avoids division by zero (typically 10^{-5}). This is called a *smoothing term*.
- $\mathbf{z}^{(i)}$ is the output of the BN operation. It is a rescaled and shifted version of the inputs.

So during training, BN standardizes its inputs, then rescales and offsets them. Good! What about at test time? Well, it's not that simple. Indeed, we may need to make predictions for individual instances rather than for batches of instances: in this case, we will have no way to compute each input's mean and standard deviation. Moreover, even if we do have a batch of instances, it may be too small, or the instances may not be independent and identically distributed, so computing statistics over the batch instances would be unreliable. One solution could be to wait until the end of training, then run the whole training set through the neural network and compute the mean and standard deviation of each input of the BN layer. These "final" input means and standard deviations could then be used instead of the batch input means and standard deviations when making predictions. However, most implementations of Batch Normalization estimate these final statistics during training by using a moving average of the layer's input means and standard deviations. This is what Keras does automatically when you use the `BatchNormalization` layer. To sum up, four parameter vectors are learned in each batch-normalized layer: $\boldsymbol{\gamma}$ (the output scale vector) and $\boldsymbol{\beta}$ (the output offset vector) are learned through regular backpropagation, and $\boldsymbol{\mu}$ (the final input mean vector) and $\boldsymbol{\sigma}$ (the final input standard deviation vector) are estimated using an exponential moving average. Note that $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are estimated during training, but they are used only after training (to replace the batch input means and standard deviations in Equation 11-3).

Ioffe and Szegedy demonstrated that Batch Normalization considerably improved all the deep neural networks they experimented with, leading to a huge improvement in the ImageNet classification task (ImageNet is a large database of images classified into many classes, commonly used to evaluate computer vision systems). The vanishing gradients problem was strongly reduced, to the point that they could use saturating activation functions such as the tanh and even the logistic activation function. The networks were also much less sensitive to the weight initialization. The authors were able to use much larger learning rates, significantly speeding up the learning process. Specifically, they note that:

> Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. […] Using an ensemble of batch-normalized networks, we improve upon the best published result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters.

Finally, like a gift that keeps on giving, Batch Normalization acts like a regularizer, reducing the need for other regularization techniques (such as dropout, described later in this chapter).

Batch Normalization does, however, add some complexity to the model (although it can remove the need for normalizing the input data, as we discussed earlier). Moreover, there is a runtime penalty: the neural network makes slower predictions due to the extra computations required at each layer. Fortunately, it's often possible to fuse the BN layer with the previous layer, after training, thereby avoiding the runtime penalty. This is done by updating the previous layer's weights and biases so that it directly produces outputs of the appropriate scale and offset. For example, if the previous layer computes $\mathbf{XW} + \mathbf{b}$, then the BN layer will compute $\boldsymbol{\gamma} \otimes (\mathbf{XW} + \mathbf{b} - \boldsymbol{\mu})/\boldsymbol{\sigma} + \boldsymbol{\beta}$ (ignoring the smoothing term ε in the denominator). If we define $\mathbf{W}' = \boldsymbol{\gamma} \otimes \mathbf{W}/\boldsymbol{\sigma}$ and $\mathbf{b}' = \boldsymbol{\gamma} \otimes (\mathbf{b} - \boldsymbol{\mu})/\boldsymbol{\sigma} + \boldsymbol{\beta}$, the equation simplifies to $\mathbf{XW}' + \mathbf{b}'$. So if we replace the previous layer's weights and biases ($\mathbf{W}$ and $\mathbf{b}$) with the updated weights and biases ($\mathbf{W}'$ and $\mathbf{b}'$), we can get rid of the BN layer (TFLite's optimizer does this automatically; see Chapter 19).

You may find that training is rather slow, because each epoch takes much more time when you use Batch Normalization. This is usually counterbalanced by the fact that convergence is much faster with BN, so it will take fewer epochs to reach the same performance. All in all, *wall time* will usually be shorter (this is the time measured by the clock on your wall).

Implementing Batch Normalization with Keras

As with most things with Keras, implementing Batch Normalization is simple and intuitive. Just add a `BatchNormalization` layer before or after each hidden layer's activation function, and optionally add a BN layer as well as the first layer in your model. For example, this model applies BN after every hidden layer and as the first layer in the model (after flattening the input images):

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

That's all! In this tiny example with just two hidden layers, it's unlikely that Batch Normalization will have a very positive impact; but for deeper networks it can make a tremendous difference.

Let's display the model summary:

```
>>> model.summary()
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 784)               0
_________________________________________________________________
batch_normalization_v2 (Batc (None, 784)               3136
_________________________________________________________________
dense_50 (Dense)             (None, 300)               235500
_________________________________________________________________
batch_normalization_v2_1 (Ba (None, 300)               1200
_________________________________________________________________
dense_51 (Dense)             (None, 100)               30100
_________________________________________________________________
batch_normalization_v2_2 (Ba (None, 100)               400
_________________________________________________________________
dense_52 (Dense)             (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

As you can see, each BN layer adds four parameters per input: $\boldsymbol{\gamma}$, $\boldsymbol{\beta}$, $\boldsymbol{\mu}$, and $\boldsymbol{\sigma}$ (for example, the first BN layer adds 3,136 parameters, which is 4×784). The last two parameters, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$, are the moving averages; they are not affected by backpropagation, so Keras calls them "non-trainable"[9] (if you count the total number of BN parameters, 3,136 + 1,200 + 400, and divide by 2, you get 2,368, which is the total number of non-trainable parameters in this model).

9 However, they are estimated during training, based on the training data, so arguably they *are* trainable. In Keras, "non-trainable" really means "untouched by backpropagation."

Let's look at the parameters of the first BN layer. Two are trainable (by backpropagation), and two are not:

```
>>> [(var.name, var.trainable) for var in model.layers[1].variables]
[('batch_normalization_v2/gamma:0', True),
 ('batch_normalization_v2/beta:0', True),
 ('batch_normalization_v2/moving_mean:0', False),
 ('batch_normalization_v2/moving_variance:0', False)]
```

Now when you create a BN layer in Keras, it also creates two operations that will be called by Keras at each iteration during training. These operations will update the moving averages. Since we are using the TensorFlow backend, these operations are TensorFlow operations (we will discuss TF operations in Chapter 12):

```
>>> model.layers[1].updates
[<tf.Operation 'cond_2/Identity' type=Identity>,
 <tf.Operation 'cond_3/Identity' type=Identity>]
```

The authors of the BN paper argued in favor of adding the BN layers before the activation functions, rather than after (as we just did). There is some debate about this, as which is preferable seems to depend on the task—you can experiment with this too to see which option works best on your dataset. To add the BN layers before the activation functions, you must remove the activation function from the hidden layers and add them as separate layers after the BN layers. Moreover, since a Batch Normalization layer includes one offset parameter per input, you can remove the bias term from the previous layer (just pass `use_bias=False` when creating it):

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(100, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(10, activation="softmax")
])
```

The `BatchNormalization` class has quite a few hyperparameters you can tweak. The defaults will usually be fine, but you may occasionally need to tweak the `momentum`. This hyperparameter is used by the `BatchNormalization` layer when it updates the exponential moving averages; given a new value $\mathbf{v}$ (i.e., a new vector of input means or standard deviations computed over the current batch), the layer updates the running average $\widehat{\mathbf{v}}$ using the following equation:

$$\widehat{\mathbf{v}} \leftarrow \widehat{\mathbf{v}} \times \text{momentum} + \mathbf{v} \times (1 - \text{momentum})$$

A good momentum value is typically close to 1; for example, 0.9, 0.99, or 0.999 (you want more 9s for larger datasets and smaller mini-batches).

Another important hyperparameter is `axis`: it determines which axis should be normalized. It defaults to –1, meaning that by default it will normalize the last axis (using the means and standard deviations computed across the *other* axes). When the input batch is 2D (i.e., the batch shape is [*batch size, features*]), this means that each input feature will be normalized based on the mean and standard deviation computed across all the instances in the batch. For example, the first BN layer in the previous code example will independently normalize (and rescale and shift) each of the 784 input features. If we move the first BN layer before the `Flatten` layer, then the input batches will be 3D, with shape [*batch size, height, width*]; therefore, the BN layer will compute 28 means and 28 standard deviations (1 per column of pixels, computed across all instances in the batch and across all rows in the column), and it will normalize all pixels in a given column using the same mean and standard deviation. There will also be just 28 scale parameters and 28 shift parameters. If instead you still want to treat each of the 784 pixels independently, then you should set `axis=[1, 2]`.

Notice that the BN layer does not perform the same computation during training and after training: it uses batch statistics during training and the "final" statistics after training (i.e., the final values of the moving averages). Let's take a peek at the source code of this class to see how this is handled:

```
class BatchNormalization(keras.layers.Layer):
    [...]
    def call(self, inputs, training=None):
        [...]
```

The `call()` method is the one that performs the computations; as you can see, it has an extra `training` argument, which is set to `None` by default, but the `fit()` method sets to it to `1` during training. If you ever need to write a custom layer, and it must behave differently during training and testing, add a `training` argument to the `call()` method and use this argument in the method to decide what to compute[10] (we will discuss custom layers in Chapter 12).

`BatchNormalization` has become one of the most-used layers in deep neural networks, to the point that it is often omitted in the diagrams, as it is assumed that BN is added after every layer. But a recent paper[11] by Hongyi Zhang et al. may change this assumption: by using a novel *fixed-update* (fixup) weight initialization technique, the authors managed to train a very deep neural network (10,000 layers!) without BN,

10 The Keras API also specifies a `keras.backend.learning_phase()` function that should return `1` during training and `0` otherwise.

11 Hongyi Zhang et al., "Fixup Initialization: Residual Learning Without Normalization," arXiv preprint arXiv: 1901.09321 (2019).

achieving state-of-the-art performance on complex image classification tasks. As this is bleeding-edge research, however, you may want to wait for additional research to confirm this finding before you drop Batch Normalization.

Gradient Clipping

Another popular technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold. This is called *Gradient Clipping*.[12] This technique is most often used in recurrent neural networks, as Batch Normalization is tricky to use in RNNs, as we will see in Chapter 15. For other types of networks, BN is usually sufficient.

In Keras, implementing Gradient Clipping is just a matter of setting the `clipvalue` or `clipnorm` argument when creating an optimizer, like this:

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

This optimizer will clip every component of the gradient vector to a value between −1.0 and 1.0. This means that all the partial derivatives of the loss (with regard to each and every trainable parameter) will be clipped between −1.0 and 1.0. The threshold is a hyperparameter you can tune. Note that it may change the orientation of the gradient vector. For instance, if the original gradient vector is [0.9, 100.0], it points mostly in the direction of the second axis; but once you clip it by value, you get [0.9, 1.0], which points roughly in the diagonal between the two axes. In practice, this approach works well. If you want to ensure that Gradient Clipping does not change the direction of the gradient vector, you should clip by norm by setting `clipnorm` instead of `clipvalue`. This will clip the whole gradient if its ℓ_2 norm is greater than the threshold you picked. For example, if you set `clipnorm=1.0`, then the vector [0.9, 100.0] will be clipped to [0.00899964, 0.9999595], preserving its orientation but almost eliminating the first component. If you observe that the gradients explode during training (you can track the size of the gradients using TensorBoard), you may want to try both clipping by value and clipping by norm, with different thresholds, and see which option performs best on the validation set.

Reusing Pretrained Layers

It is generally not a good idea to train a very large DNN from scratch: instead, you should always try to find an existing neural network that accomplishes a similar task to the one you are trying to tackle (we will discuss how to find them in Chapter 14), then reuse the lower layers of this network. This technique is called *transfer learning*.

12 Razvan Pascanu et al., "On the Difficulty of Training Recurrent Neural Networks," *Proceedings of the 30th International Conference on Machine Learning* (2013): 1310–1318.

It will not only speed up training considerably, but also require significantly less training data.

Suppose you have access to a DNN that was trained to classify pictures into 100 different categories, including animals, plants, vehicles, and everyday objects. You now want to train a DNN to classify specific types of vehicles. These tasks are very similar, even partly overlapping, so you should try to reuse parts of the first network (see Figure 11-4).

Figure 11-4. Reusing pretrained layers

If the input pictures of your new task don't have the same size as the ones used in the original task, you will usually have to add a preprocessing step to resize them to the size expected by the original model. More generally, transfer learning will work best when the inputs have similar low-level features.

The output layer of the original model should usually be replaced because it is most likely not useful at all for the new task, and it may not even have the right number of outputs for the new task.

Similarly, the upper hidden layers of the original model are less likely to be as useful as the lower layers, since the high-level features that are most useful for the new task may differ significantly from the ones that were most useful for the original task. You want to find the right number of layers to reuse.

The more similar the tasks are, the more layers you want to reuse (starting with the lower layers). For very similar tasks, try keeping all the hidden layers and just replacing the output layer.

Try freezing all the reused layers first (i.e., make their weights non-trainable so that Gradient Descent won't modify them), then train your model and see how it performs. Then try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves. The more training data you have, the more layers you can unfreeze. It is also useful to reduce the learning rate when you unfreeze reused layers: this will avoid wrecking their fine-tuned weights.

If you still cannot get good performance, and you have little training data, try dropping the top hidden layer(s) and freezing all the remaining hidden layers again. You can iterate until you find the right number of layers to reuse. If you have plenty of training data, you may try replacing the top hidden layers instead of dropping them, and even adding more hidden layers.

Transfer Learning with Keras

Let's look at an example. Suppose the Fashion MNIST dataset only contained eight classes—for example, all the classes except for sandal and shirt. Someone built and trained a Keras model on that set and got reasonably good performance (>90% accuracy). Let's call this model A. You now want to tackle a different task: you have images of sandals and shirts, and you want to train a binary classifier (positive=shirt, negative=sandal). Your dataset is quite small; you only have 200 labeled images. When you train a new model for this task (let's call it model B) with the same architecture as model A, it performs reasonably well (97.2% accuracy). But since it's a much easier task (there are just two classes), you were hoping for more. While drinking your morning coffee, you realize that your task is quite similar to task A, so perhaps transfer learning can help? Let's find out!

First, you need to load model A and create a new model based on that model's layers. Let's reuse all the layers except for the output layer:

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

Note that `model_A` and `model_B_on_A` now share some layers. When you train `model_B_on_A`, it will also affect `model_A`. If you want to avoid that, you need to *clone* `model_A` before you reuse its layers. To do this, you clone model A's architecture with `clone_model()`, then copy its weights (since `clone_model()` does not clone the weights):

```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

Now you could train `model_B_on_A` for task B, but since the new output layer was initialized randomly it will make large errors (at least during the first few epochs), so there will be large error gradients that may wreck the reused weights. To avoid this, one approach is to freeze the reused layers during the first few epochs, giving the new layer some time to learn reasonable weights. To do this, set every layer's `trainable` attribute to `False` and compile the model:

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",
                     metrics=["accuracy"])
```

You must always compile your model after you freeze or unfreeze layers.

Now you can train the model for a few epochs, then unfreeze the reused layers (which requires compiling the model again) and continue training to fine-tune the reused layers for task B. After unfreezing the reused layers, it is usually a good idea to reduce the learning rate, once again to avoid damaging the reused weights:

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                           validation_data=(X_valid_B, y_valid_B))

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=1e-4) # the default lr is 1e-2
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,
                     metrics=["accuracy"])
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                           validation_data=(X_valid_B, y_valid_B))
```

So, what's the final verdict? Well, this model's test accuracy is 99.25%, which means that transfer learning reduced the error rate from 2.8% down to almost 0.7%! That's a factor of four!

```
>>> model_B_on_A.evaluate(X_test_B, y_test_B)
[0.06887910133600235, 0.9925]
```

Are you convinced? You shouldn't be: I cheated! I tried many configurations until I found one that demonstrated a strong improvement. If you try to change the classes or the random seed, you will see that the improvement generally drops, or even vanishes or reverses. What I did is called "torturing the data until it confesses." When a

paper just looks too positive, you should be suspicious: perhaps the flashy new technique does not actually help much (in fact, it may even degrade performance), but the authors tried many variants and reported only the best results (which may be due to sheer luck), without mentioning how many failures they encountered on the way. Most of the time, this is not malicious at all, but it is part of the reason so many results in science can never be reproduced.

Why did I cheat? It turns out that transfer learning does not work very well with small dense networks, presumably because small networks learn few patterns, and dense networks learn very specific patterns, which are unlikely to be useful in other tasks. Transfer learning works best with deep convolutional neural networks, which tend to learn feature detectors that are much more general (especially in the lower layers). We will revisit transfer learning in Chapter 14, using the techniques we just discussed (and this time there will be no cheating, I promise!).

Unsupervised Pretraining

Suppose you want to tackle a complex task for which you don't have much labeled training data, but unfortunately you cannot find a model trained on a similar task. Don't lose hope! First, you should try to gather more labeled training data, but if you can't, you may still be able to perform *unsupervised pretraining* (see Figure 11-5). Indeed, it is often cheap to gather unlabeled training examples, but expensive to label them. If you can gather plenty of unlabeled training data, you can try to use it to train an unsupervised model, such as an autoencoder or a generative adversarial network (see Chapter 17). Then you can reuse the lower layers of the autoencoder or the lower layers of the GAN's discriminator, add the output layer for your task on top, and fine-tune the final network using supervised learning (i.e., with the labeled training examples).

It is this technique that Geoffrey Hinton and his team used in 2006 and which led to the revival of neural networks and the success of Deep Learning. Until 2010, unsupervised pretraining—typically with restricted Boltzmann machines (RBMs; see Appendix E)—was the norm for deep nets, and only after the vanishing gradients problem was alleviated did it become much more common to train DNNs purely using supervised learning. Unsupervised pretraining (today typically using autoencoders or GANs rather than RBMs) is still a good option when you have a complex task to solve, no similar model you can reuse, and little labeled training data but plenty of unlabeled training data.

Note that in the early days of Deep Learning it was difficult to train deep models, so people would use a technique called *greedy layer-wise pretraining* (depicted in Figure 11-5). They would first train an unsupervised model with a single layer, typically an RBM, then they would freeze that layer and add another one on top of it, then train the model again (effectively just training the new layer), then freeze the

new layer and add another layer on top of it, train the model again, and so on. Nowadays, things are much simpler: people generally train the full unsupervised model in one shot (i.e., in Figure 11-5, just start directly at step three) and use autoencoders or GANs rather than RBMs.

Figure 11-5. In unsupervised training, a model is trained on the unlabeled data (or on all the data) using an unsupervised learning technique, then it is fine-tuned for the final task on the labeled data using a supervised learning technique; the unsupervised part may train one layer at a time as shown here, or it may train the full model directly

Pretraining on an Auxiliary Task

If you do not have much labeled training data, one last option is to train a first neural network on an auxiliary task for which you can easily obtain or generate labeled training data, then reuse the lower layers of that network for your actual task. The first neural network's lower layers will learn feature detectors that will likely be reusable by the second neural network.

For example, if you want to build a system to recognize faces, you may only have a few pictures of each individual—clearly not enough to train a good classifier. Gathering hundreds of pictures of each person would not be practical. You could, however, gather a lot of pictures of random people on the web and train a first neural network to detect whether or not two different pictures feature the same person. Such a

network would learn good feature detectors for faces, so reusing its lower layers would allow you to train a good face classifier that uses little training data.

For *natural language processing* (NLP) applications, you can download a corpus of millions of text documents and automatically generate labeled data from it. For example, you could randomly mask out some words and train a model to predict what the missing words are (e.g., it should predict that the missing word in the sentence "What ___ you saying?" is probably "are" or "were"). If you can train a model to reach good performance on this task, then it will already know quite a lot about language, and you can certainly reuse it for your actual task and fine-tune it on your labeled data (we will discuss more pretraining tasks in Chapter 15).

Self-supervised learning is when you automatically generate the labels from the data itself, then you train a model on the resulting "labeled" dataset using supervised learning techniques. Since this approach requires no human labeling whatsoever, it is best classified as a form of unsupervised learning.

Faster Optimizers

Training a very large deep neural network can be painfully slow. So far we have seen four ways to speed up training (and reach a better solution): applying a good initialization strategy for the connection weights, using a good activation function, using Batch Normalization, and reusing parts of a pretrained network (possibly built on an auxiliary task or using unsupervised learning). Another huge speed boost comes from using a faster optimizer than the regular Gradient Descent optimizer. In this section we will present the most popular algorithms: momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and finally Adam and Nadam optimization.

Momentum Optimization

Imagine a bowling ball rolling down a gentle slope on a smooth surface: it will start out slowly, but it will quickly pick up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance). This is the very simple idea behind *momentum optimization*, proposed by Boris Polyak in 1964.[13] In contrast, regular Gradient Descent will simply take small, regular steps down the slope, so the algorithm will take much more time to reach the bottom.

13 Boris T. Polyak, "Some Methods of Speeding Up the Convergence of Iteration Methods," *USSR Computational Mathematics and Mathematical Physics* 4, no. 5 (1964): 1–17.

Recall that Gradient Descent updates the weights $\boldsymbol{\theta}$ by directly subtracting the gradient of the cost function $J(\boldsymbol{\theta})$ with regard to the weights ($\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$) multiplied by the learning rate η. The equation is: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.

Momentum optimization cares a great deal about what previous gradients were: at each iteration, it subtracts the local gradient from the *momentum vector* $\mathbf{m}$ (multiplied by the learning rate η), and it updates the weights by adding this momentum vector (see Equation 11-4). In other words, the gradient is used for acceleration, not for speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter β, called the *momentum*, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

Equation 11-4. Momentum algorithm

1. $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$

You can easily verify that if the gradient remains constant, the terminal velocity (i.e., the maximum size of the weight updates) is equal to that gradient multiplied by the learning rate η multiplied by $1/(1-\beta)$ (ignoring the sign). For example, if $\beta = 0.9$, then the terminal velocity is equal to 10 times the gradient times the learning rate, so momentum optimization ends up going 10 times faster than Gradient Descent! This allows momentum optimization to escape from plateaus much faster than Gradient Descent. We saw in Chapter 4 that when the inputs have very different scales, the cost function will look like an elongated bowl (see Figure 4-7). Gradient Descent goes down the steep slope quite fast, but then it takes a very long time to go down the valley. In contrast, momentum optimization will roll down the valley faster and faster until it reaches the bottom (the optimum). In deep neural networks that don't use Batch Normalization, the upper layers will often end up having inputs with very different scales, so using momentum optimization helps a lot. It can also help roll past local optima.

Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum. This is one of the reasons it's good to have a bit of friction in the system: it gets rid of these oscillations and thus speeds up convergence.

Implementing momentum optimization in Keras is a no-brainer: just use the `SGD` optimizer and set its `momentum` hyperparameter, then lie back and profit!

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

The one drawback of momentum optimization is that it adds yet another hyperparameter to tune. However, the momentum value of 0.9 usually works well in practice and almost always goes faster than regular Gradient Descent.

Nesterov Accelerated Gradient

One small variant to momentum optimization, proposed by Yurii Nesterov in 1983,[14] is almost always faster than vanilla momentum optimization. The *Nesterov Accelerated Gradient* (NAG) method, also known as *Nesterov momentum optimization*, measures the gradient of the cost function not at the local position $\boldsymbol{\theta}$ but slightly ahead in the direction of the momentum, at $\boldsymbol{\theta} + \beta\mathbf{m}$ (see Equation 11-5).

Equation 11-5. Nesterov Accelerated Gradient algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta} + \beta\mathbf{m})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$

This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than the gradient at the original position, as you can see in Figure 11-6 (where ∇_1 represents the gradient of the cost function measured at the starting point $\boldsymbol{\theta}$, and ∇_2 represents the gradient at the point located at $\boldsymbol{\theta} + \beta\mathbf{m}$).

As you can see, the Nesterov update ends up slightly closer to the optimum. After a while, these small improvements add up and NAG ends up being significantly faster than regular momentum optimization. Moreover, note that when the momentum pushes the weights across a valley, ∇_1 continues to push farther across the valley, while ∇_2 pushes back toward the bottom of the valley. This helps reduce oscillations and thus NAG converges faster.

NAG is generally faster than regular momentum optimization. To use it, simply set `nesterov=True` when creating the `SGD` optimizer:

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

14 Yurii Nesterov, "A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence $O(1/k^2)$," *Doklady AN USSR* 269 (1983): 543–547.

Figure 11-6. Regular versus Nesterov momentum optimization: the former applies the gradients computed before the momentum step, while the latter applies the gradients computed after

AdaGrad

Consider the elongated bowl problem again: Gradient Descent starts by quickly going down the steepest slope, which does not point straight toward the global optimum, then it very slowly goes down to the bottom of the valley. It would be nice if the algorithm could correct its direction earlier to point a bit more toward the global optimum. The *AdaGrad* algorithm[15] achieves this correction by scaling down the gradient vector along the steepest dimensions (see Equation 11-6).

Equation 11-6. AdaGrad algorithm

1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$

The first step accumulates the square of the gradients into the vector **s** (recall that the $\otimes$ symbol represents the element-wise multiplication). This vectorized form is equivalent to computing $s_i \leftarrow s_i + (\partial J(\boldsymbol{\theta}) / \partial \theta_i)^2$ for each element s_i of the vector **s**; in other words, each s_i accumulates the squares of the partial derivative of the cost function with regard to parameter θ_i. If the cost function is steep along the ith dimension, then s_i will get larger and larger at each iteration.

The second step is almost identical to Gradient Descent, but with one big difference: the gradient vector is scaled down by a factor of $\sqrt{\mathbf{s} + \varepsilon}$ (the $\oslash$ symbol represents the

15 John Duchi et al., "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research* 12 (2011): 2121–2159.

element-wise division, and ε is a smoothing term to avoid division by zero, typically set to 10^{-10}). This vectorized form is equivalent to simultaneously computing $\theta_i \leftarrow \theta_i - \eta\, \partial J(\boldsymbol{\theta}) / \partial \theta_i / \sqrt{s_i + \varepsilon}$ for all parameters θ_i.

In short, this algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an *adaptive learning rate*. It helps point the resulting updates more directly toward the global optimum (see Figure 11-7). One additional benefit is that it requires much less tuning of the learning rate hyperparameter η.

Figure 11-7. AdaGrad versus Gradient Descent: the former can correct its direction earlier to point to the optimum

AdaGrad frequently performs well for simple quadratic problems, but it often stops too early when training neural networks. The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum. So even though Keras has an `Adagrad` optimizer, you should not use it to train deep neural networks (it may be efficient for simpler tasks such as Linear Regression, though). Still, understanding AdaGrad is helpful to grasp the other adaptive learning rate optimizers.

RMSProp

As we've seen, AdaGrad runs the risk of slowing down a bit too fast and never converging to the global optimum. The *RMSProp* algorithm[16] fixes this by accumulating only the gradients from the most recent iterations (as opposed to all the gradients

16 This algorithm was created by Geoffrey Hinton and Tijmen Tieleman in 2012 and presented by Geoffrey Hinton in his Coursera class on neural networks (slides: *https://homl.info/57*; video: *https://homl.info/58*). Amusingly, since the authors did not write a paper to describe the algorithm, researchers often cite "slide 29 in lecture 6" in their papers.

since the beginning of training). It does so by using exponential decay in the first step (see Equation 11-7).

Equation 11-7. RMSProp algorithm

1. $\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$

The decay rate β is typically set to 0.9. Yes, it is once again a new hyperparameter, but this default value often works well, so you may not need to tune it at all.

As you might expect, Keras has an `RMSprop` optimizer:

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Note that the `rho` argument corresponds to β in Equation 11-7. Except on very simple problems, this optimizer almost always performs much better than AdaGrad. In fact, it was the preferred optimization algorithm of many researchers until Adam optimization came around.

Adam and Nadam Optimization

Adam,[17] which stands for *adaptive moment estimation*, combines the ideas of momentum optimization and RMSProp: just like momentum optimization, it keeps track of an exponentially decaying average of past gradients; and just like RMSProp, it keeps track of an exponentially decaying average of past squared gradients (see Equation 11-8).[18]

Equation 11-8. Adam algorithm

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - \left(1 - \beta_1\right) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + \left(1 - \beta_2\right) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3. $\widehat{\mathbf{m}} \leftarrow \dfrac{\mathbf{m}}{1 - {\beta_1}^t}$
4. $\widehat{\mathbf{s}} \leftarrow \dfrac{\mathbf{s}}{1 - {\beta_2}^t}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \varepsilon}$

17 Diederik P. Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization," arXiv preprint arXiv: 1412.6980 (2014).

18 These are estimations of the mean and (uncentered) variance of the gradients. The mean is often called the *first moment* while the variance is often called the *second moment*, hence the name of the algorithm.

In this equation, t represents the iteration number (starting at 1).

If you just look at steps 1, 2, and 5, you will notice Adam's close similarity to both momentum optimization and RMSProp. The only difference is that step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just $1 - \beta_1$ times the decaying sum). Steps 3 and 4 are somewhat of a technical detail: since **m** and **s** are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost **m** and **s** at the beginning of training.

The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999. As earlier, the smoothing term ε is usually initialized to a tiny number such as 10^{-7}. These are the default values for the `Adam` class (to be precise, `epsilon` defaults to `None`, which tells Keras to use `keras.backend.epsilon()`, which defaults to 10^{-7}; you can change it using `keras.backend.set_epsilon()`). Here is how to create an Adam optimizer using Keras:

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

Since Adam is an adaptive learning rate algorithm (like AdaGrad and RMSProp), it requires less tuning of the learning rate hyperparameter η. You can often use the default value $\eta = 0.001$, making Adam even easier to use than Gradient Descent.

If you are starting to feel overwhelmed by all these different techniques and are wondering how to choose the right ones for your task, don't worry: some practical guidelines are provided at the end of this chapter.

Finally, two variants of Adam are worth mentioning:

AdaMax

Notice that in step 2 of Equation 11-8, Adam accumulates the squares of the gradients in **s** (with a greater weight for more recent gradients). In step 5, if we ignore ε and steps 3 and 4 (which are technical details anyway), Adam scales down the parameter updates by the square root of **s**. In short, Adam scales down the parameter updates by the ℓ_2 norm of the time-decayed gradients (recall that the ℓ_2 norm is the square root of the sum of squares). AdaMax, introduced in the same paper as Adam, replaces the ℓ_2 norm with the ℓ_∞ norm (a fancy way of saying the max). Specifically, it replaces step 2 in Equation 11-8 with $\mathbf{s} \leftarrow \max(\beta_2\mathbf{s}, \nabla_\theta J(\boldsymbol{\theta}))$, it drops step 4, and in step 5 it scales down the gradient updates by a factor of **s**, which is just the max of the time-decayed gradients. In practice, this can make AdaMax more stable than Adam, but it really depends on the dataset,

and in general Adam performs better. So, this is just one more optimizer you can try if you experience problems with Adam on some task.

Nadam

Nadam optimization is Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam. In his report introducing this technique,[19] the researcher Timothy Dozat compares many different optimizers on various tasks and finds that Nadam generally outperforms Adam but is sometimes outperformed by RMSProp.

Adaptive optimization methods (including RMSProp, Adam, and Nadam optimization) are often great, converging fast to a good solution. However, a 2017 paper[20] by Ashia C. Wilson et al. showed that they can lead to solutions that generalize poorly on some datasets. So when you are disappointed by your model's performance, try using plain Nesterov Accelerated Gradient instead: your dataset may just be allergic to adaptive gradients. Also check out the latest research, because it's moving fast.

All the optimization techniques discussed so far only rely on the *first-order partial derivatives* (*Jacobians*). The optimization literature also contains amazing algorithms based on the *second-order partial derivatives* (the *Hessians*, which are the partial derivatives of the Jacobians). Unfortunately, these algorithms are very hard to apply to deep neural networks because there are n^2 Hessians per output (where n is the number of parameters), as opposed to just n Jacobians per output. Since DNNs typically have tens of thousands of parameters, the second-order optimization algorithms often don't even fit in memory, and even when they do, computing the Hessians is just too slow.

19 Timothy Dozat, "Incorporating Nesterov Momentum into Adam" (2016).

20 Ashia C. Wilson et al., "The Marginal Value of Adaptive Gradient Methods in Machine Learning," *Advances in Neural Information Processing Systems* 30 (2017): 4148–4158.

Training Sparse Models

All the optimization algorithms just presented produce dense models, meaning that most parameters will be nonzero. If you need a blazingly fast model at runtime, or if you need it to take up less memory, you may prefer to end up with a sparse model instead.

One easy way to achieve this is to train the model as usual, then get rid of the tiny weights (set them to zero). Note that this will typically not lead to a very sparse model, and it may degrade the model's performance.

A better option is to apply strong ℓ_1 regularization during training (we will see how later in this chapter), as it pushes the optimizer to zero out as many weights as it can (as discussed in "Lasso Regression" on page 137 in Chapter 4).

If these techniques remain insufficient, check out the TensorFlow Model Optimization Toolkit (TF-MOT), which provides a pruning API capable of iteratively removing connections during training based on their magnitude.

Table 11-2 compares all the optimizers we've discussed so far (* is bad, ** is average, and *** is good).

Table 11-2. Optimizer comparison

Class	Convergence speed	Convergence quality
`SGD`	*	***
`SGD(momentum=...)`	**	***
`SGD(momentum=..., nesterov=True)`	**	***
`Adagrad`	***	* (stops too early)
`RMSprop`	***	** or ***
`Adam`	***	** or ***
`Nadam`	***	** or ***
`AdaMax`	***	** or ***

Learning Rate Scheduling

Finding a good learning rate is very important. If you set it much too high, training may diverge (as we discussed in "Gradient Descent" on page 118). If you set it too low, training will eventually converge to the optimum, but it will take a very long time. If you set it slightly too high, it will make progress very quickly at first, but it will end up dancing around the optimum, never really settling down. If you have a limited computing budget, you may have to interrupt training before it has converged properly, yielding a suboptimal solution (see Figure 11-8).

Figure 11-8. Learning curves for various learning rates η

As we discussed in Chapter 10, you can find a good learning rate by training the model for a few hundred iterations, exponentially increasing the learning rate from a very small value to a very large value, and then looking at the learning curve and picking a learning rate slightly lower than the one at which the learning curve starts shooting back up. You can then reinitialize your model and train it with that learning rate.

But you can do better than a constant learning rate: if you start with a large learning rate and then reduce it once training stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate. There are many different strategies to reduce the learning rate during training. It can also be beneficial to start with a low learning rate, increase it, then drop it again. These strategies are called *learning schedules* (we briefly introduced this concept in Chapter 4). These are the most commonly used learning schedules:

Power scheduling

Set the learning rate to a function of the iteration number t: $\eta(t) = \eta_0 / (1 + t/s)^c$. The initial learning rate η_0, the power c (typically set to 1), and the steps s are hyperparameters. The learning rate drops at each step. After s steps, it is down to $\eta_0 / 2$. After s more steps, it is down to $\eta_0 / 3$, then it goes down to $\eta_0 / 4$, then $\eta_0 / 5$, and so on. As you can see, this schedule first drops quickly, then more and more slowly. Of course, power scheduling requires tuning η_0 and s (and possibly c).

Exponential scheduling

Set the learning rate to $\eta(t) = \eta_0\, 0.1^{t/s}$. The learning rate will gradually drop by a factor of 10 every s steps. While power scheduling reduces the learning rate more and more slowly, exponential scheduling keeps slashing it by a factor of 10 every s steps.

Piecewise constant scheduling

Use a constant learning rate for a number of epochs (e.g., $\eta_0 = 0.1$ for 5 epochs), then a smaller learning rate for another number of epochs (e.g., $\eta_1 = 0.001$ for 50 epochs), and so on. Although this solution can work very well, it requires fiddling around to figure out the right sequence of learning rates and how long to use each of them.

Performance scheduling

Measure the validation error every N steps (just like for early stopping), and reduce the learning rate by a factor of λ when the error stops dropping.

1cycle scheduling

Contrary to the other approaches, *1cycle* (introduced in a 2018 paper[21] by Leslie Smith) starts by increasing the initial learning rate η_0, growing linearly up to η_1 halfway through training. Then it decreases the learning rate linearly down to η_0 again during the second half of training, finishing the last few epochs by dropping the rate down by several orders of magnitude (still linearly). The maximum learning rate η_1 is chosen using the same approach we used to find the optimal learning rate, and the initial learning rate η_0 is chosen to be roughly 10 times lower. When using a momentum, we start with a high momentum first (e.g., 0.95), then drop it down to a lower momentum during the first half of training (e.g., down to 0.85, linearly), and then bring it back up to the maximum value (e.g., 0.95) during the second half of training, finishing the last few epochs with that maximum value. Smith did many experiments showing that this approach was often able to speed up training considerably and reach better performance. For example, on the popular CIFAR10 image dataset, this approach reached 91.9% validation accuracy in just 100 epochs, instead of 90.3% accuracy in 800 epochs through a standard approach (with the same neural network architecture).

A 2013 paper[22] by Andrew Senior et al. compared the performance of some of the most popular learning schedules when using momentum optimization to train deep neural networks for speech recognition. The authors concluded that, in this setting, both performance scheduling and exponential scheduling performed well. They favored exponential scheduling because it was easy to tune and it converged slightly faster to the optimal solution (they also mentioned that it was easier to implement

21 Leslie N. Smith, "A Disciplined Approach to Neural Network Hyper-Parameters: Part 1—Learning Rate, Batch Size, Momentum, and Weight Decay," arXiv preprint arXiv:1803.09820 (2018).

22 Andrew Senior et al., "An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition," *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2013): 6724–6728.

than performance scheduling, but in Keras both options are easy). That said, the 1cycle approach seems to perform even better.

Implementing power scheduling in Keras is the easiest option: just set the `decay` hyperparameter when creating an optimizer:

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

The `decay` is the inverse of s (the number of steps it takes to divide the learning rate by one more unit), and Keras assumes that c is equal to 1.

Exponential scheduling and piecewise scheduling are quite simple too. You first need to define a function that takes the current epoch and returns the learning rate. For example, let's implement exponential scheduling:

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1**(epoch / 20)
```

If you do not want to hardcode η_0 and s, you can create a function that returns a configured function:

```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1**(epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

Next, create a `LearningRateScheduler` callback, giving it the schedule function, and pass this callback to the `fit()` method:

```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, [...], callbacks=[lr_scheduler])
```

The `LearningRateScheduler` will update the optimizer's `learning_rate` attribute at the beginning of each epoch. Updating the learning rate once per epoch is usually enough, but if you want it to be updated more often, for example at every step, you can always write your own callback (see the "Exponential Scheduling" section of the notebook for an example). Updating the learning rate at every step makes sense if there are many steps per epoch. Alternatively, you can use the `keras.optimizers.schedules` approach, described shortly.

The schedule function can optionally take the current learning rate as a second argument. For example, the following schedule function multiplies the previous learning rate by $0.1^{1/20}$, which results in the same exponential decay (except the decay now starts at the beginning of epoch 0 instead of 1):

```
def exponential_decay_fn(epoch, lr):
    return lr * 0.1**(1 / 20)
```

This implementation relies on the optimizer's initial learning rate (contrary to the previous implementation), so make sure to set it appropriately.

When you save a model, the optimizer and its learning rate get saved along with it. This means that with this new schedule function, you could just load a trained model and continue training where it left off, no problem. Things are not so simple if your schedule function uses the `epoch` argument, however: the epoch does not get saved, and it gets reset to 0 every time you call the `fit()` method. If you were to continue training a model where it left off, this could lead to a very large learning rate, which would likely damage your model's weights. One solution is to manually set the `fit()` method's `initial_epoch` argument so the `epoch` starts at the right value.

For piecewise constant scheduling, you can use a schedule function like the following one (as earlier, you can define a more general function if you want; see the "Piecewise Constant Scheduling" section of the notebook for an example), then create a `LearningRateScheduler` callback with this function and pass it to the `fit()` method, just like we did for exponential scheduling:

```
def piecewise_constant_fn(epoch):
    if epoch < 5:
        return 0.01
    elif epoch < 15:
        return 0.005
    else:
        return 0.001
```

For performance scheduling, use the `ReduceLROnPlateau` callback. For example, if you pass the following callback to the `fit()` method, it will multiply the learning rate by 0.5 whenever the best validation loss does not improve for five consecutive epochs (other options are available; please check the documentation for more details):

```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)
```

Lastly, tf.keras offers an alternative way to implement learning rate scheduling: define the learning rate using one of the schedules available in `keras.optimizers.schedules`, then pass this learning rate to any optimizer. This approach updates the learning rate at each step rather than at each epoch. For example, here is how to implement the same exponential schedule as the `exponential_decay_fn()` function we defined earlier:

```
s = 20 * len(X_train) // 32 # number of steps in 20 epochs (batch size = 32)
learning_rate = keras.optimizers.schedules.ExponentialDecay(0.01, s, 0.1)
optimizer = keras.optimizers.SGD(learning_rate)
```

This is nice and simple, plus when you save the model, the learning rate and its schedule (including its state) get saved as well. This approach, however, is not part of the Keras API; it is specific to tf.keras.

As for the 1cycle approach, the implementation poses no particular difficulty: just create a custom callback that modifies the learning rate at each iteration (you can update the optimizer's learning rate by changing `self.model.optimizer.lr`). See the "1Cycle scheduling" section of the notebook for an example.

To sum up, exponential decay, performance scheduling, and 1cycle can considerably speed up convergence, so give them a try!

Avoiding Overfitting Through Regularization

> With four parameters I can fit an elephant and with five I can make him wiggle his trunk.
>
> —John von Neumann, cited by Enrico Fermi in *Nature* 427

With thousands of parameters, you can fit the whole zoo. Deep neural networks typically have tens of thousands of parameters, sometimes even millions. This gives them an incredible amount of freedom and means they can fit a huge variety of complex datasets. But this great flexibility also makes the network prone to overfitting the training set. We need regularization.

We already implemented one of the best regularization techniques in Chapter 10: early stopping. Moreover, even though Batch Normalization was designed to solve the unstable gradients problems, it also acts like a pretty good regularizer. In this section we will examine other popular regularization techniques for neural networks: ℓ_1 and ℓ_2 regularization, dropout, and max-norm regularization.

ℓ_1 and ℓ_2 Regularization

Just like you did in Chapter 4 for simple linear models, you can use ℓ_2 regularization to constrain a neural network's connection weights, and/or ℓ_1 regularization if you want a sparse model (with many weights equal to 0). Here is how to apply ℓ_2 regularization to a Keras layer's connection weights, using a regularization factor of 0.01:

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

The `l2()` function returns a regularizer that will be called at each step during training to compute the regularization loss. This is then added to the final loss. As you might expect, you can just use `keras.regularizers.l1()` if you want ℓ_1 regularization; if you want both ℓ_1 and ℓ_2 regularization, use `keras.regularizers.l1_l2()` (specifying both regularization factors).

Since you will typically want to apply the same regularizer to all layers in your network, as well as using the same activation function and the same initialization strategy in all hidden layers, you may find yourself repeating the same arguments. This

makes the code ugly and error-prone. To avoid this, you can try refactoring your code to use loops. Another option is to use Python's `functools.partial()` function, which lets you create a thin wrapper for any callable, with some default argument values:

```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
])
```

Dropout

Dropout is one of the most popular regularization techniques for deep neural networks. It was proposed in a paper[23] by Geoffrey Hinton in 2012 and further detailed in a 2014 paper[24] by Nitish Srivastava et al., and it has proven to be highly successful: even the state-of-the-art neural networks get a 1–2% accuracy boost simply by adding dropout. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons, but always excluding the output neurons) has a probability p of being temporarily "dropped out," meaning it will be entirely ignored during this training step, but it may be active during the next step (see Figure 11-9). The hyperparameter p is called the *dropout rate*, and it is typically set between 10% and 50%: closer to 20–30% in recurrent neural nets (see Chapter 15), and closer to 40–50% in convolutional neural networks (see Chapter 14). After training, neurons don't get dropped anymore. And that's all (except for a technical detail we will discuss momentarily).

23 Geoffrey E. Hinton et al., "Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors," arXiv preprint arXiv:1207.0580 (2012).

24 Nitish Srivastava et al., "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research* 15 (2014): 1929–1958.

Figure 11-9. With dropout regularization, at each training iteration a random subset of all neurons in one or more layers—except the output layer—are "dropped out"; these neurons output 0 at this iteration (represented by the dashed arrows)

It's surprising at first that this destructive technique works at all. Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work? Well, who knows; perhaps it would! The company would be forced to adapt its organization; it could not rely on any single person to work the coffee machine or perform any other critical tasks, so this expertise would have to be spread across several people. Employees would have to learn to cooperate with many of their coworkers, not just a handful of them. The company would become much more resilient. If one person quit, it wouldn't make much of a difference. It's unclear whether this idea would actually work for companies, but it certainly does for neural networks. Neurons trained with dropout cannot co-adapt with their neighboring neurons; they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons; they must pay attention to each of their input neurons. They end up being less sensitive to slight changes in the inputs. In the end, you get a more robust network that generalizes better.

Another way to understand the power of dropout is to realize that a unique neural network is generated at each training step. Since each neuron can be either present or absent, there are a total of 2^N possible networks (where N is the total number of droppable neurons). This is such a huge number that it is virtually impossible for the same neural network to be sampled twice. Once you have run 10,000 training steps, you have essentially trained 10,000 different neural networks (each with just one training instance). These neural networks are obviously not independent because they share many of their weights, but they are nevertheless all different. The resulting neural network can be seen as an averaging ensemble of all these smaller neural networks.

In practice, you can usually apply dropout only to the neurons in the top one to three layers (excluding the output layer).

There is one small but important technical detail. Suppose $p = 50\%$, in which case during testing a neuron would be connected to twice as many input neurons as it would be (on average) during training. To compensate for this fact, we need to multiply each neuron's input connection weights by 0.5 after training. If we don't, each neuron will get a total input signal roughly twice as large as what the network was trained on and will be unlikely to perform well. More generally, we need to multiply each input connection weight by the *keep probability* $(1 - p)$ after training. Alternatively, we can divide each neuron's output by the keep probability during training (these alternatives are not perfectly equivalent, but they work equally well).

To implement dropout using Keras, you can use the `keras.layers.Dropout` layer. During training, it randomly drops some inputs (setting them to 0) and divides the remaining inputs by the keep probability. After training, it does nothing at all; it just passes the inputs to the next layer. The following code applies dropout regularization before every `Dense` layer, using a dropout rate of 0.2:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

Since dropout is only active during training, comparing the training loss and the validation loss can be misleading. In particular, a model may be overfitting the training set and yet have similar training and validation losses. So make sure to evaluate the training loss without dropout (e.g., after training).

If you observe that the model is overfitting, you can increase the dropout rate. Conversely, you should try decreasing the dropout rate if the model underfits the training set. It can also help to increase the dropout rate for large layers, and reduce it for small ones. Moreover, many state-of-the-art architectures only use dropout after the last hidden layer, so you may want to try this if full dropout is too strong.

Dropout does tend to significantly slow down convergence, but it usually results in a much better model when tuned properly. So, it is generally well worth the extra time and effort.

If you want to regularize a self-normalizing network based on the SELU activation function (as discussed earlier), you should use *alpha dropout*: this is a variant of dropout that preserves the mean and standard deviation of its inputs (it was introduced in the same paper as SELU, as regular dropout would break self-normalization).

Monte Carlo (MC) Dropout

In 2016, a paper[25] by Yarin Gal and Zoubin Ghahramani added a few more good reasons to use dropout:

- First, the paper established a profound connection between dropout networks (i.e., neural networks containing a `Dropout` layer before every weight layer) and approximate Bayesian inference,[26] giving dropout a solid mathematical justification.
- Second, the authors introduced a powerful technique called *MC Dropout*, which can boost the performance of any trained dropout model without having to retrain it or even modify it at all, provides a much better measure of the model's uncertainty, and is also amazingly simple to implement.

If this all sounds like a "one weird trick" advertisement, then take a look at the following code. It is the full implementation of *MC Dropout*, boosting the dropout model we trained earlier without retraining it:

```
y_probas = np.stack([model(X_test_scaled, training=True)
                     for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

We just make 100 predictions over the test set, setting `training=True` to ensure that the `Dropout` layer is active, and stack the predictions. Since dropout is active, all the predictions will be different. Recall that `predict()` returns a matrix with one row per instance and one column per class. Because there are 10,000 instances in the test set and 10 classes, this is a matrix of shape [10000, 10]. We stack 100 such matrices, so `y_probas` is an array of shape [100, 10000, 10]. Once we average over the first

25 Yarin Gal and Zoubin Ghahramani, "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning," *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1050–1059.

26 Specifically, they show that training a dropout network is mathematically equivalent to approximate Bayesian inference in a specific type of probabilistic model called a *Deep Gaussian Process*.

dimension (`axis=0`), we get `y_proba`, an array of shape [10000, 10], like we would get with a single prediction. That's all! Averaging over multiple predictions with dropout on gives us a Monte Carlo estimate that is generally more reliable than the result of a single prediction with dropout off. For example, let's look at the model's prediction for the first instance in the Fashion MNIST test set, with dropout off:

```
>>> np.round(model.predict(X_test_scaled[:1]), 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.99]],
      dtype=float32)
```

The model seems almost certain that this image belongs to class 9 (ankle boot). Should you trust it? Is there really so little room for doubt? Compare this with the predictions made when dropout is activated:

```
>>> np.round(y_probas[:, :1], 2)
array([[[0.  , 0.  , 0.  , 0.  , 0.  , 0.14, 0.  , 0.17, 0.  , 0.68]],
       [[0.  , 0.  , 0.  , 0.  , 0.  , 0.16, 0.  , 0.2 , 0.  , 0.64]],
       [[0.  , 0.  , 0.  , 0.  , 0.  , 0.02, 0.  , 0.01, 0.  , 0.97]],
       [...]
```

This tells a very different story: apparently, when we activate dropout, the model is not sure anymore. It still seems to prefer class 9, but sometimes it hesitates with classes 5 (sandal) and 7 (sneaker), which makes sense given they're all footwear. Once we average over the first dimension, we get the following MC Dropout predictions:

```
>>> np.round(y_proba[:1], 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.22, 0.  , 0.16, 0.  , 0.62]],
      dtype=float32)
```

The model still thinks this image belongs to class 9, but only with a 62% confidence, which seems much more reasonable than 99%. Plus it's useful to know exactly which other classes it thinks are likely. And you can also take a look at the standard deviation of the probability estimates:

```
>>> y_std = y_probas.std(axis=0)
>>> np.round(y_std[:1], 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.28, 0.  , 0.21, 0.02, 0.32]],
      dtype=float32)
```

Apparently there's quite a lot of variance in the probability estimates: if you were building a risk-sensitive system (e.g., a medical or financial system), you should probably treat such an uncertain prediction with extreme caution. You definitely would not treat it like a 99% confident prediction. Moreover, the model's accuracy got a small boost from 86.8 to 86.9:

```
>>> accuracy = np.sum(y_pred == y_test) / len(y_test)
>>> accuracy
0.8694
```

The number of Monte Carlo samples you use (100 in this example) is a hyperparameter you can tweak. The higher it is, the more accurate the predictions and their uncertainty estimates will be. However, if you double it, inference time will also be doubled. Moreover, above a certain number of samples, you will notice little improvement. So your job is to find the right trade-off between latency and accuracy, depending on your application.

If your model contains other layers that behave in a special way during training (such as `BatchNormalization` layers), then you should not force training mode like we just did. Instead, you should replace the `Dropout` layers with the following `MCDropout` class:[27]

```
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

Here, we just subclass the `Dropout` layer and override the `call()` method to force its `training` argument to `True` (see Chapter 12). Similarly, you could define an `MCAlpha Dropout` class by subclassing `AlphaDropout` instead. If you are creating a model from scratch, it's just a matter of using `MCDropout` rather than `Dropout`. But if you have a model that was already trained using `Dropout`, you need to create a new model that's identical to the existing model except that it replaces the `Dropout` layers with `MCDrop out`, then copy the existing model's weights to your new model.

In short, MC Dropout is a fantastic technique that boosts dropout models and provides better uncertainty estimates. And of course, since it is just regular dropout during training, it also acts like a regularizer.

Max-Norm Regularization

Another regularization technique that is popular for neural networks is called *max-norm regularization*: for each neuron, it constrains the weights $\mathbf{w}$ of the incoming connections such that $\| \mathbf{w} \|_2 \leq r$, where r is the max-norm hyperparameter and $\| \cdot \|_2$ is the ℓ_2 norm.

Max-norm regularization does not add a regularization loss term to the overall loss function. Instead, it is typically implemented by computing $\|\mathbf{w}\|_2$ after each training step and rescaling $\mathbf{w}$ if needed ($\mathbf{w} \leftarrow \mathbf{w}\, r/\| \mathbf{w} \|_2$).

27 This `MCDropout` class will work with all Keras APIs, including the Sequential API. If you only care about the Functional API or the Subclassing API, you do not have to create an `MCDropout` class; you can create a regular `Dropout` layer and call it with `training=True`.

Reducing *r* increases the amount of regularization and helps reduce overfitting. Max-norm regularization can also help alleviate the unstable gradients problems (if you are not using Batch Normalization).

To implement max-norm regularization in Keras, set the `kernel_constraint` argument of each hidden layer to a `max_norm()` constraint with the appropriate max value, like this:

```
keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal",
                   kernel_constraint=keras.constraints.max_norm(1.))
```

After each training iteration, the model's `fit()` method will call the object returned by `max_norm()`, passing it the layer's weights and getting rescaled weights in return, which then replace the layer's weights. As you'll see in Chapter 12, you can define your own custom constraint function if necessary and use it as the `kernel_constraint`. You can also constrain the bias terms by setting the `bias_constraint` argument.

The `max_norm()` function has an `axis` argument that defaults to `0`. A `Dense` layer usually has weights of shape [*number of inputs, number of neurons*], so using `axis=0` means that the max-norm constraint will apply independently to each neuron's weight vector. If you want to use max-norm with convolutional layers (see Chapter 14), make sure to set the `max_norm()` constraint's `axis` argument appropriately (usually `axis=[0, 1, 2]`).

Summary and Practical Guidelines

In this chapter we have covered a wide range of techniques, and you may be wondering which ones you should use. This depends on the task, and there is no clear consensus yet, but I have found the configuration in Table 11-3 to work fine in most cases, without requiring much hyperparameter tuning. That said, please do not consider these defaults as hard rules!

Table 11-3. Default DNN configuration

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ELU
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping (+ℓ_2 reg. if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

If the network is a simple stack of dense layers, then it can self-normalize, and you should use the configuration in Table 11-4 instead.

Table 11-4. DNN configuration for a self-normalizing net

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

Don't forget to normalize the input features! You should also try to reuse parts of a pretrained neural network if you can find one that solves a similar problem, or use unsupervised pretraining if you have a lot of unlabeled data, or use pretraining on an auxiliary task if you have a lot of labeled data for a similar task.

While the previous guidelines should cover most cases, here are some exceptions:

- If you need a sparse model, you can use ℓ_1 regularization (and optionally zero out the tiny weights after training). If you need an even sparser model, you can use the TensorFlow Model Optimization Toolkit. This will break self-normalization, so you should use the default configuration in this case.
- If you need a low-latency model (one that performs lightning-fast predictions), you may need to use fewer layers, fold the Batch Normalization layers into the previous layers, and possibly use a faster activation function such as leaky ReLU or just ReLU. Having a sparse model will also help. Finally, you may want to reduce the float precision from 32 bits to 16 or even 8 bits (see "Deploying a Model to a Mobile or Embedded Device" on page 685). Again, check out TF-MOT.
- If you are building a risk-sensitive application, or inference latency is not very important in your application, you can use MC Dropout to boost performance and get more reliable probability estimates, along with uncertainty estimates.

With these guidelines, you are now ready to train very deep nets! I hope you are now convinced that you can go quite a long way using just Keras. There may come a time, however, when you need to have even more control; for example, to write a custom loss function or to tweak the training algorithm. For such cases you will need to use TensorFlow's lower-level API, as you will see in the next chapter.

Exercises

1. Is it OK to initialize all the weights to the same value as long as that value is selected randomly using He initialization?
2. Is it OK to initialize the bias terms to 0?
3. Name three advantages of the SELU activation function over ReLU.
4. In which cases would you want to use each of the following activation functions: SELU, leaky ReLU (and its variants), ReLU, tanh, logistic, and softmax?
5. What may happen if you set the `momentum` hyperparameter too close to 1 (e.g., 0.99999) when using an `SGD` optimizer?
6. Name three ways you can produce a sparse model.
7. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What about MC Dropout?
8. Practice training a deep neural network on the CIFAR10 image dataset:
   a. Build a DNN with 20 hidden layers of 100 neurons each (that's too many, but it's the point of this exercise). Use He initialization and the ELU activation function.
   b. Using Nadam optimization and early stopping, train the network on the CIFAR10 dataset. You can load it with `keras.datasets.cifar10.load_data()`. The dataset is composed of 60,000 32 × 32–pixel color images (50,000 for training, 10,000 for testing) with 10 classes, so you'll need a softmax output layer with 10 neurons. Remember to search for the right learning rate each time you change the model's architecture or hyperparameters.
   c. Now try adding Batch Normalization and compare the learning curves: Is it converging faster than before? Does it produce a better model? How does it affect training speed?
   d. Try replacing Batch Normalization with SELU, and make the necessary adjustements to ensure the network self-normalizes (i.e., standardize the input features, use LeCun normal initialization, make sure the DNN contains only a sequence of dense layers, etc.).
   e. Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC Dropout.
   f. Retrain your model using 1cycle scheduling and see if it improves training speed and model accuracy.

Solutions to these exercises are available in Appendix A.

CHAPTER 12

Custom Models and Training with TensorFlow

Up until now, we've used only TensorFlow's high-level API, tf.keras, but it already got us pretty far: we built various neural network architectures, including regression and classification nets, Wide & Deep nets, and self-normalizing nets, using all sorts of techniques, such as Batch Normalization, dropout, and learning rate schedules. In fact, 95% of the use cases you will encounter will not require anything other than tf.keras (and tf.data; see Chapter 13). But now it's time to dive deeper into TensorFlow and take a look at its lower-level Python API. This will be useful when you need extra control to write custom loss functions, custom metrics, layers, models, initializers, regularizers, weight constraints, and more. You may even need to fully control the training loop itself, for example to apply special transformations or constraints to the gradients (beyond just clipping them) or to use multiple optimizers for different parts of the network. We will cover all these cases in this chapter, and we will also look at how you can boost your custom models and training algorithms using TensorFlow's automatic graph generation feature. But first, let's take a quick tour of TensorFlow.

TensorFlow 2.0 (beta) was released in June 2019, making TensorFlow much easier to use. The first edition of this book used TF 1, while this edition uses TF 2.

A Quick Tour of TensorFlow

As you know, TensorFlow is a powerful library for numerical computation, particularly well suited and fine-tuned for large-scale Machine Learning (but you could use it for anything else that requires heavy computations). It was developed by the Google Brain team and it powers many of Google's large-scale services, such as Google Cloud Speech, Google Photos, and Google Search. It was open sourced in November 2015, and it is now the most popular Deep Learning library (in terms of citations in papers, adoption in companies, stars on GitHub, etc.). Countless projects use TensorFlow for all sorts of Machine Learning tasks, such as image classification, natural language processing, recommender systems, and time series forecasting.

So what does TensorFlow offer? Here's a summary:

- Its core is very similar to NumPy, but with GPU support.
- It supports distributed computing (across multiple devices and servers).
- It includes a kind of just-in-time (JIT) compiler that allows it to optimize computations for speed and memory usage. It works by extracting the *computation graph* from a Python function, then optimizing it (e.g., by pruning unused nodes), and finally running it efficiently (e.g., by automatically running independent operations in parallel).
- Computation graphs can be exported to a portable format, so you can train a TensorFlow model in one environment (e.g., using Python on Linux) and run it in another (e.g., using Java on an Android device).
- It implements autodiff (see Chapter 10 and Appendix D) and provides some excellent optimizers, such as RMSProp and Nadam (see Chapter 11), so you can easily minimize all sorts of loss functions.

TensorFlow offers many more features built on top of these core features: the most important is of course `tf.keras`,[1] but it also has data loading and preprocessing ops (`tf.data`, `tf.io`, etc.), image processing ops (`tf.image`), signal processing ops (`tf.signal`), and more (see Figure 12-1 for an overview of TensorFlow's Python API).

1 TensorFlow includes another Deep Learning API called the *Estimators API*, but the TensorFlow team recommends using `tf.keras` instead.

We will cover many of the packages and functions of the TensorFlow API, but it's impossible to cover them all, so you should really take some time to browse through the API; you will find that it is quite rich and well documented.

Figure 12-1. TensorFlow's Python API

At the lowest level, each TensorFlow operation (*op* for short) is implemented using highly efficient C++ code.[2] Many operations have multiple implementations called *kernels*: each kernel is dedicated to a specific device type, such as CPUs, GPUs, or even TPUs (*tensor processing units*). As you may know, GPUs can dramatically speed up computations by splitting them into many smaller chunks and running them in parallel across many GPU threads. TPUs are even faster: they are custom ASIC chips built specifically for Deep Learning operations[3] (we will discuss how to use TensorFlow with GPUs or TPUs in Chapter 19).

TensorFlow's architecture is shown in Figure 12-2. Most of the time your code will use the high-level APIs (especially tf.keras and tf.data); but when you need more flexibility, you will use the lower-level Python API, handling tensors directly. Note that

2 If you ever need to (but you probably won't), you can write your own operations using the C++ API.

3 To learn more about TPUs and how they work, check out *https://homl.info/tpus*.

APIs for other languages are also available. In any case, TensorFlow's execution engine will take care of running the operations efficiently, even across multiple devices and machines if you tell it to.

Figure 12-2. TensorFlow's architecture

TensorFlow runs not only on Windows, Linux, and macOS, but also on mobile devices (using *TensorFlow Lite*), including both iOS and Android (see Chapter 19). If you do not want to use the Python API, there are C++, Java, Go, and Swift APIs. There is even a JavaScript implementation called *TensorFlow.js* that makes it possible to run your models directly in your browser.

There's more to TensorFlow than the library. TensorFlow is at the center of an extensive ecosystem of libraries. First, there's TensorBoard for visualization (see Chapter 10). Next, there's TensorFlow Extended (TFX), which is a set of libraries built by Google to productionize TensorFlow projects: it includes tools for data validation, preprocessing, model analysis, and serving (with TF Serving; see Chapter 19). Google's *TensorFlow Hub* provides a way to easily download and reuse pretrained neural networks. You can also get many neural network architectures, some of them pretrained, in TensorFlow's model garden. Check out the TensorFlow Resources and *https://github.com/jtoy/awesome-tensorflow* for more TensorFlow-based projects. You will find hundreds of TensorFlow projects on GitHub, so it is often easy to find existing code for whatever you are trying to do.

More and more ML papers are released along with their implementations, and sometimes even with pretrained models. Check out *https://paperswithcode.com/* to easily find them.

Last but not least, TensorFlow has a dedicated team of passionate and helpful developers, as well as a large community contributing to improving it. To ask technical questions, you should use *http://stackoverflow.com/* and tag your question with *tensorflow* and *python*. You can file bugs and feature requests through GitHub. For general discussions, join the Google group.

OK, it's time to start coding!

Using TensorFlow like NumPy

TensorFlow's API revolves around *tensors*, which flow from operation to operation—hence the name Tensor*Flow*. A tensor is very similar to a NumPy `ndarray`: it is usually a multidimensional array, but it can also hold a scalar (a simple value, such as `42`). These tensors will be important when we create custom cost functions, custom metrics, custom layers, and more, so let's see how to create and manipulate them.

Tensors and Operations

You can create a tensor with `tf.constant()`. For example, here is a tensor representing a matrix with two rows and three columns of floats:

```
>>> tf.constant([[1., 2., 3.], [4., 5., 6.]]) # matrix
<tf.Tensor: id=0, shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
>>> tf.constant(42) # scalar
<tf.Tensor: id=1, shape=(), dtype=int32, numpy=42>
```

Just like an `ndarray`, a `tf.Tensor` has a shape and a data type (`dtype`):

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> t.shape
TensorShape([2, 3])
>>> t.dtype
tf.float32
```

Indexing works much like in NumPy:

```
>>> t[:, 1:]
<tf.Tensor: id=5, shape=(2, 2), dtype=float32, numpy=
array([[2., 3.],
       [5., 6.]], dtype=float32)>
>>> t[..., 1, tf.newaxis]
<tf.Tensor: id=15, shape=(2, 1), dtype=float32, numpy=
array([[2.],
       [5.]], dtype=float32)>
```

Most importantly, all sorts of tensor operations are available:

```
>>> t + 10
<tf.Tensor: id=18, shape=(2, 3), dtype=float32, numpy=
```

```
array([[11., 12., 13.],
       [14., 15., 16.]], dtype=float32)>
>>> tf.square(t)
<tf.Tensor: id=20, shape=(2, 3), dtype=float32, numpy=
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)>
>>> t @ tf.transpose(t)
<tf.Tensor: id=24, shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

Note that writing `t + 10` is equivalent to calling `tf.add(t, 10)` (indeed, Python calls the magic method `t.__add__(10)`, which just calls `tf.add(t, 10)`). Other operators like `-` and `*` are also supported. The `@` operator was added in Python 3.5, for matrix multiplication: it is equivalent to calling the `tf.matmul()` function.

You will find all the basic math operations you need (`tf.add()`, `tf.multiply()`, `tf.square()`, `tf.exp()`, `tf.sqrt()`, etc.) and most operations that you can find in NumPy (e.g., `tf.reshape()`, `tf.squeeze()`, `tf.tile()`). Some functions have a different name than in NumPy; for instance, `tf.reduce_mean()`, `tf.reduce_sum()`, `tf.reduce_max()`, and `tf.math.log()` are the equivalent of `np.mean()`, `np.sum()`, `np.max()` and `np.log()`. When the name differs, there is often a good reason for it. For example, in TensorFlow you must write `tf.transpose(t)`; you cannot just write `t.T` like in NumPy. The reason is that the `tf.transpose()` function does not do exactly the same thing as NumPy's `T` attribute: in TensorFlow, a new tensor is created with its own copy of the transposed data, while in NumPy, `t.T` is just a transposed view on the same data. Similarly, the `tf.reduce_sum()` operation is named this way because its GPU kernel (i.e., GPU implementation) uses a reduce algorithm that does not guarantee the order in which the elements are added: because 32-bit floats have limited precision, the result may change ever so slightly every time you call this operation. The same is true of `tf.reduce_mean()` (but of course `tf.reduce_max()` is deterministic).

Many functions and classes have aliases. For example, `tf.add()` and `tf.math.add()` are the same function. This allows TensorFlow to have concise names for the most common operations[4] while preserving well-organized packages.

4 A notable exception is `tf.math.log()`, which is commonly used but doesn't have a `tf.log()` alias (as it might be confused with logging).

Keras' Low-Level API

The Keras API has its own low-level API, located in `keras.backend`. It includes functions like `square()`, `exp()`, and `sqrt()`. In tf.keras, these functions generally just call the corresponding TensorFlow operations. If you want to write code that will be portable to other Keras implementations, you should use these Keras functions. However, they only cover a subset of all functions available in TensorFlow, so in this book we will use the TensorFlow operations directly. Here is as simple example using `keras.backend`, which is commonly named `K` for short:

```
>>> from tensorflow import keras
>>> K = keras.backend
>>> K.square(K.transpose(t)) + 10
<tf.Tensor: id=39, shape=(3, 2), dtype=float32, numpy=
array([[11., 26.],
       [14., 35.],
       [19., 46.]], dtype=float32)>
```

Tensors and NumPy

Tensors play nice with NumPy: you can create a tensor from a NumPy array, and vice versa. You can even apply TensorFlow operations to NumPy arrays and NumPy operations to tensors:

```
>>> a = np.array([2., 4., 5.])
>>> tf.constant(a)
<tf.Tensor: id=111, shape=(3,), dtype=float64, numpy=array([2., 4., 5.])>
>>> t.numpy() # or np.array(t)
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
>>> tf.square(a)
<tf.Tensor: id=116, shape=(3,), dtype=float64, numpy=array([4., 16., 25.])>
>>> np.square(t)
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)
```

Notice that NumPy uses 64-bit precision by default, while TensorFlow uses 32-bit. This is because 32-bit precision is generally more than enough for neural networks, plus it runs faster and uses less RAM. So when you create a tensor from a NumPy array, make sure to set `dtype=tf.float32`.

Type Conversions

Type conversions can significantly hurt performance, and they can easily go unnoticed when they are done automatically. To avoid this, TensorFlow does not perform

any type conversions automatically: it just raises an exception if you try to execute an operation on tensors with incompatible types. For example, you cannot add a float tensor and an integer tensor, and you cannot even add a 32-bit float and a 64-bit float:

```
>>> tf.constant(2.) + tf.constant(40)
Traceback[...]InvalidArgumentError[...]expected to be a float[...]
>>> tf.constant(2.) + tf.constant(40., dtype=tf.float64)
Traceback[...]InvalidArgumentError[...]expected to be a double[...]
```

This may be a bit annoying at first, but remember that it's for a good cause! And of course you can use `tf.cast()` when you really need to convert types:

```
>>> t2 = tf.constant(40., dtype=tf.float64)
>>> tf.constant(2.0) + tf.cast(t2, tf.float32)
<tf.Tensor: id=136, shape=(), dtype=float32, numpy=42.0>
```

Variables

The `tf.Tensor` values we've seen so far are immutable: you cannot modify them. This means that we cannot use regular tensors to implement weights in a neural network, since they need to be tweaked by backpropagation. Plus, other parameters may also need to change over time (e.g., a momentum optimizer keeps track of past gradients). What we need is a `tf.Variable`:

```
>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
>>> v
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

A `tf.Variable` acts much like a `tf.Tensor`: you can perform the same operations with it, it plays nicely with NumPy as well, and it is just as picky with types. But it can also be modified in place using the `assign()` method (or `assign_add()` or `assign_sub()`, which increment or decrement the variable by the given value). You can also modify individual cells (or slices), by using the cell's (or slice's) `assign()` method (direct item assignment will not work) or by using the `scatter_update()` or `scatter_nd_update()` methods:

```
v.assign(2 * v)           # => [[2., 4., 6.], [8., 10., 12.]]
v[0, 1].assign(42)        # => [[2., 42., 6.], [8., 10., 12.]]
v[:, 2].assign([0., 1.])  # => [[2., 42., 0.], [8., 10., 1.]]
v.scatter_nd_update(indices=[[0, 0], [1, 2]], updates=[100., 200.])
                          # => [[100., 42., 0.], [8., 10., 200.]]
```

In practice you will rarely have to create variables manually, since Keras provides an `add_weight()` method that will take care of it for you, as we will see. Moreover, model parameters will generally be updated directly by the optimizers, so you will rarely need to update variables manually.

Other Data Structures

TensorFlow supports several other data structures, including the following (please see the "Tensors and Operations" section in the notebook or Appendix F for more details):

Sparse tensors (`tf.SparseTensor`)
: Efficiently represent tensors containing mostly zeros. The `tf.sparse` package contains operations for sparse tensors.

Tensor arrays (`tf.TensorArray`)
: Are lists of tensors. They have a fixed size by default but can optionally be made dynamic. All tensors they contain must have the same shape and data type.

Ragged tensors (`tf.RaggedTensor`)
: Represent static lists of lists of tensors, where every tensor has the same shape and data type. The `tf.ragged` package contains operations for ragged tensors.

String tensors
: Are regular tensors of type `tf.string`. These represent byte strings, not Unicode strings, so if you create a string tensor using a Unicode string (e.g., a regular Python 3 string like `"café"`), then it will get encoded to UTF-8 automatically (e.g., `b"caf\xc3\xa9"`). Alternatively, you can represent Unicode strings using tensors of type `tf.int32`, where each item represents a Unicode code point (e.g., `[99, 97, 102, 233]`). The `tf.strings` package (with an `s`) contains ops for byte strings and Unicode strings (and to convert one into the other). It's important to note that a `tf.string` is atomic, meaning that its length does not appear in the tensor's shape. Once you convert it to a Unicode tensor (i.e., a tensor of type `tf.int32` holding Unicode code points), the length appears in the shape.

Sets
: Are represented as regular tensors (or sparse tensors). For example, `tf.constant([[1, 2], [3, 4]])` represents the two sets {1, 2} and {3, 4}. More generally, each set is represented by a vector in the tensor's last axis. You can manipulate sets using operations from the `tf.sets` package.

Queues
: Store tensors across multiple steps. TensorFlow offers various kinds of queues: simple First In, First Out (FIFO) queues (FIFOQueue), queues that can prioritize

some items (`PriorityQueue`), shuffle their items (`RandomShuffleQueue`), and batch items of different shapes by padding (`PaddingFIFOQueue`). These classes are all in the `tf.queue` package.

With tensors, operations, variables, and various data structures at your disposal, you are now ready to customize your models and training algorithms!

Customizing Models and Training Algorithms

Let's start by creating a custom loss function, which is a simple and common use case.

Custom Loss Functions

Suppose you want to train a regression model, but your training set is a bit noisy. Of course, you start by trying to clean up your dataset by removing or fixing the outliers, but that turns out to be insufficient; the dataset is still noisy. Which loss function should you use? The mean squared error might penalize large errors too much and cause your model to be imprecise. The mean absolute error would not penalize outliers as much, but training might take a while to converge, and the trained model might not be very precise. This is probably a good time to use the Huber loss (introduced in Chapter 10) instead of the good old MSE. The Huber loss is not currently part of the official Keras API, but it is available in tf.keras (just use an instance of the `keras.losses.Huber` class). But let's pretend it's not there: implementing it is easy as pie! Just create a function that takes the labels and predictions as arguments, and use TensorFlow operations to compute every instance's loss:

```
def huber_fn(y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) < 1
    squared_loss = tf.square(error) / 2
    linear_loss  = tf.abs(error) - 0.5
    return tf.where(is_small_error, squared_loss, linear_loss)
```

For better performance, you should use a vectorized implementation, as in this example. Moreover, if you want to benefit from TensorFlow's graph features, you should use only TensorFlow operations.

It is also preferable to return a tensor containing one loss per instance, rather than returning the mean loss. This way, Keras can apply class weights or sample weights when requested (see Chapter 10).

Now you can use this loss when you compile the Keras model, then train your model:

```
model.compile(loss=huber_fn, optimizer="nadam")
model.fit(X_train, y_train, [...])
```

And that's it! For each batch during training, Keras will call the `huber_fn()` function to compute the loss and use it to perform a Gradient Descent step. Moreover, it will keep track of the total loss since the beginning of the epoch, and it will display the mean loss.

But what happens to this custom loss when you save the model?

Saving and Loading Models That Contain Custom Components

Saving a model containing a custom loss function works fine, as Keras saves the name of the function. Whenever you load it, you'll need to provide a dictionary that maps the function name to the actual function. More generally, when you load a model containing custom objects, you need to map the names to the objects:

```
model = keras.models.load_model("my_model_with_a_custom_loss.h5",
                                custom_objects={"huber_fn": huber_fn})
```

With the current implementation, any error between −1 and 1 is considered "small." But what if you want a different threshold? One solution is to create a function that creates a configured loss function:

```
def create_huber(threshold=1.0):
    def huber_fn(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < threshold
        squared_loss = tf.square(error) / 2
        linear_loss  = threshold * tf.abs(error) - threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    return huber_fn

model.compile(loss=create_huber(2.0), optimizer="nadam")
```

Unfortunately, when you save the model, the `threshold` will not be saved. This means that you will have to specify the `threshold` value when loading the model (note that the name to use is `"huber_fn"`, which is the name of the function you gave Keras, not the name of the function that created it):

```
model = keras.models.load_model("my_model_with_a_custom_loss_threshold_2.h5",
                                custom_objects={"huber_fn": create_huber(2.0)})
```

You can solve this by creating a subclass of the `keras.losses.Loss` class, and then implementing its `get_config()` method:

```
class HuberLoss(keras.losses.Loss):
    def __init__(self, threshold=1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)
    def call(self, y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < self.threshold
        squared_loss = tf.square(error) / 2
        linear_loss  = self.threshold * tf.abs(error) - self.threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

The Keras API currently only specifies how to use subclassing to define layers, models, callbacks, and regularizers. If you build other components (such as losses, metrics, initializers, or constraints) using subclassing, they may not be portable to other Keras implementations. It's likely that the Keras API will be updated to specify subclassing for all these components as well.

Let's walk through this code:

- The constructor accepts `**kwargs` and passes them to the parent constructor, which handles standard hyperparameters: the `name` of the loss and the `reduction` algorithm to use to aggregate the individual instance losses. By default, it is `"sum_over_batch_size"`, which means that the loss will be the sum of the instance losses, weighted by the sample weights, if any, and divided by the batch size (not by the sum of weights, so this is *not* the weighted mean).[5] Other possible values are `"sum"` and `"none"`.
- The `call()` method takes the labels and predictions, computes all the instance losses, and returns them.
- The `get_config()` method returns a dictionary mapping each hyperparameter name to its value. It first calls the parent class's `get_config()` method, then adds the new hyperparameters to this dictionary (note that the convenient `{**x}` syntax was added in Python 3.5).

You can then use any instance of this class when you compile the model:

```
model.compile(loss=HuberLoss(2.), optimizer="nadam")
```

5 It would not be a good idea to use a weighted mean: if you did, then two instances with the same weight but in different batches would have a different impact on training, depending on the total weight of each batch.

When you save the model, the threshold will be saved along with it; and when you load the model, you just need to map the class name to the class itself:

```
model = keras.models.load_model("my_model_with_a_custom_loss_class.h5",
                                custom_objects={"HuberLoss": HuberLoss})
```

When you save a model, Keras calls the loss instance's `get_config()` method and saves the config as JSON in the HDF5 file. When you load the model, it calls the `from_config()` class method on the `HuberLoss` class: this method is implemented by the base class (`Loss`) and creates an instance of the class, passing `**config` to the constructor.

That's it for losses! That wasn't too hard, was it? Just as simple are custom activation functions, initializers, regularizers, and constraints. Let's look at these now.

Custom Activation Functions, Initializers, Regularizers, and Constraints

Most Keras functionalities, such as losses, regularizers, constraints, initializers, metrics, activation functions, layers, and even full models, can be customized in very much the same way. Most of the time, you will just need to write a simple function with the appropriate inputs and outputs. Here are examples of a custom activation function (equivalent to `keras.activations.softplus()` or `tf.nn.softplus()`), a custom Glorot initializer (equivalent to `keras.initializers.glorot_normal()`), a custom ℓ_1 regularizer (equivalent to `keras.regularizers.l1(0.01)`), and a custom constraint that ensures weights are all positive (equivalent to `keras.constraints.nonneg()` or `tf.nn.relu()`):

```
def my_softplus(z): # return value is just tf.nn.softplus(z)
    return tf.math.log(tf.exp(z) + 1.0)

def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)

def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))

def my_positive_weights(weights): # return value is just tf.nn.relu(weights)
    return tf.where(weights < 0., tf.zeros_like(weights), weights)
```

As you can see, the arguments depend on the type of custom function. These custom functions can then be used normally; for example:

```
layer = keras.layers.Dense(30, activation=my_softplus,
                           kernel_initializer=my_glorot_initializer,
                           kernel_regularizer=my_l1_regularizer,
                           kernel_constraint=my_positive_weights)
```

The activation function will be applied to the output of this `Dense` layer, and its result will be passed on to the next layer. The layer's weights will be initialized using the value returned by the initializer. At each training step the weights will be passed to the regularization function to compute the regularization loss, which will be added to the main loss to get the final loss used for training. Finally, the constraint function will be called after each training step, and the layer's weights will be replaced by the constrained weights.

If a function has hyperparameters that need to be saved along with the model, then you will want to subclass the appropriate class, such as `keras.regularizers.Regularizer`, `keras.constraints.Constraint`, `keras.initializers.Initializer`, or `keras.layers.Layer` (for any layer, including activation functions). Much like we did for the custom loss, here is a simple class for ℓ_1 regularization that saves its `factor` hyperparameter (this time we do not need to call the parent constructor or the `get_config()` method, as they are not defined by the parent class):

```
class MyL1Regularizer(keras.regularizers.Regularizer):
    def __init__(self, factor):
        self.factor = factor
    def __call__(self, weights):
        return tf.reduce_sum(tf.abs(self.factor * weights))
    def get_config(self):
        return {"factor": self.factor}
```

Note that you must implement the `call()` method for losses, layers (including activation functions), and models, or the `__call__()` method for regularizers, initializers, and constraints. For metrics, things are a bit different, as we will see now.

Custom Metrics

Losses and metrics are conceptually not the same thing: losses (e.g., cross entropy) are used by Gradient Descent to *train* a model, so they must be differentiable (at least where they are evaluated), and their gradients should not be 0 everywhere. Plus, it's OK if they are not easily interpretable by humans. In contrast, metrics (e.g., accuracy) are used to *evaluate* a model: they must be more easily interpretable, and they can be non-differentiable or have 0 gradients everywhere.

That said, in most cases, defining a custom metric function is exactly the same as defining a custom loss function. In fact, we could even use the Huber loss function we created earlier as a metric;[6] it would work just fine (and persistence would also work the same way, in this case only saving the name of the function, `"huber_fn"`):

6 However, the Huber loss is seldom used as a metric (the MAE or MSE is preferred).

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])
```

For each batch during training, Keras will compute this metric and keep track of its mean since the beginning of the epoch. Most of the time, this is exactly what you want. But not always! Consider a binary classifier's precision, for example. As we saw in Chapter 3, precision is the number of true positives divided by the number of positive predictions (including both true positives and false positives). Suppose the model made five positive predictions in the first batch, four of which were correct: that's 80% precision. Then suppose the model made three positive predictions in the second batch, but they were all incorrect: that's 0% precision for the second batch. If you just compute the mean of these two precisions, you get 40%. But wait a second—that's *not* the model's precision over these two batches! Indeed, there were a total of four true positives (4 + 0) out of eight positive predictions (5 + 3), so the overall precision is 50%, not 40%. What we need is an object that can keep track of the number of true positives and the number of false positives and that can compute their ratio when requested. This is precisely what the `keras.metrics.Precision` class does:

```
>>> precision = keras.metrics.Precision()
>>> precision([0, 1, 1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1])
<tf.Tensor: id=581729, shape=(), dtype=float32, numpy=0.8>
>>> precision([0, 1, 0, 0, 1, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0])
<tf.Tensor: id=581780, shape=(), dtype=float32, numpy=0.5>
```

In this example, we created a `Precision` object, then we used it like a function, passing it the labels and predictions for the first batch, then for the second batch (note that we could also have passed sample weights). We used the same number of true and false positives as in the example we just discussed. After the first batch, it returns a precision of 80%; then after the second batch, it returns 50% (which is the overall precision so far, not the second batch's precision). This is called a *streaming metric* (or *stateful metric*), as it is gradually updated, batch after batch.

At any point, we can call the `result()` method to get the current value of the metric. We can also look at its variables (tracking the number of true and false positives) by using the `variables` attribute, and we can reset these variables using the `reset_states()` method:

```
>>> precision.result()
<tf.Tensor: id=581794, shape=(), dtype=float32, numpy=0.5>
>>> precision.variables
[<tf.Variable 'true_positives:0' [...] numpy=array([4.], dtype=float32)>,
 <tf.Variable 'false_positives:0' [...] numpy=array([4.], dtype=float32)>]
>>> precision.reset_states() # both variables get reset to 0.0
```

If you need to create such a streaming metric, create a subclass of the `keras.metrics.Metric` class. Here is a simple example that keeps track of the total Huber loss

and the number of instances seen so far. When asked for the result, it returns the ratio, which is simply the mean Huber loss:

```
class HuberMetric(keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs) # handles base args (e.g., dtype)
        self.threshold = threshold
        self.huber_fn = create_huber(threshold)
        self.total = self.add_weight("total", initializer="zeros")
        self.count = self.add_weight("count", initializer="zeros")
    def update_state(self, y_true, y_pred, sample_weight=None):
        metric = self.huber_fn(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(metric))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))
    def result(self):
        return self.total / self.count
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```

Let's walk through this code:[7]

- The constructor uses the `add_weight()` method to create the variables needed to keep track of the metric's state over multiple batches—in this case, the sum of all Huber losses (`total`) and the number of instances seen so far (`count`). You could just create variables manually if you preferred. Keras tracks any `tf.Variable` that is set as an attribute (and more generally, any "trackable" object, such as layers or models).
- The `update_state()` method is called when you use an instance of this class as a function (as we did with the `Precision` object). It updates the variables, given the labels and predictions for one batch (and sample weights, but in this case we ignore them).
- The `result()` method computes and returns the final result, in this case the mean Huber metric over all instances. When you use the metric as a function, the `update_state()` method gets called first, then the `result()` method is called, and its output is returned.
- We also implement the `get_config()` method to ensure the `threshold` gets saved along with the model.
- The default implementation of the `reset_states()` method resets all variables to 0.0 (but you can override it if needed).

7 This class is for illustration purposes only. A simpler and better implementation would just subclass the `keras.metrics.Mean` class; see the "Streaming metrics" section of the notebook for an example.

Keras will take care of variable persistence seamlessly; no action is required.

When you define a metric using a simple function, Keras automatically calls it for each batch, and it keeps track of the mean during each epoch, just like we did manually. So the only benefit of our `HuberMetric` class is that the `threshold` will be saved. But of course, some metrics, like precision, cannot simply be averaged over batches: in those cases, there's no other option than to implement a streaming metric.

Now that we have built a streaming metric, building a custom layer will seem like a walk in the park!

Custom Layers

You may occasionally want to build an architecture that contains an exotic layer for which TensorFlow does not provide a default implementation. In this case, you will need to create a custom layer. Or you may simply want to build a very repetitive architecture, containing identical blocks of layers repeated many times, and it would be convenient to treat each block of layers as a single layer. For example, if the model is a sequence of layers A, B, C, A, B, C, A, B, C, then you might want to define a custom layer D containing layers A, B, C, so your model would then simply be D, D, D. Let's see how to build custom layers.

First, some layers have no weights, such as `keras.layers.Flatten` or `keras.layers.ReLU`. If you want to create a custom layer without any weights, the simplest option is to write a function and wrap it in a `keras.layers.Lambda` layer. For example, the following layer will apply the exponential function to its inputs:

```
exponential_layer = keras.layers.Lambda(lambda x: tf.exp(x))
```

This custom layer can then be used like any other layer, using the Sequential API, the Functional API, or the Subclassing API. You can also use it as an activation function (or you could use `activation=tf.exp`, `activation=keras.activations.exponential`, or simply `activation="exponential"`). The exponential layer is sometimes used in the output layer of a regression model when the values to predict have very different scales (e.g., 0.001, 10., 1,000.).

As you've probably guessed by now, to build a custom stateful layer (i.e., a layer with weights), you need to create a subclass of the `keras.layers.Layer` class. For example, the following class implements a simplified version of the `Dense` layer:

```
class MyDense(keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = keras.activations.get(activation)

    def build(self, batch_input_shape):
        self.kernel = self.add_weight(
            name="kernel", shape=[batch_input_shape[-1], self.units],
            initializer="glorot_normal")
        self.bias = self.add_weight(
            name="bias", shape=[self.units], initializer="zeros")
        super().build(batch_input_shape) # must be at the end

    def call(self, X):
        return self.activation(X @ self.kernel + self.bias)

    def compute_output_shape(self, batch_input_shape):
        return tf.TensorShape(batch_input_shape.as_list()[:-1] + [self.units])

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "units": self.units,
                "activation": keras.activations.serialize(self.activation)}
```

Let's walk through this code:

- The constructor takes all the hyperparameters as arguments (in this example, `units` and `activation`), and importantly it also takes a `**kwargs` argument. It calls the parent constructor, passing it the `kwargs`: this takes care of standard arguments such as `input_shape`, `trainable`, and `name`. Then it saves the hyperparameters as attributes, converting the `activation` argument to the appropriate activation function using the `keras.activations.get()` function (it accepts functions, standard strings like `"relu"` or `"selu"`, or simply `None`).[8]
- The `build()` method's role is to create the layer's variables by calling the `add_weight()` method for each weight. The `build()` method is called the first time the layer is used. At that point, Keras will know the shape of this layer's inputs, and it will pass it to the `build()` method,[9] which is often necessary to create some of the weights. For example, we need to know the number of neurons in the previous layer in order to create the connection weights matrix (i.e., the `"kernel"`): this corresponds to the size of the last dimension of the inputs. At the end of the `build()` method (and only at the end), you must call the parent's

8 This function is specific to tf.keras. You could use `keras.layers.Activation` instead.

9 The Keras API calls this argument `input_shape`, but since it also includes the batch dimension, I prefer to call it `batch_input_shape`. Same for `compute_output_shape()`.

`build()` method: this tells Keras that the layer is built (it just sets `self.built=True`).

- The `call()` method performs the desired operations. In this case, we compute the matrix multiplication of the inputs `X` and the layer's kernel, we add the bias vector, and we apply the activation function to the result, and this gives us the output of the layer.
- The `compute_output_shape()` method simply returns the shape of this layer's outputs. In this case, it is the same shape as the inputs, except the last dimension is replaced with the number of neurons in the layer. Note that in tf.keras, shapes are instances of the `tf.TensorShape` class, which you can convert to Python lists using `as_list()`.
- The `get_config()` method is just like in the previous custom classes. Note that we save the activation function's full configuration by calling `keras.activations.serialize()`.

You can now use a `MyDense` layer just like any other layer!

You can generally omit the `compute_output_shape()` method, as tf.keras automatically infers the output shape, except when the layer is dynamic (as we will see shortly). In other Keras implementations, this method is either required or its default implementation assumes the output shape is the same as the input shape.

To create a layer with multiple inputs (e.g., `Concatenate`), the argument to the `call()` method should be a tuple containing all the inputs, and similarly the argument to the `compute_output_shape()` method should be a tuple containing each input's batch shape. To create a layer with multiple outputs, the `call()` method should return the list of outputs, and `compute_output_shape()` should return the list of batch output shapes (one per output). For example, the following toy layer takes two inputs and returns three outputs:

```
class MyMultiLayer(keras.layers.Layer):
    def call(self, X):
        X1, X2 = X
        return [X1 + X2, X1 * X2, X1 / X2]

    def compute_output_shape(self, batch_input_shape):
        b1, b2 = batch_input_shape
        return [b1, b1, b1] # should probably handle broadcasting rules
```

This layer may now be used like any other layer, but of course only using the Functional and Subclassing APIs, not the Sequential API (which only accepts layers with one input and one output).

If your layer needs to have a different behavior during training and during testing (e.g., if it uses `Dropout` or `BatchNormalization` layers), then you must add a `training` argument to the `call()` method and use this argument to decide what to do. For example, let's create a layer that adds Gaussian noise during training (for regularization) but does nothing during testing (Keras has a layer that does the same thing, `keras.layers.GaussianNoise`):

```
class MyGaussianNoise(keras.layers.Layer):
    def __init__(self, stddev, **kwargs):
        super().__init__(**kwargs)
        self.stddev = stddev

    def call(self, X, training=None):
        if training:
            noise = tf.random.normal(tf.shape(X), stddev=self.stddev)
            return X + noise
        else:
            return X

    def compute_output_shape(self, batch_input_shape):
        return batch_input_shape
```

With that, you can now build any custom layer you need! Now let's create custom models.

Custom Models

We already looked at creating custom model classes in Chapter 10, when we discussed the Subclassing API.[10] It's straightforward: subclass the `keras.Model` class, create layers and variables in the constructor, and implement the `call()` method to do whatever you want the model to do. Suppose you want to build the model represented in Figure 12-3.

10 The name "Subclassing API" usually refers only to the creation of custom models by subclassing, although many other things can be created by subclassing, as we saw in this chapter.

Figure 12-3. Custom model example: an arbitrary model with a custom ResidualBlock layer containing a skip connection

The inputs go through a first dense layer, then through a *residual block* composed of two dense layers and an addition operation (as we will see in Chapter 14, a residual block adds its inputs to its outputs), then through this same residual block three more times, then through a second residual block, and the final result goes through a dense output layer. Note that this model does not make much sense; it's just an example to illustrate the fact that you can easily build any kind of model you want, even one that contains loops and skip connections. To implement this model, it is best to first create a `ResidualBlock` layer, since we are going to create a couple of identical blocks (and we might want to reuse it in another model):

```
class ResidualBlock(keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(n_neurons, activation="elu",
                                          kernel_initializer="he_normal")
                       for _ in range(n_layers)]

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        return inputs + Z
```

This layer is a bit special since it contains other layers. This is handled transparently by Keras: it automatically detects that the `hidden` attribute contains trackable objects (layers in this case), so their variables are automatically added to this layer's list of

variables. The rest of this class is self-explanatory. Next, let's use the Subclassing API to define the model itself:

```
class ResidualRegressor(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.Dense(30, activation="elu",
                                          kernel_initializer="he_normal")
        self.block1 = ResidualBlock(2, 30)
        self.block2 = ResidualBlock(2, 30)
        self.out = keras.layers.Dense(output_dim)

    def call(self, inputs):
        Z = self.hidden1(inputs)
        for _ in range(1 + 3):
            Z = self.block1(Z)
        Z = self.block2(Z)
        return self.out(Z)
```

We create the layers in the constructor and use them in the `call()` method. This model can then be used like any other model (compile it, fit it, evaluate it, and use it to make predictions). If you also want to be able to save the model using the `save()` method and load it using the `keras.models.load_model()` function, you must implement the `get_config()` method (as we did earlier) in both the `ResidualBlock` class and the `ResidualRegressor` class. Alternatively, you can save and load the weights using the `save_weights()` and `load_weights()` methods.

The `Model` class is a subclass of the `Layer` class, so models can be defined and used exactly like layers. But a model has some extra functionalities, including of course its `compile()`, `fit()`, `evaluate()`, and `predict()` methods (and a few variants), plus the `get_layers()` method (which can return any of the model's layers by name or by index) and the `save()` method (and support for `keras.models.load_model()` and `keras.models.clone_model()`).

If models provide more functionality than layers, why not just define every layer as a model? Well, technically you could, but it is usually cleaner to distinguish the internal components of your model (i.e., layers or reusable blocks of layers) from the model itself (i.e., the object you will train). The former should subclass the `Layer` class, while the latter should subclass the `Model` class.

With that, you can naturally and concisely build almost any model that you find in a paper, using the Sequential API, the Functional API, the Subclassing API, or even a mix of these. "Almost" any model? Yes, there are still a few things that we need to look

at: first, how to define losses or metrics based on model internals, and second, how to build a custom training loop.

Losses and Metrics Based on Model Internals

The custom losses and metrics we defined earlier were all based on the labels and the predictions (and optionally sample weights). There will be times when you want to define losses based on other parts of your model, such as the weights or activations of its hidden layers. This may be useful for regularization purposes or to monitor some internal aspect of your model.

To define a custom loss based on model internals, compute it based on any part of the model you want, then pass the result to the `add_loss()` method.For example, let's build a custom regression MLP model composed of a stack of five hidden layers plus an output layer. This custom model will also have an auxiliary output on top of the upper hidden layer. The loss associated to this auxiliary output will be called the *reconstruction loss* (see Chapter 17): it is the mean squared difference between the reconstruction and the inputs. By adding this reconstruction loss to the main loss, we will encourage the model to preserve as much information as possible through the hidden layers—even information that is not directly useful for the regression task itself. In practice, this loss sometimes improves generalization (it is a regularization loss). Here is the code for this custom model with a custom reconstruction loss:

```
class ReconstructingRegressor(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(30, activation="selu",
                                          kernel_initializer="lecun_normal")
                       for _ in range(5)]
        self.out = keras.layers.Dense(output_dim)

    def build(self, batch_input_shape):
        n_inputs = batch_input_shape[-1]
        self.reconstruct = keras.layers.Dense(n_inputs)
        super().build(batch_input_shape)

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        reconstruction = self.reconstruct(Z)
        recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
        self.add_loss(0.05 * recon_loss)
        return self.out(Z)
```

Let's go through this code:

- The constructor creates the DNN with five dense hidden layers and one dense output layer.
- The `build()` method creates an extra dense layer which will be used to reconstruct the inputs of the model. It must be created here because its number of units must be equal to the number of inputs, and this number is unknown before the `build()` method is called.
- The `call()` method processes the inputs through all five hidden layers, then passes the result through the reconstruction layer, which produces the reconstruction.
- Then the `call()` method computes the reconstruction loss (the mean squared difference between the reconstruction and the inputs), and adds it to the model's list of losses using the `add_loss()` method.[11] Notice that we scale down the reconstruction loss by multiplying it by 0.05 (this is a hyperparameter you can tune). This ensures that the reconstruction loss does not dominate the main loss.
- Finally, the `call()` method passes the output of the hidden layers to the output layer and returns its output.

Similarly, you can add a custom metric based on model internals by computing it in any way you want, as long as the result is the output of a metric object. For example, you can create a `keras.metrics.Mean` object in the constructor, then call it in the `call()` method, passing it the `recon_loss`, and finally add it to the model by calling the model's `add_metric()` method. This way, when you train the model, Keras will display both the mean loss over each epoch (the loss is the sum of the main loss plus 0.05 times the reconstruction loss) and the mean reconstruction error over each epoch. Both will go down during training:

```
Epoch 1/5
11610/11610 [=============] [...] loss: 4.3092 - reconstruction_error: 1.7360
Epoch 2/5
11610/11610 [=============] [...] loss: 1.1232 - reconstruction_error: 0.8964
[...]
```

In over 99% of cases, everything we have discussed so far will be sufficient to implement whatever model you want to build, even with complex architectures, losses, and metrics. However, in some rare cases you may need to customize the training loop

11 You can also call `add_loss()` on any layer inside the model, as the model recursively gathers losses from all of its layers.

itself. Before we get there, we need to look at how to compute gradients automatically in TensorFlow.

Computing Gradients Using Autodiff

To understand how to use autodiff (see Chapter 10 and Appendix D) to compute gradients automatically, let's consider a simple toy function:

```
def f(w1, w2):
    return 3 * w1 ** 2 + 2 * w1 * w2
```

If you know calculus, you can analytically find that the partial derivative of this function with regard to `w1` is `6 * w1 + 2 * w2`. You can also find that its partial derivative with regard to `w2` is `2 * w1`. For example, at the point `(w1, w2) = (5, 3)`, these partial derivatives are equal to 36 and 10, respectively, so the gradient vector at this point is (36, 10). But if this were a neural network, the function would be much more complex, typically with tens of thousands of parameters, and finding the partial derivatives analytically by hand would be an almost impossible task. One solution could be to compute an approximation of each partial derivative by measuring how much the function's output changes when you tweak the corresponding parameter:

```
>>> w1, w2 = 5, 3
>>> eps = 1e-6
>>> (f(w1 + eps, w2) - f(w1, w2)) / eps
36.000003007075065
>>> (f(w1, w2 + eps) - f(w1, w2)) / eps
10.000000003174137
```

Looks about right! This works rather well and is easy to implement, but it is just an approximation, and importantly you need to call `f()` at least once per parameter (not twice, since we could compute `f(w1, w2)` just once). Needing to call `f()` at least once per parameter makes this approach intractable for large neural networks. So instead, we should use autodiff. TensorFlow makes this pretty simple:

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
```

We first define two variables `w1` and `w2`, then we create a `tf.GradientTape` context that will automatically record every operation that involves a variable, and finally we ask this tape to compute the gradients of the result `z` with regard to both variables `[w1, w2]`. Let's take a look at the gradients that TensorFlow computed:

```
>>> gradients
[<tf.Tensor: id=828234, shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: id=828229, shape=(), dtype=float32, numpy=10.0>]
```

Perfect! Not only is the result accurate (the precision is only limited by the floating-point errors), but the `gradient()` method only goes through the recorded computations once (in reverse order), no matter how many variables there are, so it is incredibly efficient. It's like magic!

To save memory, only put the strict minimum inside the `tf.GradientTape()` block. Alternatively, pause recording by creating a `with tape.stop_recording()` block inside the `tf.GradientTape()` block.

The tape is automatically erased immediately after you call its `gradient()` method, so you will get an exception if you try to call `gradient()` twice:

```
with tf.GradientTape() as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) # => tensor 36.0
dz_dw2 = tape.gradient(z, w2) # RuntimeError!
```

If you need to call `gradient()` more than once, you must make the tape persistent and delete it each time you are done with it to free resources:[12]

```
with tf.GradientTape(persistent=True) as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) # => tensor 36.0
dz_dw2 = tape.gradient(z, w2) # => tensor 10.0, works fine now!
del tape
```

By default, the tape will only track operations involving variables, so if you try to compute the gradient of `z` with regard to anything other than a variable, the result will be `None`:

```
c1, c2 = tf.constant(5.), tf.constant(3.)
with tf.GradientTape() as tape:
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # returns [None, None]
```

However, you can force the tape to watch any tensors you like, to record every operation that involves them. You can then compute gradients with regard to these tensors, as if they were variables:

12 If the tape goes out of scope, for example when the function that used it returns, Python's garbage collector will delete it for you.

```
with tf.GradientTape() as tape:
    tape.watch(c1)
    tape.watch(c2)
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # returns [tensor 36., tensor 10.]
```

This can be useful in some cases, like if you want to implement a regularization loss that penalizes activations that vary a lot when the inputs vary little: the loss will be based on the gradient of the activations with regard to the inputs. Since the inputs are not variables, you would need to tell the tape to watch them.

Most of the time a gradient tape is used to compute the gradients of a single value (usually the loss) with regard to a set of values (usually the model parameters). This is where reverse-mode autodiff shines, as it just needs to do one forward pass and one reverse pass to get all the gradients at once. If you try to compute the gradients of a vector, for example a vector containing multiple losses, then TensorFlow will compute the gradients of the vector's sum. So if you ever need to get the individual gradients (e.g., the gradients of each loss with regard to the model parameters), you must call the tape's `jacobian()` method: it will perform reverse-mode autodiff once for each loss in the vector (all in parallel by default). It is even possible to compute second-order partial derivatives (the Hessians, i.e., the partial derivatives of the partial derivatives), but this is rarely needed in practice (see the "Computing Gradients with Autodiff" section of the notebook for an example).

In some cases you may want to stop gradients from backpropagating through some part of your neural network. To do this, you must use the `tf.stop_gradient()` function. The function returns its inputs during the forward pass (like `tf.identity()`), but it does not let gradients through during backpropagation (it acts like a constant):

```
def f(w1, w2):
    return 3 * w1 ** 2 + tf.stop_gradient(2 * w1 * w2)

with tf.GradientTape() as tape:
    z = f(w1, w2) # same result as without stop_gradient()

gradients = tape.gradient(z, [w1, w2]) # => returns [tensor 30., None]
```

Finally, you may occasionally run into some numerical issues when computing gradients. For example, if you compute the gradients of the `my_softplus()` function for large inputs, the result will be NaN:

```
>>> x = tf.Variable([100.])
>>> with tf.GradientTape() as tape:
...     z = my_softplus(x)
...
>>> tape.gradient(z, [x])
<tf.Tensor: [...] numpy=array([nan], dtype=float32)>
```

This is because computing the gradients of this function using autodiff leads to some numerical difficulties: due to floating-point precision errors, autodiff ends up computing infinity divided by infinity (which returns NaN). Fortunately, we can analytically find that the derivative of the softplus function is just 1 / (1 + 1 / exp(x)), which is numerically stable. Next, we can tell TensorFlow to use this stable function when computing the gradients of the `my_softplus()` function by decorating it with `@tf.custom_gradient` and making it return both its normal output and the function that computes the derivatives (note that it will receive as input the gradients that were backpropagated so far, down to the softplus function; and according to the chain rule, we should multiply them with this function's gradients):

```
@tf.custom_gradient
def my_better_softplus(z):
    exp = tf.exp(z)
    def my_softplus_gradients(grad):
        return grad / (1 + 1 / exp)
    return tf.math.log(exp + 1), my_softplus_gradients
```

Now when we compute the gradients of the `my_better_softplus()` function, we get the proper result, even for large input values (however, the main output still explodes because of the exponential; one workaround is to use `tf.where()` to return the inputs when they are large).

Congratulations! You can now compute the gradients of any function (provided it is differentiable at the point where you compute it), even blocking backpropagation when needed, and write your own gradient functions! This is probably more flexibility than you will ever need, even if you build your own custom training loops, as we will see now.

Custom Training Loops

In some rare cases, the `fit()` method may not be flexible enough for what you need to do. For example, the Wide & Deep paper we discussed in Chapter 10 uses two different optimizers: one for the wide path and the other for the deep path. Since the `fit()` method only uses one optimizer (the one that we specify when compiling the model), implementing this paper requires writing your own custom loop.

You may also like to write custom training loops simply to feel more confident that they do precisely what you intend them to do (perhaps you are unsure about some details of the `fit()` method). It can sometimes feel safer to make everything explicit. However, remember that writing a custom training loop will make your code longer, more error-prone, and harder to maintain.

Unless you really need the extra flexibility, you should prefer using the `fit()` method rather than implementing your own training loop, especially if you work in a team.

First, let's build a simple model. No need to compile it, since we will handle the training loop manually:

```
l2_reg = keras.regularizers.l2(0.05)
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="elu", kernel_initializer="he_normal",
                       kernel_regularizer=l2_reg),
    keras.layers.Dense(1, kernel_regularizer=l2_reg)
])
```

Next, let's create a tiny function that will randomly sample a batch of instances from the training set (in Chapter 13 we will discuss the Data API, which offers a much better alternative):

```
def random_batch(X, y, batch_size=32):
    idx = np.random.randint(len(X), size=batch_size)
    return X[idx], y[idx]
```

Let's also define a function that will display the training status, including the number of steps, the total number of steps, the mean loss since the start of the epoch (i.e., we will use the `Mean` metric to compute it), and other metrics:

```
def print_status_bar(iteration, total, loss, metrics=None):
    metrics = " - ".join(["{}: {:.4f}".format(m.name, m.result())
                          for m in [loss] + (metrics or [])])
    end = "" if iteration < total else "\n"
    print("\r{}/{} - ".format(iteration, total) + metrics,
          end=end)
```

This code is self-explanatory, unless you are unfamiliar with Python string formatting: `{:.4f}` will format a float with four digits after the decimal point, and using `\r` (carriage return) along with `end=""` ensures that the status bar always gets printed on the same line. In the notebook, the `print_status_bar()` function includes a progress bar, but you could use the handy `tqdm` library instead.

With that, let's get down to business! First, we need to define some hyperparameters and choose the optimizer, the loss function, and the metrics (just the MAE in this example):

```
n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = keras.optimizers.Nadam(lr=0.01)
loss_fn = keras.losses.mean_squared_error
```

```
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.MeanAbsoluteError()]
```

And now we are ready to build the custom loop!

```
for epoch in range(1, n_epochs + 1):
    print("Epoch {}/{}".format(epoch, n_epochs))
    for step in range(1, n_steps + 1):
        X_batch, y_batch = random_batch(X_train_scaled, y_train)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch, training=True)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))
        mean_loss(loss)
        for metric in metrics:
            metric(y_batch, y_pred)
        print_status_bar(step * batch_size, len(y_train), mean_loss, metrics)
    print_status_bar(len(y_train), len(y_train), mean_loss, metrics)
    for metric in [mean_loss] + metrics:
        metric.reset_states()
```

There's a lot going on in this code, so let's walk through it:

- We create two nested loops: one for the epochs, the other for the batches within an epoch.
- Then we sample a random batch from the training set.
- Inside the `tf.GradientTape()` block, we make a prediction for one batch (using the model as a function), and we compute the loss: it is equal to the main loss plus the other losses (in this model, there is one regularization loss per layer). Since the `mean_squared_error()` function returns one loss per instance, we compute the mean over the batch using `tf.reduce_mean()` (if you wanted to apply different weights to each instance, this is where you would do it). The regularization losses are already reduced to a single scalar each, so we just need to sum them (using `tf.add_n()`, which sums multiple tensors of the same shape and data type).
- Next, we ask the tape to compute the gradient of the loss with regard to each trainable variable (*not* all variables!), and we apply them to the optimizer to perform a Gradient Descent step.
- Then we update the mean loss and the metrics (over the current epoch), and we display the status bar.

- At the end of each epoch, we display the status bar again to make it look complete[13] and to print a line feed, and we reset the states of the mean loss and the metrics.

If you set the optimizer's `clipnorm` or `clipvalue` hyperparameter, it will take care of this for you. If you want to apply any other transformation to the gradients, simply do so before calling the `apply_gradients()` method.

If you add weight constraints to your model (e.g., by setting `kernel_constraint` or `bias_constraint` when creating a layer), you should update the training loop to apply these constraints just after `apply_gradients()`:

```
for variable in model.variables:
    if variable.constraint is not None:
        variable.assign(variable.constraint(variable))
```

Most importantly, this training loop does not handle layers that behave differently during training and testing (e.g., `BatchNormalization` or `Dropout`). To handle these, you need to call the model with `training=True` and make sure it propagates this to every layer that needs it.

As you can see, there are quite a lot of things you need to get right, and it's easy to make a mistake. But on the bright side, you get full control, so it's your call.

Now that you know how to customize any part of your models[14] and training algorithms, let's see how you can use TensorFlow's automatic graph generation feature: it can speed up your custom code considerably, and it will also make it portable to any platform supported by TensorFlow (see Chapter 19).

TensorFlow Functions and Graphs

In TensorFlow 1, graphs were unavoidable (as were the complexities that came with them) because they were a central part of TensorFlow's API. In TensorFlow 2, they are still there, but not as central, and they're much (much!) simpler to use. To show just how simple, let's start with a trivial function that computes the cube of its input:

```
def cube(x):
    return x ** 3
```

13 The truth is we did not process every single instance in the training set, because we sampled instances randomly: some were processed more than once, while others were not processed at all. Likewise, if the training set size is not a multiple of the batch size, we will miss a few instances. In practice that's fine.

14 With the exception of optimizers, as very few people ever customize these; see the "Custom Optimizers" section in the notebook for an example.

We can obviously call this function with a Python value, such as an int or a float, or we can call it with a tensor:

```
>>> cube(2)
8
>>> cube(tf.constant(2.0))
<tf.Tensor: id=18634148, shape=(), dtype=float32, numpy=8.0>
```

Now, let's use `tf.function()` to convert this Python function to a *TensorFlow Function*:

```
>>> tf_cube = tf.function(cube)
>>> tf_cube
<tensorflow.python.eager.def_function.Function at 0x1546fc080>
```

This TF Function can then be used exactly like the original Python function, and it will return the same result (but as tensors):

```
>>> tf_cube(2)
<tf.Tensor: id=18634201, shape=(), dtype=int32, numpy=8>
>>> tf_cube(tf.constant(2.0))
<tf.Tensor: id=18634211, shape=(), dtype=float32, numpy=8.0>
```

Under the hood, `tf.function()` analyzed the computations performed by the `cube()` function and generated an equivalent computation graph! As you can see, it was rather painless (we will see how this works shortly). Alternatively, we could have used `tf.function` as a decorator; this is actually more common:

```
@tf.function
def tf_cube(x):
    return x ** 3
```

The original Python function is still available via the TF Function's `python_function` attribute, in case you ever need it:

```
>>> tf_cube.python_function(2)
8
```

TensorFlow optimizes the computation graph, pruning unused nodes, simplifying expressions (e.g., 1 + 2 would get replaced with 3), and more. Once the optimized graph is ready, the TF Function efficiently executes the operations in the graph, in the appropriate order (and in parallel when it can). As a result, a TF Function will usually run much faster than the original Python function, especially if it performs complex computations.[15] Most of the time you will not really need to know more than that: when you want to boost a Python function, just transform it into a TF Function. That's all!

15 However, in this trivial example, the computation graph is so small that there is nothing at all to optimize, so `tf_cube()` actually runs much slower than `cube()`.

Moreover, when you write a custom loss function, a custom metric, a custom layer, or any other custom function and you use it in a Keras model (as we did throughout this chapter), Keras automatically converts your function into a TF Function—no need to use `tf.function()`. So most of the time, all this magic is 100% transparent.

You can tell Keras *not* to convert your Python functions to TF Functions by setting `dynamic=True` when creating a custom layer or a custom model. Alternatively, you can set `run_eagerly=True` when calling the model's `compile()` method.

By default, a TF Function generates a new graph for every unique set of input shapes and data types and caches it for subsequent calls. For example, if you call `tf_cube(tf.constant(10))`, a graph will be generated for int32 tensors of shape []. Then if you call `tf_cube(tf.constant(20))`, the same graph will be reused. But if you then call `tf_cube(tf.constant([10, 20]))`, a new graph will be generated for int32 tensors of shape [2]. This is how TF Functions handle polymorphism (i.e., varying argument types and shapes). However, this is only true for tensor arguments: if you pass numerical Python values to a TF Function, a new graph will be generated for every distinct value: for example, calling `tf_cube(10)` and `tf_cube(20)` will generate two graphs.

If you call a TF Function many times with different numerical Python values, then many graphs will be generated, slowing down your program and using up a lot of RAM (you must delete the TF Function to release it). Python values should be reserved for arguments that will have few unique values, such as hyperparameters like the number of neurons per layer. This allows TensorFlow to better optimize each variant of your model.

AutoGraph and Tracing

So how does TensorFlow generate graphs? It starts by analyzing the Python function's source code to capture all the control flow statements, such as `for` loops, `while` loops, and `if` statements, as well as `break`, `continue`, and `return` statements. This first step is called *AutoGraph*. The reason TensorFlow has to analyze the source code is that Python does not provide any other way to capture control flow statements: it offers magic methods like `__add__()` and `__mul__()` to capture operators like `+` and `*`, but there are no `__while__()` or `__if__()` magic methods. After analyzing the function's code, AutoGraph outputs an upgraded version of that function in which all the control flow statements are replaced by the appropriate TensorFlow operations, such as `tf.while_loop()` for loops and `tf.cond()` for `if` statements. For example, in Figure 12-4, AutoGraph analyzes the source code of the `sum_squares()` Python

function, and it generates the `tf__sum_squares()` function. In this function, the `for` loop is replaced by the definition of the `loop_body()` function (containing the body of the original `for` loop), followed by a call to the `for_stmt()` function. This call will build the appropriate `tf.while_loop()` operation in the computation graph.

Figure 12-4. How TensorFlow generates graphs using AutoGraph and tracing

Next, TensorFlow calls this "upgraded" function, but instead of passing the argument, it passes a *symbolic tensor*—a tensor without any actual value, only a name, a data type, and a shape. For example, if you call `sum_squares(tf.constant(10))`, then the `tf__sum_squares()` function will be called with a symbolic tensor of type int32 and shape []. The function will run in *graph mode*, meaning that each TensorFlow operation will add a node in the graph to represent itself and its output tensor(s) (as opposed to the regular mode, called *eager execution*, or *eager mode*). In graph mode, TF operations do not perform any computations. This should feel familiar if you know TensorFlow 1, as graph mode was the default mode. In Figure 12-4, you can see the `tf__sum_squares()` function being called with a symbolic tensor as its argument (in this case, an int32 tensor of shape []) and the final graph being generated during tracing. The nodes represent operations, and the arrows represent tensors (both the generated function and the graph are simplified).

To view the generated function's source code, you can call `tf.autograph.to_code(sum_squares.python_function)`. The code is not meant to be pretty, but it can sometimes help for debugging.

TF Function Rules

Most of the time, converting a Python function that performs TensorFlow operations into a TF Function is trivial: decorate it with `@tf.function` or let Keras take care of it for you. However, there are a few rules to respect:

- If you call any external library, including NumPy or even the standard library, this call will run only during tracing; it will not be part of the graph. Indeed, a TensorFlow graph can only include TensorFlow constructs (tensors, operations, variables, datasets, and so on). So, make sure you use `tf.reduce_sum()` instead of `np.sum()`, `tf.sort()` instead of the built-in `sorted()` function, and so on (unless you really want the code to run only during tracing). This has a few additional implications:
  - If you define a TF Function `f(x)` that just returns `np.random.rand()`, a random number will only be generated when the function is traced, so `f(tf.constant(2.))` and `f(tf.constant(3.))` will return the same random number, but `f(tf.constant([2., 3.]))` will return a different one. If you replace `np.random.rand()` with `tf.random.uniform([])`, then a new random number will be generated upon every call, since the operation will be part of the graph.
  - If your non-TensorFlow code has side effects (such as logging something or updating a Python counter), then you should not expect those side effects to occur every time you call the TF Function, as they will only occur when the function is traced.
  - You can wrap arbitrary Python code in a `tf.py_function()` operation, but doing so will hinder performance, as TensorFlow will not be able to do any graph optimization on this code. It will also reduce portability, as the graph will only run on platforms where Python is available (and where the right libraries are installed).
- You can call other Python functions or TF Functions, but they should follow the same rules, as TensorFlow will capture their operations in the computation graph. Note that these other functions do not need to be decorated with `@tf.function`.
- If the function creates a TensorFlow variable (or any other stateful TensorFlow object, such as a dataset or a queue), it must do so upon the very first call, and only then, or else you will get an exception. It is usually preferable to create

variables outside of the TF Function (e.g., in the `build()` method of a custom layer). If you want to assign a new value to the variable, make sure you call its `assign()` method, instead of using the `=` operator.

- The source code of your Python function should be available to TensorFlow. If the source code is unavailable (for example, if you define your function in the Python shell, which does not give access to the source code, or if you deploy only the compiled **.pyc* Python files to production), then the graph generation process will fail or have limited functionality.
- TensorFlow will only capture `for` loops that iterate over a tensor or a dataset. So make sure you use `for i in tf.range(x)` rather than `for i in range(x)`, or else the loop will not be captured in the graph. Instead, it will run during tracing. (This may be what you want if the `for` loop is meant to build the graph, for example to create each layer in a neural network.)
- As always, for performance reasons, you should prefer a vectorized implementation whenever you can, rather than using loops.

It's time to sum up! In this chapter we started with a brief overview of TensorFlow, then we looked at TensorFlow's low-level API, including tensors, operations, variables, and special data structures. We then used these tools to customize almost every component in tf.keras. Finally, we looked at how TF Functions can boost performance, how graphs are generated using AutoGraph and tracing, and what rules to follow when you write TF Functions (if you would like to open the black box a bit further, for example to explore the generated graphs, you will find technical details in Appendix G).

In the next chapter, we will look at how to efficiently load and preprocess data with TensorFlow.

Exercises

1. How would you describe TensorFlow in a short sentence? What are its main features? Can you name other popular Deep Learning libraries?
2. Is TensorFlow a drop-in replacement for NumPy? What are the main differences between the two?
3. Do you get the same result with `tf.range(10)` and `tf.constant(np.arange(10))`?
4. Can you name six other data structures available in TensorFlow, beyond regular tensors?

5. A custom loss function can be defined by writing a function or by subclassing the `keras.losses.Loss` class. When would you use each option?
6. Similarly, a custom metric can be defined in a function or a subclass of `keras.metrics.Metric`. When would you use each option?
7. When should you create a custom layer versus a custom model?
8. What are some use cases that require writing your own custom training loop?
9. Can custom Keras components contain arbitrary Python code, or must they be convertible to TF Functions?
10. What are the main rules to respect if you want a function to be convertible to a TF Function?
11. When would you need to create a dynamic Keras model? How do you do that? Why not make all your models dynamic?
12. Implement a custom layer that performs *Layer Normalization* (we will use this type of layer in Chapter 15):
    a. The `build()` method should define two trainable weights $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, both of shape `input_shape[-1:]` and data type `tf.float32`. $\boldsymbol{\alpha}$ should be initialized with 1s, and $\boldsymbol{\beta}$ with 0s.
    b. The `call()` method should compute the mean μ and standard deviation σ of each instance's features. For this, you can use `tf.nn.moments(inputs, axes=-1, keepdims=True)`, which returns the mean μ and the variance σ^2 of all instances (compute the square root of the variance to get the standard deviation). Then the function should compute and return $\boldsymbol{\alpha}\otimes(\mathbf{X} - \mu)/(\sigma + \varepsilon) + \boldsymbol{\beta}$, where $\otimes$ represents itemwise multiplication (`*`) and ε is a smoothing term (small constant to avoid division by zero, e.g., 0.001).
    c. Ensure that your custom layer produces the same (or very nearly the same) output as the `keras.layers.LayerNormalization` layer.
13. Train a model using a custom training loop to tackle the Fashion MNIST dataset (see Chapter 10).
    a. Display the epoch, iteration, mean training loss, and mean accuracy over each epoch (updated at each iteration), as well as the validation loss and accuracy at the end of each epoch.
    b. Try using a different optimizer with a different learning rate for the upper layers and the lower layers.

Solutions to these exercises are available in Appendix A.

CHAPTER 13

Loading and Preprocessing Data with TensorFlow

So far we have used only datasets that fit in memory, but Deep Learning systems are often trained on very large datasets that will not fit in RAM. Ingesting a large dataset and preprocessing it efficiently can be tricky to implement with other Deep Learning libraries, but TensorFlow makes it easy thanks to the *Data API*: you just create a dataset object, and tell it where to get the data and how to transform it. TensorFlow takes care of all the implementation details, such as multithreading, queuing, batching, and prefetching. Moreover, the Data API works seamlessly with tf.keras!

Off the shelf, the Data API can read from text files (such as CSV files), binary files with fixed-size records, and binary files that use TensorFlow's TFRecord format, which supports records of varying sizes. TFRecord is a flexible and efficient binary format usually containing protocol buffers (an open source binary format). The Data API also has support for reading from SQL databases. Moreover, many open source extensions are available to read from all sorts of data sources, such as Google's BigQuery service.

Reading huge datasets efficiently is not the only difficulty: the data also needs to be preprocessed, usually normalized. Moreover, it is not always composed strictly of convenient numerical fields: there may be text features, categorical features, and so on. These need to be encoded, for example using one-hot encoding, bag-of-words encoding, or *embeddings* (as we will see, an embedding is a trainable dense vector that represents a category or token). One option to handle all this preprocessing is to write your own custom preprocessing layers. Another is to use the standard preprocessing layers provided by Keras.

In this chapter, we will cover the Data API, the TFRecord format, and how to create custom preprocessing layers and use the standard Keras ones. We will also take a quick look at a few related projects from TensorFlow's ecosystem:

TF Transform (tf.Transform)
: Makes it possible to write a single preprocessing function that can be run in batch mode on your full training set, before training (to speed it up), and then exported to a TF Function and incorporated into your trained model so that once it is deployed in production it can take care of preprocessing new instances on the fly.

TF Datasets (TFDS)
: Provides a convenient function to download many common datasets of all kinds, including large ones like ImageNet, as well as convenient dataset objects to manipulate them using the Data API.

So let's get started!

The Data API

The whole Data API revolves around the concept of a *dataset*: as you might suspect, this represents a sequence of data items. Usually you will use datasets that gradually read data from disk, but for simplicity let's create a dataset entirely in RAM using `tf.data.Dataset.from_tensor_slices()`:

```
>>> X = tf.range(10)  # any data tensor
>>> dataset = tf.data.Dataset.from_tensor_slices(X)
>>> dataset
<TensorSliceDataset shapes: (), types: tf.int32>
```

The `from_tensor_slices()` function takes a tensor and creates a `tf.data.Dataset` whose elements are all the slices of `X` (along the first dimension), so this dataset contains 10 items: tensors 0, 1, 2, …, 9. In this case we would have obtained the same dataset if we had used `tf.data.Dataset.range(10)`.

You can simply iterate over a dataset's items like this:

```
>>> for item in dataset:
...     print(item)
...
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
[...]
tf.Tensor(9, shape=(), dtype=int32)
```

Chaining Transformations

Once you have a dataset, you can apply all sorts of transformations to it by calling its transformation methods. Each method returns a new dataset, so you can chain transformations like this (this chain is illustrated in Figure 13-1):

```
>>> dataset = dataset.repeat(3).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)
tf.Tensor([8 9], shape=(2,), dtype=int32)
```

Figure 13-1. Chaining dataset transformations

In this example, we first call the `repeat()` method on the original dataset, and it returns a new dataset that will repeat the items of the original dataset three times. Of course, this will not copy all the data in memory three times! (If you call this method with no arguments, the new dataset will repeat the source dataset forever, so the code that iterates over the dataset will have to decide when to stop.) Then we call the `batch()` method on this new dataset, and again this creates a new dataset. This one will group the items of the previous dataset in batches of seven items. Finally, we iterate over the items of this final dataset. As you can see, the `batch()` method had to output a final batch of size two instead of seven, but you can call it with `drop_remainder=True` if you want it to drop this final batch so that all batches have the exact same size.

The dataset methods do *not* modify datasets, they create new ones, so make sure to keep a reference to these new datasets (e.g., with `dataset = ...`), or else nothing will happen.

You can also transform the items by calling the `map()` method. For example, this creates a new dataset with all items doubled:

```
>>> dataset = dataset.map(lambda x: x * 2) # Items: [0,2,4,6,8,10,12]
```

This function is the one you will call to apply any preprocessing you want to your data. Sometimes this will include computations that can be quite intensive, such as reshaping or rotating an image, so you will usually want to spawn multiple threads to speed things up: it's as simple as setting the `num_parallel_calls` argument. Note that the function you pass to the `map()` method must be convertible to a TF Function (see Chapter 12).

While the `map()` method applies a transformation to each item, the `apply()` method applies a transformation to the dataset as a whole. For example, the following code applies the `unbatch()` function to the dataset (this function is currently experimental, but it will most likely move to the core API in a future release). Each item in the new dataset will be a single-integer tensor instead of a batch of seven integers:

```
>>> dataset = dataset.apply(tf.data.experimental.unbatch()) # Items: 0,2,4,...
```

It is also possible to simply filter the dataset using the `filter()` method:

```
>>> dataset = dataset.filter(lambda x: x < 10) # Items: 0 2 4 6 8 0 2 4 6...
```

You will often want to look at just a few items from a dataset. You can use the `take()` method for that:

```
>>> for item in dataset.take(3):
...     print(item)
...
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
```

Shuffling the Data

As you know, Gradient Descent works best when the instances in the training set are independent and identically distributed (see Chapter 4). A simple way to ensure this is to shuffle the instances, using the `shuffle()` method. It will create a new dataset that will start by filling up a buffer with the first items of the source dataset. Then, whenever it is asked for an item, it will pull one out randomly from the buffer and replace it with a fresh one from the source dataset, until it has iterated entirely through the source dataset. At this point it continues to pull out items randomly from

the buffer until it is empty. You must specify the buffer size, and it is important to make it large enough, or else shuffling will not be very effective.[1] Just don't exceed the amount of RAM you have, and even if you have plenty of it, there's no need to go beyond the dataset's size. You can provide a random seed if you want the same random order every time you run your program. For example, the following code creates and displays a dataset containing the integers 0 to 9, repeated 3 times, shuffled using a buffer of size 5 and a random seed of 42, and batched with a batch size of 7:

```
>>> dataset = tf.data.Dataset.range(10).repeat(3) # 0 to 9, three times
>>> dataset = dataset.shuffle(buffer_size=5, seed=42).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 2 3 6 7 9 4], shape=(7,), dtype=int64)
tf.Tensor([5 0 1 1 8 6 5], shape=(7,), dtype=int64)
tf.Tensor([4 8 7 1 2 3 0], shape=(7,), dtype=int64)
tf.Tensor([5 4 2 7 8 9 9], shape=(7,), dtype=int64)
tf.Tensor([3 6], shape=(2,), dtype=int64)
```

If you call `repeat()` on a shuffled dataset, by default it will generate a new order at every iteration. This is generally a good idea, but if you prefer to reuse the same order at each iteration (e.g., for tests or debugging), you can set `reshuffle_each_iteration=False`.

For a large dataset that does not fit in memory, this simple shuffling-buffer approach may not be sufficient, since the buffer will be small compared to the dataset. One solution is to shuffle the source data itself (for example, on Linux you can shuffle text files using the `shuf` command). This will definitely improve shuffling a lot! Even if the source data is shuffled, you will usually want to shuffle it some more, or else the same order will be repeated at each epoch, and the model may end up being biased (e.g., due to some spurious patterns present by chance in the source data's order). To shuffle the instances some more, a common approach is to split the source data into multiple files, then read them in a random order during training. However, instances located in the same file will still end up close to each other. To avoid this you can pick multiple files randomly and read them simultaneously, interleaving their records. Then on top of that you can add a shuffling buffer using the `shuffle()` method. If all

1 Imagine a sorted deck of cards on your left: suppose you just take the top three cards and shuffle them, then pick one randomly and put it to your right, keeping the other two in your hands. Take another card on your left, shuffle the three cards in your hands and pick one of them randomly, and put it on your right. When you are done going through all the cards like this, you will have a deck of cards on your right: do you think it will be perfectly shuffled?

this sounds like a lot of work, don't worry: the Data API makes all this possible in just a few lines of code. Let's see how to do this.

Interleaving lines from multiple files

First, let's suppose that you've loaded the California housing dataset, shuffled it (unless it was already shuffled), and split it into a training set, a validation set, and a test set. Then you split each set into many CSV files that each look like this (each row contains eight input features plus the target median house value):

```
MedInc,HouseAge,AveRooms,AveBedrms,Popul,AveOccup,Lat,Long,MedianHouseValue
3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442
5.3275,5.0,6.4900,0.9910,3464.0,3.4433,33.69,-117.39,1.687
3.1,29.0,7.5423,1.5915,1328.0,2.2508,38.44,-122.98,1.621
[...]
```

Let's also suppose `train_filepaths` contains the list of training file paths (and you also have `valid_filepaths` and `test_filepaths`):

```
>>> train_filepaths
['datasets/housing/my_train_00.csv', 'datasets/housing/my_train_01.csv',...]
```

Alternatively, you could use file patterns; for example, `train_filepaths = "datasets/housing/my_train_*.csv"`. Now let's create a dataset containing only these file paths:

```
filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

By default, the `list_files()` function returns a dataset that shuffles the file paths. In general this is a good thing, but you can set `shuffle=False` if you do not want that for some reason.

Next, you can call the `interleave()` method to read from five files at a time and interleave their lines (skipping the first line of each file, which is the header row, using the `skip()` method):

```
n_readers = 5
dataset = filepath_dataset.interleave(
    lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
    cycle_length=n_readers)
```

The `interleave()` method will create a dataset that will pull five file paths from the `filepath_dataset`, and for each one it will call the function you gave it (a lambda in this example) to create a new dataset (in this case a `TextLineDataset`). To be clear, at this stage there will be seven datasets in all: the filepath dataset, the interleave dataset, and the five `TextLineDatasets` created internally by the interleave dataset. When we iterate over the interleave dataset, it will cycle through these five `TextLineDatasets`, reading one line at a time from each until all datasets are out of items. Then it will get

the next five file paths from the `filepath_dataset` and interleave them the same way, and so on until it runs out of file paths.

For interleaving to work best, it is preferable to have files of identical length; otherwise the ends of the longest files will not be interleaved.

By default, `interleave()` does not use parallelism; it just reads one line at a time from each file, sequentially. If you want it to actually read files in parallel, you can set the `num_parallel_calls` argument to the number of threads you want (note that the `map()` method also has this argument). You can even set it to `tf.data.experimental.AUTOTUNE` to make TensorFlow choose the right number of threads dynamically based on the available CPU (however, this is an experimental feature for now). Let's look at what the dataset contains now:

```
>>> for line in dataset.take(5):
...     print(line.numpy())
...
b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782'
b'4.1812,52.0,5.7013,0.9965,692.0,2.4027,33.73,-118.31,3.215'
b'3.6875,44.0,4.5244,0.9930,457.0,3.1958,34.04,-118.15,1.625'
b'3.3456,37.0,4.5140,0.9084,458.0,3.2253,36.67,-121.7,2.526'
b'3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442'
```

These are the first rows (ignoring the header row) of five CSV files, chosen randomly. Looks good! But as you can see, these are just byte strings; we need to parse them and scale the data.

Preprocessing the Data

Let's implement a small function that will perform this preprocessing:

```
X_mean, X_std = [...] # mean and scale of each feature in the training set
n_inputs = 8

def preprocess(line):
  defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
  fields = tf.io.decode_csv(line, record_defaults=defs)
  x = tf.stack(fields[:-1])
  y = tf.stack(fields[-1:])
  return (x - X_mean) / X_std, y
```

Let's walk through this code:

- First, the code assumes that we have precomputed the mean and standard deviation of each feature in the training set. `X_mean` and `X_std` are just 1D tensors (or NumPy arrays) containing eight floats, one per input feature.
- The `preprocess()` function takes one CSV line and starts by parsing it. For this it uses the `tf.io.decode_csv()` function, which takes two arguments: the first is the line to parse, and the second is an array containing the default value for each column in the CSV file. This array tells TensorFlow not only the default value for each column, but also the number of columns and their types. In this example, we tell it that all feature columns are floats and that missing values should default to 0, but we provide an empty array of type `tf.float32` as the default value for the last column (the target): the array tells TensorFlow that this column contains floats, but that there is no default value, so it will raise an exception if it encounters a missing value.
- The `decode_csv()` function returns a list of scalar tensors (one per column), but we need to return 1D tensor arrays. So we call `tf.stack()` on all tensors except for the last one (the target): this will stack these tensors into a 1D array. We then do the same for the target value (this makes it a 1D tensor array with a single value, rather than a scalar tensor).
- Finally, we scale the input features by subtracting the feature means and then dividing by the feature standard deviations, and we return a tuple containing the scaled features and the target.

Let's test this preprocessing function:

```
>>> preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782')
(<tf.Tensor: id=6227, shape=(8,), dtype=float32, numpy=
 array([ 0.16579159,  1.216324  , -0.05204564, -0.39215982, -0.5277444 ,
        -0.2633488 ,  0.8543046 , -1.3072058 ], dtype=float32)>,
 <tf.Tensor: [...], numpy=array([2.782], dtype=float32)>)
```

Looks good! We can now apply the function to the dataset.

Putting Everything Together

To make the code reusable, let's put together everything we have discussed so far into a small helper function: it will create and return a dataset that will efficiently load California housing data from multiple CSV files, preprocess it, shuffle it, optionally repeat it, and batch it (see Figure 13-2):

```
def csv_reader_dataset(filepaths, repeat=1, n_readers=5,
                       n_read_threads=None, shuffle_buffer_size=10000,
                       n_parse_threads=5, batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths)
    dataset = dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
```

```
        cycle_length=n_readers, num_parallel_calls=n_read_threads)
    dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
    dataset = dataset.shuffle(shuffle_buffer_size).repeat(repeat)
    return dataset.batch(batch_size).prefetch(1)
```

Everything should make sense in this code, except the very last line (`prefetch(1)`), which is important for performance.

Figure 13-2. Loading and preprocessing data from multiple CSV files

Prefetching

By calling `prefetch(1)` at the end, we are creating a dataset that will do its best to always be one batch ahead.[2] In other words, while our training algorithm is working on one batch, the dataset will already be working in parallel on getting the next batch ready (e.g., reading the data from disk and preprocessing it). This can improve performance dramatically, as is illustrated in Figure 13-3. If we also ensure that loading and preprocessing are multithreaded (by setting `num_parallel_calls` when calling `interleave()` and `map()`), we can exploit multiple cores on the CPU and hopefully make preparing one batch of data shorter than running a training step on the GPU:

2 In general, just prefetching one batch is fine, but in some cases you may need to prefetch a few more. Alternatively, you can let TensorFlow decide automatically by passing `tf.data.experimental.AUTOTUNE` (this is an experimental feature for now).

this way the GPU will be almost 100% utilized (except for the data transfer time from the CPU to the GPU[3]), and training will run much faster.

Figure 13-3. With prefetching, the CPU and the GPU work in parallel: as the GPU works on one batch, the CPU works on the next

If you plan to purchase a GPU card, its processing power and its memory size are of course very important (in particular, a large amount of RAM is crucial for computer vision). Just as important to get good performance is its *memory bandwidth*; this is the number of gigabytes of data it can get into or out of its RAM per second.

If the dataset is small enough to fit in memory, you can significantly speed up training by using the dataset's `cache()` method to cache its content to RAM. You should generally do this after loading and preprocessing the data, but before shuffling, repeating, batching, and prefetching. This way, each instance will only be read and

3 But check out the `tf.data.experimental.prefetch_to_device()` function, which can prefetch data directly to the GPU.

preprocessed once (instead of once per epoch), but the data will still be shuffled differently at each epoch, and the next batch will still be prepared in advance.

You now know how to build efficient input pipelines to load and preprocess data from multiple text files. We have discussed the most common dataset methods, but there are a few more you may want to look at: `concatenate()`, `zip()`, `window()`, `reduce()`, `shard()`, `flat_map()`, and `padded_batch()`. There are also a couple more class methods: `from_generator()` and `from_tensors()`, which create a new dataset from a Python generator or a list of tensors, respectively. Please check the API documentation for more details. Also note that there are experimental features available in `tf.data.experimental`, many of which will likely make it to the core API in future releases (e.g., check out the `CsvDataset` class, as well as the `make_csv_dataset()` method, which takes care of inferring the type of each column).

Using the Dataset with tf.keras

Now we can use the `csv_reader_dataset()` function to create a dataset for the training set. Note that we do not need to repeat it, as this will be taken care of by tf.keras. We also create datasets for the validation set and the test set:

```
train_set = csv_reader_dataset(train_filepaths)
valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)
```

And now we can simply build and train a Keras model using these datasets.[4] All we need to do is pass the training and validation datasets to the `fit()` method, instead of `X_train, y_train`, `X_valid`, and `y_valid`:[5]

```
model = keras.models.Sequential([...])
model.compile([...])
model.fit(train_set, epochs=10, validation_data=valid_set)
```

Similarly, we can pass a dataset to the `evaluate()` and `predict()` methods:

```
model.evaluate(test_set)
new_set = test_set.take(3).map(lambda X, y: X) # pretend we have 3 new instances
model.predict(new_set) # a dataset containing new instances
```

Unlike the other sets, the `new_set` will usually not contain labels (if it does, Keras will ignore them). Note that in all these cases, you can still use NumPy arrays instead of

4 Support for datasets is specific to tf.keras; this will not work in other implementations of the Keras API.

5 The `fit()` method will take care of repeating the training dataset. Alternatively, you could call `repeat()` on the training dataset so that it repeats forever and specify the `steps_per_epoch` argument when calling the `fit()` method. This may be useful in some rare cases, for example if you want to use a shuffle buffer that crosses over epochs.

datasets if you want (but of course they need to have been loaded and preprocessed first).

If you want to build your own custom training loop (as in Chapter 12), you can just iterate over the training set, very naturally:

```
for X_batch, y_batch in train_set:
    [...] # perform one Gradient Descent step
```

In fact, it is even possible to create a TF Function (see Chapter 12) that performs the whole training loop:

```
@tf.function
def train(model, optimizer, loss_fn, n_epochs, [...]):
    train_set = csv_reader_dataset(train_filepaths, repeat=n_epochs, [...])
    for X_batch, y_batch in train_set:
        with tf.GradientTape() as tape:
            y_pred = model(X_batch)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Congratulations, you now know how to build powerful input pipelines using the Data API! However, so far we have used CSV files, which are common, simple, and convenient but not really efficient, and do not support large or complex data structures (such as images or audio) very well. So let's see how to use TFRecords instead.

If you are happy with CSV files (or whatever other format you are using), you do not *have* to use TFRecords. As the saying goes, if it ain't broke, don't fix it! TFRecords are useful when the bottleneck during training is loading and parsing the data.

The TFRecord Format

The TFRecord format is TensorFlow's preferred format for storing large amounts of data and reading it efficiently. It is a very simple binary format that just contains a sequence of binary records of varying sizes (each record is comprised of a length, a CRC checksum to check that the length was not corrupted, then the actual data, and finally a CRC checksum for the data). You can easily create a TFRecord file using the `tf.io.TFRecordWriter` class:

```
with tf.io.TFRecordWriter("my_data.tfrecord") as f:
    f.write(b"This is the first record")
    f.write(b"And this is the second record")
```

And you can then use a `tf.data.TFRecordDataset` to read one or more TFRecord files:

```
filepaths = ["my_data.tfrecord"]
dataset = tf.data.TFRecordDataset(filepaths)
for item in dataset:
    print(item)
```

This will output:

```
tf.Tensor(b'This is the first record', shape=(), dtype=string)
tf.Tensor(b'And this is the second record', shape=(), dtype=string)
```

By default, a `TFRecordDataset` will read files one by one, but you can make it read multiple files in parallel and interleave their records by setting `num_parallel_reads`. Alternatively, you could obtain the same result by using `list_files()` and `interleave()` as we did earlier to read multiple CSV files.

Compressed TFRecord Files

It can sometimes be useful to compress your TFRecord files, especially if they need to be loaded via a network connection. You can create a compressed TFRecord file by setting the `options` argument:

```
options = tf.io.TFRecordOptions(compression_type="GZIP")
with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:
  [...]
```

When reading a compressed TFRecord file, you need to specify the compression type:

```
dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"],
                                  compression_type="GZIP")
```

A Brief Introduction to Protocol Buffers

Even though each record can use any binary format you want, TFRecord files usually contain serialized protocol buffers (also called *protobufs*). This is a portable, extensible, and efficient binary format developed at Google back in 2001 and made open source in 2008; protobufs are now widely used, in particular in gRPC, Google's remote procedure call system. They are defined using a simple language that looks like this:

```
syntax = "proto3";
message Person {
  string name = 1;
  int32 id = 2;
  repeated string email = 3;
}
```

This definition says we are using version 3 of the protobuf format, and it specifies that each `Person` object[6] may (optionally) have a `name` of type `string`, an `id` of type `int32`, and zero or more `email` fields, each of type `string`. The numbers `1`, `2`, and `3` are the field identifiers: they will be used in each record's binary representation. Once you have a definition in a *.proto* file, you can compile it. This requires `protoc`, the protobuf compiler, to generate access classes in Python (or some other language). Note that the protobuf definitions we will use have already been compiled for you, and their Python classes are part of TensorFlow, so you will not need to use `protoc`. All you need to know is how to use protobuf access classes in Python. To illustrate the basics, let's look at a simple example that uses the access classes generated for the `Person` protobuf (the code is explained in the comments):

```
>>> from person_pb2 import Person  # import the generated access class
>>> person = Person(name="Al", id=123, email=["a@b.com"])  # create a Person
>>> print(person)  # display the Person
name: "Al"
id: 123
email: "a@b.com"
>>> person.name  # read a field
"Al"
>>> person.name = "Alice"  # modify a field
>>> person.email[0]  # repeated fields can be accessed like arrays
"a@b.com"
>>> person.email.append("c@d.com")  # add an email address
>>> s = person.SerializeToString()  # serialize the object to a byte string
>>> s
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person()  # create a new Person
>>> person2.ParseFromString(s)  # parse the byte string (27 bytes long)
27
>>> person == person2  # now they are equal
True
```

In short, we import the `Person` class generated by `protoc`, we create an instance and play with it, visualizing it and reading and writing some fields, then we serialize it using the `SerializeToString()` method. This is the binary data that is ready to be saved or transmitted over the network. When reading or receiving this binary data, we can parse it using the `ParseFromString()` method, and we get a copy of the object that was serialized.[7]

We could save the serialized `Person` object to a TFRecord file, then we could load and parse it: everything would work fine. However, `SerializeToString()` and `ParseFrom`

6 Since protobuf objects are meant to be serialized and transmitted, they are called *messages*.

7 This chapter contains the bare minimum you need to know about protobufs to use TFRecords. To learn more about protobufs, please visit *https://homl.info/protobuf*.

`String()` are not TensorFlow operations (and neither are the other operations in this code), so they cannot be included in a TensorFlow Function (except by wrapping them in a `tf.py_function()` operation, which would make the code slower and less portable, as we saw in Chapter 12). Fortunately, TensorFlow does include special protobuf definitions for which it provides parsing operations.

TensorFlow Protobufs

The main protobuf typically used in a TFRecord file is the `Example` protobuf, which represents one instance in a dataset. It contains a list of named features, where each feature can either be a list of byte strings, a list of floats, or a list of integers. Here is the protobuf definition:

```
syntax = "proto3";
message BytesList { repeated bytes value = 1; }
message FloatList { repeated float value = 1 [packed = true]; }
message Int64List { repeated int64 value = 1 [packed = true]; }
message Feature {
    oneof kind {
        BytesList bytes_list = 1;
        FloatList float_list = 2;
        Int64List int64_list = 3;
    }
};
message Features { map<string, Feature> feature = 1; };
message Example { Features features = 1; };
```

The definitions of `BytesList`, `FloatList`, and `Int64List` are straightforward enough. Note that `[packed = true]` is used for repeated numerical fields, for a more efficient encoding. A `Feature` contains either a `BytesList`, a `FloatList`, or an `Int64List`. A `Features` (with an `s`) contains a dictionary that maps a feature name to the corresponding feature value. And finally, an `Example` contains only a `Features` object.[8] Here is how you could create a `tf.train.Example` representing the same person as earlier and write it to a TFRecord file:

```
from tensorflow.train import BytesList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example

person_example = Example(
    features=Features(
        feature={
            "name": Feature(bytes_list=BytesList(value=[b"Alice"])),
```

8 Why was `Example` even defined, since it contains no more than a `Features` object? Well, TensorFlow's developers may one day decide to add more fields to it. As long as the new `Example` definition still contains the `features` field, with the same ID, it will be backward compatible. This extensibility is one of the great features of protobufs.

```
            "id": Feature(int64_list=Int64List(value=[123])),
            "emails": Feature(bytes_list=BytesList(value=[b"a@b.com",
                                                          b"c@d.com"]))
        }))
```

The code is a bit verbose and repetitive, but it's rather straightforward (and you could easily wrap it inside a small helper function). Now that we have an `Example` protobuf, we can serialize it by calling its `SerializeToString()` method, then write the resulting data to a TFRecord file:

```
with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    f.write(person_example.SerializeToString())
```

Normally you would write much more than one `Example`! Typically, you would create a conversion script that reads from your current format (say, CSV files), creates an `Example` protobuf for each instance, serializes them, and saves them to several TFRecord files, ideally shuffling them in the process. This requires a bit of work, so once again make sure it is really necessary (perhaps your pipeline works fine with CSV files).

Now that we have a nice TFRecord file containing a serialized `Example`, let's try to load it.

Loading and Parsing Examples

To load the serialized `Example` protobufs, we will use a `tf.data.TFRecordDataset` once again, and we will parse each `Example` using `tf.io.parse_single_example()`. This is a TensorFlow operation, so it can be included in a TF Function. It requires at least two arguments: a string scalar tensor containing the serialized data, and a description of each feature. The description is a dictionary that maps each feature name to either a `tf.io.FixedLenFeature` descriptor indicating the feature's shape, type, and default value, or a `tf.io.VarLenFeature` descriptor indicating only the type (if the length of the feature's list may vary, such as for the `"emails"` feature).

The following code defines a description dictionary, then it iterates over the `TFRecordDataset` and parses the serialized `Example` protobuf this dataset contains:

```
feature_description = {
    "name": tf.io.FixedLenFeature([], tf.string, default_value=""),
    "id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
    "emails": tf.io.VarLenFeature(tf.string),
}

for serialized_example in tf.data.TFRecordDataset(["my_contacts.tfrecord"]):
    parsed_example = tf.io.parse_single_example(serialized_example,
                                                feature_description)
```

The fixed-length features are parsed as regular tensors, but the variable-length features are parsed as sparse tensors. You can convert a sparse tensor to a dense tensor using `tf.sparse.to_dense()`, but in this case it is simpler to just access its values:

```
>>> tf.sparse.to_dense(parsed_example["emails"], default_value=b"")
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
>>> parsed_example["emails"].values
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
```

A `BytesList` can contain any binary data you want, including any serialized object. For example, you can use `tf.io.encode_jpeg()` to encode an image using the JPEG format and put this binary data in a `BytesList`. Later, when your code reads the TFRecord, it will start by parsing the `Example`, then it will need to call `tf.io.decode_jpeg()` to parse the data and get the original image (or you can use `tf.io.decode_image()`, which can decode any BMP, GIF, JPEG, or PNG image). You can also store any tensor you want in a `BytesList` by serializing the tensor using `tf.io.serialize_tensor()` then putting the resulting byte string in a `BytesList` feature. Later, when you parse the TFRecord, you can parse this data using `tf.io.parse_tensor()`.

Instead of parsing examples one by one using `tf.io.parse_single_example()`, you may want to parse them batch by batch using `tf.io.parse_example()`:

```
dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).batch(10)
for serialized_examples in dataset:
    parsed_examples = tf.io.parse_example(serialized_examples,
                                          feature_description)
```

As you can see, the `Example` protobuf will probably be sufficient for most use cases. However, it may be a bit cumbersome to use when you are dealing with lists of lists. For example, suppose you want to classify text documents. Each document may be represented as a list of sentences, where each sentence is represented as a list of words. And perhaps each document also has a list of comments, where each comment is represented as a list of words. There may be some contextual data too, such as the document's author, title, and publication date. TensorFlow's `SequenceExample` protobuf is designed for such use cases.

Handling Lists of Lists Using the SequenceExample Protobuf

Here is the definition of the `SequenceExample` protobuf:

```
message FeatureList { repeated Feature feature = 1; };
message FeatureLists { map<string, FeatureList> feature_list = 1; };
message SequenceExample {
    Features context = 1;
    FeatureLists feature_lists = 2;
};
```

A `SequenceExample` contains a `Features` object for the contextual data and a `FeatureLists` object that contains one or more named `FeatureList` objects (e.g., a `FeatureList` named `"content"` and another named `"comments"`). Each `FeatureList` contains a list of `Feature` objects, each of which may be a list of byte strings, a list of 64-bit integers, or a list of floats (in this example, each `Feature` would represent a sentence or a comment, perhaps in the form of a list of word identifiers). Building a `SequenceExample`, serializing it, and parsing it is similar to building, serializing, and parsing an `Example`, but you must use `tf.io.parse_single_sequence_example()` to parse a single `SequenceExample` or `tf.io.parse_sequence_example()` to parse a batch. Both functions return a tuple containing the context features (as a dictionary) and the feature lists (also as a dictionary). If the feature lists contain sequences of varying sizes (as in the preceding example), you may want to convert them to ragged tensors, using `tf.RaggedTensor.from_sparse()` (see the notebook for the full code):

```
parsed_context, parsed_feature_lists = tf.io.parse_single_sequence_example(
    serialized_sequence_example, context_feature_descriptions,
    sequence_feature_descriptions)
parsed_content = tf.RaggedTensor.from_sparse(parsed_feature_lists["content"])
```

Now that you know how to efficiently store, load, and parse data, the next step is to prepare it so that it can be fed to a neural network.

Preprocessing the Input Features

Preparing your data for a neural network requires converting all features into numerical features, generally normalizing them, and more. In particular, if your data contains categorical features or text features, they need to be converted to numbers. This can be done ahead of time when preparing your data files, using any tool you like (e.g., NumPy, pandas, or Scikit-Learn). Alternatively, you can preprocess your data on the fly when loading it with the Data API (e.g., using the dataset's `map()` method, as we saw earlier), or you can include a preprocessing layer directly in your model. Let's look at this last option now.

For example, here is how you can implement a standardization layer using a `Lambda` layer. For each feature, it subtracts the mean and divides by its standard deviation (plus a tiny smoothing term to avoid division by zero):

```
means = np.mean(X_train, axis=0, keepdims=True)
stds = np.std(X_train, axis=0, keepdims=True)
eps = keras.backend.epsilon()
model = keras.models.Sequential([
    keras.layers.Lambda(lambda inputs: (inputs - means) / (stds + eps)),
    [...] # other layers
])
```

That's not too hard! However, you may prefer to use a nice self-contained custom layer (much like Scikit-Learn's `StandardScaler`), rather than having global variables like `means` and `stds` dangling around:

```
class Standardization(keras.layers.Layer):
    def adapt(self, data_sample):
        self.means_ = np.mean(data_sample, axis=0, keepdims=True)
        self.stds_ = np.std(data_sample, axis=0, keepdims=True)
    def call(self, inputs):
        return (inputs - self.means_) / (self.stds_ + keras.backend.epsilon())
```

Before you can use this standardization layer, you will need to adapt it to your dataset by calling the `adapt()` method and passing it a data sample. This will allow it to use the appropriate mean and standard deviation for each feature:

```
std_layer = Standardization()
std_layer.adapt(data_sample)
```

This sample must be large enough to be representative of your dataset, but it does not have to be the full training set: in general, a few hundred randomly selected instances will suffice (however, this depends on your task). Next, you can use this preprocessing layer like a normal layer:

```
model = keras.Sequential()
model.add(std_layer)
[...] # create the rest of the model
model.compile([...])
model.fit([...])
```

If you are thinking that Keras should contain a standardization layer like this one, here's some good news for you: by the time you read this, the `keras.layers.Normalization` layer will probably be available. It will work very much like our custom `Standardization` layer: first, create the layer, then adapt it to your dataset by passing a data sample to the `adapt()` method, and finally use the layer normally.

Now let's look at categorical features. We will start by encoding them as one-hot vectors.

Encoding Categorical Features Using One-Hot Vectors

Consider the `ocean_proximity` feature in the California housing dataset we explored in Chapter 2: it is a categorical feature with five possible values: `"<1H OCEAN"`, `"INLAND"`, `"NEAR OCEAN"`, `"NEAR BAY"`, and `"ISLAND"`. We need to encode this feature before we feed it to a neural network. Since there are very few categories, we can use one-hot encoding. For this, we first need to map each category to its index (0 to 4), which can be done using a lookup table:

```
vocab = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
indices = tf.range(len(vocab), dtype=tf.int64)
```

```
table_init = tf.lookup.KeyValueTensorInitializer(vocab, indices)
num_oov_buckets = 2
table = tf.lookup.StaticVocabularyTable(table_init, num_oov_buckets)
```

Let's go through this code:

- We first define the *vocabulary*: this is the list of all possible categories.
- Then we create a tensor with the corresponding indices (0 to 4).
- Next, we create an initializer for the lookup table, passing it the list of categories and their corresponding indices. In this example, we already have this data, so we use a `KeyValueTensorInitializer`; but if the categories were listed in a text file (with one category per line), we would use a `TextFileInitializer` instead.
- In the last two lines we create the lookup table, giving it the initializer and specifying the number of *out-of-vocabulary* (oov) buckets. If we look up a category that does not exist in the vocabulary, the lookup table will compute a hash of this category and use it to assign the unknown category to one of the oov buckets. Their indices start after the known categories, so in this example the indices of the two oov buckets are 5 and 6.

Why use oov buckets? Well, if the number of categories is large (e.g., zip codes, cities, words, products, or users) and the dataset is large as well, or it keeps changing, then getting the full list of categories may not be convenient. One solution is to define the vocabulary based on a data sample (rather than the whole training set) and add some oov buckets for the other categories that were not in the data sample. The more unknown categories you expect to find during training, the more oov buckets you should use. Indeed, if there are not enough oov buckets, there will be collisions: different categories will end up in the same bucket, so the neural network will not be able to distinguish them (at least not based on this feature).

Now let's use the lookup table to encode a small batch of categorical features to one-hot vectors:

```
>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=514, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> cat_one_hot = tf.one_hot(cat_indices, depth=len(vocab) + num_oov_buckets)
>>> cat_one_hot
<tf.Tensor: id=524, shape=(4, 7), dtype=float32, numpy=
array([[0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.]], dtype=float32)>
```

As you can see, `"NEAR BAY"` was mapped to index 3, the unknown category `"DESERT"` was mapped to one of the two oov buckets (at index 5), and `"INLAND"` was mapped to

index 1, twice. Then we used `tf.one_hot()` to one-hot encode these indices. Notice that we have to tell this function the total number of indices, which is equal to the vocabulary size plus the number of oov buckets. Now you know how to encode categorical features to one-hot vectors using TensorFlow!

Just like earlier, it wouldn't be too difficult to bundle all of this logic into a nice self-contained class. Its `adapt()` method would take a data sample and extract all the distinct categories it contains. It would create a lookup table to map each category to its index (including unknown categories using oov buckets). Then its `call()` method would use the lookup table to map the input categories to their indices. Well, here's more good news: by the time you read this, Keras will probably include a layer called `keras.layers.TextVectorization`, which will be capable of doing exactly that: its `adapt()` method will extract the vocabulary from a data sample, and its `call()` method will convert each category to its index in the vocabulary. You could add this layer at the beginning of your model, followed by a `Lambda` layer that would apply the `tf.one_hot()` function, if you want to convert these indices to one-hot vectors.

This may not be the best solution, though. The size of each one-hot vector is the vocabulary length plus the number of oov buckets. This is fine when there are just a few possible categories, but if the vocabulary is large, it is much more efficient to encode them using *embeddings* instead.

As a rule of thumb, if the number of categories is lower than 10, then one-hot encoding is generally the way to go (but your mileage may vary!). If the number of categories is greater than 50 (which is often the case when you use hash buckets), then embeddings are usually preferable. In between 10 and 50 categories, you may want to experiment with both options and see which one works best for your use case.

Encoding Categorical Features Using Embeddings

An embedding is a trainable dense vector that represents a category. By default, embeddings are initialized randomly, so for example the `"NEAR BAY"` category could be represented initially by a random vector such as `[0.131, 0.890]`, while the `"NEAR OCEAN"` category might be represented by another random vector such as `[0.631, 0.791]`. In this example, we use 2D embeddings, but the number of dimensions is a hyperparameter you can tweak. Since these embeddings are trainable, they will gradually improve during training; and as they represent fairly similar categories, Gradient Descent will certainly end up pushing them closer together, while it will tend to move them away from the `"INLAND"` category's embedding (see Figure 13-4). Indeed, the better the representation, the easier it will be for the neural network to make accurate predictions, so training tends to make embeddings useful representations of

the categories. This is called *representation learning* (we will see other types of representation learning in Chapter 17).

Figure 13-4. Embeddings will gradually improve during training

Word Embeddings

Not only will embeddings generally be useful representations for the task at hand, but quite often these same embeddings can be reused successfully for other tasks. The most common example of this is *word embeddings* (i.e., embeddings of individual words): when you are working on a natural language processing task, you are often better off reusing pretrained word embeddings than training your own.

The idea of using vectors to represent words dates back to the 1960s, and many sophisticated techniques have been used to generate useful vectors, including using neural networks. But things really took off in 2013, when Tomáš Mikolov and other Google researchers published a paper[9] describing an efficient technique to learn word embeddings using neural networks, significantly outperforming previous attempts. This allowed them to learn embeddings on a very large corpus of text: they trained a neural network to predict the words near any given word, and obtained astounding word embeddings. For example, synonyms had very close embeddings, and semantically related words such as France, Spain, and Italy ended up clustered together.

It's not just about proximity, though: word embeddings were also organized along meaningful axes in the embedding space. Here is a famous example: if you compute King – Man + Woman (adding and subtracting the embedding vectors of these words), then the result will be very close to the embedding of the word Queen (see Figure 13-5). In other words, the word embeddings encode the concept of gender!

9 Tomas Mikolov et al., "Distributed Representations of Words and Phrases and Their Compositionality," *Proceedings of the 26th International Conference on Neural Information Processing Systems* 2 (2013): 3111–3119.

Similarly, you can compute Madrid – Spain + France, and the result is close to Paris, which seems to show that the notion of capital city was also encoded in the embeddings.

Figure 13-5. Word embeddings of similar words tend to be close, and some axes seem to encode meaningful concepts

Unfortunately, word embeddings sometimes capture our worst biases. For example, although they correctly learn that Man is to King as Woman is to Queen, they also seem to learn that Man is to Doctor as Woman is to Nurse: quite a sexist bias! To be fair, this particular example is probably exaggerated, as was pointed out in a 2019 paper[10] by Malvina Nissim et al. Nevertheless, ensuring fairness in Deep Learning algorithms is an important and active research topic.

Let's look at how we could implement embeddings manually, to understand how they work (then we will use a simple Keras layer instead). First, we need to create an *embedding matrix* containing each category's embedding, initialized randomly; it will have one row per category and per oov bucket, and one column per embedding dimension:

```
embedding_dim = 2
embed_init = tf.random.uniform([len(vocab) + num_oov_buckets, embedding_dim])
embedding_matrix = tf.Variable(embed_init)
```

10 Malvina Nissim et al., "Fair Is Better Than Sensational: Man Is to Doctor as Woman Is to Doctor," arXiv preprint arXiv:1905.09866 (2019).

In this example we are using 2D embeddings, but as a rule of thumb embeddings typically have 10 to 300 dimensions, depending on the task and the vocabulary size (you will have to tune this hyperparameter).

This embedding matrix is a random 6 × 2 matrix, stored in a variable (so it can be tweaked by Gradient Descent during training):

```
>>> embedding_matrix
<tf.Variable 'Variable:0' shape=(6, 2) dtype=float32, numpy=
array([[0.6645621 , 0.44100678],
       [0.3528825 , 0.46448255],
       [0.03366041, 0.68467236],
       [0.74011743, 0.8724445 ],
       [0.22632635, 0.22319686],
       [0.3103881 , 0.7223358 ]], dtype=float32)>
```

Now let's encode the same batch of categorical features as earlier, but this time using these embeddings:

```
>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=741, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> tf.nn.embedding_lookup(embedding_matrix, cat_indices)
<tf.Tensor: id=864, shape=(4, 2), dtype=float32, numpy=
array([[0.74011743, 0.8724445 ],
       [0.3103881 , 0.7223358 ],
       [0.3528825 , 0.46448255],
       [0.3528825 , 0.46448255]], dtype=float32)>
```

The `tf.nn.embedding_lookup()` function looks up the rows in the embedding matrix, at the given indices—that's all it does. For example, the lookup table says that the `"INLAND"` category is at index 1, so the `tf.nn.embedding_lookup()` function returns the embedding at row 1 in the embedding matrix (twice): `[0.3528825, 0.46448255]`.

Keras provides a `keras.layers.Embedding` layer that handles the embedding matrix (trainable, by default); when the layer is created it initializes the embedding matrix randomly, and then when it is called with some category indices it returns the rows at those indices in the embedding matrix:

```
>>> embedding = keras.layers.Embedding(input_dim=len(vocab) + num_oov_buckets,
...                                    output_dim=embedding_dim)
...
>>> embedding(cat_indices)
<tf.Tensor: id=814, shape=(4, 2), dtype=float32, numpy=
array([[ 0.02401174,  0.03724445],
       [-0.01896119,  0.02223358],
       [-0.01471175, -0.00355174],
       [-0.01471175, -0.00355174]], dtype=float32)>
```

Putting everything together, we can now create a Keras model that can process categorical features (along with regular numerical features) and learn an embedding for each category (as well as for each oov bucket):

```
regular_inputs = keras.layers.Input(shape=[8])
categories = keras.layers.Input(shape=[], dtype=tf.string)
cat_indices = keras.layers.Lambda(lambda cats: table.lookup(cats))(categories)
cat_embed = keras.layers.Embedding(input_dim=6, output_dim=2)(cat_indices)
encoded_inputs = keras.layers.concatenate([regular_inputs, cat_embed])
outputs = keras.layers.Dense(1)(encoded_inputs)
model = keras.models.Model(inputs=[regular_inputs, categories],
                           outputs=[outputs])
```

This model takes two inputs: a regular input containing eight numerical features per instance, plus a categorical input (containing one categorical feature per instance). It uses a `Lambda` layer to look up each category's index, then it looks up the embeddings for these indices. Next, it concatenates the embeddings and the regular inputs in order to give the encoded inputs, which are ready to be fed to a neural network. We could add any kind of neural network at this point, but we just add a dense output layer, and we create the Keras model.

When the `keras.layers.TextVectorization` layer is available, you can call its `adapt()` method to make it extract the vocabulary from a data sample (it will take care of creating the lookup table for you). Then you can add it to your model, and it will perform the index lookup (replacing the `Lambda` layer in the previous code example).

One-hot encoding followed by a `Dense` layer (with no activation function and no biases) is equivalent to an `Embedding` layer. However, the `Embedding` layer uses way fewer computations (the performance difference becomes clear when the size of the embedding matrix grows). The `Dense` layer's weight matrix plays the role of the embedding matrix. For example, using one-hot vectors of size 20 and a `Dense` layer with 10 units is equivalent to using an `Embedding` layer with `input_dim=20` and `output_dim=10`. As a result, it would be wasteful to use more embedding dimensions than the number of units in the layer that follows the `Embedding` layer.

Now let's look a bit more closely at the Keras preprocessing layers.

Keras Preprocessing Layers

The TensorFlow team is working on providing a set of standard Keras preprocessing layers. They will probably be available by the time you read this; however, the API may change slightly by then, so please refer to the notebook for this chapter if anything behaves unexpectedly. This new API will likely supersede the existing Feature

Columns API, which is harder to use and less intuitive (if you want to learn more about the Feature Columns API anyway, please check out the notebook for this chapter).

We already discussed two of these layers: the `keras.layers.Normalization` layer that will perform feature standardization (it will be equivalent to the `Standardization` layer we defined earlier), and the `TextVectorization` layer that will be capable of encoding each word in the inputs into its index in the vocabulary. In both cases, you create the layer, you call its `adapt()` method with a data sample, and then you use the layer normally in your model. The other preprocessing layers will follow the same pattern.

The API will also include a `keras.layers.Discretization` layer that will chop continuous data into different bins and encode each bin as a one-hot vector. For example, you could use it to discretize prices into three categories, (low, medium, high), which would be encoded as [1, 0, 0], [0, 1, 0], and [0, 0, 1], respectively. Of course this loses a lot of information, but in some cases it can help the model detect patterns that would otherwise not be obvious when just looking at the continuous values.

The `Discretization` layer will not be differentiable, and it should only be used at the start of your model. Indeed, the model's preprocessing layers will be frozen during training, so their parameters will not be affected by Gradient Descent, and thus they do not need to be differentiable. This also means that you should not use an `Embedding` layer directly in a custom preprocessing layer, if you want it to be trainable: instead, it should be added separately to your model, as in the previous code example.

It will also be possible to chain multiple preprocessing layers using the `PreprocessingStage` class. For example, the following code will create a preprocessing pipeline that will first normalize the inputs, then discretize them (this may remind you of Scikit-Learn pipelines). After you adapt this pipeline to a data sample, you can use it like a regular layer in your models (but again, only at the start of the model, since it contains a nondifferentiable preprocessing layer):

```
normalization = keras.layers.Normalization()
discretization = keras.layers.Discretization([...])
pipeline = keras.layers.PreprocessingStage([normalization, discretization])
pipeline.adapt(data_sample)
```

The `TextVectorization` layer will also have an option to output word-count vectors instead of word indices. For example, if the vocabulary contains three words, say `["and", "basketball", "more"]`, then the text `"more and more"` will be mapped to the vector `[1, 0, 2]`: the word `"and"` appears once, the word `"basketball"` does not appear at all, and the word `"more"` appears twice. This text representation is called a

bag of words, since it completely loses the order of the words. Common words like `"and"` will have a large value in most texts, even though they are usually the least interesting (e.g., in the text `"more and more basketball"` the word `"basketball"` is clearly the most important, precisely because it is not a very frequent word). So, the word counts should be normalized in a way that reduces the importance of frequent words. A common way to do this is to divide each word count by the log of the total number of training instances in which the word appears. This technique is called *Term-Frequency* × *Inverse-Document-Frequency* (TF-IDF). For example, let's imagine that the words `"and"`, `"basketball"`, and `"more"` appear respectively in 200, 10, and 100 text instances in the training set: in this case, the final vector will be `[1/log(200), 0/log(10), 2/log(100)]`, which is approximately equal to `[0.19, 0., 0.43]`. The `TextVectorization` layer will (likely) have an option to perform TF-IDF.

If the standard preprocessing layers are insufficient for your task, you will still have the option to create your own custom preprocessing layer, much like we did earlier with the `Standardization` class. Create a subclass of the `keras.layers.PreprocessingLayer` class with an `adapt()` method, which should take a `data_sample` argument and optionally an extra `reset_state` argument: if `True`, then the `adapt()` method should reset any existing state before computing the new state; if `False`, it should try to update the existing state.

As you can see, these Keras preprocessing layers will make preprocessing much easier! Now, whether you choose to write your own preprocessing layers or use Keras's (or even use the Feature Columns API), all the preprocessing will be done on the fly. During training, however, it may be preferable to perform preprocessing ahead of time. Let's see why we'd want to do that and how we'd go about it.

TF Transform

If preprocessing is computationally expensive, then handling it before training rather than on the fly may give you a significant speedup: the data will be preprocessed just once per instance *before* training, rather than once per instance and per epoch *during* training. As mentioned earlier, if the dataset is small enough to fit in RAM, you can use its `cache()` method. But if it is too large, then tools like Apache Beam or Spark will help. They let you run efficient data processing pipelines over large amounts of data, even distributed across multiple servers, so you can use them to preprocess all the training data before training.

This works great and indeed can speed up training, but there is one problem: once your model is trained, suppose you want to deploy it to a mobile app. In that case you will need to write some code in your app to take care of preprocessing the data before

it is fed to the model. And suppose you also want to deploy the model to TensorFlow.js so that it runs in a web browser? Once again, you will need to write some preprocessing code. This can become a maintenance nightmare: whenever you want to change the preprocessing logic, you will need to update your Apache Beam code, your mobile app code, and your JavaScript code. This is not only time-consuming, but also error-prone: you may end up with subtle differences between the preprocessing operations performed before training and the ones performed in your app or in the browser. This *training/serving skew* will lead to bugs or degraded performance.

One improvement would be to take the trained model (trained on data that was preprocessed by your Apache Beam or Spark code) and, before deploying it to your app or the browser, add extra preprocessing layers to take care of preprocessing on the fly. That's definitely better, since now you just have two versions of your preprocessing code: the Apache Beam or Spark code, and the preprocessing layers' code.

But what if you could define your preprocessing operations just once? This is what TF Transform was designed for. It is part of TensorFlow Extended (TFX), an end-to-end platform for productionizing TensorFlow models. First, to use a TFX component such as TF Transform, you must install it; it does not come bundled with TensorFlow. You then define your preprocessing function just once (in Python), by using TF Transform functions for scaling, bucketizing, and more. You can also use any TensorFlow operation you need. Here is what this preprocessing function might look like if we just had two features:

```
import tensorflow_transform as tft

def preprocess(inputs):  # inputs = a batch of input features
    median_age = inputs["housing_median_age"]
    ocean_proximity = inputs["ocean_proximity"]
    standardized_age = tft.scale_to_z_score(median_age)
    ocean_proximity_id = tft.compute_and_apply_vocabulary(ocean_proximity)
    return {
        "standardized_median_age": standardized_age,
        "ocean_proximity_id": ocean_proximity_id
    }
```

Next, TF Transform lets you apply this `preprocess()` function to the whole training set using Apache Beam (it provides an `AnalyzeAndTransformDataset` class that you can use for this purpose in your Apache Beam pipeline). In the process, it will also compute all the necessary statistics over the whole training set: in this example, the mean and standard deviation of the `housing_median_age` feature, and the vocabulary for the `ocean_proximity` feature. The components that compute these statistics are called *analyzers*.

Importantly, TF Transform will also generate an equivalent TensorFlow Function that you can plug into the model you deploy. This TF Function includes some constants

that correspond to all the all the necessary statistics computed by Apache Beam (the mean, standard deviation, and vocabulary).

With the Data API, TFRecords, the Keras preprocessing layers, and TF Transform, you can build highly scalable input pipelines for training and benefit from fast and portable data preprocessing in production.

But what if you just wanted to use a standard dataset? Well in that case, things are much simpler: just use TFDS!

The TensorFlow Datasets (TFDS) Project

The TensorFlow Datasets project makes it very easy to download common datasets, from small ones like MNIST or Fashion MNIST to huge datasets like ImageNet (you will need quite a bit of disk space!). The list includes image datasets, text datasets (including translation datasets), and audio and video datasets. You can visit *https://homl.info/tfds* to view the full list, along with a description of each dataset.

TFDS is not bundled with TensorFlow, so you need to install the `tensorflow-datasets` library (e.g., using pip). Then call the `tfds.load()` function, and it will download the data you want (unless it was already downloaded earlier) and return the data as a dictionary of datasets (typically one for training and one for testing, but this depends on the dataset you choose). For example, let's download MNIST:

```
import tensorflow_datasets as tfds

dataset = tfds.load(name="mnist")
mnist_train, mnist_test = dataset["train"], dataset["test"]
```

You can then apply any transformation you want (typically shuffling, batching, and prefetching), and you're ready to train your model. Here is a simple example:

```
mnist_train = mnist_train.shuffle(10000).batch(32).prefetch(1)
for item in mnist_train:
    images = item["image"]
    labels = item["label"]
    [...]
```

The `load()` function shuffles each data shard it downloads (only for the training set). This may not be sufficient, so it's best to shuffle the training data some more.

Note that each item in the dataset is a dictionary containing both the features and the labels. But Keras expects each item to be a tuple containing two elements (again, the features and the labels). You could transform the dataset using the `map()` method, like this:

```
mnist_train = mnist_train.shuffle(10000).batch(32)
mnist_train = mnist_train.map(lambda items: (items["image"], items["label"]))
mnist_train = mnist_train.prefetch(1)
```

But it's simpler to ask the `load()` function to do this for you by setting `as_supervised=True` (obviously this works only for labeled datasets). You can also specify the batch size if you want. Then you can pass the dataset directly to your tf.keras model:

```
dataset = tfds.load(name="mnist", batch_size=32, as_supervised=True)
mnist_train = dataset["train"].prefetch(1)
model = keras.models.Sequential([...])
model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd")
model.fit(mnist_train, epochs=5)
```

This was quite a technical chapter, and you may feel that it is a bit far from the abstract beauty of neural networks, but the fact is Deep Learning often involves large amounts of data, and knowing how to load, parse, and preprocess it efficiently is a crucial skill to have. In the next chapter, we will look at convolutional neural networks, which are among the most successful neural net architectures for image processing and many other applications.

Exercises

1. Why would you want to use the Data API?
2. What are the benefits of splitting a large dataset into multiple files?
3. During training, how can you tell that your input pipeline is the bottleneck? What can you do to fix it?
4. Can you save any binary data to a TFRecord file, or only serialized protocol buffers?
5. Why would you go through the hassle of converting all your data to the `Example` protobuf format? Why not use your own protobuf definition?
6. When using TFRecords, when would you want to activate compression? Why not do it systematically?
7. Data can be preprocessed directly when writing the data files, or within the tf.data pipeline, or in preprocessing layers within your model, or using TF Transform. Can you list a few pros and cons of each option?
8. Name a few common techniques you can use to encode categorical features. What about text?
9. Load the Fashion MNIST dataset (introduced in Chapter 10); split it into a training set, a validation set, and a test set; shuffle the training set; and save each dataset to multiple TFRecord files. Each record should be a serialized `Example` protobuf with two features: the serialized image (use `tf.io.serialize_tensor()`

to serialize each image), and the label.[11] Then use tf.data to create an efficient dataset for each set. Finally, use a Keras model to train these datasets, including a preprocessing layer to standardize each input feature. Try to make the input pipeline as efficient as possible, using TensorBoard to visualize profiling data.

10. In this exercise you will download a dataset, split it, create a `tf.data.Dataset` to load it and preprocess it efficiently, then build and train a binary classification model containing an `Embedding` layer:

    a. Download the Large Movie Review Dataset, which contains 50,000 movies reviews from the Internet Movie Database. The data is organized in two directories, *train* and *test*, each containing a *pos* subdirectory with 12,500 positive reviews and a *neg* subdirectory with 12,500 negative reviews. Each review is stored in a separate text file. There are other files and folders (including preprocessed bag-of-words), but we will ignore them in this exercise.

    b. Split the test set into a validation set (15,000) and a test set (10,000).

    c. Use tf.data to create an efficient dataset for each set.

    d. Create a binary classification model, using a `TextVectorization` layer to preprocess each review. If the `TextVectorization` layer is not yet available (or if you like a challenge), try to create your own custom preprocessing layer: you can use the functions in the `tf.strings` package, for example `lower()` to make everything lowercase, `regex_replace()` to replace punctuation with spaces, and `split()` to split words on spaces. You should use a lookup table to output word indices, which must be prepared in the `adapt()` method.

    e. Add an `Embedding` layer and compute the mean embedding for each review, multiplied by the square root of the number of words (see Chapter 16). This rescaled mean embedding can then be passed to the rest of your model.

    f. Train the model and see what accuracy you get. Try to optimize your pipelines to make training as fast as possible.

    g. Use TFDS to load the same dataset more easily: `tfds.load("imdb_reviews")`.

Solutions to these exercises are available in Appendix A.

11 For large images, you could use `tf.io.encode_jpeg()` instead. This would save a lot of space, but it would lose a bit of image quality.

CHAPTER 14

Deep Computer Vision Using Convolutional Neural Networks

Although IBM's Deep Blue supercomputer beat the chess world champion Garry Kasparov back in 1996, it wasn't until fairly recently that computers were able to reliably perform seemingly trivial tasks such as detecting a puppy in a picture or recognizing spoken words. Why are these tasks so effortless to us humans? The answer lies in the fact that perception largely takes place outside the realm of our consciousness, within specialized visual, auditory, and other sensory modules in our brains. By the time sensory information reaches our consciousness, it is already adorned with high-level features; for example, when you look at a picture of a cute puppy, you cannot choose *not* to see the puppy, *not* to notice its cuteness. Nor can you explain *how* you recognize a cute puppy; it's just obvious to you. Thus, we cannot trust our subjective experience: perception is not trivial at all, and to understand it we must look at how the sensory modules work.

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and they have been used in image recognition since the 1980s. In the last few years, thanks to the increase in computational power, the amount of available training data, and the tricks presented in Chapter 11 for training deep nets, CNNs have managed to achieve superhuman performance on some complex visual tasks. They power image search services, self-driving cars, automatic video classification systems, and more. Moreover, CNNs are not restricted to visual perception: they are also successful at many other tasks, such as voice recognition and natural language processing. However, we will focus on visual applications for now.

In this chapter we will explore where CNNs came from, what their building blocks look like, and how to implement them using TensorFlow and Keras. Then we will discuss some of the best CNN architectures, as well as other visual tasks, including

object detection (classifying multiple objects in an image and placing bounding boxes around them) and semantic segmentation (classifying each pixel according to the class of the object it belongs to).

The Architecture of the Visual Cortex

David H. Hubel and Torsten Wiesel performed a series of experiments on cats in 1958[1] and 1959[2] (and a few years later on monkeys[3]), giving crucial insights into the structure of the visual cortex (the authors received the Nobel Prize in Physiology or Medicine in 1981 for their work). In particular, they showed that many neurons in the visual cortex have a small *local receptive field*, meaning they react only to visual stimuli located in a limited region of the visual field (see Figure 14-1, in which the local receptive fields of five neurons are represented by dashed circles). The receptive fields of different neurons may overlap, and together they tile the whole visual field.

Moreover, the authors showed that some neurons react only to images of horizontal lines, while others react only to lines with different orientations (two neurons may have the same receptive field but react to different line orientations). They also noticed that some neurons have larger receptive fields, and they react to more complex patterns that are combinations of the lower-level patterns. These observations led to the idea that the higher-level neurons are based on the outputs of neighboring lower-level neurons (in Figure 14-1, notice that each neuron is connected only to a few neurons from the previous layer). This powerful architecture is able to detect all sorts of complex patterns in any area of the visual field.

1 David H. Hubel, "Single Unit Activity in Striate Cortex of Unrestrained Cats," *The Journal of Physiology* 147 (1959): 226–238.

2 David H. Hubel and Torsten N. Wiesel, "Receptive Fields of Single Neurons in the Cat's Striate Cortex," *The Journal of Physiology* 148 (1959): 574–591.

3 David H. Hubel and Torsten N. Wiesel, "Receptive Fields and Functional Architecture of Monkey Striate Cortex," *The Journal of Physiology* 195 (1968): 215–243.

Figure 14-1. Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field called receptive fields; as the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields.

These studies of the visual cortex inspired the neocognitron,[4] introduced in 1980, which gradually evolved into what we now call *convolutional neural networks*. An important milestone was a 1998 paper[5] by Yann LeCun et al. that introduced the famous *LeNet-5* architecture, widely used by banks to recognize handwritten check numbers. This architecture has some building blocks that you already know, such as fully connected layers and sigmoid activation functions, but it also introduces two new building blocks: *convolutional layers* and *pooling layers*. Let's look at them now.

Why not simply use a deep neural network with fully connected layers for image recognition tasks? Unfortunately, although this works fine for small images (e.g., MNIST), it breaks down for larger images because of the huge number of parameters it requires. For example, a 100 × 100–pixel image has 10,000 pixels, and if the first layer has just 1,000 neurons (which already severely restricts the amount of information transmitted to the next layer), this means a total of 10 million connections. And that's just the first layer. CNNs solve this problem using partially connected layers and weight sharing.

4 Kunihiko Fukushima, "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position," *Biological Cybernetics* 36 (1980): 193–202.

5 Yann LeCun et al., "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE* 86, no. 11 (1998): 2278–2324.

Convolutional Layers

The most important building block of a CNN is the *convolutional layer*:[6] neurons in the first convolutional layer are not connected to every single pixel in the input image (like they were in the layers discussed in previous chapters), but only to pixels in their receptive fields (see Figure 14-2). In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer. This architecture allows the network to concentrate on small low-level features in the first hidden layer, then assemble them into larger higher-level features in the next hidden layer, and so on. This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition.

Figure 14-2. CNN layers with rectangular local receptive fields

All the multilayer neural networks we've looked at so far had layers composed of a long line of neurons, and we had to flatten input images to 1D before feeding them to the neural network. In a CNN each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs.

6 A convolution is a mathematical operation that slides one function over another and measures the integral of their pointwise multiplication. It has deep connections with the Fourier transform and the Laplace transform and is heavily used in signal processing. Convolutional layers actually use cross-correlations, which are very similar to convolutions (see *https://homl.info/76* for more details).

A neuron located in row i, column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$, where f_h and f_w are the height and width of the receptive field (see Figure 14-3). In order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs, as shown in the diagram. This is called *zero padding*.

Figure 14-3. Connections between layers and zero padding

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields, as shown in Figure 14-4. This dramatically reduces the model's computational complexity. The shift from one receptive field to the next is called the *stride*. In the diagram, a 5×7 input layer (plus zero padding) is connected to a 3×4 layer, using 3×3 receptive fields and a stride of 2 (in this example the stride is the same in both directions, but it does not have to be so). A neuron located in row i, column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows $i \times s_h$ to $i \times s_h + f_h - 1$, columns $j \times s_w$ to $j \times s_w + f_w - 1$, where s_h and s_w are the vertical and horizontal strides.

Figure 14-4. Reducing dimensionality using a stride of 2

Filters

A neuron's weights can be represented as a small image the size of the receptive field. For example, Figure 14-5 shows two possible sets of weights, called *filters* (or *convolution kernels*). The first one is represented as a black square with a vertical white line in the middle (it is a 7 × 7 matrix full of 0s except for the central column, which is full of 1s); neurons using these weights will ignore everything in their receptive field except for the central vertical line (since all inputs will get multiplied by 0, except for the ones located in the central vertical line). The second filter is a black square with a horizontal white line in the middle. Once again, neurons using these weights will ignore everything in their receptive field except for the central horizontal line.

Now if all neurons in a layer use the same vertical line filter (and the same bias term), and you feed the network the input image shown in Figure 14-5 (the bottom image), the layer will output the top-left image. Notice that the vertical white lines get enhanced while the rest gets blurred. Similarly, the upper-right image is what you get if all neurons use the same horizontal line filter; notice that the horizontal white lines get enhanced while the rest is blurred out. Thus, a layer full of neurons using the same filter outputs a *feature map*, which highlights the areas in an image that activate the filter the most. Of course, you do not have to define the filters manually: instead, during training the convolutional layer will automatically learn the most useful filters for its task, and the layers above will learn to combine them into more complex patterns.

Figure 14-5. Applying two different filters to get two feature maps

Stacking Multiple Feature Maps

Up to now, for simplicity, I have represented the output of each convolutional layer as a 2D layer, but in reality a convolutional layer has multiple filters (you decide how many) and outputs one feature map per filter, so it is more accurately represented in 3D (see Figure 14-6). It has one neuron per pixel in each feature map, and all neurons within a given feature map share the same parameters (i.e., the same weights and bias term). Neurons in different feature maps use different parameters. A neuron's receptive field is the same as described earlier, but it extends across all the previous layers' feature maps. In short, a convolutional layer simultaneously applies multiple trainable filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.

The fact that all neurons in a feature map share the same parameters dramatically reduces the number of parameters in the model. Once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location. In contrast, once a regular DNN has learned to recognize a pattern in one location, it can recognize it only in that particular location.

Input images are also composed of multiple sublayers: one per *color channel*. There are typically three: red, green, and blue (RGB). Grayscale images have just one

channel, but some images may have much more—for example, satellite images that capture extra light frequencies (such as infrared).

Figure 14-6. Convolutional layers with multiple feature maps, and images with three color channels

Specifically, a neuron located in row i, column j of the feature map k in a given convolutional layer l is connected to the outputs of the neurons in the previous layer $l - 1$, located in rows $i \times s_h$ to $i \times s_h + f_h - 1$ and columns $j \times s_w$ to $j \times s_w + f_w - 1$, across all feature maps (in layer $l - 1$). Note that all neurons located in the same row i and column j but in different feature maps are connected to the outputs of the exact same neurons in the previous layer.

Equation 14-1 summarizes the preceding explanations in one big mathematical equation: it shows how to compute the output of a given neuron in a convolutional layer.

It is a bit ugly due to all the different indices, but all it does is calculate the weighted sum of all the inputs, plus the bias term.

Equation 14-1. Computing the output of a neuron in a convolutional layer

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h - 1} \sum_{v=0}^{f_w - 1} \sum_{k'=0}^{f_{n'} - 1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with} \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

In this equation:

- $z_{i,j,k}$ is the output of the neuron located in row i, column j in feature map k of the convolutional layer (layer l).
- As explained earlier, s_h and s_w are the vertical and horizontal strides, f_h and f_w are the height and width of the receptive field, and $f_{n'}$ is the number of feature maps in the previous layer (layer $l - 1$).
- $x_{i',j',k'}$ is the output of the neuron located in layer $l - 1$, row i', column j', feature map k' (or channel k' if the previous layer is the input layer).
- b_k is the bias term for feature map k (in layer l). You can think of it as a knob that tweaks the overall brightness of the feature map k.
- $w_{u,v,k',k}$ is the connection weight between any neuron in feature map k of the layer l and its input located at row u, column v (relative to the neuron's receptive field), and feature map k'.

TensorFlow Implementation

In TensorFlow, each input image is typically represented as a 3D tensor of shape [*height, width, channels*]. A mini-batch is represented as a 4D tensor of shape [*mini-batch size, height, width, channels*]. The weights of a convolutional layer are represented as a 4D tensor of shape $[f_h, f_w, f_{n'}, f_n]$. The bias terms of a convolutional layer are simply represented as a 1D tensor of shape $[f_n]$.

Let's look at a simple example. The following code loads two sample images, using Scikit-Learn's `load_sample_image()` (which loads two color images, one of a Chinese temple, and the other of a flower), then it creates two filters and applies them to both images, and finally it displays one of the resulting feature maps. Note that you must pip install the `Pillow` package to use `load_sample_image()`.

```
from sklearn.datasets import load_sample_image

# Load sample images
china = load_sample_image("china.jpg") / 255
flower = load_sample_image("flower.jpg") / 255
```

```
images = np.array([china, flower])
batch_size, height, width, channels = images.shape

# Create 2 filters
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1  # vertical line
filters[3, :, :, 1] = 1  # horizontal line

outputs = tf.nn.conv2d(images, filters, strides=1, padding="SAME")

plt.imshow(outputs[0, :, :, 1], cmap="gray") # plot 1st image's 2nd feature map
plt.show()
```

Let's go through this code:

- The pixel intensity for each color channel is represented as a byte from 0 to 255, so we scale these features simply by dividing by 255, to get floats ranging from 0 to 1.
- Then we create two 7×7 filters (one with a vertical white line in the middle, and the other with a horizontal white line in the middle).
- We apply them to both images using the `tf.nn.conv2d()` function, which is part of TensorFlow's low-level Deep Learning API. In this example, we use zero padding (`padding="SAME"`) and a stride of 1.
- Finally, we plot one of the resulting feature maps (similar to the top-right image in Figure 14-5).

The `tf.nn.conv2d()` line deserves a bit more explanation:

- `images` is the input mini-batch (a 4D tensor, as explained earlier).
- `filters` is the set of filters to apply (also a 4D tensor, as explained earlier).
- `strides` is equal to `1`, but it could also be a 1D array with four elements, where the two central elements are the vertical and horizontal strides (s_h and s_w). The first and last elements must currently be equal to `1`. They may one day be used to specify a batch stride (to skip some instances) and a channel stride (to skip some of the previous layer's feature maps or channels).
- `padding` must be either `"SAME"` or `"VALID"`:
  - If set to `"SAME"`, the convolutional layer uses zero padding if necessary. The output size is set to the number of input neurons divided by the stride, rounded up. For example, if the input size is 13 and the stride is 5 (see Figure 14-7), then the output size is 3 (i.e., 13 / 5 = 2.6, rounded up to 3). Then zeros are added as evenly as possible around the inputs, as needed. When `strides=1`, the layer's outputs will have the same spatial dimensions (width and height) as its inputs, hence the name *same*.

— If set to "VALID", the convolutional layer does *not* use zero padding and may ignore some rows and columns at the bottom and right of the input image, depending on the stride, as shown in Figure 14-7 (for simplicity, only the horizontal dimension is shown here, but of course the same logic applies to the vertical dimension). This means that every neuron's receptive field lies strictly within valid positions inside the input (it does not go out of bounds), hence the name *valid*.

Figure 14-7. Padding="SAME" or "VALID" (with input width 13, filter width 6, stride 5)

In this example we manually defined the filters, but in a real CNN you would normally define filters as trainable variables so the neural net can learn which filters work best, as explained earlier. Instead of manually creating the variables, use the `keras.layers.Conv2D` layer:

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,
                           padding="same", activation="relu")
```

This code creates a `Conv2D` layer with 32 filters, each 3 × 3, using a stride of 1 (both horizontally and vertically) and `"same"` padding, and applying the ReLU activation function to its outputs. As you can see, convolutional layers have quite a few hyperparameters: you must choose the number of filters, their height and width, the strides, and the padding type. As always, you can use cross-validation to find the right hyperparameter values, but this is very time-consuming. We will discuss common CNN architectures later, to give you some idea of which hyperparameter values work best in practice.

Memory Requirements

Another problem with CNNs is that the convolutional layers require a huge amount of RAM. This is especially true during training, because the reverse pass of backpropagation requires all the intermediate values computed during the forward pass.

For example, consider a convolutional layer with 5 × 5 filters, outputting 200 feature maps of size 150 × 100, with stride 1 and `"same"` padding. If the input is a 150 × 100 RGB image (three channels), then the number of parameters is (5 × 5 × 3 + 1) × 200 = 15,200 (the + 1 corresponds to the bias terms), which is fairly small compared to a fully connected layer.[7] However, each of the 200 feature maps contains 150 × 100 neurons, and each of these neurons needs to compute a weighted sum of its 5 × 5 × 3 = 75 inputs: that's a total of 225 million float multiplications. Not as bad as a fully connected layer, but still quite computationally intensive. Moreover, if the feature maps are represented using 32-bit floats, then the convolutional layer's output will occupy 200 × 150 × 100 × 32 = 96 million bits (12 MB) of RAM.[8] And that's just for one instance—if a training batch contains 100 instances, then this layer will use up 1.2 GB of RAM!

During inference (i.e., when making a prediction for a new instance) the RAM occupied by one layer can be released as soon as the next layer has been computed, so you only need as much RAM as required by two consecutive layers. But during training everything computed during the forward pass needs to be preserved for the reverse pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers.

If training crashes because of an out-of-memory error, you can try reducing the mini-batch size. Alternatively, you can try reducing dimensionality using a stride, or removing a few layers. Or you can try using 16-bit floats instead of 32-bit floats. Or you could distribute the CNN across multiple devices.

Now let's look at the second common building block of CNNs: the *pooling layer*.

Pooling Layers

Once you understand how convolutional layers work, the pooling layers are quite easy to grasp. Their goal is to *subsample* (i.e., shrink) the input image in order to

7 A fully connected layer with 150 × 100 neurons, each connected to all 150 × 100 × 3 inputs, would have $150^2 \times 100^2 \times 3$ = 675 million parameters!

8 In the international system of units (SI), 1 MB = 1,000 KB = 1,000 × 1,000 bytes = 1,000 × 1,000 × 8 bits.

reduce the computational load, the memory usage, and the number of parameters (thereby limiting the risk of overfitting).

Just like in convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. You must define its size, the stride, and the padding type, just like before. However, a pooling neuron has no weights; all it does is aggregate the inputs using an aggregation function such as the max or mean. Figure 14-8 shows a *max pooling layer*, which is the most common type of pooling layer. In this example, we use a 2 × 2 *pooling kernel*,[9] with a stride of 2 and no padding. Only the max input value in each receptive field makes it to the next layer, while the other inputs are dropped. For example, in the lower-left receptive field in Figure 14-8, the input values are 1, 5, 3, 2, so only the max value, 5, is propagated to the next layer. Because of the stride of 2, the output image has half the height and half the width of the input image (rounded down since we use no padding).

Figure 14-8. Max pooling layer (2 × 2 pooling kernel, stride 2, no padding)

A pooling layer typically works on every input channel independently, so the output depth is the same as the input depth.

Other than reducing computations, memory usage, and the number of parameters, a max pooling layer also introduces some level of *invariance* to small translations, as shown in Figure 14-9. Here we assume that the bright pixels have a lower value than dark pixels, and we consider three images (A, B, C) going through a max pooling layer with a 2 × 2 kernel and stride 2. Images B and C are the same as image A, but

9 Other kernels we've discussed so far had weights, but pooling kernels do not: they are just stateless sliding windows.

shifted by one and two pixels to the right. As you can see, the outputs of the max pooling layer for images A and B are identical. This is what translation invariance means. For image C, the output is different: it is shifted one pixel to the right (but there is still 75% invariance). By inserting a max pooling layer every few layers in a CNN, it is possible to get some level of translation invariance at a larger scale. Moreover, max pooling offers a small amount of rotational invariance and a slight scale invariance. Such invariance (even if it is limited) can be useful in cases where the prediction should not depend on these details, such as in classification tasks.

Figure 14-9. Invariance to small translations

However, max pooling has some downsides too. Firstly, it is obviously very destructive: even with a tiny 2 × 2 kernel and a stride of 2, the output will be two times smaller in both directions (so its area will be four times smaller), simply dropping 75% of the input values. And in some applications, invariance is not desirable. Take semantic segmentation (the task of classifying each pixel in an image according to the object that pixel belongs to, which we'll explore later in this chapter): obviously, if the input image is translated by one pixel to the right, the output should also be translated by one pixel to the right. The goal in this case is *equivariance*, not invariance: a small change to the inputs should lead to a corresponding small change in the output.

TensorFlow Implementation

Implementing a max pooling layer in TensorFlow is quite easy. The following code creates a max pooling layer using a 2 × 2 kernel. The strides default to the kernel size, so this layer will use a stride of 2 (both horizontally and vertically). By default, it uses `"valid"` padding (i.e., no padding at all):

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

To create an *average pooling layer*, just use `AvgPool2D` instead of `MaxPool2D`. As you might expect, it works exactly like a max pooling layer, except it computes the mean rather than the max. Average pooling layers used to be very popular, but people mostly use max pooling layers now, as they generally perform better. This may seem surprising, since computing the mean generally loses less information than computing the max. But on the other hand, max pooling preserves only the strongest features, getting rid of all the meaningless ones, so the next layers get a cleaner signal to work with. Moreover, max pooling offers stronger translation invariance than average pooling, and it requires slightly less compute.

Note that max pooling and average pooling can be performed along the depth dimension rather than the spatial dimensions, although this is not as common. This can allow the CNN to learn to be invariant to various features. For example, it could learn multiple filters, each detecting a different rotation of the same pattern (such as handwritten digits; see Figure 14-10), and the depthwise max pooling layer would ensure that the output is the same regardless of the rotation. The CNN could similarly learn to be invariant to anything else: thickness, brightness, skew, color, and so on.

Figure 14-10. Depthwise max pooling can help the CNN learn any invariance

Keras does not include a depthwise max pooling layer, but TensorFlow's low-level Deep Learning API does: just use the `tf.nn.max_pool()` function, and specify the kernel size and strides as 4-tuples (i.e., tuples of size 4). The first three values of each should be 1: this indicates that the kernel size and stride along the batch, height, and width dimensions should be 1. The last value should be whatever kernel size and stride you want along the depth dimension—for example, 3 (this must be a divisor of the input depth; it will not work if the previous layer outputs 20 feature maps, since 20 is not a multiple of 3):

```
output = tf.nn.max_pool(images,
                        ksize=(1, 1, 1, 3),
                        strides=(1, 1, 1, 3),
                        padding="valid")
```

If you want to include this as a layer in your Keras models, wrap it in a `Lambda` layer (or create a custom Keras layer):

```
depth_pool = keras.layers.Lambda(
    lambda X: tf.nn.max_pool(X, ksize=(1, 1, 1, 3), strides=(1, 1, 1, 3),
                             padding="valid"))
```

One last type of pooling layer that you will often see in modern architectures is the *global average pooling layer*. It works very differently: all it does is compute the mean of each entire feature map (it's like an average pooling layer using a pooling kernel with the same spatial dimensions as the inputs). This means that it just outputs a single number per feature map and per instance. Although this is of course extremely destructive (most of the information in the feature map is lost), it can be useful as the output layer, as we will see later in this chapter. To create such a layer, simply use the `keras.layers.GlobalAvgPool2D` class:

```
global_avg_pool = keras.layers.GlobalAvgPool2D()
```

It's equivalent to this simple `Lambda` layer, which computes the mean over the spatial dimensions (height and width):

```
global_avg_pool = keras.layers.Lambda(lambda X: tf.reduce_mean(X, axis=[1, 2]))
```

Now you know all the building blocks to create convolutional neural networks. Let's see how to assemble them.

CNN Architectures

Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps), thanks to the convolutional layers (see Figure 14-11). At the top of the stack, a regular feedforward neural network is added, composed of a few

fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).

Figure 14-11. Typical CNN architecture

A common mistake is to use convolution kernels that are too large. For example, instead of using a convolutional layer with a 5 × 5 kernel, stack two layers with 3 × 3 kernels: it will use fewer parameters and require fewer computations, and it will usually perform better. One exception is for the first convolutional layer: it can typically have a large kernel (e.g., 5 × 5), usually with a stride of 2 or more: this will reduce the spatial dimension of the image without losing too much information, and since the input image only has three channels in general, it will not be too costly.

Here is how you can implement a simple CNN to tackle the Fashion MNIST dataset (introduced in Chapter 10):

```
model = keras.models.Sequential([
    keras.layers.Conv2D(64, 7, activation="relu", padding="same",
                        input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation="softmax")
])
```

Let's go through this model:

- The first layer uses 64 fairly large filters (7 × 7) but no stride because the input images are not very large. It also sets `input_shape=[28, 28, 1]`, because the images are 28 × 28 pixels, with a single color channel (i.e., grayscale).
- Next we have a max pooling layer which uses a pool size of 2, so it divides each spatial dimension by a factor of 2.
- Then we repeat the same structure twice: two convolutional layers followed by a max pooling layer. For larger images, we could repeat this structure several more times (the number of repetitions is a hyperparameter you can tune).
- Note that the number of filters grows as we climb up the CNN toward the output layer (it is initially 64, then 128, then 256): it makes sense for it to grow, since the number of low-level features is often fairly low (e.g., small circles, horizontal lines), but there are many different ways to combine them into higher-level features. It is a common practice to double the number of filters after each pooling layer: since a pooling layer divides each spatial dimension by a factor of 2, we can afford to double the number of feature maps in the next layer without fear of exploding the number of parameters, memory usage, or computational load.
- Next is the fully connected network, composed of two hidden dense layers and a dense output layer. Note that we must flatten its inputs, since a dense network expects a 1D array of features for each instance. We also add two dropout layers, with a dropout rate of 50% each, to reduce overfitting.

This CNN reaches over 92% accuracy on the test set. It's not state of the art, but it is pretty good, and clearly much better than what we achieved with dense networks in Chapter 10.

Over the years, variants of this fundamental architecture have been developed, leading to amazing advances in the field. A good measure of this progress is the error rate in competitions such as the ILSVRC ImageNet challenge. In this competition the top-five error rate for image classification fell from over 26% to less than 2.3% in just six years. The top-five error rate is the number of test images for which the system's top five predictions did not include the correct answer. The images are large (256 pixels high) and there are 1,000 classes, some of which are really subtle (try distinguishing 120 dog breeds). Looking at the evolution of the winning entries is a good way to understand how CNNs work.

We will first look at the classical LeNet-5 architecture (1998), then three of the winners of the ILSVRC challenge: AlexNet (2012), GoogLeNet (2014), and ResNet (2015).

LeNet-5

The LeNet-5 architecture[10] is perhaps the most widely known CNN architecture. As mentioned earlier, it was created by Yann LeCun in 1998 and has been widely used for handwritten digit recognition (MNIST). It is composed of the layers shown in Table 14-1.

Table 14-1. LeNet-5 architecture

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully connected	–	10	–	–	RBF
F6	Fully connected	–	84	–	–	tanh
C5	Convolution	120	1×1	5×5	1	tanh
S4	Avg pooling	16	5×5	2×2	2	tanh
C3	Convolution	16	10×10	5×5	1	tanh
S2	Avg pooling	6	14×14	2×2	2	tanh
C1	Convolution	6	28×28	5×5	1	tanh
In	Input	1	32×32	–	–	–

There are a few extra details to be noted:

- MNIST images are 28×28 pixels, but they are zero-padded to 32×32 pixels and normalized before being fed to the network. The rest of the network does not use any padding, which is why the size keeps shrinking as the image progresses through the network.
- The average pooling layers are slightly more complex than usual: each neuron computes the mean of its inputs, then multiplies the result by a learnable coefficient (one per map) and adds a learnable bias term (again, one per map), then finally applies the activation function.
- Most neurons in C3 maps are connected to neurons in only three or four S2 maps (instead of all six S2 maps). See table 1 (page 8) in the original paper[10] for details.
- The output layer is a bit special: instead of computing the matrix multiplication of the inputs and the weight vector, each neuron outputs the square of the Euclidian distance between its input vector and its weight vector. Each output measures how much the image belongs to a particular digit class. The cross-entropy

10 Yann LeCun et al., "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE* 86, no. 11 (1998): 2278–2324.

cost function is now preferred, as it penalizes bad predictions much more, producing larger gradients and converging faster.

Yann LeCun's website features great demos of LeNet-5 classifying digits.

AlexNet

The AlexNet CNN architecture[11] won the 2012 ImageNet ILSVRC challenge by a large margin: it achieved a top-five error rate of 17%, while the second best achieved only 26%! It was developed by Alex Krizhevsky (hence the name), Ilya Sutskever, and Geoffrey Hinton. It is similar to LeNet-5, only much larger and deeper, and it was the first to stack convolutional layers directly on top of one another, instead of stacking a pooling layer on top of each convolutional layer. Table 14-2 presents this architecture.

Table 14-2. AlexNet architecture

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully connected	–	1,000	–	–	–	Softmax
F10	Fully connected	–	4,096	–	–	–	ReLU
F9	Fully connected	–	4,096	–	–	–	ReLU
S8	Max pooling	256	6×6	3×3	2	`valid`	–
C7	Convolution	256	13×13	3×3	1	`same`	ReLU
C6	Convolution	384	13×13	3×3	1	`same`	ReLU
C5	Convolution	384	13×13	3×3	1	`same`	ReLU
S4	Max pooling	256	13×13	3×3	2	`valid`	–
C3	Convolution	256	27×27	5×5	1	`same`	ReLU
S2	Max pooling	96	27×27	3×3	2	`valid`	–
C1	Convolution	96	55×55	11×11	4	`valid`	ReLU
In	Input	3 (RGB)	227×227	–	–	–	–

To reduce overfitting, the authors used two regularization techniques. First, they applied dropout (introduced in Chapter 11) with a 50% dropout rate during training to the outputs of layers F9 and F10. Second, they performed *data augmentation* by randomly shifting the training images by various offsets, flipping them horizontally, and changing the lighting conditions.

11 Alex Krizhevsky et al., "ImageNet Classification with Deep Convolutional Neural Networks," _*Proceedings of the 25th International Conference on Neural Information Processing Systems* 1 (2012): 1097–1105.

Data Augmentation

Data augmentation artificially increases the size of the training set by generating many realistic variants of each training instance. This reduces overfitting, making this a regularization technique. The generated instances should be as realistic as possible: ideally, given an image from the augmented training set, a human should not be able to tell whether it was augmented or not. Simply adding white noise will not help; the modifications should be learnable (white noise is not).

For example, you can slightly shift, rotate, and resize every picture in the training set by various amounts and add the resulting pictures to the training set (see Figure 14-12). This forces the model to be more tolerant to variations in the position, orientation, and size of the objects in the pictures. For a model that's more tolerant of different lighting conditions, you can similarly generate many images with various contrasts. In general, you can also flip the pictures horizontally (except for text, and other asymmetrical objects). By combining these transformations, you can greatly increase the size of your training set.

Figure 14-12. Generating new training instances from existing ones

AlexNet also uses a competitive normalization step immediately after the ReLU step of layers C1 and C3, called *local response normalization* (LRN): the most strongly activated neurons inhibit other neurons located at the same position in neighboring feature maps (such competitive activation has been observed in biological neurons). This encourages different feature maps to specialize, pushing them apart and forcing

them to explore a wider range of features, ultimately improving generalization. Equation 14-2 shows how to apply LRN.

Equation 14-2. Local response normalization (LRN)

$$b_i = a_i\left(k + \alpha \sum_{j=j_{\text{low}}}^{j_{\text{high}}} a_j^2\right)^{-\beta} \quad \text{with} \begin{cases} j_{\text{high}} = \min\left(i + \frac{r}{2}, f_n - 1\right) \\ j_{\text{low}} = \max\left(0, i - \frac{r}{2}\right) \end{cases}$$

In this equation:

- b_i is the normalized output of the neuron located in feature map i, at some row u and column v (note that in this equation we consider only neurons located at this row and column, so u and v are not shown).
- a_i is the activation of that neuron after the ReLU step, but before normalization.
- k, α, β, and r are hyperparameters. k is called the *bias*, and r is called the *depth radius*.
- f_n is the number of feature maps.

For example, if $r = 2$ and a neuron has a strong activation, it will inhibit the activation of the neurons located in the feature maps immediately above and below its own.

In AlexNet, the hyperparameters are set as follows: $r = 2$, $\alpha = 0.00002$, $\beta = 0.75$, and $k = 1$. This step can be implemented using the `tf.nn.local_response_normalization()` function (which you can wrap in a `Lambda` layer if you want to use it in a Keras model).

A variant of AlexNet called *ZF Net*[12] was developed by Matthew Zeiler and Rob Fergus and won the 2013 ILSVRC challenge. It is essentially AlexNet with a few tweaked hyperparameters (number of feature maps, kernel size, stride, etc.).

GoogLeNet

The GoogLeNet architecture was developed by Christian Szegedy et al. from Google Research,[13] and it won the ILSVRC 2014 challenge by pushing the top-five error rate below 7%. This great performance came in large part from the fact that the network was much deeper than previous CNNs (as you'll see in Figure 14-14). This was made

12 Matthew D. Zeiler and Rob Fergus, "Visualizing and Understanding Convolutional Networks," *Proceedings of the European Conference on Computer Vision* (2014): 818-833.

13 Christian Szegedy et al., "Going Deeper with Convolutions," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015): 1–9.

possible by subnetworks called *inception modules*,[14] which allow GoogLeNet to use parameters much more efficiently than previous architectures: GoogLeNet actually has 10 times fewer parameters than AlexNet (roughly 6 million instead of 60 million).

Figure 14-13 shows the architecture of an inception module. The notation "3 × 3 + 1(S)" means that the layer uses a 3 × 3 kernel, stride 1, and `"same"` padding. The input signal is first copied and fed to four different layers. All convolutional layers use the ReLU activation function. Note that the second set of convolutional layers uses different kernel sizes (1 × 1, 3 × 3, and 5 × 5), allowing them to capture patterns at different scales. Also note that every single layer uses a stride of 1 and `"same"` padding (even the max pooling layer), so their outputs all have the same height and width as their inputs. This makes it possible to concatenate all the outputs along the depth dimension in the final *depth concatenation layer* (i.e., stack the feature maps from all four top convolutional layers). This concatenation layer can be implemented in TensorFlow using the `tf.concat()` operation, with `axis=3` (the axis is the depth).

Figure 14-13. Inception module

You may wonder why inception modules have convolutional layers with 1 × 1 kernels. Surely these layers cannot capture any features because they look at only one pixel at a time? In fact, the layers serve three purposes:

- Although they cannot capture spatial patterns, they can capture patterns along the depth dimension.
- They are configured to output fewer feature maps than their inputs, so they serve as *bottleneck layers*, meaning they reduce dimensionality. This cuts the computa-

14 In the 2010 movie *Inception*, the characters keep going deeper and deeper into multiple layers of dreams; hence the name of these modules.

tional cost and the number of parameters, speeding up training and improving generalization.

- Each pair of convolutional layers ([1 × 1, 3 × 3] and [1 × 1, 5 × 5]) acts like a single powerful convolutional layer, capable of capturing more complex patterns. Indeed, instead of sweeping a simple linear classifier across the image (as a single convolutional layer does), this pair of convolutional layers sweeps a two-layer neural network across the image.

In short, you can think of the whole inception module as a convolutional layer on steroids, able to output feature maps that capture complex patterns at various scales.

The number of convolutional kernels for each convolutional layer is a hyperparameter. Unfortunately, this means that you have six more hyperparameters to tweak for every inception layer you add.

Now let's look at the architecture of the GoogLeNet CNN (see Figure 14-14). The number of feature maps output by each convolutional layer and each pooling layer is shown before the kernel size. The architecture is so deep that it has to be represented in three columns, but GoogLeNet is actually one tall stack, including nine inception modules (the boxes with the spinning tops). The six numbers in the inception modules represent the number of feature maps output by each convolutional layer in the module (in the same order as in Figure 14-13). Note that all the convolutional layers use the ReLU activation function.

Figure 14-14. GoogLeNet architecture

Let's go through this network:

- The first two layers divide the image's height and width by 4 (so its area is divided by 16), to reduce the computational load. The first layer uses a large kernel size so that much of the information is preserved.
- Then the local response normalization layer ensures that the previous layers learn a wide variety of features (as discussed earlier).
- Two convolutional layers follow, where the first acts like a bottleneck layer. As explained earlier, you can think of this pair as a single smarter convolutional layer.
- Again, a local response normalization layer ensures that the previous layers capture a wide variety of patterns.
- Next, a max pooling layer reduces the image height and width by 2, again to speed up computations.
- Then comes the tall stack of nine inception modules, interleaved with a couple max pooling layers to reduce dimensionality and speed up the net.

- Next, the global average pooling layer outputs the mean of each feature map: this drops any remaining spatial information, which is fine because there was not much spatial information left at that point. Indeed, GoogLeNet input images are typically expected to be 224 × 224 pixels, so after 5 max pooling layers, each dividing the height and width by 2, the feature maps are down to 7 × 7. Moreover, it is a classification task, not localization, so it does not matter where the object is. Thanks to the dimensionality reduction brought by this layer, there is no need to have several fully connected layers at the top of the CNN (like in AlexNet), and this considerably reduces the number of parameters in the network and limits the risk of overfitting.
- The last layers are self-explanatory: dropout for regularization, then a fully connected layer with 1,000 units (since there are 1,000 classes) and a softmax activation function to output estimated class probabilities.

This diagram is slightly simplified: the original GoogLeNet architecture also included two auxiliary classifiers plugged on top of the third and sixth inception modules. They were both composed of one average pooling layer, one convolutional layer, two fully connected layers, and a softmax activation layer. During training, their loss (scaled down by 70%) was added to the overall loss. The goal was to fight the vanishing gradients problem and regularize the network. However, it was later shown that their effect was relatively minor.

Several variants of the GoogLeNet architecture were later proposed by Google researchers, including Inception-v3 and Inception-v4, using slightly different inception modules and reaching even better performance.

VGGNet

The runner-up in the ILSVRC 2014 challenge was VGGNet,[15] developed by Karen Simonyan and Andrew Zisserman from the Visual Geometry Group (VGG) research lab at Oxford University. It had a very simple and classical architecture, with 2 or 3 convolutional layers and a pooling layer, then again 2 or 3 convolutional layers and a pooling layer, and so on (reaching a total of just 16 or 19 convolutional layers, depending on the VGG variant), plus a final dense network with 2 hidden layers and the output layer. It used only 3 × 3 filters, but many filters.

15 Karen Simonyan and Andrew Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," arXiv preprint arXiv:1409.1556 (2014).

ResNet

Kaiming He et al. won the ILSVRC 2015 challenge using a *Residual Network* (or *ResNet*),[16] that delivered an astounding top-five error rate under 3.6%. The winning variant used an extremely deep CNN composed of 152 layers (other variants had 34, 50, and 101 layers). It confirmed the general trend: models are getting deeper and deeper, with fewer and fewer parameters. The key to being able to train such a deep network is to use *skip connections* (also called *shortcut connections*): the signal feeding into a layer is also added to the output of a layer located a bit higher up the stack. Let's see why this is useful.

When training a neural network, the goal is to make it model a target function $h(\mathbf{x})$. If you add the input $\mathbf{x}$ to the output of the network (i.e., you add a skip connection), then the network will be forced to model $f(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x}$ rather than $h(\mathbf{x})$. This is called *residual learning* (see Figure 14-15).

Figure 14-15. Residual learning

When you initialize a regular neural network, its weights are close to zero, so the network just outputs values close to zero. If you add a skip connection, the resulting network just outputs a copy of its inputs; in other words, it initially models the identity function. If the target function is fairly close to the identity function (which is often the case), this will speed up training considerably.

Moreover, if you add many skip connections, the network can start making progress even if several layers have not started learning yet (see Figure 14-16). Thanks to skip connections, the signal can easily make its way across the whole network. The deep residual network can be seen as a stack of *residual units* (RUs), where each residual unit is a small neural network with a skip connection.

16 Kaiming He et al., "Deep Residual Learning for Image Recognition," arXiv preprint arXiv:1512:03385 (2015).

Figure 14-16. Regular deep neural network (left) and deep residual network (right)

Now let's look at ResNet's architecture (see Figure 14-17). It is surprisingly simple. It starts and ends exactly like GoogLeNet (except without a dropout layer), and in between is just a very deep stack of simple residual units. Each residual unit is composed of two convolutional layers (and no pooling layer!), with Batch Normalization (BN) and ReLU activation, using 3 × 3 kernels and preserving spatial dimensions (stride 1, `"same"` padding).

Figure 14-17. ResNet architecture

Note that the number of feature maps is doubled every few residual units, at the same time as their height and width are halved (using a convolutional layer with stride 2). When this happens, the inputs cannot be added directly to the outputs of the residual unit because they don't have the same shape (for example, this problem affects the skip connection represented by the dashed arrow in Figure 14-17). To solve this problem, the inputs are passed through a 1 × 1 convolutional layer with stride 2 and the right number of output feature maps (see Figure 14-18).

Figure 14-18. Skip connection when changing feature map size and depth

ResNet-34 is the ResNet with 34 layers (only counting the convolutional layers and the fully connected layer)[17] containing 3 residual units that output 64 feature maps, 4 RUs with 128 maps, 6 RUs with 256 maps, and 3 RUs with 512 maps. We will implement this architecture later in this chapter.

ResNets deeper than that, such as ResNet-152, use slightly different residual units. Instead of two 3 × 3 convolutional layers with, say, 256 feature maps, they use three convolutional layers: first a 1 × 1 convolutional layer with just 64 feature maps (4 times less), which acts as a bottleneck layer (as discussed already), then a 3 × 3 layer with 64 feature maps, and finally another 1 × 1 convolutional layer with 256 feature maps (4 times 64) that restores the original depth. ResNet-152 contains 3 such RUs that output 256 maps, then 8 RUs with 512 maps, a whopping 36 RUs with 1,024 maps, and finally 3 RUs with 2,048 maps.

Google's Inception-v4[18] architecture merged the ideas of GoogLeNet and ResNet and achieved a top-five error rate of close to 3% on ImageNet classification.

Xception

Another variant of the GoogLeNet architecture is worth noting: Xception[19] (which stands for *Extreme Inception*) was proposed in 2016 by François Chollet (the author of Keras), and it significantly outperformed Inception-v3 on a huge vision task (350 million images and 17,000 classes). Just like Inception-v4, it merges the ideas of GoogLeNet and ResNet, but it replaces the inception modules with a special type of layer called a *depthwise separable convolution layer* (or *separable convolution layer* for short[20]). These layers had been used before in some CNN architectures, but they were not as central as in the Xception architecture. While a regular convolutional layer uses filters that try to simultaneously capture spatial patterns (e.g., an oval) and cross-channel patterns (e.g., mouth + nose + eyes = face), a separable convolutional layer makes the strong assumption that spatial patterns and cross-channel patterns can be modeled separately (see Figure 14-19). Thus, it is composed of two parts: the first part applies a single spatial filter for each input feature map, then the second part looks

17 It is a common practice when describing a neural network to count only layers with parameters.

18 Christian Szegedy et al., "Inception–v4, Inception-ResNet and the Impact of Residual Connections on Learning," arXiv preprint arXiv:1602.07261 (2016).

19 François Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," arXiv preprint arXiv: 1610.02357 (2016).

20 This name can sometimes be ambiguous, since spatially separable convolutions are often called "separable convolutions" as well.

exclusively for cross-channel patterns—it is just a regular convolutional layer with 1 × 1 filters.

Figure 14-19. Depthwise separable convolutional layer

Since separable convolutional layers only have one spatial filter per input channel, you should avoid using them after layers that have too few channels, such as the input layer (granted, that's what Figure 14-19 represents, but it is just for illustration purposes). For this reason, the Xception architecture starts with 2 regular convolutional layers, but then the rest of the architecture uses only separable convolutions (34 in all), plus a few max pooling layers and the usual final layers (a global average pooling layer and a dense output layer).

You might wonder why Xception is considered a variant of GoogLeNet, since it contains no inception module at all. Well, as we discussed earlier, an inception module contains convolutional layers with 1 × 1 filters: these look exclusively for cross-channel patterns. However, the convolutional layers that sit on top of them are regular convolutional layers that look both for spatial and cross-channel patterns. So you can think of an inception module as an intermediate between a regular convolutional layer (which considers spatial patterns and cross-channel patterns jointly) and a separable convolutional layer (which considers them separately). In practice, it seems that separable convolutional layers generally perform better.

Separable convolutional layers use fewer parameters, less memory, and fewer computations than regular convolutional layers, and in general they even perform better, so you should consider using them by default (except after layers with few channels).

The ILSVRC 2016 challenge was won by the CUImage team from the Chinese University of Hong Kong. They used an ensemble of many different techniques, including a sophisticated object-detection system called GBD-Net,[21] to achieve a top-five error rate below 3%. Although this result is unquestionably impressive, the complexity of the solution contrasted with the simplicity of ResNets. Moreover, one year later another fairly simple architecture performed even better, as we will see now.

SENet

The winning architecture in the ILSVRC 2017 challenge was the Squeeze-and-Excitation Network (SENet).[22] This architecture extends existing architectures such as inception networks and ResNets, and boosts their performance. This allowed SENet to win the competition with an astonishing 2.25% top-five error rate! The extended versions of inception networks and ResNets are called *SE-Inception* and *SE-ResNet*, respectively. The boost comes from the fact that a SENet adds a small neural network, called an *SE block*, to every unit in the original architecture (i.e., every inception module or every residual unit), as shown in Figure 14-20.

Figure 14-20. SE-Inception module (left) and SE-ResNet unit (right)

21 Xingyu Zeng et al., "Crafting GBD-Net for Object Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, no. 9 (2018): 2109–2123.

22 Jie Hu et al., "Squeeze-and-Excitation Networks," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018): 7132–7141.

An SE block analyzes the output of the unit it is attached to, focusing exclusively on the depth dimension (it does not look for any spatial pattern), and it learns which features are usually most active together. It then uses this information to recalibrate the feature maps, as shown in Figure 14-21. For example, an SE block may learn that mouths, noses, and eyes usually appear together in pictures: if you see a mouth and a nose, you should expect to see eyes as well. So if the block sees a strong activation in the mouth and nose feature maps, but only mild activation in the eye feature map, it will boost the eye feature map (more accurately, it will reduce irrelevant feature maps). If the eyes were somewhat confused with something else, this feature map recalibration will help resolve the ambiguity.

Figure 14-21. An SE block performs feature map recalibration

An SE block is composed of just three layers: a global average pooling layer, a hidden dense layer using the ReLU activation function, and a dense output layer using the sigmoid activation function (see Figure 14-22).

Figure 14-22. SE block architecture

As earlier, the global average pooling layer computes the mean activation for each feature map: for example, if its input contains 256 feature maps, it will output 256

numbers representing the overall level of response for each filter. The next layer is where the "squeeze" happens: this layer has significantly fewer than 256 neurons—typically 16 times fewer than the number of feature maps (e.g., 16 neurons)—so the 256 numbers get compressed into a small vector (e.g., 16 dimensions). This is a low-dimensional vector representation (i.e., an embedding) of the distribution of feature responses. This bottleneck step forces the SE block to learn a general representation of the feature combinations (we will see this principle in action again when we discuss autoencoders in Chapter 17). Finally, the output layer takes the embedding and outputs a recalibration vector containing one number per feature map (e.g., 256), each between 0 and 1. The feature maps are then multiplied by this recalibration vector, so irrelevant features (with a low recalibration score) get scaled down while relevant features (with a recalibration score close to 1) are left alone.

Implementing a ResNet-34 CNN Using Keras

Most CNN architectures described so far are fairly straightforward to implement (although generally you would load a pretrained network instead, as we will see). To illustrate the process, let's implement a ResNet-34 from scratch using Keras. First, let's create a `ResidualUnit` layer:

```
class ResidualUnit(keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = keras.activations.get(activation)
        self.main_layers = [
            keras.layers.Conv2D(filters, 3, strides=strides,
                                padding="same", use_bias=False),
            keras.layers.BatchNormalization(),
            self.activation,
            keras.layers.Conv2D(filters, 3, strides=1,
                                padding="same", use_bias=False),
            keras.layers.BatchNormalization()]
        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                keras.layers.Conv2D(filters, 1, strides=strides,
                                    padding="same", use_bias=False),
                keras.layers.BatchNormalization()]

    def call(self, inputs):
        Z = inputs
        for layer in self.main_layers:
            Z = layer(Z)
        skip_Z = inputs
        for layer in self.skip_layers:
            skip_Z = layer(skip_Z)
        return self.activation(Z + skip_Z)
```

As you can see, this code matches Figure 14-18 pretty closely. In the constructor, we create all the layers we will need: the main layers are the ones on the right side of the diagram, and the skip layers are the ones on the left (only needed if the stride is greater than 1). Then in the `call()` method, we make the inputs go through the main layers and the skip layers (if any), then we add both outputs and apply the activation function.

Next, we can build the ResNet-34 using a `Sequential` model, since it's really just a long sequence of layers (we can treat each residual unit as a single layer now that we have the `ResidualUnit` class):

```
model = keras.models.Sequential()
model.add(keras.layers.Conv2D(64, 7, strides=2, input_shape=[224, 224, 3],
                              padding="same", use_bias=False))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation("relu"))
model.add(keras.layers.MaxPool2D(pool_size=3, strides=2, padding="same"))
prev_filters = 64
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    strides = 1 if filters == prev_filters else 2
    model.add(ResidualUnit(filters, strides=strides))
    prev_filters = filters
model.add(keras.layers.GlobalAvgPool2D())
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(10, activation="softmax"))
```

The only slightly tricky part in this code is the loop that adds the `ResidualUnit` layers to the model: as explained earlier, the first 3 RUs have 64 filters, then the next 4 RUs have 128 filters, and so on. We then set the stride to 1 when the number of filters is the same as in the previous RU, or else we set it to 2. Then we add the `ResidualUnit`, and finally we update `prev_filters`.

It is amazing that in fewer than 40 lines of code, we can build the model that won the ILSVRC 2015 challenge! This demonstrates both the elegance of the ResNet model and the expressiveness of the Keras API. Implementing the other CNN architectures is not much harder. However, Keras comes with several of these architectures built in, so why not use them instead?

Using Pretrained Models from Keras

In general, you won't have to implement standard models like GoogLeNet or ResNet manually, since pretrained networks are readily available with a single line of code in the `keras.applications` package. For example, you can load the ResNet-50 model, pretrained on ImageNet, with the following line of code:

```
model = keras.applications.resnet50.ResNet50(weights="imagenet")
```

That's all! This will create a ResNet-50 model and download weights pretrained on the ImageNet dataset. To use it, you first need to ensure that the images have the right size. A ResNet-50 model expects 224 × 224-pixel images (other models may expect other sizes, such as 299 × 299), so let's use TensorFlow's `tf.image.resize()` function to resize the images we loaded earlier:

```
images_resized = tf.image.resize(images, [224, 224])
```

The `tf.image.resize()` will not preserve the aspect ratio. If this is a problem, try cropping the images to the appropriate aspect ratio before resizing. Both operations can be done in one shot with `tf.image.crop_and_resize()`.

The pretrained models assume that the images are preprocessed in a specific way. In some cases they may expect the inputs to be scaled from 0 to 1, or –1 to 1, and so on. Each model provides a `preprocess_input()` function that you can use to preprocess your images. These functions assume that the pixel values range from 0 to 255, so we must multiply them by 255 (since earlier we scaled them to the 0–1 range):

```
inputs = keras.applications.resnet50.preprocess_input(images_resized * 255)
```

Now we can use the pretrained model to make predictions:

```
Y_proba = model.predict(inputs)
```

As usual, the output `Y_proba` is a matrix with one row per image and one column per class (in this case, there are 1,000 classes). If you want to display the top *K* predictions, including the class name and the estimated probability of each predicted class, use the `decode_predictions()` function. For each image, it returns an array containing the top *K* predictions, where each prediction is represented as an array containing the class identifier,[23] its name, and the corresponding confidence score:

```
top_K = keras.applications.resnet50.decode_predictions(Y_proba, top=3)
for image_index in range(len(images)):
    print("Image #{}".format(image_index))
    for class_id, name, y_proba in top_K[image_index]:
        print("  {} - {:12s} {:.2f}%".format(class_id, name, y_proba * 100))
    print()
```

The output looks like this:

```
Image #0
  n03877845 - palace       42.87%
  n02825657 - bell_cote    40.57%
  n03781244 - monastery    14.56%
```

23 In the ImageNet dataset, each image is associated to a word in the WordNet dataset: the class ID is just a WordNet ID.

```
Image #1
  n04522168 - vase         46.83%
  n07930864 - cup          7.78%
  n11939491 - daisy        4.87%
```

The correct classes (monastery and daisy) appear in the top three results for both images. That's pretty good, considering that the model had to choose from among 1,000 classes.

As you can see, it is very easy to create a pretty good image classifier using a pretrained model. Other vision models are available in `keras.applications`, including several ResNet variants, GoogLeNet variants like Inception-v3 and Xception, VGGNet variants, and MobileNet and MobileNetV2 (lightweight models for use in mobile applications).

But what if you want to use an image classifier for classes of images that are not part of ImageNet? In that case, you may still benefit from the pretrained models to perform transfer learning.

Pretrained Models for Transfer Learning

If you want to build an image classifier but you do not have enough training data, then it is often a good idea to reuse the lower layers of a pretrained model, as we discussed in Chapter 11. For example, let's train a model to classify pictures of flowers, reusing a pretrained Xception model. First, let's load the dataset using TensorFlow Datasets (see Chapter 13):

```
import tensorflow_datasets as tfds

dataset, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)
dataset_size = info.splits["train"].num_examples # 3670
class_names = info.features["label"].names # ["dandelion", "daisy", ...]
n_classes = info.features["label"].num_classes # 5
```

Note that you can get information about the dataset by setting `with_info=True`. Here, we get the dataset size and the names of the classes. Unfortunately, there is only a `"train"` dataset, no test set or validation set, so we need to split the training set. The TF Datasets project provides an API for this. For example, let's take the first 10% of the dataset for testing, the next 15% for validation, and the remaining 75% for training:

```
test_split, valid_split, train_split = tfds.Split.TRAIN.subsplit([10, 15, 75])

test_set = tfds.load("tf_flowers", split=test_split, as_supervised=True)
valid_set = tfds.load("tf_flowers", split=valid_split, as_supervised=True)
train_set = tfds.load("tf_flowers", split=train_split, as_supervised=True)
```

Next we must preprocess the images. The CNN expects 224 × 224 images, so we need to resize them. We also need to run the images through Xception's `preprocess_input()` function:

```
def preprocess(image, label):
    resized_image = tf.image.resize(image, [224, 224])
    final_image = keras.applications.xception.preprocess_input(resized_image)
    return final_image, label
```

Let's apply this preprocessing function to all three datasets, shuffle the training set, and add batching and prefetching to all the datasets:

```
batch_size = 32
train_set = train_set.shuffle(1000)
train_set = train_set.map(preprocess).batch(batch_size).prefetch(1)
valid_set = valid_set.map(preprocess).batch(batch_size).prefetch(1)
test_set = test_set.map(preprocess).batch(batch_size).prefetch(1)
```

If you want to perform some data augmentation, change the preprocessing function for the training set, adding some random transformations to the training images. For example, use `tf.image.random_crop()` to randomly crop the images, use `tf.image.random_flip_left_right()` to randomly flip the images horizontally, and so on (see the "Pretrained Models for Transfer Learning" section of the notebook for an example).

The `keras.preprocessing.image.ImageDataGenerator` class makes it easy to load images from disk and augment them in various ways: you can shift each image, rotate it, rescale it, flip it horizontally or vertically, shear it, or apply any transformation function you want to it. This is very convenient for simple projects. However, building a tf.data pipeline has many advantages: it can read the images efficiently (e.g., in parallel) from any source, not just the local disk; you can manipulate the `Dataset` as you wish; and if you write a preprocessing function based on `tf.image` operations, this function can be used both in the tf.data pipeline and in the model you will deploy to production (see Chapter 19).

Next let's load an Xception model, pretrained on ImageNet. We exclude the top of the network by setting `include_top=False`: this excludes the global average pooling layer and the dense output layer. We then add our own global average pooling layer, based on the output of the base model, followed by a dense output layer with one unit per class, using the softmax activation function. Finally, we create the Keras `Model`:

```
base_model = keras.applications.xception.Xception(weights="imagenet",
                                                  include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
output = keras.layers.Dense(n_classes, activation="softmax")(avg)
model = keras.Model(inputs=base_model.input, outputs=output)
```

As explained in Chapter 11, it's usually a good idea to freeze the weights of the pre-trained layers, at least at the beginning of training:

```
for layer in base_model.layers:
    layer.trainable = False
```

Since our model uses the base model's layers directly, rather than the `base_model` object itself, setting `base_model.trainable=False` would have no effect.

Finally, we can compile the model and start training:

```
optimizer = keras.optimizers.SGD(lr=0.2, momentum=0.9, decay=0.01)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set, epochs=5, validation_data=valid_set)
```

This will be very slow, unless you have a GPU. If you do not, then you should run this chapter's notebook in Colab, using a GPU runtime (it's free!). See the instructions at *https://github.com/ageron/handson-ml2*.

After training the model for a few epochs, its validation accuracy should reach about 75–80% and stop making much progress. This means that the top layers are now pretty well trained, so we are ready to unfreeze all the layers (or you could try unfreezing just the top ones) and continue training (don't forget to compile the model when you freeze or unfreeze layers). This time we use a much lower learning rate to avoid damaging the pretrained weights:

```
for layer in base_model.layers:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.001)
model.compile(...)
history = model.fit(...)
```

It will take a while, but this model should reach around 95% accuracy on the test set. With that, you can start training amazing image classifiers! But there's more to computer vision than just classification. For example, what if you also want to know *where* the flower is in the picture? Let's look at this now.

Classification and Localization

Localizing an object in a picture can be expressed as a regression task, as discussed in Chapter 10: to predict a bounding box around the object, a common approach is to

predict the horizontal and vertical coordinates of the object's center, as well as its height and width. This means we have four numbers to predict. It does not require much change to the model; we just need to add a second dense output layer with four units (typically on top of the global average pooling layer), and it can be trained using the MSE loss:

```
base_model = keras.applications.xception.Xception(weights="imagenet",
                                                  include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
class_output = keras.layers.Dense(n_classes, activation="softmax")(avg)
loc_output = keras.layers.Dense(4)(avg)
model = keras.Model(inputs=base_model.input,
                    outputs=[class_output, loc_output])
model.compile(loss=["sparse_categorical_crossentropy", "mse"],
              loss_weights=[0.8, 0.2], # depends on what you care most about
              optimizer=optimizer, metrics=["accuracy"])
```

But now we have a problem: the flowers dataset does not have bounding boxes around the flowers. So, we need to add them ourselves. This is often one of the hardest and most costly parts of a Machine Learning project: getting the labels. It's a good idea to spend time looking for the right tools. To annotate images with bounding boxes, you may want to use an open source image labeling tool like VGG Image Annotator, LabelImg, OpenLabeler, or ImgLab, or perhaps a commercial tool like LabelBox or Supervisely. You may also want to consider crowdsourcing platforms such as Amazon Mechanical Turk if you have a very large number of images to annotate. However, it is quite a lot of work to set up a crowdsourcing platform, prepare the form to be sent to the workers, supervise them, and ensure that the quality of the bounding boxes they produce is good, so make sure it is worth the effort. If there are just a few thousand images to label, and you don't plan to do this frequently, it may be preferable to do it yourself. Adriana Kovashka et al. wrote a very practical paper[24] about crowdsourcing in computer vision. I recommend you check it out, even if you do not plan to use crowdsourcing.

Let's suppose you've obtained the bounding boxes for every image in the flowers dataset (for now we will assume there is a single bounding box per image). You then need to create a dataset whose items will be batches of preprocessed images along with their class labels and their bounding boxes. Each item should be a tuple of the form `(images, (class_labels, bounding_boxes))`. Then you are ready to train your model!

24 Adriana Kovashka et al., "Crowdsourcing in Computer Vision," *Foundations and Trends in Computer Graphics and Vision* 10, no. 3 (2014): 177–243.

The bounding boxes should be normalized so that the horizontal and vertical coordinates, as well as the height and width, all range from 0 to 1. Also, it is common to predict the square root of the height and width rather than the height and width directly: this way, a 10-pixel error for a large bounding box will not be penalized as much as a 10-pixel error for a small bounding box.

The MSE often works fairly well as a cost function to train the model, but it is not a great metric to evaluate how well the model can predict bounding boxes. The most common metric for this is the *Intersection over Union* (IoU): the area of overlap between the predicted bounding box and the target bounding box, divided by the area of their union (see Figure 14-23). In tf.keras, it is implemented by the `tf.keras.metrics.MeanIoU` class.

Figure 14-23. Intersection over Union (IoU) metric for bounding boxes

Classifying and localizing a single object is nice, but what if the images contain multiple objects (as is often the case in the flowers dataset)?

Object Detection

The task of classifying and localizing multiple objects in an image is called *object detection*. Until a few years ago, a common approach was to take a CNN that was trained to classify and locate a single object, then slide it across the image, as shown in Figure 14-24. In this example, the image was chopped into a 6 × 8 grid, and we show a CNN (the thick black rectangle) sliding across all 3 × 3 regions. When the CNN was looking at the top left of the image, it detected part of the leftmost rose, and then it detected that same rose again when it was first shifted one step to the right. At

the next step, it started detecting part of the topmost rose, and then it detected it again once it was shifted one more step to the right. You would then continue to slide the CNN through the whole image, looking at all 3 × 3 regions. Moreover, since objects can have varying sizes, you would also slide the CNN across regions of different sizes. For example, once you are done with the 3 × 3 regions, you might want to slide the CNN across all 4 × 4 regions as well.

Figure 14-24. Detecting multiple objects by sliding a CNN across the image

This technique is fairly straightforward, but as you can see it will detect the same object multiple times, at slightly different positions. Some post-processing will then be needed to get rid of all the unnecessary bounding boxes. A common approach for this is called *non-max suppression*. Here's how you do it:

1. First, you need to add an extra *objectness* output to your CNN, to estimate the probability that a flower is indeed present in the image (alternatively, you could add a "no-flower" class, but this usually does not work as well). It must use the sigmoid activation function, and you can train it using binary cross-entropy loss. Then get rid of all the bounding boxes for which the objectness score is below some threshold: this will drop all the bounding boxes that don't actually contain a flower.
2. Find the bounding box with the highest objectness score, and get rid of all the other bounding boxes that overlap a lot with it (e.g., with an IoU greater than 60%). For example, in Figure 14-24, the bounding box with the max objectness score is the thick bounding box over the topmost rose (the objectness score is represented by the thickness of the bounding boxes). The other bounding box

over that same rose overlaps a lot with the max bounding box, so we will get rid of it.

3. Repeat step two until there are no more bounding boxes to get rid of.

This simple approach to object detection works pretty well, but it requires running the CNN many times, so it is quite slow. Fortunately, there is a much faster way to slide a CNN across an image: using a *fully convolutional network* (FCN).

Fully Convolutional Networks

The idea of FCNs was first introduced in a 2015 paper[25] by Jonathan Long et al., for semantic segmentation (the task of classifying every pixel in an image according to the class of the object it belongs to). The authors pointed out that you could replace the dense layers at the top of a CNN by convolutional layers. To understand this, let's look at an example: suppose a dense layer with 200 neurons sits on top of a convolutional layer that outputs 100 feature maps, each of size 7×7 (this is the feature map size, not the kernel size). Each neuron will compute a weighted sum of all $100 \times 7 \times 7$ activations from the convolutional layer (plus a bias term). Now let's see what happens if we replace the dense layer with a convolutional layer using 200 filters, each of size 7×7, and with `"valid"` padding. This layer will output 200 feature maps, each 1×1 (since the kernel is exactly the size of the input feature maps and we are using `"valid"` padding). In other words, it will output 200 numbers, just like the dense layer did; and if you look closely at the computations performed by a convolutional layer, you will notice that these numbers will be precisely the same as those the dense layer produced. The only difference is that the dense layer's output was a tensor of shape [*batch size*, 200], while the convolutional layer will output a tensor of shape [*batch size*, 1, 1, 200].

To convert a dense layer to a convolutional layer, the number of filters in the convolutional layer must be equal to the number of units in the dense layer, the filter size must be equal to the size of the input feature maps, and you must use `"valid"` padding. The stride may be set to 1 or more, as we will see shortly.

Why is this important? Well, while a dense layer expects a specific input size (since it has one weight per input feature), a convolutional layer will happily process images of any size[26] (however, it does expect its inputs to have a specific number of channels,

25 Jonathan Long et al., "Fully Convolutional Networks for Semantic Segmentation," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015): 3431–3440.

26 There is one small exception: a convolutional layer using `"valid"` padding will complain if the input size is smaller than the kernel size.

since each kernel contains a different set of weights for each input channel). Since an FCN contains only convolutional layers (and pooling layers, which have the same property), it can be trained and executed on images of any size!

For example, suppose we'd already trained a CNN for flower classification and localization. It was trained on 224 × 224 images, and it outputs 10 numbers: outputs 0 to 4 are sent through the softmax activation function, and this gives the class probabilities (one per class); output 5 is sent through the logistic activation function, and this gives the objectness score; outputs 6 to 9 do not use any activation function, and they represent the bounding box's center coordinates, as well as its height and width. We can now convert its dense layers to convolutional layers. In fact, we don't even need to retrain it; we can just copy the weights from the dense layers to the convolutional layers! Alternatively, we could have converted the CNN into an FCN before training.

Now suppose the last convolutional layer before the output layer (also called the bottleneck layer) outputs 7 × 7 feature maps when the network is fed a 224 × 224 image (see the left side of Figure 14-25). If we feed the FCN a 448 × 448 image (see the right side of Figure 14-25), the bottleneck layer will now output 14 × 14 feature maps.[27] Since the dense output layer was replaced by a convolutional layer using 10 filters of size 7 × 7, with `"valid"` padding and stride 1, the output will be composed of 10 features maps, each of size 8 × 8 (since 14 – 7 + 1 = 8). In other words, the FCN will process the whole image only once, and it will output an 8 × 8 grid where each cell contains 10 numbers (5 class probabilities, 1 objectness score, and 4 bounding box coordinates). It's exactly like taking the original CNN and sliding it across the image using 8 steps per row and 8 steps per column. To visualize this, imagine chopping the original image into a 14 × 14 grid, then sliding a 7 × 7 window across this grid; there will be 8 × 8 = 64 possible locations for the window, hence 8 × 8 predictions. However, the FCN approach is *much* more efficient, since the network only looks at the image once. In fact, *You Only Look Once* (YOLO) is the name of a very popular object detection architecture, which we'll look at next.

27 This assumes we used only `"same"` padding in the network: indeed, `"valid"` padding would reduce the size of the feature maps. Moreover, 448 can be neatly divided by 2 several times until we reach 7, without any rounding error. If any layer uses a different stride than 1 or 2, then there may be some rounding error, so again the feature maps may end up being smaller.

Figure 14-25. The same fully convolutional network processing a small image (left) and a large one (right)

You Only Look Once (YOLO)

YOLO is an extremely fast and accurate object detection architecture proposed by Joseph Redmon et al. in a 2015 paper,[28] and subsequently improved in 2016[29] (YOLOv2) and in 2018[30] (YOLOv3). It is so fast that it can run in real time on a video, as seen in Redmon's demo.

YOLOv3's architecture is quite similar to the one we just discussed, but with a few important differences:

28 Joseph Redmon et al., "You Only Look Once: Unified, Real-Time Object Detection," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016): 779–788.

29 Joseph Redmon and Ali Farhadi, "YOLO9000: Better, Faster, Stronger," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2017): 6517–6525.

30 Joseph Redmon and Ali Farhadi, "YOLOv3: An Incremental Improvement," arXiv preprint arXiv:1804.02767 (2018).

- It outputs five bounding boxes for each grid cell (instead of just one), and each bounding box comes with an objectness score. It also outputs 20 class probabilities per grid cell, as it was trained on the PASCAL VOC dataset, which contains 20 classes. That's a total of 45 numbers per grid cell: 5 bounding boxes, each with 4 coordinates, plus 5 objectness scores, plus 20 class probabilities.
- Instead of predicting the absolute coordinates of the bounding box centers, YOLOv3 predicts an offset relative to the coordinates of the grid cell, where (0, 0) means the top left of that cell and (1, 1) means the bottom right. For each grid cell, YOLOv3 is trained to predict only bounding boxes whose center lies in that cell (but the bounding box itself generally extends well beyond the grid cell). YOLOv3 applies the logistic activation function to the bounding box coordinates to ensure they remain in the 0 to 1 range.
- Before training the neural net, YOLOv3 finds five representative bounding box dimensions, called *anchor boxes* (or *bounding box priors*). It does this by applying the K-Means algorithm (see Chapter 9) to the height and width of the training set bounding boxes. For example, if the training images contain many pedestrians, then one of the anchor boxes will likely have the dimensions of a typical pedestrian. Then when the neural net predicts five bounding boxes per grid cell, it actually predicts how much to rescale each of the anchor boxes. For example, suppose one anchor box is 100 pixels tall and 50 pixels wide, and the network predicts, say, a vertical rescaling factor of 1.5 and a horizontal rescaling of 0.9 (for one of the grid cells). This will result in a predicted bounding box of size 150 × 45 pixels. To be more precise, for each grid cell and each anchor box, the network predicts the log of the vertical and horizontal rescaling factors. Having these priors makes the network more likely to predict bounding boxes of the appropriate dimensions, and it also speeds up training because it will more quickly learn what reasonable bounding boxes look like.
- The network is trained using images of different scales: every few batches during training, the network randomly chooses a new image dimension (from 330 × 330 to 608 × 608 pixels). This allows the network to learn to detect objects at different scales. Moreover, it makes it possible to use YOLOv3 at different scales: the smaller scale will be less accurate but faster than the larger scale, so you can choose the right trade-off for your use case.

There are a few more innovations you might be interested in, such as the use of skip connections to recover some of the spatial resolution that is lost in the CNN (we will discuss this shortly, when we look at semantic segmentation). In the 2016 paper, the authors introduce the YOLO9000 model that uses hierarchical classification: the model predicts a probability for each node in a visual hierarchy called *WordTree*. This makes it possible for the network to predict with high confidence that an image represents, say, a dog, even though it is unsure what specific type of dog. I encourage you

to go ahead and read all three papers: they are quite pleasant to read, and they provide excellent examples of how Deep Learning systems can be incrementally improved.

Mean Average Precision (mAP)

A very common metric used in object detection tasks is the *mean Average Precision* (mAP). "Mean Average" sounds a bit redundant, doesn't it? To understand this metric, let's go back to two classification metrics we discussed in Chapter 3: precision and recall. Remember the trade-off: the higher the recall, the lower the precision. You can visualize this in a precision/recall curve (see Figure 3-5). To summarize this curve into a single number, we could compute its area under the curve (AUC). But note that the precision/recall curve may contain a few sections where precision actually goes up when recall increases, especially at low recall values (you can see this at the top left of Figure 3-5). This is one of the motivations for the mAP metric.

Suppose the classifier has 90% precision at 10% recall, but 96% precision at 20% recall. There's really no trade-off here: it simply makes more sense to use the classifier at 20% recall rather than at 10% recall, as you will get both higher recall and higher precision. So instead of looking at the precision *at* 10% recall, we should really be looking at the *maximum* precision that the classifier can offer with *at least* 10% recall. It would be 96%, not 90%. Therefore, one way to get a fair idea of the model's performance is to compute the maximum precision you can get with at least 0% recall, then 10% recall, 20%, and so on up to 100%, and then calculate the mean of these maximum precisions. This is called the *Average Precision* (AP) metric. Now when there are more than two classes, we can compute the AP for each class, and then compute the mean AP (mAP). That's it!

In an object detection system, there is an additional level of complexity: what if the system detected the correct class, but at the wrong location (i.e., the bounding box is completely off)? Surely we should not count this as a positive prediction. One approach is to define an IOU threshold: for example, we may consider that a prediction is correct only if the IOU is greater than, say, 0.5, and the predicted class is correct. The corresponding mAP is generally noted mAP@0.5 (or mAP@50%, or sometimes just AP_{50}). In some competitions (such as the PASCAL VOC challenge), this is what is done. In others (such as the COCO competition), the mAP is computed for different IOU thresholds (0.50, 0.55, 0.60, …, 0.95), and the final metric is the mean of all these mAPs (noted AP@[.50:.95] or AP@[.50:0.05:.95]). Yes, that's a mean mean average.

Several YOLO implementations built using TensorFlow are available on GitHub. In particular, check out Zihao Zang's TensorFlow 2 implementation. Other object detection models are available in the TensorFlow Models project, many with pretrained

weights; and some have even been ported to TF Hub, such as SSD[31] and Faster-RCNN,[32] which are both quite popular. SSD is also a "single shot" detection model, similar to YOLO. Faster R-CNN is more complex: the image first goes through a CNN, then the output is passed to a *Region Proposal Network* (RPN) that proposes bounding boxes that are most likely to contain an object, and a classifier is run for each bounding box, based on the cropped output of the CNN.

The choice of detection system depends on many factors: speed, accuracy, available pretrained models, training time, complexity, etc. The papers contain tables of metrics, but there is quite a lot of variability in the testing environments, and the technologies evolve so fast that it is difficult to make a fair comparison that will be useful for most people and remain valid for more than a few months.

So, we can locate objects by drawing bounding boxes around them. Great! But perhaps you want to be a bit more precise. Let's see how to go down to the pixel level.

Semantic Segmentation

In *semantic segmentation*, each pixel is classified according to the class of the object it belongs to (e.g., road, car, pedestrian, building, etc.), as shown in Figure 14-26. Note that different objects of the same class are *not* distinguished. For example, all the bicycles on the right side of the segmented image end up as one big lump of pixels. The main difficulty in this task is that when images go through a regular CNN, they gradually lose their spatial resolution (due to the layers with strides greater than 1); so, a regular CNN may end up knowing that there's a person somewhere in the bottom left of the image, but it will not be much more precise than that.

Just like for object detection, there are many different approaches to tackle this problem, some quite complex. However, a fairly simple solution was proposed in the 2015 paper by Jonathan Long et al. we discussed earlier. The authors start by taking a pretrained CNN and turning it into an FCN. The CNN applies an overall stride of 32 to the input image (i.e., if you add up all the strides greater than 1), meaning the last layer outputs feature maps that are 32 times smaller than the input image. This is clearly too coarse, so they add a single *upsampling layer* that multiplies the resolution by 32.

31 Wei Liu et al., "SSD: Single Shot Multibox Detector," *Proceedings of the 14th European Conference on Computer Vision* 1 (2016): 21–37.

32 Shaoqing Ren et al., "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *Proceedings of the 28th International Conference on Neural Information Processing Systems* 1 (2015): 91–99.

Figure 14-26. Semantic segmentation

There are several solutions available for upsampling (increasing the size of an image), such as bilinear interpolation, but that only works reasonably well up to ×4 or ×8. Instead, they use a *transposed convolutional layer*:[33] it is equivalent to first stretching the image by inserting empty rows and columns (full of zeros), then performing a regular convolution (see Figure 14-27). Alternatively, some people prefer to think of it as a regular convolutional layer that uses fractional strides (e.g., 1/2 in Figure 14-27). The transposed convolutional layer can be initialized to perform something close to linear interpolation, but since it is a trainable layer, it will learn to do better during training. In tf.keras, you can use the `Conv2DTranspose` layer.

Figure 14-27. Upsampling using a transposed convolutional layer

33 This type of layer is sometimes referred to as a *deconvolution layer*, but it does *not* perform what mathematicians call a deconvolution, so this name should be avoided.

In a transposed convolutional layer, the stride defines how much the input will be stretched, not the size of the filter steps, so the larger the stride, the larger the output (unlike for convolutional layers or pooling layers).

TensorFlow Convolution Operations

TensorFlow also offers a few other kinds of convolutional layers:

`keras.layers.Conv1D`
: Creates a convolutional layer for 1D inputs, such as time series or text (sequences of letters or words), as we will see in Chapter 15.

`keras.layers.Conv3D`
: Creates a convolutional layer for 3D inputs, such as 3D PET scans.

`dilation_rate`
: Setting the `dilation_rate` hyperparameter of any convolutional layer to a value of 2 or more creates an *à-trous convolutional layer* ("à trous" is French for "with holes"). This is equivalent to using a regular convolutional layer with a filter dilated by inserting rows and columns of zeros (i.e., holes). For example, a 1×3 filter equal to `[[1,2,3]]` may be dilated with a *dilation rate* of 4, resulting in a *dilated filter* of `[[1, 0, 0, 0, 2, 0, 0, 0, 3]]`. This lets the convolutional layer have a larger receptive field at no computational price and using no extra parameters.

`tf.nn.depthwise_conv2d()`
: Can be used to create a *depthwise convolutional layer* (but you need to create the variables yourself). It applies every filter to every individual input channel independently. Thus, if there are f_n filters and $f_{n'}$ input channels, then this will output $f_n \times f_{n'}$ feature maps.

This solution is OK, but still too imprecise. To do better, the authors added skip connections from lower layers: for example, they upsampled the output image by a factor of 2 (instead of 32), and they added the output of a lower layer that had this double resolution. Then they upsampled the result by a factor of 16, leading to a total upsampling factor of 32 (see Figure 14-28). This recovered some of the spatial resolution that was lost in earlier pooling layers. In their best architecture, they used a second similar skip connection to recover even finer details from an even lower layer. In short, the output of the original CNN goes through the following extra steps: upscale ×2, add the output of a lower layer (of the appropriate scale), upscale ×2, add the output of an even lower layer, and finally upscale ×8. It is even possible to scale up beyond the size of the original image: this can be used to increase the resolution of an image, which is a technique called *super-resolution*.

Figure 14-28. Skip layers recover some spatial resolution from lower layers

Once again, many GitHub repositories provide TensorFlow implementations of semantic segmentation (TensorFlow 1 for now), and you will even find pretrained *instance segmentation* models in the TensorFlow Models project. Instance segmentation is similar to semantic segmentation, but instead of merging all objects of the same class into one big lump, each object is distinguished from the others (e.g., it identifies each individual bicycle). At present, the instance segmentation models available in the TensorFlow Models project are based on the *Mask R-CNN* architecture, which was proposed in a 2017 paper:[34] it extends the Faster R-CNN model by additionally producing a pixel mask for each bounding box. So not only do you get a bounding box around each object, with a set of estimated class probabilities, but you also get a pixel mask that locates pixels in the bounding box that belong to the object.

As you can see, the field of Deep Computer Vision is vast and moving fast, with all sorts of architectures popping out every year, all based on convolutional neural networks. The progress made in just a few years has been astounding, and researchers are now focusing on harder and harder problems, such as *adversarial learning* (which attempts to make the network more resistant to images designed to fool it), explainability (understanding why the network makes a specific classification), realistic *image generation* (which we will come back to in Chapter 17), and *single-shot learning* (a system that can recognize an object after it has seen it just once). Some even explore completely novel architectures, such as Geoffrey Hinton's *capsule networks*[35] (I presented them in a couple of videos, with the corresponding code in a notebook). Now on to the next chapter, where we will look at how to process sequential data such as time series using recurrent neural networks and convolutional neural networks.

34 Kaiming He et al., "Mask R-CNN," arXiv preprint arXiv:1703.06870 (2017).

35 Geoffrey Hinton et al., "Matrix Capsules with EM Routing," *Proceedings of the International Conference on Learning Representations* (2018).

Exercises

1. What are the advantages of a CNN over a fully connected DNN for image classification?
2. Consider a CNN composed of three convolutional layers, each with 3×3 kernels, a stride of 2, and `"same"` padding. The lowest layer outputs 100 feature maps, the middle one outputs 200, and the top one outputs 400. The input images are RGB images of 200×300 pixels.

   What is the total number of parameters in the CNN? If we are using 32-bit floats, at least how much RAM will this network require when making a prediction for a single instance? What about when training on a mini-batch of 50 images?
3. If your GPU runs out of memory while training a CNN, what are five things you could try to solve the problem?
4. Why would you want to add a max pooling layer rather than a convolutional layer with the same stride?
5. When would you want to add a local response normalization layer?
6. Can you name the main innovations in AlexNet, compared to LeNet-5? What about the main innovations in GoogLeNet, ResNet, SENet, and Xception?
7. What is a fully convolutional network? How can you convert a dense layer into a convolutional layer?
8. What is the main technical difficulty of semantic segmentation?
9. Build your own CNN from scratch and try to achieve the highest possible accuracy on MNIST.
10. Use transfer learning for large image classification, going through these steps:
    a. Create a training set containing at least 100 images per class. For example, you could classify your own pictures based on the location (beach, mountain, city, etc.), or alternatively you can use an existing dataset (e.g., from TensorFlow Datasets).
    b. Split it into a training set, a validation set, and a test set.
    c. Build the input pipeline, including the appropriate preprocessing operations, and optionally add data augmentation.
    d. Fine-tune a pretrained model on this dataset.
11. Go through TensorFlow's Style Transfer tutorial. It is a fun way to generate art using Deep Learning.

Solutions to these exercises are available in Appendix A.

CHAPTER 15

Processing Sequences Using RNNs and CNNs

The batter hits the ball. The outfielder immediately starts running, anticipating the ball's trajectory. He tracks it, adapts his movements, and finally catches it (under a thunder of applause). Predicting the future is something you do all the time, whether you are finishing a friend's sentence or anticipating the smell of coffee at breakfast. In this chapter we will discuss recurrent neural networks (RNNs), a class of nets that can predict the future (well, up to a point, of course). They can analyze time series data such as stock prices, and tell you when to buy or sell. In autonomous driving systems, they can anticipate car trajectories and help avoid accidents. More generally, they can work on sequences of arbitrary lengths, rather than on fixed-sized inputs like all the nets we have considered so far. For example, they can take sentences, documents, or audio samples as input, making them extremely useful for natural language processing applications such as automatic translation or speech-to-text.

In this chapter we will first look at the fundamental concepts underlying RNNs and how to train them using backpropagation through time, then we will use them to forecast a time series. After that we'll explore the two main difficulties that RNNs face:

- Unstable gradients (discussed in Chapter 11), which can be alleviated using various techniques, including recurrent dropout and recurrent layer normalization
- A (very) limited short-term memory, which can be extended using LSTM and GRU cells

RNNs are not the only types of neural networks capable of handling sequential data: for small sequences, a regular dense network can do the trick; and for very long sequences, such as audio samples or text, convolutional neural networks can actually

work quite well too. We will discuss both of these possibilities, and we will finish this chapter by implementing a *WaveNet*: this is a CNN architecture capable of handling sequences of tens of thousands of time steps. In Chapter 16, we will continue to explore RNNs and see how to use them for natural language processing, along with more recent architectures based on attention mechanisms. Let's get started!

Recurrent Neurons and Layers

Up to now we have focused on feedforward neural networks, where the activations flow only in one direction, from the input layer to the output layer (a few exceptions are discussed in Appendix E). A recurrent neural network looks very much like a feedforward neural network, except it also has connections pointing backward. Let's look at the simplest possible RNN, composed of one neuron receiving inputs, producing an output, and sending that output back to itself, as shown in Figure 15-1 (left). At each *time step t* (also called a *frame*), this *recurrent neuron* receives the inputs $\mathbf{x}_{(t)}$ as well as its own output from the previous time step, $y_{(t-1)}$. Since there is no previous output at the first time step, it is generally set to 0. We can represent this tiny network against the time axis, as shown in Figure 15-1 (right). This is called *unrolling the network through time* (it's the same recurrent neuron represented once per time step).

Figure 15-1. A recurrent neuron (left) unrolled through time (right)

You can easily create a layer of recurrent neurons. At each time step *t*, every neuron receives both the input vector $\mathbf{x}_{(t)}$ and the output vector from the previous time step $\mathbf{y}_{(t-1)}$, as shown in Figure 15-2. Note that both the inputs and outputs are vectors now (when there was just a single neuron, the output was a scalar).

Figure 15-2. A layer of recurrent neurons (left) unrolled through time (right)

Each recurrent neuron has two sets of weights: one for the inputs $\mathbf{x}_{(t)}$ and the other for the outputs of the previous time step, $\mathbf{y}_{(t-1)}$. Let's call these weight vectors $\mathbf{w}_x$ and $\mathbf{w}_y$. If we consider the whole recurrent layer instead of just one recurrent neuron, we can place all the weight vectors in two weight matrices, $\mathbf{W}_x$ and $\mathbf{W}_y$. The output vector of the whole recurrent layer can then be computed pretty much as you might expect, as shown in Equation 15-1 ($\mathbf{b}$ is the bias vector and $\phi(\cdot)$ is the activation function (e.g., ReLU[1]).

Equation 15-1. Output of a recurrent layer for a single instance

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{W}_x^{\top}\mathbf{x}_{(t)} + \mathbf{W}_y^{\top}\mathbf{y}_{(t-1)} + \mathbf{b}\right)$$

Just as with feedforward neural networks, we can compute a recurrent layer's output in one shot for a whole mini-batch by placing all the inputs at time step t in an input matrix $\mathbf{X}_{(t)}$ (see Equation 15-2).

Equation 15-2. Outputs of a layer of recurrent neurons for all instances in a mini-batch

$$\begin{aligned}\mathbf{Y}_{(t)} &= \phi\left(\mathbf{X}_{(t)}\mathbf{W}_x + \mathbf{Y}_{(t-1)}\mathbf{W}_y + \mathbf{b}\right)\\ &= \phi\left(\left[\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}\right]\mathbf{W} + \mathbf{b}\right) \text{ with } \mathbf{W} = \begin{bmatrix}\mathbf{W}_x\\ \mathbf{W}_y\end{bmatrix}\end{aligned}$$

1 Note that many researchers prefer to use the hyperbolic tangent (tanh) activation function in RNNs rather than the ReLU activation function. For example, take a look at Vu Pham et al.'s 2013 paper "Dropout Improves Recurrent Neural Networks for Handwriting Recognition". ReLU-based RNNs are also possible, as shown in Quoc V. Le et al.'s 2015 paper "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units".

In this equation:

- $\mathbf{Y}_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch (m is the number of instances in the mini-batch and n_{neurons} is the number of neurons).
- $\mathbf{X}_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances (n_{inputs} is the number of input features).
- $\mathbf{W}_x$ is an $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix containing the connection weights for the inputs of the current time step.
- $\mathbf{W}_y$ is an $n_{\text{neurons}} \times n_{\text{neurons}}$ matrix containing the connection weights for the outputs of the previous time step.
- $\mathbf{b}$ is a vector of size n_{neurons} containing each neuron's bias term.
- The weight matrices $\mathbf{W}_x$ and $\mathbf{W}_y$ are often concatenated vertically into a single weight matrix $\mathbf{W}$ of shape $(n_{\text{inputs}} + n_{\text{neurons}}) \times n_{\text{neurons}}$ (see the second line of Equation 15-2).
- The notation $[\mathbf{X}_{(t)}\ \mathbf{Y}_{(t-1)}]$ represents the horizontal concatenation of the matrices $\mathbf{X}_{(t)}$ and $\mathbf{Y}_{(t-1)}$.

Notice that $\mathbf{Y}_{(t)}$ is a function of $\mathbf{X}_{(t)}$ and $\mathbf{Y}_{(t-1)}$, which is a function of $\mathbf{X}_{(t-1)}$ and $\mathbf{Y}_{(t-2)}$, which is a function of $\mathbf{X}_{(t-2)}$ and $\mathbf{Y}_{(t-3)}$, and so on. This makes $\mathbf{Y}_{(t)}$ a function of all the inputs since time $t = 0$ (that is, $\mathbf{X}_{(0)}, \mathbf{X}_{(1)}, \ldots, \mathbf{X}_{(t)}$). At the first time step, $t = 0$, there are no previous outputs, so they are typically assumed to be all zeros.

Memory Cells

Since the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, you could say it has a form of *memory*. A part of a neural network that preserves some state across time steps is called a *memory cell* (or simply a *cell*). A single recurrent neuron, or a layer of recurrent neurons, is a very basic cell, capable of learning only short patterns (typically about 10 steps long, but this varies depending on the task). Later in this chapter, we will look at some more complex and powerful types of cells capable of learning longer patterns (roughly 10 times longer, but again, this depends on the task).

In general a cell's state at time step t, denoted $\mathbf{h}_{(t)}$ (the "h" stands for "hidden"), is a function of some inputs at that time step and its state at the previous time step: $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$. Its output at time step t, denoted $\mathbf{y}_{(t)}$, is also a function of the previous state and the current inputs. In the case of the basic cells we have discussed so far, the output is simply equal to the state, but in more complex cells this is not always the case, as shown in Figure 15-3.

Figure 15-3. A cell's hidden state and its output may be different

Input and Output Sequences

An RNN can simultaneously take a sequence of inputs and produce a sequence of outputs (see the top-left network in Figure 15-4). This type of *sequence-to-sequence network* is useful for predicting time series such as stock prices: you feed it the prices over the last *N* days, and it must output the prices shifted by one day into the future (i.e., from *N* – 1 days ago to tomorrow).

Alternatively, you could feed the network a sequence of inputs and ignore all outputs except for the last one (see the top-right network in Figure 15-4). In other words, this is a *sequence-to-vector network*. For example, you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score (e.g., from –1 [hate] to +1 [love]).

Conversely, you could feed the network the same input vector over and over again at each time step and let it output a sequence (see the bottom-left network of Figure 15-4). This is a *vector-to-sequence network*. For example, the input could be an image (or the output of a CNN), and the output could be a caption for that image.

Lastly, you could have a sequence-to-vector network, called an *encoder*, followed by a vector-to-sequence network, called a *decoder* (see the bottom-right network of Figure 15-4). For example, this could be used for translating a sentence from one language to another. You would feed the network a sentence in one language, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in another language. This two-step model, called an *Encoder–Decoder*, works much better than trying to translate on the fly with a single sequence-to-sequence RNN (like the one represented at the top left): the last words of a sentence can affect the first words of the translation, so you need to wait until you have seen the whole sentence before translating it. We will see how to implement an Encoder–Decoder in Chapter 16 (as we will see, it is a bit more complex than in Figure 15-4 suggests).

Figure 15-4. Seq-to-seq (top left), seq-to-vector (top right), vector-to-seq (bottom left), and Encoder–Decoder (bottom right) networks

Sounds promising, but how do you train a recurrent neural network?

Training RNNs

To train an RNN, the trick is to unroll it through time (like we just did) and then simply use regular backpropagation (see Figure 15-5). This strategy is called *backpropagation through time* (BPTT).

Just like in regular backpropagation, there is a first forward pass through the unrolled network (represented by the dashed arrows). Then the output sequence is evaluated using a cost function $C(\mathbf{Y}_{(0)}, \mathbf{Y}_{(1)}, \ldots \mathbf{Y}_{(T)})$ (where T is the max time step). Note that this cost function may ignore some outputs, as shown in Figure 15-5 (for example, in a sequence-to-vector RNN, all outputs are ignored except for the very last one). The gradients of that cost function are then propagated backward through the unrolled network (represented by the solid arrows). Finally the model parameters are updated using the gradients computed during BPTT. Note that the gradients flow backward through all the outputs used by the cost function, not just through the final output (for example, in Figure 15-5 the cost function is computed using the last three outputs of the network, $\mathbf{Y}_{(2)}$, $\mathbf{Y}_{(3)}$, and $\mathbf{Y}_{(4)}$, so gradients flow through these three outputs,

but not through $\mathbf{Y}_{(0)}$ and $\mathbf{Y}_{(1)}$). Moreover, since the same parameters **W** and **b** are used at each time step, backpropagation will do the right thing and sum over all time steps.

Figure 15-5. Backpropagation through time

Fortunately, tf.keras takes care of all of this complexity for you—so let's start coding!

Forecasting a Time Series

Suppose you are studying the number of active users per hour on your website, or the daily temperature in your city, or your company's financial health, measured quarterly using multiple metrics. In all these cases, the data will be a sequence of one or more values per time step. This is called a *time series*. In the first two examples there is a single value per time step, so these are *univariate time series*, while in the financial example there are multiple values per time step (e.g., the company's revenue, debt, and so on), so it is a *multivariate time series*. A typical task is to predict future values, which is called *forecasting*. Another common task is to fill in the blanks: to predict (or rather "postdict") missing values from the past. This is called *imputation*. For example, Figure 15-6 shows 3 univariate time series, each of them 50 time steps long, and the goal here is to forecast the value at the next time step (represented by the X) for each of them.

Figure 15-6. Time series forecasting

For simplicity, we are using a time series generated by the `generate_time_series()` function, shown here:

```
def generate_time_series(batch_size, n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)
    time = np.linspace(0, 1, n_steps)
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10))  #   wave 1
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20)) # + wave 2
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5)   # + noise
    return series[..., np.newaxis].astype(np.float32)
```

This function creates as many time series as requested (via the `batch_size` argument), each of length `n_steps`, and there is just one value per time step in each series (i.e., all series are univariate). The function returns a NumPy array of shape [*batch size*, *time steps*, 1], where each series is the sum of two sine waves of fixed amplitudes but random frequencies and phases, plus a bit of noise.

When dealing with time series (and other types of sequences such as sentences), the input features are generally represented as 3D arrays of shape [*batch size*, *time steps*, *dimensionality*], where *dimensionality* is 1 for univariate time series and more for multivariate time series.

Now let's create a training set, a validation set, and a test set using this function:

```
n_steps = 50
series = generate_time_series(10000, n_steps + 1)
X_train, y_train = series[:7000, :n_steps], series[:7000, -1]
X_valid, y_valid = series[7000:9000, :n_steps], series[7000:9000, -1]
X_test, y_test = series[9000:, :n_steps], series[9000:, -1]
```

`X_train` contains 7,000 time series (i.e., its shape is [7000, 50, 1]), while `X_valid` contains 2,000 (from the 7,000th time series to the 8,999th) and `X_test` contains 1,000 (from the 9,000th to the 9,999th). Since we want to forecast a single value for each series, the targets are column vectors (e.g., `y_train` has a shape of [7000, 1]).

Baseline Metrics

Before we start using RNNs, it is often a good idea to have a few baseline metrics, or else we may end up thinking our model works great when in fact it is doing worse than basic models. For example, the simplest approach is to predict the last value in each series. This is called *naive forecasting*, and it is sometimes surprisingly difficult to outperform. In this case, it gives us a mean squared error of about 0.020:

```
>>> y_pred = X_valid[:, -1]
>>> np.mean(keras.losses.mean_squared_error(y_valid, y_pred))
0.020211367
```

Another simple approach is to use a fully connected network. Since it expects a flat list of features for each input, we need to add a `Flatten` layer. Let's just use a simple Linear Regression model so that each prediction will be a linear combination of the values in the time series:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])
```

If we compile this model using the MSE loss and the default Adam optimizer, then fit it on the training set for 20 epochs and evaluate it on the validation set, we get an MSE of about 0.004. That's much better than the naive approach!

Implementing a Simple RNN

Let's see if we can beat that with a simple RNN:

```
model = keras.models.Sequential([
  keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

That's really the simplest RNN you can build. It just contains a single layer, with a single neuron, as we saw in Figure 15-1. We do not need to specify the length of the input sequences (unlike in the previous model), since a recurrent neural network can process any number of time steps (this is why we set the first input dimension to `None`). By default, the `SimpleRNN` layer uses the hyperbolic tangent activation function. It works exactly as we saw earlier: the initial state $h_{(\text{init})}$ is set to 0, and it is passed to a single recurrent neuron, along with the value of the first time step, $x_{(0)}$. The neuron computes a weighted sum of these values and applies the hyperbolic tangent activation function to the result, and this gives the first output, y_0. In a simple RNN, this output is also the new state h_0. This new state is passed to the same recurrent neuron along with the next input value, $x_{(1)}$, and the process is repeated until the last time step. Then the layer just outputs the last value, y_{49}. All of this is performed simultaneously for every time series.

By default, recurrent layers in Keras only return the final output. To make them return one output per time step, you must set `return_sequences=True`, as we will see.

If you compile, fit, and evaluate this model (just like earlier, we train for 20 epochs using Adam), you will find that its MSE reaches only 0.014, so it is better than the naive approach but it does not beat a simple linear model. Note that for each neuron, a linear model has one parameter per input and per time step, plus a bias term (in the simple linear model we used, that's a total of 51 parameters). In contrast, for each recurrent neuron in a simple RNN, there is just one parameter per input and per hidden state dimension (in a simple RNN, that's just the number of recurrent neurons in the layer), plus a bias term. In this simple RNN, that's a total of just three parameters.

Trend and Seasonality

There are many other models to forecast time series, such as *weighted moving average* models or *autoregressive integrated moving average* (ARIMA) models. Some of them require you to first remove the trend and seasonality. For example, if you are studying the number of active users on your website, and it is growing by 10% every month, you would have to remove this trend from the time series. Once the model is trained and starts making predictions, you would have to add the trend back to get the final predictions. Similarly, if you are trying to predict the amount of sunscreen lotion sold every month, you will probably observe strong seasonality: since it sells well every summer, a similar pattern will be repeated every year. You would have to remove this seasonality from the time series, for example by computing the difference between the value at each time step and the value one year earlier (this technique is called *differencing*). Again, after the model is trained and makes predictions, you would have to add the seasonal pattern back to get the final predictions.

When using RNNs, it is generally not necessary to do all this, but it may improve performance in some cases, since the model will not have to learn the trend or the seasonality.

Apparently our simple RNN was too simple to get good performance. So let's try to add more recurrent layers!

Deep RNNs

It is quite common to stack multiple layers of cells, as shown in Figure 15-7. This gives you a *deep RNN*.

Figure 15-7. Deep RNN (left) unrolled through time (right)

Implementing a deep RNN with tf.keras is quite simple: just stack recurrent layers. In this example, we use three `SimpleRNN` layers (but we could add any other type of recurrent layer, such as an `LSTM` layer or a `GRU` layer, which we will discuss shortly):

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])
```

Make sure to set `return_sequences=True` for all recurrent layers (except the last one, if you only care about the last output). If you don't, they will output a 2D array (containing only the output of the last time step) instead of a 3D array (containing outputs for all time steps), and the next recurrent layer will complain that you are not feeding it sequences in the expected 3D format.

If you compile, fit, and evaluate this model, you will find that it reaches an MSE of 0.003. We finally managed to beat the linear model!

Note that the last layer is not ideal: it must have a single unit because we want to forecast a univariate time series, and this means we must have a single output value per time step. However, having a single unit means that the hidden state is just a single number. That's really not much, and it's probably not that useful; presumably, the RNN will mostly use the hidden states of the other recurrent layers to carry over all the information it needs from time step to time step, and it will not use the final layer's hidden state very much. Moreover, since a `SimpleRNN` layer uses the tanh activation function by default, the predicted values must lie within the range –1 to 1. But what if you want to use another activation function? For both these reasons, it might be preferable to replace the output layer with a `Dense` layer: it would run slightly

faster, the accuracy would be roughly the same, and it would allow us to choose any output activation function we want. If you make this change, also make sure to remove `return_sequences=True` from the second (now last) recurrent layer:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)
])
```

If you train this model, you will see that it converges faster and performs just as well. Plus, you could change the output activation function if you wanted.

Forecasting Several Time Steps Ahead

So far we have only predicted the value at the next time step, but we could just as easily have predicted the value several steps ahead by changing the targets appropriately (e.g., to predict 10 steps ahead, just change the targets to be the value 10 steps ahead instead of 1 step ahead). But what if we want to predict the next 10 values?

The first option is to use the model we already trained, make it predict the next value, then add that value to the inputs (acting as if this predicted value had actually occurred), and use the model again to predict the following value, and so on, as in the following code:

```
series = generate_time_series(1, n_steps + 10)
X_new, Y_new = series[:, :n_steps], series[:, n_steps:]
X = X_new
for step_ahead in range(10):
    y_pred_one = model.predict(X[:, step_ahead:])[:, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)

Y_pred = X[:, n_steps:]
```

As you might expect, the prediction for the next step will usually be more accurate than the predictions for later time steps, since the errors might accumulate (as you can see in Figure 15-8). If you evaluate this approach on the validation set, you will find an MSE of about 0.029. This is much higher than the previous models, but it's also a much harder task, so the comparison doesn't mean much. It's much more meaningful to compare this performance with naive predictions (just forecasting that the time series will remain constant for 10 time steps) or with a simple linear model. The naive approach is terrible (it gives an MSE of about 0.223), but the linear model gives an MSE of about 0.0188: it's much better than using our RNN to forecast the future one step at a time, and also much faster to train and run. Still, if you only want to forecast a few time steps ahead, on more complex tasks, this approach may work well.

Figure 15-8. Forecasting 10 steps ahead, 1 step at a time

The second option is to train an RNN to predict all 10 next values at once. We can still use a sequence-to-vector model, but it will output 10 values instead of 1. However, we first need to change the targets to be vectors containing the next 10 values:

```
series = generate_time_series(10000, n_steps + 10)
X_train, Y_train = series[:7000, :n_steps], series[:7000, -10:, 0]
X_valid, Y_valid = series[7000:9000, :n_steps], series[7000:9000, -10:, 0]
X_test, Y_test = series[9000:, :n_steps], series[9000:, -10:, 0]
```

Now we just need the output layer to have 10 units instead of 1:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)
])
```

After training this model, you can predict the next 10 values at once very easily:

```
Y_pred = model.predict(X_new)
```

This model works nicely: the MSE for the next 10 time steps is about 0.008. That's much better than the linear model. But we can still do better: indeed, instead of training the model to forecast the next 10 values only at the very last time step, we can train it to forecast the next 10 values at each and every time step. In other words, we can turn this sequence-to-vector RNN into a sequence-to-sequence RNN. The advantage of this technique is that the loss will contain a term for the output of the RNN at each and every time step, not just the output at the last time step. This means there will be many more error gradients flowing through the model, and they won't have to flow only through time; they will also flow from the output of each time step. This will both stabilize and speed up training.

To be clear, at time step 0 the model will output a vector containing the forecasts for time steps 1 to 10, then at time step 1 the model will forecast time steps 2 to 11, and so on. So each target must be a sequence of the same length as the input sequence, containing a 10-dimensional vector at each step. Let's prepare these target sequences:

```
Y = np.empty((10000, n_steps, 10)) # each target is a sequence of 10D vectors
for step_ahead in range(1, 10 + 1):
    Y[:, :, step_ahead - 1] = series[:, step_ahead:step_ahead + n_steps, 0]
Y_train = Y[:7000]
Y_valid = Y[7000:9000]
Y_test = Y[9000:]
```

It may be surprising that the targets will contain values that appear in the inputs (there is a lot of overlap between `X_train` and `Y_train`). Isn't that cheating? Fortunately, not at all: at each time step, the model only knows about past time steps, so it cannot look ahead. It is said to be a *causal* model.

To turn the model into a sequence-to-sequence model, we must set `return_sequences=True` in all recurrent layers (even the last one), and we must apply the output `Dense` layer at every time step. Keras offers a `TimeDistributed` layer for this very purpose: it wraps any layer (e.g., a `Dense` layer) and applies it at every time step of its input sequence. It does this efficiently, by reshaping the inputs so that each time step is treated as a separate instance (i.e., it reshapes the inputs from [*batch size*, *time steps*, *input dimensions*] to [*batch size* × *time steps*, *input dimensions*]; in this example, the number of input dimensions is 20 because the previous `SimpleRNN` layer has 20 units), then it runs the `Dense` layer, and finally it reshapes the outputs back to sequences (i.e., it reshapes the outputs from [*batch size* × *time steps*, *output dimensions*] to [*batch size*, *time steps*, *output dimensions*]; in this example the number of output dimensions is 10, since the `Dense` layer has 10 units).[2] Here is the updated model:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

The `Dense` layer actually supports sequences as inputs (and even higher-dimensional inputs): it handles them just like `TimeDistributed(Dense(…))`, meaning it is applied to the last input dimension only (independently across all time steps). Thus, we could replace the last layer with just `Dense(10)`. For the sake of clarity, however, we will keep using `TimeDistributed(Dense(10))` because it makes it clear that the `Dense`

2 Note that a `TimeDistributed(Dense(n))` layer is equivalent to a `Conv1D(n, filter_size=1)` layer.

layer is applied independently at each time step and that the model will output a sequence, not just a single vector.

All outputs are needed during training, but only the output at the last time step is useful for predictions and for evaluation. So although we will rely on the MSE over all the outputs for training, we will use a custom metric for evaluation, to only compute the MSE over the output at the last time step:

```
def last_time_step_mse(Y_true, Y_pred):
    return keras.metrics.mean_squared_error(Y_true[:, -1], Y_pred[:, -1])

optimizer = keras.optimizers.Adam(lr=0.01)
model.compile(loss="mse", optimizer=optimizer, metrics=[last_time_step_mse])
```

We get a validation MSE of about 0.006, which is 25% better than the previous model. You can combine this approach with the first one: just predict the next 10 values using this RNN, then concatenate these values to the input time series and use the model again to predict the next 10 values, and repeat the process as many times as needed. With this approach, you can generate arbitrarily long sequences. It may not be very accurate for long-term predictions, but it may be just fine if your goal is to generate original music or text, as we will see in Chapter 16.

When forecasting time series, it is often useful to have some error bars along with your predictions. For this, an efficient technique is MC Dropout, introduced in Chapter 11: add an MC Dropout layer within each memory cell, dropping part of the inputs and hidden states. After training, to forecast a new time series, use the model many times and compute the mean and standard deviation of the predictions at each time step.

Simple RNNs can be quite good at forecasting time series or handling other kinds of sequences, but they do not perform as well on long time series or sequences. Let's discuss why and see what we can do about it.

Handling Long Sequences

To train an RNN on long sequences, we must run it over many time steps, making the unrolled RNN a very deep network. Just like any deep neural network it may suffer from the unstable gradients problem, discussed in Chapter 11: it may take forever to train, or training may be unstable. Moreover, when an RNN processes a long sequence, it will gradually forget the first inputs in the sequence. Let's look at both these problems, starting with the unstable gradients problem.

Fighting the Unstable Gradients Problem

Many of the tricks we used in deep nets to alleviate the unstable gradients problem can also be used for RNNs: good parameter initialization, faster optimizers, dropout, and so on. However, nonsaturating activation functions (e.g., ReLU) may not help as much here; in fact, they may actually lead the RNN to be even more unstable during training. Why? Well, suppose Gradient Descent updates the weights in a way that increases the outputs slightly at the first time step. Because the same weights are used at every time step, the outputs at the second time step may also be slightly increased, and those at the third, and so on until the outputs explode—and a nonsaturating activation function does not prevent that. You can reduce this risk by using a smaller learning rate, but you can also simply use a saturating activation function like the hyperbolic tangent (this explains why it is the default). In much the same way, the gradients themselves can explode. If you notice that training is unstable, you may want to monitor the size of the gradients (e.g., using TensorBoard) and perhaps use Gradient Clipping.

Moreover, Batch Normalization cannot be used as efficiently with RNNs as with deep feedforward nets. In fact, you cannot use it between time steps, only between recurrent layers. To be more precise, it is technically possible to add a BN layer to a memory cell (as we will see shortly) so that it will be applied at each time step (both on the inputs for that time step and on the hidden state from the previous step). However, the same BN layer will be used at each time step, with the same parameters, regardless of the actual scale and offset of the inputs and hidden state. In practice, this does not yield good results, as was demonstrated by César Laurent et al. in a 2015 paper:[3] the authors found that BN was slightly beneficial only when it was applied to the inputs, not to the hidden states. In other words, it was slightly better than nothing when applied between recurrent layers (i.e., vertically in Figure 15-7), but not within recurrent layers (i.e., horizontally). In Keras this can be done simply by adding a `Batch Normalization` layer before each recurrent layer, but don't expect too much from it.

Another form of normalization often works better with RNNs: *Layer Normalization*. This idea was introduced by Jimmy Lei Ba et al. in a 2016 paper:[4] it is very similar to Batch Normalization, but instead of normalizing across the batch dimension, it normalizes across the features dimension. One advantage is that it can compute the required statistics on the fly, at each time step, independently for each instance. This also means that it behaves the same way during training and testing (as opposed to BN), and it does not need to use exponential moving averages to estimate the feature statistics across all instances in the training set. Like BN, Layer Normalization learns a

3 César Laurent et al., "Batch Normalized Recurrent Neural Networks," *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2016): 2657–2661.

4 Jimmy Lei Ba et al., "Layer Normalization," arXiv preprint arXiv:1607.06450 (2016).

scale and an offset parameter for each input. In an RNN, it is typically used right after the linear combination of the inputs and the hidden states.

Let's use tf.keras to implement Layer Normalization within a simple memory cell. For this, we need to define a custom memory cell. It is just like a regular layer, except its `call()` method takes two arguments: the `inputs` at the current time step and the hidden `states` from the previous time step. Note that the `states` argument is a list containing one or more tensors. In the case of a simple RNN cell it contains a single tensor equal to the outputs of the previous time step, but other cells may have multiple state tensors (e.g., an `LSTMCell` has a long-term state and a short-term state, as we will see shortly). A cell must also have a `state_size` attribute and an `output_size` attribute. In a simple RNN, both are simply equal to the number of units. The following code implements a custom memory cell which will behave like a `SimpleRNNCell`, except it will also apply Layer Normalization at each time step:

```
class LNSimpleRNNCell(keras.layers.Layer):
    def __init__(self, units, activation="tanh", **kwargs):
        super().__init__(**kwargs)
        self.state_size = units
        self.output_size = units
        self.simple_rnn_cell = keras.layers.SimpleRNNCell(units,
                                                          activation=None)
        self.layer_norm = keras.layers.LayerNormalization()
        self.activation = keras.activations.get(activation)
    def call(self, inputs, states):
        outputs, new_states = self.simple_rnn_cell(inputs, states)
        norm_outputs = self.activation(self.layer_norm(outputs))
        return norm_outputs, [norm_outputs]
```

The code is quite straightforward.[5] Our `LNSimpleRNNCell` class inherits from the `keras.layers.Layer` class, just like any custom layer. The constructor takes the number of units and the desired activation function, and it sets the `state_size` and `output_size` attributes, then creates a `SimpleRNNCell` with no activation function (because we want to perform Layer Normalization after the linear operation but before the activation function). Then the constructor creates the `LayerNormalization` layer, and finally it fetches the desired activation function. The `call()` method starts by applying the simple RNN cell, which computes a linear combination of the current inputs and the previous hidden states, and it returns the result twice (indeed, in a `SimpleRNNCell`, the outputs are just equal to the hidden states: in other words, `new_states[0]` is equal to `outputs`, so we can safely ignore `new_states` in the rest of the `call()` method). Next, the `call()` method applies Layer Normalization, followed

5 It would have been simpler to inherit from `SimpleRNNCell` instead so that we wouldn't have to create an internal `SimpleRNNCell` or handle the `state_size` and `output_size` attributes, but the goal here was to show how to create a custom cell from scratch.

by the activation function. Finally, it returns the outputs twice (once as the outputs, and once as the new hidden states). To use this custom cell, all we need to do is create a `keras.layers.RNN` layer, passing it a cell instance:

```
model = keras.models.Sequential([
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True,
                     input_shape=[None, 1]),
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Similarly, you could create a custom cell to apply dropout between each time step. But there's a simpler way: all recurrent layers (except for `keras.layers.RNN`) and all cells provided by Keras have a `dropout` hyperparameter and a `recurrent_dropout` hyperparameter: the former defines the dropout rate to apply to the inputs (at each time step), and the latter defines the dropout rate for the hidden states (also at each time step). No need to create a custom cell to apply dropout at each time step in an RNN.

With these techniques, you can alleviate the unstable gradients problem and train an RNN much more efficiently. Now let's look at how to deal with the short-term memory problem.

Tackling the Short-Term Memory Problem

Due to the transformations that the data goes through when traversing an RNN, some information is lost at each time step. After a while, the RNN's state contains virtually no trace of the first inputs. This can be a showstopper. Imagine Dory the fish[6] trying to translate a long sentence; by the time she's finished reading it, she has no clue how it started. To tackle this problem, various types of cells with long-term memory have been introduced. They have proven so successful that the basic cells are not used much anymore. Let's first look at the most popular of these long-term memory cells: the LSTM cell.

LSTM cells

The *Long Short-Term Memory* (LSTM) cell was proposed in 1997[7] by Sepp Hochreiter and Jürgen Schmidhuber and gradually improved over the years by several researchers, such as Alex Graves, Haşim Sak,[8] and Wojciech Zaremba.[9] If you consider the

6 A character from the animated movies *Finding Nemo* and *Finding Dory* who has short-term memory loss.

7 Sepp Hochreiter and Jürgen Schmidhuber, "Long Short-Term Memory," *Neural Computation* 9, no. 8 (1997): 1735–1780.

8 Haşim Sak et al., "Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition," arXiv preprint arXiv:1402.1128 (2014).

9 Wojciech Zaremba et al., "Recurrent Neural Network Regularization," arXiv preprint arXiv:1409.2329 (2014).

LSTM cell as a black box, it can be used very much like a basic cell, except it will perform much better; training will converge faster, and it will detect long-term dependencies in the data. In Keras, you can simply use the `LSTM` layer instead of the `SimpleRNN` layer:

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Alternatively, you could use the general-purpose `keras.layers.RNN` layer, giving it an `LSTMCell` as an argument:

```
model = keras.models.Sequential([
    keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True,
                     input_shape=[None, 1]),
    keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

However, the `LSTM` layer uses an optimized implementation when running on a GPU (see Chapter 19), so in general it is preferable to use it (the `RNN` layer is mostly useful when you define custom cells, as we did earlier).

So how does an LSTM cell work? Its architecture is shown in Figure 15-9.

If you don't look at what's inside the box, the LSTM cell looks exactly like a regular cell, except that its state is split into two vectors: $\mathbf{h}_{(t)}$ and $\mathbf{c}_{(t)}$ ("c" stands for "cell"). You can think of $\mathbf{h}_{(t)}$ as the short-term state and $\mathbf{c}_{(t)}$ as the long-term state.

Figure 15-9. LSTM cell

Now let's open the box! The key idea is that the network can learn what to store in the long-term state, what to throw away, and what to read from it. As the long-term state $\mathbf{c}_{(t-1)}$ traverses the network from left to right, you can see that it first goes through a *forget gate*, dropping some memories, and then it adds some new memories via the addition operation (which adds the memories that were selected by an *input gate*). The result $\mathbf{c}_{(t)}$ is sent straight out, without any further transformation. So, at each time step, some memories are dropped and some memories are added. Moreover, after the addition operation, the long-term state is copied and passed through the tanh function, and then the result is filtered by the *output gate*. This produces the short-term state $\mathbf{h}_{(t)}$ (which is equal to the cell's output for this time step, $\mathbf{y}_{(t)}$). Now let's look at where new memories come from and how the gates work.

First, the current input vector $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t-1)}$ are fed to four different fully connected layers. They all serve a different purpose:

- The main layer is the one that outputs $\mathbf{g}_{(t)}$. It has the usual role of analyzing the current inputs $\mathbf{x}_{(t)}$ and the previous (short-term) state $\mathbf{h}_{(t-1)}$. In a basic cell, there is nothing other than this layer, and its output goes straight out to $\mathbf{y}_{(t)}$ and $\mathbf{h}_{(t)}$. In contrast, in an LSTM cell this layer's output does not go straight out, but instead its most important parts are stored in the long-term state (and the rest is dropped).
- The three other layers are *gate controllers*. Since they use the logistic activation function, their outputs range from 0 to 1. As you can see, their outputs are fed to

element-wise multiplication operations, so if they output 0s they close the gate, and if they output 1s they open it. Specifically:

— The *forget gate* (controlled by $\mathbf{f}_{(t)}$) controls which parts of the long-term state should be erased.

— The *input gate* (controlled by $\mathbf{i}_{(t)}$) controls which parts of $\mathbf{g}_{(t)}$ should be added to the long-term state.

— Finally, the *output gate* (controlled by $\mathbf{o}_{(t)}$) controls which parts of the long-term state should be read and output at this time step, both to $\mathbf{h}_{(t)}$ and to $\mathbf{y}_{(t)}$.

In short, an LSTM cell can learn to recognize an important input (that's the role of the input gate), store it in the long-term state, preserve it for as long as it is needed (that's the role of the forget gate), and extract it whenever it is needed. This explains why these cells have been amazingly successful at capturing long-term patterns in time series, long texts, audio recordings, and more.

Equation 15-3 summarizes how to compute the cell's long-term state, its short-term state, and its output at each time step for a single instance (the equations for a whole mini-batch are very similar).

Equation 15-3. LSTM computations

$$
\begin{aligned}
\mathbf{i}_{(t)} &= \sigma\left(\mathbf{W}_{xi}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hi}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_i\right) \\
\mathbf{f}_{(t)} &= \sigma\left(\mathbf{W}_{xf}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hf}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_f\right) \\
\mathbf{o}_{(t)} &= \sigma\left(\mathbf{W}_{xo}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{ho}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_o\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_g\right) \\
\mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} \\
\mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right)
\end{aligned}
$$

In this equation:

- $\mathbf{W}_{xi}$, $\mathbf{W}_{xf}$, $\mathbf{W}_{xo}$, $\mathbf{W}_{xg}$ are the weight matrices of each of the four layers for their connection to the input vector $\mathbf{x}_{(t)}$.
- $\mathbf{W}_{hi}$, $\mathbf{W}_{hf}$, $\mathbf{W}_{ho}$, and $\mathbf{W}_{hg}$ are the weight matrices of each of the four layers for their connection to the previous short-term state $\mathbf{h}_{(t-1)}$.
- $\mathbf{b}_i$, $\mathbf{b}_f$, $\mathbf{b}_o$, and $\mathbf{b}_g$ are the bias terms for each of the four layers. Note that TensorFlow initializes $\mathbf{b}_f$ to a vector full of 1s instead of 0s. This prevents forgetting everything at the beginning of training.

Peephole connections

In a regular LSTM cell, the gate controllers can look only at the input $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t-1)}$. It may be a good idea to give them a bit more context by letting them peek at the long-term state as well. This idea was proposed by Felix Gers and Jürgen Schmidhuber in 2000.[10] They proposed an LSTM variant with extra connections called *peephole connections*: the previous long-term state $\mathbf{c}_{(t-1)}$ is added as an input to the controllers of the forget gate and the input gate, and the current long-term state $\mathbf{c}_{(t)}$ is added as input to the controller of the output gate. This often improves performance, but not always, and there is no clear pattern for which tasks are better off with or without them: you will have to try it on your task and see if it helps.

In Keras, the `LSTM` layer is based on the `keras.layers.LSTMCell` cell, which does not support peepholes. The experimental `tf.keras.experimental.PeepholeLSTMCell` does, however, so you can create a `keras.layers.RNN` layer and pass a `PeepholeLSTMCell` to its constructor.

There are many other variants of the LSTM cell. One particularly popular variant is the GRU cell, which we will look at now.

GRU cells

The *Gated Recurrent Unit* (GRU) cell (see Figure 15-10) was proposed by Kyunghyun Cho et al. in a 2014 paper[11] that also introduced the Encoder–Decoder network we discussed earlier.

10 F. A. Gers and J. Schmidhuber, "Recurrent Nets That Time and Count," *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks* (2000): 189–194.

11 Kyunghyun Cho et al., "Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation," *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing* (2014): 1724–1734.

Figure 15-10. GRU cell

The GRU cell is a simplified version of the LSTM cell, and it seems to perform just as well[12] (which explains its growing popularity). These are the main simplifications:

- Both state vectors are merged into a single vector $\mathbf{h}_{(t)}$.
- A single gate controller $\mathbf{z}_{(t)}$ controls both the forget gate and the input gate. If the gate controller outputs a 1, the forget gate is open (= 1) and the input gate is closed (1 – 1 = 0). If it outputs a 0, the opposite happens. In other words, whenever a memory must be stored, the location where it will be stored is erased first. This is actually a frequent variant to the LSTM cell in and of itself.
- There is no output gate; the full state vector is output at every time step. However, there is a new gate controller $\mathbf{r}_{(t)}$ that controls which part of the previous state will be shown to the main layer ($\mathbf{g}_{(t)}$).

12 A 2015 paper by Klaus Greff et al., "LSTM: A Search Space Odyssey", seems to show that all LSTM variants perform roughly the same.

Equation 15-4 summarizes how to compute the cell's state at each time step for a single instance.

Equation 15-4. GRU computations

$$
\begin{aligned}
\mathbf{z}_{(t)} &= \sigma\left(\mathbf{W}_{xz}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_{z}\right) \\
\mathbf{r}_{(t)} &= \sigma\left(\mathbf{W}_{xr}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_{r}\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{\top}\left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right) + \mathbf{b}_{g}\right) \\
\mathbf{h}_{(t)} &= \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + \left(1 - \mathbf{z}_{(t)}\right) \otimes \mathbf{g}_{(t)}
\end{aligned}
$$

Keras provides a `keras.layers.GRU` layer (based on the `keras.layers.GRUCell` memory cell); using it is just a matter of replacing `SimpleRNN` or `LSTM` with `GRU`.

LSTM and GRU cells are one of the main reasons behind the success of RNNs. Yet while they can tackle much longer sequences than simple RNNs, they still have a fairly limited short-term memory, and they have a hard time learning long-term patterns in sequences of 100 time steps or more, such as audio samples, long time series, or long sentences. One way to solve this is to shorten the input sequences, for example using 1D convolutional layers.

Using 1D convolutional layers to process sequences

In Chapter 14, we saw that a 2D convolutional layer works by sliding several fairly small kernels (or filters) across an image, producing multiple 2D feature maps (one per kernel). Similarly, a 1D convolutional layer slides several kernels across a sequence, producing a 1D feature map per kernel. Each kernel will learn to detect a single very short sequential pattern (no longer than the kernel size). If you use 10 kernels, then the layer's output will be composed of 10 1-dimensional sequences (all of the same length), or equivalently you can view this output as a single 10-dimensional sequence. This means that you can build a neural network composed of a mix of recurrent layers and 1D convolutional layers (or even 1D pooling layers). If you use a 1D convolutional layer with a stride of 1 and `"same"` padding, then the output sequence will have the same length as the input sequence. But if you use `"valid"` padding or a stride greater than 1, then the output sequence will be shorter than the input sequence, so make sure you adjust the targets accordingly. For example, the following model is the same as earlier, except it starts with a 1D convolutional layer that downsamples the input sequence by a factor of 2, using a stride of 2. The kernel size is larger than the stride, so all inputs will be used to compute the layer's output, and therefore the model can learn to preserve the useful information, dropping only the unimportant details. By shortening the sequences, the convolutional layer may help the `GRU` layers detect longer patterns. Note that we must also crop off the first three

time steps in the targets (since the kernel's size is 4, the first output of the convolutional layer will be based on the input time steps 0 to 3), and downsample the targets by a factor of 2:

```
model = keras.models.Sequential([
    keras.layers.Conv1D(filters=20, kernel_size=4, strides=2, padding="valid",
                        input_shape=[None, 1]),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])

model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train[:, 3::2], epochs=20,
                    validation_data=(X_valid, Y_valid[:, 3::2]))
```

If you train and evaluate this model, you will find that it is the best model so far. The convolutional layer really helps. In fact, it is actually possible to use only 1D convolutional layers and drop the recurrent layers entirely!

WaveNet

In a 2016 paper,[13] Aaron van den Oord and other DeepMind researchers introduced an architecture called *WaveNet*. They stacked 1D convolutional layers, doubling the dilation rate (how spread apart each neuron's inputs are) at every layer: the first convolutional layer gets a glimpse of just two time steps at a time, while the next one sees four time steps (its receptive field is four time steps long), the next one sees eight time steps, and so on (see Figure 15-11). This way, the lower layers learn short-term patterns, while the higher layers learn long-term patterns. Thanks to the doubling dilation rate, the network can process extremely large sequences very efficiently.

13 Aaron van den Oord et al., "WaveNet: A Generative Model for Raw Audio," arXiv preprint arXiv:1609.03499 (2016).

Figure 15-11. WaveNet architecture

In the WaveNet paper, the authors actually stacked 10 convolutional layers with dilation rates of 1, 2, 4, 8, …, 256, 512, then they stacked another group of 10 identical layers (also with dilation rates 1, 2, 4, 8, …, 256, 512), then again another identical group of 10 layers. They justified this architecture by pointing out that a single stack of 10 convolutional layers with these dilation rates will act like a super-efficient convolutional layer with a kernel of size 1,024 (except way faster, more powerful, and using significantly fewer parameters), which is why they stacked 3 such blocks. They also left-padded the input sequences with a number of zeros equal to the dilation rate before every layer, to preserve the same sequence length throughout the network. Here is how to implement a simplified WaveNet to tackle the same sequences as earlier:[14]

```
model = keras.models.Sequential()
model.add(keras.layers.InputLayer(input_shape=[None, 1]))
for rate in (1, 2, 4, 8) * 2:
    model.add(keras.layers.Conv1D(filters=20, kernel_size=2, padding="causal",
                                  activation="relu", dilation_rate=rate))
model.add(keras.layers.Conv1D(filters=10, kernel_size=1))
model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train, epochs=20,
                    validation_data=(X_valid, Y_valid))
```

This `Sequential` model starts with an explicit input layer (this is simpler than trying to set `input_shape` only on the first layer), then continues with a 1D convolutional layer using `"causal"` padding: this ensures that the convolutional layer does not peek into the future when making predictions (it is equivalent to padding the inputs with the right amount of zeros on the left and using `"valid"` padding). We then add

14 The complete WaveNet uses a few more tricks, such as skip connections like in a ResNet, and *Gated Activation Units* similar to those found in a GRU cell. Please see the notebook for more details.

similar pairs of layers using growing dilation rates: 1, 2, 4, 8, and again 1, 2, 4, 8. Finally, we add the output layer: a convolutional layer with 10 filters of size 1 and without any activation function. Thanks to the padding layers, every convolutional layer outputs a sequence of the same length as the input sequences, so the targets we use during training can be the full sequences: no need to crop them or downsample them.

The last two models offer the best performance so far in forecasting our time series! In the WaveNet paper, the authors achieved state-of-the-art performance on various audio tasks (hence the name of the architecture), including text-to-speech tasks, producing incredibly realistic voices across several languages. They also used the model to generate music, one audio sample at a time. This feat is all the more impressive when you realize that a single second of audio can contain tens of thousands of time steps—even LSTMs and GRUs cannot handle such long sequences.

In Chapter 16, we will continue to explore RNNs, and we will see how they can tackle various NLP tasks.

Exercises

1. Can you think of a few applications for a sequence-to-sequence RNN? What about a sequence-to-vector RNN, and a vector-to-sequence RNN?
2. How many dimensions must the inputs of an RNN layer have? What does each dimension represent? What about its outputs?
3. If you want to build a deep sequence-to-sequence RNN, which RNN layers should have `return_sequences=True`? What about a sequence-to-vector RNN?
4. Suppose you have a daily univariate time series, and you want to forecast the next seven days. Which RNN architecture should you use?
5. What are the main difficulties when training RNNs? How can you handle them?
6. Can you sketch the LSTM cell's architecture?
7. Why would you want to use 1D convolutional layers in an RNN?
8. Which neural network architecture could you use to classify videos?
9. Train a classification model for the SketchRNN dataset, available in TensorFlow Datasets.
10. Download the Bach chorales dataset and unzip it. It is composed of 382 chorales composed by Johann Sebastian Bach. Each chorale is 100 to 640 time steps long, and each time step contains 4 integers, where each integer corresponds to a note's index on a piano (except for the value 0, which means that no note is played). Train a model—recurrent, convolutional, or both—that can predict the next time step (four notes), given a sequence of time steps from a chorale. Then use this

model to generate Bach-like music, one note at a time: you can do this by giving the model the start of a chorale and asking it to predict the next time step, then appending these time steps to the input sequence and asking the model for the next note, and so on. Also make sure to check out Google's Coconet model, which was used for a nice Google doodle about Bach.

Solutions to these exercises are available in Appendix A.

CHAPTER 16

Natural Language Processing with RNNs and Attention

When Alan Turing imagined his famous Turing test[1] in 1950, his objective was to evaluate a machine's ability to match human intelligence. He could have tested for many things, such as the ability to recognize cats in pictures, play chess, compose music, or escape a maze, but, interestingly, he chose a linguistic task. More specifically, he devised a *chatbot* capable of fooling its interlocutor into thinking it was human.[2] This test does have its weaknesses: a set of hardcoded rules can fool unsuspecting or naive humans (e.g., the machine could give vague predefined answers in response to some keywords; it could pretend that it is joking or drunk, to get a pass on its weirdest answers; or it could escape difficult questions by answering them with its own questions), and many aspects of human intelligence are utterly ignored (e.g., the ability to interpret nonverbal communication such as facial expressions, or to learn a manual task). But the test does highlight the fact that mastering language is arguably *Homo sapiens*'s greatest cognitive ability. Can we build a machine that can read and write natural language?

A common approach for natural language tasks is to use recurrent neural networks. We will therefore continue to explore RNNs (introduced in Chapter 15), starting with a *character RNN*, trained to predict the next character in a sentence. This will allow us to generate some original text, and in the process we will see how to build a TensorFlow Dataset on a very long sequence. We will first use a *stateless RNN* (which learns

1 Alan Turing, "Computing Machinery and Intelligence," *Mind* 49 (1950): 433–460.

2 Of course, the word *chatbot* came much later. Turing called his test the *imitation game*: machine A and human B chat with human interrogator C via text messages; the interrogator asks questions to figure out which one is the machine (A or B). The machine passes the test if it can fool the interrogator, while the human B must try to help the interrogator.

on random portions of text at each iteration, without any information on the rest of the text), then we will build a *stateful RNN* (which preserves the hidden state between training iterations and continues reading where it left off, allowing it to learn longer patterns). Next, we will build an RNN to perform sentiment analysis (e.g., reading movie reviews and extracting the rater's feeling about the movie), this time treating sentences as sequences of words, rather than characters. Then we will show how RNNs can be used to build an Encoder–Decoder architecture capable of performing neural machine translation (NMT). For this, we will use the seq2seq API provided by the TensorFlow Addons project.

In the second part of this chapter, we will look at *attention mechanisms*. As their name suggests, these are neural network components that learn to select the part of the inputs that the rest of the model should focus on at each time step. First we will see how to boost the performance of an RNN-based Encoder–Decoder architecture using attention, then we will drop RNNs altogether and look at a very successful attention-only architecture called the *Transformer*. Finally, we will take a look at some of the most important advances in NLP in 2018 and 2019, including incredibly powerful language models such as GPT-2 and BERT, both based on Transformers.

Let's start with a simple and fun model that can write like Shakespeare (well, sort of).

Generating Shakespearean Text Using a Character RNN

In a famous 2015 blog post titled "The Unreasonable Effectiveness of Recurrent Neural Networks," Andrej Karpathy showed how to train an RNN to predict the next character in a sentence. This *Char-RNN* can then be used to generate novel text, one character at a time. Here is a small sample of the text generated by a Char-RNN model after it was trained on all of Shakespeare's work:

> PANDARUS:
>
> Alas, I think he shall be come approached and the day
>
> When little srain would be attain'd into being never fed,
>
> And who is but a chain and subjects of his death,
>
> I should not sleep.

Not exactly a masterpiece, but it is still impressive that the model was able to learn words, grammar, proper punctuation, and more, just by learning to predict the next character in a sentence. Let's look at how to build a Char-RNN, step by step, starting with the creation of the dataset.

Creating the Training Dataset

First, let's download all of Shakespeare's work, using Keras's handy `get_file()` function and downloading the data from Andrej Karpathy's Char-RNN project:

```
shakespeare_url = "https://homl.info/shakespeare" # shortcut URL
filepath = keras.utils.get_file("shakespeare.txt", shakespeare_url)
with open(filepath) as f:
    shakespeare_text = f.read()
```

Next, we must encode every character as an integer. One option is to create a custom preprocessing layer, as we did in Chapter 13. But in this case, it will be simpler to use Keras's `Tokenizer` class. First we need to fit a tokenizer to the text: it will find all the characters used in the text and map each of them to a different character ID, from 1 to the number of distinct characters (it does not start at 0, so we can use that value for masking, as we will see later in this chapter):

```
tokenizer = keras.preprocessing.text.Tokenizer(char_level=True)
tokenizer.fit_on_texts([shakespeare_text])
```

We set `char_level=True` to get character-level encoding rather than the default word-level encoding. Note that this tokenizer converts the text to lowercase by default (but you can set `lower=False` if you do not want that). Now the tokenizer can encode a sentence (or a list of sentences) to a list of character IDs and back, and it tells us how many distinct characters there are and the total number of characters in the text:

```
>>> tokenizer.texts_to_sequences(["First"])
[[20, 6, 9, 8, 3]]
>>> tokenizer.sequences_to_texts([[20, 6, 9, 8, 3]])
['f i r s t']
>>> max_id = len(tokenizer.word_index) # number of distinct characters
>>> dataset_size = tokenizer.document_count # total number of characters
```

Let's encode the full text so each character is represented by its ID (we subtract 1 to get IDs from 0 to 38, rather than from 1 to 39):

```
[encoded] = np.array(tokenizer.texts_to_sequences([shakespeare_text])) - 1
```

Before we continue, we need to split the dataset into a training set, a validation set, and a test set. We can't just shuffle all the characters in the text, so how do you split a sequential dataset?

How to Split a Sequential Dataset

It is very important to avoid any overlap between the training set, the validation set, and the test set. For example, we can take the first 90% of the text for the training set, then the next 5% for the validation set, and the final 5% for the test set. It would also

be a good idea to leave a gap between these sets to avoid the risk of a paragraph overlapping over two sets.

When dealing with time series, you would in general split across time,: for example, you might take the years 2000 to 2012 for the training set, the years 2013 to 2015 for the validation set, and the years 2016 to 2018 for the test set. However, in some cases you may be able to split along other dimensions, which will give you a longer time period to train on. For example, if you have data about the financial health of 10,000 companies from 2000 to 2018, you might be able to split this data across the different companies. It's very likely that many of these companies will be strongly correlated, though (e.g., whole economic sectors may go up or down jointly), and if you have correlated companies across the training set and the test set your test set will not be as useful, as its measure of the generalization error will be optimistically biased.

So, it is often safer to split across time—but this implicitly assumes that the patterns the RNN can learn in the past (in the training set) will still exist in the future. In other words, we assume that the time series is *stationary* (at least in a wide sense).[3] For many time series this assumption is reasonable (e.g., chemical reactions should be fine, since the laws of chemistry don't change every day), but for many others it is not (e.g., financial markets are notoriously not stationary since patterns disappear as soon as traders spot them and start exploiting them). To make sure the time series is indeed sufficiently stationary, you can plot the model's errors on the validation set across time: if the model performs much better on the first part of the validation set than on the last part, then the time series may not be stationary enough, and you might be better off training the model on a shorter time span.

In short, splitting a time series into a training set, a validation set, and a test set is not a trivial task, and how it's done will depend strongly on the task at hand.

Now back to Shakespeare! Let's take the first 90% of the text for the training set (keeping the rest for the validation set and the test set), and create a `tf.data.Dataset` that will return each character one by one from this set:

```
train_size = dataset_size * 90 // 100
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
```

Chopping the Sequential Dataset into Multiple Windows

The training set now consists of a single sequence of over a million characters, so we can't just train the neural network directly on it: the RNN would be equivalent to a

3 By definition, a stationary time series's mean, variance, and *autocorrelations* (i.e., correlations between values in the time series separated by a given interval) do not change over time. This is quite restrictive; for example, it excludes time series with trends or cyclical patterns. RNNs are more tolerant in that they can learn trends and cyclical patterns.

deep net with over a million layers, and we would have a single (very long) instance to train it. Instead, we will use the dataset's `window()` method to convert this long sequence of characters into many smaller windows of text. Every instance in the dataset will be a fairly short substring of the whole text, and the RNN will be unrolled only over the length of these substrings. This is called *truncated backpropagation through time*. Let's call the `window()` method to create a dataset of short text windows:

```
n_steps = 100
window_length = n_steps + 1 # target = input shifted 1 character ahead
dataset = dataset.window(window_length, shift=1, drop_remainder=True)
```

You can try tuning `n_steps`: it is easier to train RNNs on shorter input sequences, but of course the RNN will not be able to learn any pattern longer than `n_steps`, so don't make it too small.

By default, the `window()` method creates nonoverlapping windows, but to get the largest possible training set we use `shift=1` so that the first window contains characters 0 to 100, the second contains characters 1 to 101, and so on. To ensure that all windows are exactly 101 characters long (which will allow us to create batches without having to do any padding), we set `drop_remainder=True` (otherwise the last 100 windows will contain 100 characters, 99 characters, and so on down to 1 character).

The `window()` method creates a dataset that contains windows, each of which is also represented as a dataset. It's a *nested dataset*, analogous to a list of lists. This is useful when you want to transform each window by calling its dataset methods (e.g., to shuffle them or batch them). However, we cannot use a nested dataset directly for training, as our model will expect tensors as input, not datasets. So, we must call the `flat_map()` method: it converts a nested dataset into a *flat dataset* (one that does not contain datasets). For example, suppose {1, 2, 3} represents a dataset containing the sequence of tensors 1, 2, and 3. If you flatten the nested dataset {{1, 2}, {3, 4, 5, 6}}, you get back the flat dataset {1, 2, 3, 4, 5, 6}. Moreover, the `flat_map()` method takes a function as an argument, which allows you to transform each dataset in the nested dataset before flattening. For example, if you pass the function `lambda ds: ds.batch(2)` to `flat_map()`, then it will transform the nested dataset {{1, 2}, {3, 4, 5, 6}} into the flat dataset {[1, 2], [3, 4], [5, 6]}: it's a dataset of tensors of size 2. With that in mind, we are ready to flatten our dataset:

```
dataset = dataset.flat_map(lambda window: window.batch(window_length))
```

Notice that we call `batch(window_length)` on each window: since all windows have exactly that length, we will get a single tensor for each of them. Now the dataset contains consecutive windows of 101 characters each. Since Gradient Descent works best

when the instances in the training set are independent and identically distributed (see Chapter 4), we need to shuffle these windows. Then we can batch the windows and separate the inputs (the first 100 characters) from the target (the last character):

```
batch_size = 32
dataset = dataset.shuffle(10000).batch(batch_size)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
```

Figure 16-1 summarizes the dataset preparation steps discussed so far (showing windows of length 11 rather than 101, and a batch size of 3 instead of 32).

Figure 16-1. Preparing a dataset of shuffled windows

As discussed in Chapter 13, categorical input features should generally be encoded, usually as one-hot vectors or as embeddings. Here, we will encode each character using a one-hot vector because there are fairly few distinct characters (only 39):

```
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
```

Finally, we just need to add prefetching:

```
dataset = dataset.prefetch(1)
```

That's it! Preparing the dataset was the hardest part. Now let's create the model.

Building and Training the Char-RNN Model

To predict the next character based on the previous 100 characters, we can use an RNN with 2 `GRU` layers of 128 units each and 20% dropout on both the inputs (`dropout`) and the hidden states (`recurrent_dropout`). We can tweak these hyperparameters later, if needed. The output layer is a time-distributed `Dense` layer like we saw in Chapter 15. This time this layer must have 39 units (`max_id`) because there are 39 distinct characters in the text, and we want to output a probability for each possible character (at each time step). The output probabilities should sum up to 1 at each time step, so we apply the softmax activation function to the outputs of the `Dense`

layer. We can then compile this model, using the `"sparse_categorical_crossentropy"` loss and an Adam optimizer. Finally, we are ready to train the model for several epochs (this may take many hours, depending on your hardware):

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, input_shape=[None, max_id],
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.GRU(128, return_sequences=True,
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
history = model.fit(dataset, epochs=20)
```

Using the Char-RNN Model

Now we have a model that can predict the next character in text written by Shakespeare. To feed it some text, we first need to preprocess it like we did earlier, so let's create a little function for this:

```
def preprocess(texts):
    X = np.array(tokenizer.texts_to_sequences(texts)) - 1
    return tf.one_hot(X, max_id)
```

Now let's use the model to predict the next letter in some text:

```
>>> X_new = preprocess(["How are yo"])
>>> Y_pred = model.predict_classes(X_new)
>>> tokenizer.sequences_to_texts(Y_pred + 1)[0][-1] # 1st sentence, last char
'u'
```

Success! The model guessed right. Now let's use this model to generate new text.

Generating Fake Shakespearean Text

To generate new text using the Char-RNN model, we could feed it some text, make the model predict the most likely next letter, add it at the end of the text, then give the extended text to the model to guess the next letter, and so on. But in practice this often leads to the same words being repeated over and over again. Instead, we can pick the next character randomly, with a probability equal to the estimated probability, using TensorFlow's `tf.random.categorical()` function. This will generate more diverse and interesting text. The `categorical()` function samples random class indices, given the class log probabilities (logits). To have more control over the diversity of the generated text, we can divide the logits by a number called the *temperature*, which we can tweak as we wish: a temperature close to 0 will favor the high-probability characters, while a very high temperature will give all characters an equal probability. The following `next_char()` function uses this approach to pick the next character to add to the input text:

```
def next_char(text, temperature=1):
    X_new = preprocess([text])
    y_proba = model.predict(X_new)[0, -1:, :]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits, num_samples=1) + 1
    return tokenizer.sequences_to_texts(char_id.numpy())[0]
```

Next, we can write a small function that will repeatedly call `next_char()` to get the next character and append it to the given text:

```
def complete_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(text, temperature)
    return text
```

We are now ready to generate some text! Let's try with different temperatures:

```
>>> print(complete_text("t", temperature=0.2))
the belly the great and who shall be the belly the
>>> print(complete_text("w", temperature=1))
thing? or why you gremio.
who make which the first
>>> print(complete_text("w", temperature=2))
th no cce:
yeolg-hormer firi. a play asks.
fol rusb
```

Apparently our Shakespeare model works best at a temperature close to 1. To generate more convincing text, you could try using more `GRU` layers and more neurons per layer, train for longer, and add some regularization (for example, you could set `recurrent_dropout=0.3` in the `GRU` layers). Moreover, the model is currently incapable of learning patterns longer than `n_steps`, which is just 100 characters. You could try making this window larger, but it will also make training harder, and even LSTM and GRU cells cannot handle very long sequences. Alternatively, you could use a stateful RNN.

Stateful RNN

Until now, we have used only *stateless RNNs*: at each training iteration the model starts with a hidden state full of zeros, then it updates this state at each time step, and after the last time step, it throws it away, as it is not needed anymore. What if we told the RNN to preserve this final state after processing one training batch and use it as the initial state for the next training batch? This way the model can learn long-term patterns despite only backpropagating through short sequences. This is called a *stateful RNN*. Let's see how to build one.

First, note that a stateful RNN only makes sense if each input sequence in a batch starts exactly where the corresponding sequence in the previous batch left off. So the first thing we need to do to build a stateful RNN is to use sequential and nonoverlap-

ping input sequences (rather than the shuffled and overlapping sequences we used to train stateless RNNs). When creating the `Dataset`, we must therefore use `shift=n_steps` (instead of `shift=1`) when calling the `window()` method. Moreover, we must obviously *not* call the `shuffle()` method. Unfortunately, batching is much harder when preparing a dataset for a stateful RNN than it is for a stateless RNN. Indeed, if we were to call `batch(32)`, then 32 consecutive windows would be put in the same batch, and the following batch would not continue each of these window where it left off. The first batch would contain windows 1 to 32 and the second batch would contain windows 33 to 64, so if you consider, say, the first window of each batch (i.e., windows 1 and 33), you can see that they are not consecutive. The simplest solution to this problem is to just use "batches" containing a single window:

```
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
dataset = dataset.window(window_length, shift=n_steps, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(window_length))
dataset = dataset.batch(1)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
dataset = dataset.prefetch(1)
```

Figure 16-2 summarizes the first steps.

Figure 16-2. Preparing a dataset of consecutive sequence fragments for a stateful RNN

Batching is harder, but it is not impossible. For example, we could chop Shakespeare's text into 32 texts of equal length, create one dataset of consecutive input sequences for each of them, and finally use `tf.train.Dataset.zip(datasets).map(lambda *windows: tf.stack(windows))` to create proper consecutive batches, where the nth input sequence in a batch starts off exactly where the nth input sequence ended in the previous batch (see the notebook for the full code).

Now let's create the stateful RNN. First, we need to set `stateful=True` when creating every recurrent layer. Second, the stateful RNN needs to know the batch size (since it will preserve a state for each input sequence in the batch), so we must set the `batch_input_shape` argument in the first layer. Note that we can leave the second dimension unspecified, since the inputs could have any length:

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     dropout=0.2, recurrent_dropout=0.2,
                     batch_input_shape=[batch_size, None, max_id]),
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
```

At the end of each epoch, we need to reset the states before we go back to the beginning of the text. For this, we can use a small callback:

```
class ResetStatesCallback(keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

And now we can compile and fit the model (for more epochs, because each epoch is much shorter than earlier, and there is only one instance per batch):

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
model.fit(dataset, epochs=50, callbacks=[ResetStatesCallback()])
```

After this model is trained, it will only be possible to use it to make predictions for batches of the same size as were used during training. To avoid this restriction, create an identical *stateless* model, and copy the stateful model's weights to this model.

Now that we have built a character-level model, it's time to look at word-level models and tackle a common natural language processing task: *sentiment analysis*. In the process we will learn how to handle sequences of variable lengths using masking.

Sentiment Analysis

If MNIST is the "hello world" of computer vision, then the IMDb reviews dataset is the "hello world" of natural language processing: it consists of 50,000 movie reviews in English (25,000 for training, 25,000 for testing) extracted from the famous Internet Movie Database, along with a simple binary target for each review indicating whether it is negative (0) or positive (1). Just like MNIST, the IMDb reviews dataset is popular for good reasons: it is simple enough to be tackled on a laptop in a reasonable amount

of time, but challenging enough to be fun and rewarding. Keras provides a simple function to load it:

```
>>> (X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data()
>>> X_train[0][:10]
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

Where are the movie reviews? Well, as you can see, the dataset is already preprocessed for you: `X_train` consists of a list of reviews, each of which is represented as a NumPy array of integers, where each integer represents a word. All punctuation was removed, and then words were converted to lowercase, split by spaces, and finally indexed by frequency (so low integers correspond to frequent words). The integers 0, 1, and 2 are special: they represent the padding token, the *start-of-sequence* (SSS) token, and unknown words, respectively. If you want to visualize a review, you can decode it like this:

```
>>> word_index = keras.datasets.imdb.get_word_index()
>>> id_to_word = {id_ + 3: word for word, id_ in word_index.items()}
>>> for id_, token in enumerate(("<pad>", "<sos>", "<unk>")):
...     id_to_word[id_] = token
...
>>> " ".join([id_to_word[id_] for id_ in X_train[0][:10]])
'<sos> this film was just brilliant casting location scenery story'
```

In a real project, you will have to preprocess the text yourself. You can do that using the same `Tokenizer` class we used earlier, but this time setting `char_level=False` (which is the default). When encoding words, it filters out a lot of characters, including most punctuation, line breaks, and tabs (but you can change this by setting the `filters` argument). Most importantly, it uses spaces to identify word boundaries. This is OK for English and many other scripts (written languages) that use spaces between words, but not all scripts use spaces this way. Chinese does not use spaces between words, Vietnamese uses spaces even within words, and languages such as German often attach multiple words together, without spaces. Even in English, spaces are not always the best way to tokenize text: think of "San Francisco" or "#ILoveDeepLearning."

Fortunately, there are better options! The 2018 paper[4] by Taku Kudo introduced an unsupervised learning technique to tokenize and detokenize text at the subword level in a language-independent way, treating spaces like other characters. With this approach, even if your model encounters a word it has never seen before, it can still reasonably guess what it means. For example, it may never have seen the word "smartest" during training, but perhaps it learned the word "smart" and it also learned that the suffix "est" means "the most," so it can infer the meaning of

4 Taku Kudo, "Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates," arXiv preprint arXiv:1804.10959 (2018).

"smartest." Google's *SentencePiece* project provides an open source implementation, described in a paper[5] by Taku Kudo and John Richardson.

Another option was proposed in an earlier paper[6] by Rico Sennrich et al. that explored other ways of creating subword encodings (e.g., using *byte pair encoding*). Last but not least, the TensorFlow team released the TF.Text library in June 2019, which implements various tokenization strategies, including WordPiece[7] (a variant of byte pair encoding).

If you want to deploy your model to a mobile device or a web browser, and you don't want to have to write a different preprocessing function every time, then you will want to handle preprocessing using only TensorFlow operations, so it can be included in the model itself. Let's see how. First, let's load the original IMDb reviews, as text (byte strings), using TensorFlow Datasets (introduced in Chapter 13):

```
import tensorflow_datasets as tfds

datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
```

Next, let's write the preprocessing function:

```
def preprocess(X_batch, y_batch):
    X_batch = tf.strings.substr(X_batch, 0, 300)
    X_batch = tf.strings.regex_replace(X_batch, b"<br\\s*/?>", b" ")
    X_batch = tf.strings.regex_replace(X_batch, b"[^a-zA-Z']", b" ")
    X_batch = tf.strings.split(X_batch)
    return X_batch.to_tensor(default_value=b"<pad>"), y_batch
```

It starts by truncating the reviews, keeping only the first 300 characters of each: this will speed up training, and it won't impact performance too much because you can generally tell whether a review is positive or not in the first sentence or two. Then it uses *regular expressions* to replace `<br />` tags with spaces, and to replace any characters other than letters and quotes with spaces. For example, the text `"Well, I can't<br />"` will become `"Well I can't"`. Finally, the `preprocess()` function splits the reviews by the spaces, which returns a ragged tensor, and it converts this ragged tensor to a dense tensor, padding all reviews with the padding token `"<pad>"` so that they all have the same length.

5 Taku Kudo and John Richardson, "SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing," arXiv preprint arXiv:1808.06226 (2018).

6 Rico Sennrich et al., "Neural Machine Translation of Rare Words with Subword Units," *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics* 1 (2016): 1715–1725.

7 Yonghui Wu et al., "Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation," arXiv preprint arXiv:1609.08144 (2016).

Next, we need to construct the vocabulary. This requires going through the whole training set once, applying our `preprocess()` function, and using a `Counter` to count the number of occurrences of each word:

```
from collections import Counter
vocabulary = Counter()
for X_batch, y_batch in datasets["train"].batch(32).map(preprocess):
    for review in X_batch:
        vocabulary.update(list(review.numpy()))
```

Let's look at the three most common words:

```
>>> vocabulary.most_common()[:3]
[(b'<pad>', 215797), (b'the', 61137), (b'a', 38564)]
```

Great! We probably don't need our model to know all the words in the dictionary to get good performance, though, so let's truncate the vocabulary, keeping only the 10,000 most common words:

```
vocab_size = 10000
truncated_vocabulary = [
    word for word, count in vocabulary.most_common()[:vocab_size]]
```

Now we need to add a preprocessing step to replace each word with its ID (i.e., its index in the vocabulary). Just like we did in Chapter 13, we will create a lookup table for this, using 1,000 out-of-vocabulary (oov) buckets:

```
words = tf.constant(truncated_vocabulary)
word_ids = tf.range(len(truncated_vocabulary), dtype=tf.int64)
vocab_init = tf.lookup.KeyValueTensorInitializer(words, word_ids)
num_oov_buckets = 1000
table = tf.lookup.StaticVocabularyTable(vocab_init, num_oov_buckets)
```

We can then use this table to look up the IDs of a few words:

```
>>> table.lookup(tf.constant([b"This movie was faaaaaantastic".split()]))
<tf.Tensor: [...], dtype=int64, numpy=array([[   22,    12,    11, 10054]])>
```

Note that the words "this," "movie," and "was" were found in the table, so their IDs are lower than 10,000, while the word "faaaaaantastic" was not found, so it was mapped to one of the oov buckets, with an ID greater than or equal to 10,000.

TF Transform (introduced in Chapter 13) provides some useful functions to handle such vocabularies. For example, check out the `tft.compute_and_apply_vocabulary()` function: it will go through the dataset to find all distinct words and build the vocabulary, and it will generate the TF operations required to encode each word using this vocabulary.

Now we are ready to create the final training set. We batch the reviews, then convert them to short sequences of words using the `preprocess()` function, then encode

these words using a simple `encode_words()` function that uses the table we just built, and finally prefetch the next batch:

```
def encode_words(X_batch, y_batch):
    return table.lookup(X_batch), y_batch

train_set = datasets["train"].batch(32).map(preprocess)
train_set = train_set.map(encode_words).prefetch(1)
```

At last we can create the model and train it:

```
embed_size = 128
model = keras.models.Sequential([
    keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
                           input_shape=[None]),
    keras.layers.GRU(128, return_sequences=True),
    keras.layers.GRU(128),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
history = model.fit(train_set, epochs=5)
```

The first layer is an `Embedding` layer, which will convert word IDs into embeddings (introduced in Chapter 13). The embedding matrix needs to have one row per word ID (`vocab_size + num_oov_buckets`) and one column per embedding dimension (this example uses 128 dimensions, but this is a hyperparameter you could tune). Whereas the inputs of the model will be 2D tensors of shape [*batch size*, *time steps*], the output of the `Embedding` layer will be a 3D tensor of shape [*batch size*, *time steps*, *embedding size*].

The rest of the model is fairly straightforward: it is composed of two `GRU` layers, with the second one returning only the output of the last time step. The output layer is just a single neuron using the sigmoid activation function to output the estimated probability that the review expresses a positive sentiment regarding the movie. We then compile the model quite simply, and we fit it on the dataset we prepared earlier, for a few epochs.

Masking

As it stands, the model will need to learn that the padding tokens should be ignored. But we already know that! Why don't we tell the model to ignore the padding tokens, so that it can focus on the data that actually matters? It's actually trivial: simply add

`mask_zero=True` when creating the `Embedding` layer. This means that padding tokens (whose ID is 0)[8] will be ignored by all downstream layers. That's all!

The way this works is that the `Embedding` layer creates a *mask tensor* equal to `K.not_equal(inputs, 0)` (where `K = keras.backend`): it is a Boolean tensor with the same shape as the inputs, and it is equal to `False` anywhere the word IDs are 0, or `True` otherwise. This mask tensor is then automatically propagated by the model to all subsequent layers, as long as the time dimension is preserved. So in this example, both `GRU` layers will receive this mask automatically, but since the second `GRU` layer does not return sequences (it only returns the output of the last time step), the mask will not be transmitted to the `Dense` layer. Each layer may handle the mask differently, but in general they simply ignore masked time steps (i.e., time steps for which the mask is `False`). For example, when a recurrent layer encounters a masked time step, it simply copies the output from the previous time step. If the mask propagates all the way to the output (in models that output sequences, which is not the case in this example), then it will be applied to the losses as well, so the masked time steps will not contribute to the loss (their loss will be 0).

The `LSTM` and `GRU` layers have an optimized implementation for GPUs, based on Nvidia's cuDNN library. However, this implementation does not support masking. If your model uses a mask, then these layers will fall back to the (much slower) default implementation. Note that the optimized implementation also requires you to use the default values for several hyperparameters: `activation`, `recurrent_activation`, `recurrent_dropout`, `unroll`, `use_bias`, and `reset_after`.

All layers that receive the mask must support masking (or else an exception will be raised). This includes all recurrent layers, as well as the `TimeDistributed` layer and a few other layers. Any layer that supports masking must have a `supports_masking` attribute equal to `True`. If you want to implement your own custom layer with masking support, you should add a `mask` argument to the `call()` method (and obviously make the method use the mask somehow). Additionally, you should set `self.supports_masking = True` in the constructor. If your layer does not start with an `Embedding` layer, you may use the `keras.layers.Masking` layer instead: it sets the mask to `K.any(K.not_equal(inputs, 0), axis=-1)`, meaning that time steps where the last dimension is full of zeros will be masked out in subsequent layers (again, as long as the time dimension exists).

8 Their ID is 0 only because they are the most frequent "words" in the dataset. It would probably be a good idea to ensure that the padding tokens are always encoded as 0, even if they are not the most frequent.

Using masking layers and automatic mask propagation works best for simple `Sequential` models. It will not always work for more complex models, such as when you need to mix `Conv1D` layers with recurrent layers. In such cases, you will need to explicitly compute the mask and pass it to the appropriate layers, using either the Functional API or the Subclassing API. For example, the following model is identical to the previous model, except it is built using the Functional API and handles masking manually:

```
K = keras.backend
inputs = keras.layers.Input(shape=[None])
mask = keras.layers.Lambda(lambda inputs: K.not_equal(inputs, 0))(inputs)
z = keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size)(inputs)
z = keras.layers.GRU(128, return_sequences=True)(z, mask=mask)
z = keras.layers.GRU(128)(z, mask=mask)
outputs = keras.layers.Dense(1, activation="sigmoid")(z)
model = keras.Model(inputs=[inputs], outputs=[outputs])
```

After training for a few epochs, this model will become quite good at judging whether a review is positive or not. If you use the `TensorBoard()` callback, you can visualize the embeddings in TensorBoard as they are being learned: it is fascinating to see words like "awesome" and "amazing" gradually cluster on one side of the embedding space, while words like "awful" and "terrible" cluster on the other side. Some words are not as positive as you might expect (at least with this model), such as the word "good," presumably because many negative reviews contain the phrase "not good." It's impressive that the model is able to learn useful word embeddings based on just 25,000 movie reviews. Imagine how good the embeddings would be if we had billions of reviews to train on! Unfortunately we don't, but perhaps we can reuse word embeddings trained on some other large text corpus (e.g., Wikipedia articles), even if it is not composed of movie reviews? After all, the word "amazing" generally has the same meaning whether you use it to talk about movies or anything else. Moreover, perhaps embeddings would be useful for sentiment analysis even if they were trained on another task: since words like "awesome" and "amazing" have a similar meaning, they will likely cluster in the embedding space even for other tasks (e.g., predicting the next word in a sentence). If all positive words and all negative words form clusters, then this will be helpful for sentiment analysis. So instead of using so many parameters to learn word embeddings, let's see if we can't just reuse pretrained embeddings.

Reusing Pretrained Embeddings

The TensorFlow Hub project makes it easy to reuse pretrained model components in your own models. These model components are called *modules*. Simply browse the TF Hub repository, find the one you need, and copy the code example into your project, and the module will be automatically downloaded, along with its pretrained weights, and included in your model. Easy!

For example, let's use the `nnlm-en-dim50` sentence embedding module, version 1, in our sentiment analysis model:

```
import tensorflow_hub as hub

model = keras.Sequential([
    hub.KerasLayer("https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1",
                   dtype=tf.string, input_shape=[], output_shape=[50]),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
```

The `hub.KerasLayer` layer downloads the module from the given URL. This particular module is a *sentence encoder*: it takes strings as input and encodes each one as a single vector (in this case, a 50-dimensional vector). Internally, it parses the string (splitting words on spaces) and embeds each word using an embedding matrix that was pretrained on a huge corpus: the Google News 7B corpus (seven billion words long!). Then it computes the mean of all the word embeddings, and the result is the sentence embedding.[9] We can then add two simple `Dense` layers to create a good sentiment analysis model. By default, a `hub.KerasLayer` is not trainable, but you can set `trainable=True` when creating it to change that so that you can fine-tune it for your task.

Not all TF Hub modules support TensorFlow 2, so make sure you choose a module that does.

Next, we can just load the IMDb reviews dataset—no need to preprocess it (except for batching and prefetching)—and directly train the model:

```
datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
batch_size = 32
train_set = datasets["train"].batch(batch_size).prefetch(1)
history = model.fit(train_set, epochs=5)
```

Note that the last part of the TF Hub module URL specified that we wanted version 1 of the model. This versioning ensures that if a new module version is released, it will not break our model. Conveniently, if you just enter this URL in a web browser, you

9 To be precise, the sentence embedding is equal to the mean word embedding multiplied by the square root of the number of words in the sentence. This compensates for the fact that the mean of n vectors gets shorter as n grows.

will get the documentation for this module. By default, TF Hub will cache the downloaded files into the local system's temporary directory. You may prefer to download them into a more permanent directory to avoid having to download them again after every system cleanup. To do that, set the `TFHUB_CACHE_DIR` environment variable to the directory of your choice (e.g., `os.environ["TFHUB_CACHE_DIR"] = "./my_tfhub_cache"`).

So far, we have looked at time series, text generation using Char-RNN, and sentiment analysis using word-level RNN models, training our own word embeddings or reusing pretrained embeddings. Let's now look at another important NLP task: *neural machine translation* (NMT), first using a pure Encoder–Decoder model, then improving it with attention mechanisms, and finally looking the extraordinary Transformer architecture.

An Encoder–Decoder Network for Neural Machine Translation

Let's take a look at a simple neural machine translation model[10] that will translate English sentences to French (see Figure 16-3).

In short, the English sentences are fed to the encoder, and the decoder outputs the French translations. Note that the French translations are also used as inputs to the decoder, but shifted back by one step. In other words, the decoder is given as input the word that it *should* have output at the previous step (regardless of what it actually output). For the very first word, it is given the start-of-sequence (SOS) token. The decoder is expected to end the sentence with an end-of-sequence (EOS) token.

Note that the English sentences are reversed before they are fed to the encoder. For example, "I drink milk" is reversed to "milk drink I." This ensures that the beginning of the English sentence will be fed last to the encoder, which is useful because that's generally the first thing that the decoder needs to translate.

Each word is initially represented by its ID (e.g., 288 for the word "milk"). Next, an `embedding` layer returns the word embedding. These word embeddings are what is actually fed to the encoder and the decoder.

10 Ilya Sutskever et al., "Sequence to Sequence Learning with Neural Networks," arXiv preprint arXiv:1409.3215 (2014).

Figure 16-3. A simple machine translation model

At each step, the decoder outputs a score for each word in the output vocabulary (i.e., French), and then the softmax layer turns these scores into probabilities. For example, at the first step the word "Je" may have a probability of 20%, "Tu" may have a probability of 1%, and so on. The word with the highest probability is output. This is very much like a regular classification task, so you can train the model using the `"sparse_categorical_crossentropy"` loss, much like we did in the Char-RNN model.

Note that at inference time (after training), you will not have the target sentence to feed to the decoder. Instead, simply feed the decoder the word that it output at the previous step, as shown in Figure 16-4 (this will require an embedding lookup that is not shown in the diagram).

Figure 16-4. Feeding the previous output word as input at inference time

OK, now you have the big picture. Still, there are a few more details to handle if you implement this model:

- So far we have assumed that all input sequences (to the encoder and to the decoder) have a constant length. But obviously sentence lengths vary. Since regular tensors have fixed shapes, they can only contain sentences of the same length. You can use masking to handle this, as discussed earlier. However, if the sentences have very different lengths, you can't just crop them like we did for sentiment analysis (because we want full translations, not cropped translations). Instead, group sentences into buckets of similar lengths (e.g., a bucket for the 1- to 6-word sentences, another for the 7- to 12-word sentences, and so on), using padding for the shorter sequences to ensure all sentences in a bucket have the same length (check out the `tf.data.experimental.bucket_by_sequence_length()` function for this). For example, "I drink milk" becomes "<pad> <pad> <pad> milk drink I."
- We want to ignore any output past the EOS token, so these tokens should not contribute to the loss (they must be masked out). For example, if the model outputs "Je bois du lait <eos> oui," the loss for the last word should be ignored.
- When the output vocabulary is large (which is the case here), outputting a probability for each and every possible word would be terribly slow. If the target vocabulary contains, say, 50,000 French words, then the decoder would output 50,000-dimensional vectors, and then computing the softmax function over such a large vector would be very computationally intensive. To avoid this, one solution is to look only at the logits output by the model for the correct word and for a random sample of incorrect words, then compute an approximation of the loss based only on these logits. This *sampled softmax* technique was introduced in

2015 by Sébastien Jean et al..[11] In TensorFlow you can use the `tf.nn.sampled_softmax_loss()` function for this during training and use the normal softmax function at inference time (sampled softmax cannot be used at inference time because it requires knowing the target).

The TensorFlow Addons project includes many sequence-to-sequence tools to let you easily build production-ready Encoder–Decoders. For example, the following code creates a basic Encoder–Decoder model, similar to the one represented in Figure 16-3:

```
import tensorflow_addons as tfa

encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
sequence_lengths = keras.layers.Input(shape=[], dtype=np.int32)

embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)

encoder = keras.layers.LSTM(512, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_embeddings)
encoder_state = [state_h, state_c]

sampler = tfa.seq2seq.sampler.TrainingSampler()

decoder_cell = keras.layers.LSTMCell(512)
output_layer = keras.layers.Dense(vocab_size)
decoder = tfa.seq2seq.basic_decoder.BasicDecoder(decoder_cell, sampler,
                                                 output_layer=output_layer)
final_outputs, final_state, final_sequence_lengths = decoder(
    decoder_embeddings, initial_state=encoder_state,
    sequence_length=sequence_lengths)
Y_proba = tf.nn.softmax(final_outputs.rnn_output)

model = keras.Model(inputs=[encoder_inputs, decoder_inputs, sequence_lengths],
                    outputs=[Y_proba])
```

The code is mostly self-explanatory, but there are a few points to note. First, we set `return_state=True` when creating the `LSTM` layer so that we can get its final hidden state and pass it to the decoder. Since we are using an LSTM cell, it actually returns two hidden states (short term and long term). The `TrainingSampler` is one of several samplers available in TensorFlow Addons: their role is to tell the decoder at each step what it should pretend the previous output was. During inference, this should be the

11 Sébastien Jean et al., "On Using Very Large Target Vocabulary for Neural Machine Translation," *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing* 1 (2015): 1–10.

embedding of the token that was actually output. During training, it should be the embedding of the previous target token: this is why we used the `TrainingSampler`. In practice, it is often a good idea to start training with the embedding of the target of the previous time step and gradually transition to using the embedding of the actual token that was output at the previous step. This idea was introduced in a 2015 paper[12] by Samy Bengio et al. The `ScheduledEmbeddingTrainingSampler` will randomly choose between the target or the actual output, with a probability that you can gradually change during training.

Bidirectional RNNs

A each time step, a regular recurrent layer only looks at past and present inputs before generating its output. In other words, it is "causal," meaning it cannot look into the future. This type of RNN makes sense when forecasting time series, but for many NLP tasks, such as Neural Machine Translation, it is often preferable to look ahead at the next words before encoding a given word. For example, consider the phrases "the Queen of the United Kingdom," "the queen of hearts," and "the queen bee": to properly encode the word "queen," you need to look ahead. To implement this, run two recurrent layers on the same inputs, one reading the words from left to right and the other reading them from right to left. Then simply combine their outputs at each time step, typically by concatenating them. This is called a *bidirectional recurrent layer* (see Figure 16-5).

To implement a bidirectional recurrent layer in Keras, wrap a recurrent layer in a `keras.layers.Bidirectional` layer. For example, the following code creates a bidirectional `GRU` layer:

```
keras.layers.Bidirectional(keras.layers.GRU(10, return_sequences=True))
```

The `Bidirectional` layer will create a clone of the `GRU` layer (but in the reverse direction), and it will run both and concatenate their outputs. So although the `GRU` layer has 10 units, the `Bidirectional` layer will output 20 values per time step.

12 Samy Bengio et al., "Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks," arXiv preprint arXiv:1506.03099 (2015).

Figure 16-5. A bidirectional recurrent layer

Beam Search

Suppose you train an Encoder–Decoder model, and use it to translate the French sentence "Comment vas-tu?" to English. You are hoping that it will output the proper translation ("How are you?"), but unfortunately it outputs "How will you?" Looking at the training set, you notice many sentences such as "Comment vas-tu jouer?" which translates to "How will you play?" So it wasn't absurd for the model to output "How will" after seeing "Comment vas." Unfortunately, in this case it was a mistake, and the model could not go back and fix it, so it tried to complete the sentence as best it could. By greedily outputting the most likely word at every step, it ended up with a suboptimal translation. How can we give the model a chance to go back and fix mistakes it made earlier? One of the most common solutions is *beam search*: it keeps track of a short list of the *k* most promising sentences (say, the top three), and at each decoder step it tries to extend them by one word, keeping only the *k* most likely sentences. The parameter *k* is called the *beam width*.

For example, suppose you use the model to translate the sentence "Comment vas-tu?" using beam search with a beam width of 3. At the first decoder step, the model will output an estimated probability for each possible word. Suppose the top three words are "How" (75% estimated probability), "What" (3%), and "You" (1%). That's our short list so far. Next, we create three copies of our model and use them to find the next word for each sentence. Each model will output one estimated probability per word in the vocabulary. The first model will try to find the next word in the sentence "How," and perhaps it will output a probability of 36% for the word "will," 32% for the word "are," 16% for the word "do," and so on. Note that these are actually *conditional* probabilities, given that the sentence starts with "How." The second model will try to complete the sentence "What"; it might output a conditional probability of 50% for

the word "are," and so on. Assuming the vocabulary has 10,000 words, each model will output 10,000 probabilities.

Next, we compute the probabilities of each of the 30,000 two-word sentences that these models considered (3 × 10,000). We do this by multiplying the estimated conditional probability of each word by the estimated probability of the sentence it completes. For example, the estimated probability of the sentence "How" was 75%, while the estimated conditional probability of the word "will" (given that the first word is "How") was 36%, so the estimated probability of the sentence "How will" is 75% × 36% = 27%. After computing the probabilities of all 30,000 two-word sentences, we keep only the top 3. Perhaps they all start with the word "How": "How will" (27%), "How are" (24%), and "How do" (12%). Right now, the sentence "How will" is winning, but "How are" has not been eliminated.

Then we repeat the same process: we use three models to predict the next word in each of these three sentences, and we compute the probabilities of all 30,000 three-word sentences we considered. Perhaps the top three are now "How are you" (10%), "How do you" (8%), and "How will you" (2%). At the next step we may get "How do you do" (7%), "How are you <eos>" (6%), and "How are you doing" (3%). Notice that "How will" was eliminated, and we now have three perfectly reasonable translations. We boosted our Encoder–Decoder model's performance without any extra training, simply by using it more wisely.

You can implement beam search fairly easily using TensorFlow Addons:

```
beam_width = 10
decoder = tfa.seq2seq.beam_search_decoder.BeamSearchDecoder(
    cell=decoder_cell, beam_width=beam_width, output_layer=output_layer)
decoder_initial_state = tfa.seq2seq.beam_search_decoder.tile_batch(
    encoder_state, multiplier=beam_width)
outputs, _, _ = decoder(
    embedding_decoder, start_tokens=start_tokens, end_token=end_token,
    initial_state=decoder_initial_state)
```

We first create a `BeamSearchDecoder`, which wraps all the decoder clones (in this case 10 clones). Then we create one copy of the encoder's final state for each decoder clone, and we pass these states to the decoder, along with the start and end tokens.

With all this, you can get good translations for fairly short sentences (especially if you use pretrained word embeddings). Unfortunately, this model will be really bad at translating long sentences. Once again, the problem comes from the limited short-term memory of RNNs. *Attention mechanisms* are the game-changing innovation that addressed this problem.

Attention Mechanisms

Consider the path from the word "milk" to its translation "lait" in Figure 16-3: it is quite long! This means that a representation of this word (along with all the other words) needs to be carried over many steps before it is actually used. Can't we make this path shorter?

This was the core idea in a groundbreaking 2014 paper[13] by Dzmitry Bahdanau et al. They introduced a technique that allowed the decoder to focus on the appropriate words (as encoded by the encoder) at each time step. For example, at the time step where the decoder needs to output the word "lait," it will focus its attention on the word "milk." This means that the path from an input word to its translation is now much shorter, so the short-term memory limitations of RNNs have much less impact. Attention mechanisms revolutionized neural machine translation (and NLP in general), allowing a significant improvement in the state of the art, especially for long sentences (over 30 words).[14]

Figure 16-6 shows this model's architecture (slightly simplified, as we will see). On the left, you have the encoder and the decoder. Instead of just sending the encoder's final hidden state to the decoder (which is still done, although it is not shown in the figure), we now send all of its outputs to the decoder. At each time step, the decoder's memory cell computes a weighted sum of all these encoder outputs: this determines which words it will focus on at this step. The weight $\alpha_{(t,i)}$ is the weight of the ith encoder output at the tth decoder time step. For example, if the weight $\alpha_{(3,2)}$ is much larger than the weights $\alpha_{(3,0)}$ and $\alpha_{(3,1)}$, then the decoder will pay much more attention to word number 2 ("milk") than to the other two words, at least at this time step. The rest of the decoder works just like earlier: at each time step the memory cell receives the inputs we just discussed, plus the hidden state from the previous time step, and finally (although it is not represented in the diagram) it receives the target word from the previous time step (or at inference time, the output from the previous time step).

13 Dzmitry Bahdanau et al., "Neural Machine Translation by Jointly Learning to Align and Translate," arXiv preprint arXiv:1409.0473 (2014).

14 The most common metric used in NMT is the BiLingual Evaluation Understudy (BLEU) score, which compares each translation produced by the model with several good translations produced by humans: it counts the number of *n*-grams (sequences of *n* words) that appear in any of the target translations and adjusts the score to take into account the frequency of the produced *n*-grams in the target translations.

Figure 16-6. Neural machine translation using an Encoder–Decoder network with an attention model

But where do these $\alpha_{(t,i)}$ weights come from? It's actually pretty simple: they are generated by a type of small neural network called an *alignment model* (or an *attention layer*), which is trained jointly with the rest of the Encoder–Decoder model. This alignment model is illustrated on the righthand side of Figure 16-6. It starts with a time-distributed `Dense` layer[15] with a single neuron, which receives as input all the encoder outputs, concatenated with the decoder's previous hidden state (e.g., $\mathbf{h}_{(2)}$). This layer outputs a score (or energy) for each encoder output (e.g., $e_{(3,\,2)}$): this score measures how well each output is aligned with the decoder's previous hidden state. Finally, all the scores go through a softmax layer to get a final weight for each encoder output (e.g., $\alpha_{(3,2)}$). All the weights for a given decoder time step add up to 1 (since the softmax layer is not time-distributed). This particular attention mechanism is called *Bahdanau attention* (named after the paper's first author). Since it concatenates the encoder output with the decoder's previous hidden state, it is sometimes called *concatenative attention* (or *additive attention*).

15 Recall that a time-distributed `Dense` layer is equivalent to a regular `Dense` layer that you apply independently at each time step (only much faster).

If the input sentence is n words long, and assuming the output sentence is about as long, then this model will need to compute about n^2 weights. Fortunately, this quadratic computational complexity is still tractable because even long sentences don't have thousands of words.

Another common attention mechanism was proposed shortly after, in a 2015 paper[16] by Minh-Thang Luong et al. Because the goal of the attention mechanism is to measure the similarity between one of the encoder's outputs and the decoder's previous hidden state, the authors proposed to simply compute the *dot product* (see Chapter 4) of these two vectors, as this is often a fairly good similarity measure, and modern hardware can compute it much faster. For this to be possible, both vectors must have the same dimensionality. This is called *Luong attention* (again, after the paper's first author), or sometimes *multiplicative attention*. The dot product gives a score, and all the scores (at a given decoder time step) go through a softmax layer to give the final weights, just like in Bahdanau attention. Another simplification they proposed was to use the decoder's hidden state at the current time step rather than at the previous time step (i.e., $\mathbf{h}_{(t)}$) rather than $\mathbf{h}_{(t-1)}$), then to use the output of the attention mechanism (noted $\tilde{\mathbf{h}}_{(t)}$) directly to compute the decoder's predictions (rather than using it to compute the decoder's current hidden state). They also proposed a variant of the dot product mechanism where the encoder outputs first go through a linear transformation (i.e., a time-distributed `Dense` layer without a bias term) before the dot products are computed. This is called the "general" dot product approach. They compared both dot product approaches to the concatenative attention mechanism (adding a rescaling parameter vector $\mathbf{v}$), and they observed that the dot product variants performed better than concatenative attention. For this reason, concatenative attention is much less used now. The equations for these three attention mechanisms are summarized in Equation 16-1.

16 Minh-Thang Luong et al., "Effective Approaches to Attention-Based Neural Machine Translation," *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (2015): 1412–1421.

Equation 16-1. Attention mechanisms

$$\widetilde{\mathbf{h}}_{(t)} = \sum_i \alpha_{(t,i)} \mathbf{y}_{(i)}$$

$$\text{with } \alpha_{(t,i)} = \frac{\exp\left(e_{(t,i)}\right)}{\sum_{i'} \exp\left(e_{(t,i')}\right)}$$

$$\text{and } e_{(t,i)} = \begin{cases} \mathbf{h}_{(t)}{}^{\top} \mathbf{y}_{(i)} & \textit{dot} \\ \mathbf{h}_{(t)}{}^{\top} \mathbf{W}\, \mathbf{y}_{(i)} & \textit{general} \\ \mathbf{v}^{\top} \tanh\left(\mathbf{W}\left[\mathbf{h}_{(t)}; \mathbf{y}_{(i)}\right]\right) & \textit{concat} \end{cases}$$

Here is how you can add Luong attention to an Encoder–Decoder model using TensorFlow Addons:

```
attention_mechanism = tfa.seq2seq.attention_wrapper.LuongAttention(
    units, encoder_state, memory_sequence_length=encoder_sequence_length)
attention_decoder_cell = tfa.seq2seq.attention_wrapper.AttentionWrapper(
    decoder_cell, attention_mechanism, attention_layer_size=n_units)
```

We simply wrap the decoder cell in an `AttentionWrapper`, and we provide the desired attention mechanism (Luong attention in this example).

Visual Attention

Attention mechanisms are now used for a variety of purposes. One of their first applications beyond NMT was in generating image captions using visual attention:[17] a convolutional neural network first processes the image and outputs some feature maps, then a decoder RNN equipped with an attention mechanism generates the caption, one word at a time. At each decoder time step (each word), the decoder uses the attention model to focus on just the right part of the image. For example, in Figure 16-7, the model generated the caption "A woman is throwing a frisbee in a park," and you can see what part of the input image the decoder focused its attention on when it was about to output the word "frisbee": clearly, most of its attention was focused on the frisbee.

17 Kelvin Xu et al., "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention," *Proceedings of the 32nd International Conference on Machine Learning* (2015): 2048–2057.

Figure 16-7. Visual attention: an input image (left) and the model's focus before producing the word "frisbee" (right)[18]

Explainability

One extra benefit of attention mechanisms is that they make it easier to understand what led the model to produce its output. This is called *explainability*. It can be especially useful when the model makes a mistake: for example, if an image of a dog walking in the snow is labeled as "a wolf walking in the snow," then you can go back and check what the model focused on when it output the word "wolf." You may find that it was paying attention not only to the dog, but also to the snow, hinting at a possible explanation: perhaps the way the model learned to distinguish dogs from wolves is by checking whether or not there's a lot of snow around. You can then fix this by training the model with more images of wolves without snow, and dogs with snow. This example comes from a great 2016 paper[19] by Marco Tulio Ribeiro et al. that uses a different approach to explainability: learning an interpretable model locally around a classifier's prediction.

In some applications, explainability is not just a tool to debug a model; it can be a legal requirement (think of a system deciding whether or not it should grant you a loan).

18 This is a part of figure 3 from the paper. It is reproduced with the kind authorization of the authors.

19 Marco Tulio Ribeiro et al., "'Why Should I Trust You?': Explaining the Predictions of Any Classifier," *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016): 1135–1144.

Attention mechanisms are so powerful that you can actually build state-of-the-art models using only attention mechanisms.

Attention Is All You Need: The Transformer Architecture

In a groundbreaking 2017 paper,[20] a team of Google researchers suggested that "Attention Is All You Need." They managed to create an architecture called the *Transformer*, which significantly improved the state of the art in NMT without using any recurrent or convolutional layers,[21] just attention mechanisms (plus embedding layers, dense layers, normalization layers, and a few other bits and pieces). As an extra bonus, this architecture was also much faster to train and easier to parallelize, so they managed to train it at a fraction of the time and cost of the previous state-of-the-art models.

The Transformer architecture is represented in Figure 16-8.

20 Ashish Vaswani et al., "Attention Is All You Need," *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 6000–6010.

21 Since the Transformer uses time-distributed `Dense` layers, you could argue that it uses 1D convolutional layers with a kernel size of 1.

Figure 16-8. The Transformer architecture[22]

Let's walk through this figure:

- The lefthand part is the encoder. Just like earlier, it takes as input a batch of sentences represented as sequences of word IDs (the input shape is [*batch size*, *max input sentence length*]), and it encodes each word into a 512-dimensional representation (so the encoder's output shape is [*batch size*, *max input sentence length*, 512]). Note that the top part of the encoder is stacked N times (in the paper, $N = 6$).

22 This is figure 1 from the paper, reproduced with the kind authorization of the authors.

- The righthand part is the decoder. During training, it takes the target sentence as input (also represented as a sequence of word IDs), shifted one time step to the right (i.e., a start-of-sequence token is inserted at the beginning). It also receives the outputs of the encoder (i.e., the arrows coming from the left side). Note that the top part of the decoder is also stacked N times, and the encoder stack's final outputs are fed to the decoder at each of these N levels. Just like earlier, the decoder outputs a probability for each possible next word, at each time step (its output shape is [*batch size*, *max output sentence length*, *vocabulary length*]).
- During inference, the decoder cannot be fed targets, so we feed it the previously output words (starting with a start-of-sequence token). So the model needs to be called repeatedly, predicting one more word at every round (which is fed to the decoder at the next round, until the end-of-sequence token is output).
- Looking more closely, you can see that you are already familiar with most components: there are two embedding layers, $5 \times N$ skip connections, each of them followed by a layer normalization layer, $2 \times N$ "Feed Forward" modules that are composed of two dense layers each (the first one using the ReLU activation function, the second with no activation function), and finally the output layer is a dense layer using the softmax activation function. All of these layers are time-distributed, so each word is treated independently of all the others. But how can we translate a sentence by only looking at one word at a time? Well, that's where the new components come in:
  - The encoder's *Multi-Head Attention* layer encodes each word's relationship with every other word in the same sentence, paying more attention to the most relevant ones. For example, the output of this layer for the word "Queen" in the sentence "They welcomed the Queen of the United Kingdom" will depend on all the words in the sentence, but it will probably pay more attention to the words "United" and "Kingdom" than to the words "They" or "welcomed." This attention mechanism is called *self-attention* (the sentence is paying attention to itself). We will discuss exactly how it works shortly. The decoder's *Masked Multi-Head Attention* layer does the same thing, but each word is only allowed to attend to words located before it. Finally, the decoder's upper Multi-Head Attention layer is where the decoder pays attention to the words in the input sentence. For example, the decoder will probably pay close attention to the word "Queen" in the input sentence when it is about to output this word's translation.
  - The *positional embeddings* are simply dense vectors (much like word embeddings) that represent the position of a word in the sentence. The nth positional embedding is added to the word embedding of the nth word in each sentence. This gives the model access to each word's position, which is needed because the Multi-Head Attention layers do not consider the order or the position of the words; they only look at their relationships. Since all the other layers are

time-distributed, they have no way of knowing the position of each word (either relative or absolute). Obviously, the relative and absolute word positions are important, so we need to give this information to the Transformer somehow, and positional embeddings are a good way to do this.

Let's look a bit closer at both these novel components of the Transformer architecture, starting with the positional embeddings.

Positional embeddings

A positional embedding is a dense vector that encodes the position of a word within a sentence: the ith positional embedding is simply added to the word embedding of the ith word in the sentence. These positional embeddings can be learned by the model, but in the paper the authors preferred to use fixed positional embeddings, defined using the sine and cosine functions of different frequencies. The positional embedding matrix **P** is defined in Equation 16-2 and represented at the bottom of Figure 16-9 (transposed), where $P_{p,i}$ is the ith component of the embedding for the word located at the pth position in the sentence.

Equation 16-2. Sine/cosine positional embeddings

$$P_{p,2i} = \sin\left(p/10000^{2i/d}\right)$$
$$P_{p,2i+1} = \cos\left(p/10000^{2i/d}\right)$$

Figure 16-9. Sine/cosine positional embedding matrix (transposed, top) with a focus on two values of i (bottom)

This solution gives the same performance as learned positional embeddings do, but it can extend to arbitrarily long sentences, which is why it's favored. After the positional embeddings are added to the word embeddings, the rest of the model has access to the absolute position of each word in the sentence because there is a unique positional embedding for each position (e.g., the positional embedding for the word located at the 22nd position in a sentence is represented by the vertical dashed line at the bottom left of Figure 16-9, and you can see that it is unique to that position). Moreover, the choice of oscillating functions (sine and cosine) makes it possible for the model to learn relative positions as well. For example, words located 38 words apart (e.g., at positions $p = 22$ and $p = 60$) always have the same positional embedding values in the embedding dimensions $i = 100$ and $i = 101$, as you can see in Figure 16-9. This explains why we need both the sine and the cosine for each frequency: if we only used the sine (the blue wave at $i = 100$), the model would not be able to distinguish positions $p = 25$ and $p = 35$ (marked by a cross).

There is no `PositionalEmbedding` layer in TensorFlow, but it is easy to create one. For efficiency reasons, we precompute the positional embedding matrix in the constructor (so we need to know the maximum sentence length, `max_steps`, and the number of dimensions for each word representation, `max_dims`). Then the `call()` method crops this embedding matrix to the size of the inputs, and it adds it to the inputs. Since we added an extra first dimension of size 1 when creating the positional embedding matrix, the rules of broadcasting will ensure that the matrix gets added to every sentence in the inputs:

```
class PositionalEncoding(keras.layers.Layer):
    def __init__(self, max_steps, max_dims, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)
        if max_dims % 2 == 1: max_dims += 1 # max_dims must be even
        p, i = np.meshgrid(np.arange(max_steps), np.arange(max_dims // 2))
        pos_emb = np.empty((1, max_steps, max_dims))
        pos_emb[0, :, ::2] = np.sin(p / 10000**(2 * i / max_dims)).T
        pos_emb[0, :, 1::2] = np.cos(p / 10000**(2 * i / max_dims)).T
        self.positional_embedding = tf.constant(pos_emb.astype(self.dtype))
    def call(self, inputs):
        shape = tf.shape(inputs)
        return inputs + self.positional_embedding[:, :shape[-2], :shape[-1]]
```

Then we can create the first layers of the Transformer:

```
embed_size = 512; max_steps = 500; vocab_size = 10000
encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)
positional_encoding = PositionalEncoding(max_steps, max_dims=embed_size)
encoder_in = positional_encoding(encoder_embeddings)
decoder_in = positional_encoding(decoder_embeddings)
```

Now let's look deeper into the heart of the Transformer model: the Multi-Head Attention layer.

Multi-Head Attention

To understand how a Multi-Head Attention layer works, we must first understand the *Scaled Dot-Product Attention* layer, which it is based on. Let's suppose the encoder analyzed the input sentence "They played chess," and it managed to understand that the word "They" is the subject and the word "played" is the verb, so it encoded this information in the representations of these words. Now suppose the decoder has already translated the subject, and it thinks that it should translate the verb next. For this, it needs to fetch the verb from the input sentence. This is analog to a dictionary lookup: it's as if the encoder created a dictionary {"subject": "They", "verb": "played", …} and the decoder wanted to look up the value that corresponds to the key "verb." However, the model does not have discrete tokens to represent the keys (like "subject" or "verb"); it has vectorized representations of these concepts (which it learned during training), so the key it will use for the lookup (called the *query*) will not perfectly match any key in the dictionary. The solution is to compute a similarity measure between the query and each key in the dictionary, and then use the softmax function to convert these similarity scores to weights that add up to 1. If the key that represents the verb is by far the most similar to the query, then that key's weight will be close to 1. Then the model can compute a weighted sum of the corresponding values, so if the weight of the "verb" key is close to 1, then the weighted sum will be very close to the representation of the word "played." In short, you can think of this whole process as a differentiable dictionary lookup. The similarity measure used by the Transformer is just the dot product, like in Luong attention. In fact, the equation is the same as for Luong attention, except for a scaling factor. The equation is shown in Equation 16-3, in a vectorized form.

Equation 16-3. Scaled Dot-Product Attention

$$\text{Attention}\,(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\top}}{\sqrt{d_{keys}}}\right)\mathbf{V}$$

In this equation:

- **Q** is a matrix containing one row per query. Its shape is $[n_{\text{queries}}, d_{\text{keys}}]$, where n_{queries} is the number of queries and d_{keys} is the number of dimensions of each query and each key.
- **K** is a matrix containing one row per key. Its shape is $[n_{\text{keys}}, d_{\text{keys}}]$, where n_{keys} is the number of keys and values.

- **V** is a matrix containing one row per value. Its shape is $[n_{\text{keys}}, d_{\text{values}}]$, where d_{values} is the number of each value.
- The shape of $\mathbf{Q}\ \mathbf{K}^{\intercal}$ is $[n_{\text{queries}}, n_{\text{keys}}]$: it contains one similarity score for each query/key pair. The output of the softmax function has the same shape, but all rows sum up to 1. The final output has a shape of $[n_{\text{queries}}, d_{\text{values}}]$: there is one row per query, where each row represents the query result (a weighted sum of the values).
- The scaling factor scales down the similarity scores to avoid saturating the softmax function, which would lead to tiny gradients.
- It is possible to mask out some key/value pairs by adding a very large negative value to the corresponding similarity scores, just before computing the softmax. This is useful in the Masked Multi-Head Attention layer.

In the encoder, this equation is applied to every input sentence in the batch, with **Q**, **K**, and **V** all equal to the list of words in the input sentence (so each word in the sentence will be compared to every word in the same sentence, including itself). Similarly, in the decoder's masked attention layer, the equation will be applied to every target sentence in the batch, with **Q**, **K**, and **V** all equal to the list of words in the target sentence, but this time using a mask to prevent any word from comparing itself to words located after it (at inference time the decoder will only have access to the words it already output, not to future words, so during training we must mask out future output tokens). In the upper attention layer of the decoder, the keys **K** and values **V** are simply the list of word encodings produced by the encoder, and the queries **Q** are the list of word encodings produced by the decoder.

The `keras.layers.Attention` layer implements Scaled Dot-Product Attention, efficiently applying Equation 16-3 to multiple sentences in a batch. Its inputs are just like **Q**, **K**, and **V**, except with an extra batch dimension (the first dimension).

In TensorFlow, if `A` and `B` are tensors with more than two dimensions—say, of shape [2, 3, 4, 5] and [2, 3, 5, 6] respectively—then `tf.matmul(A, B)` will treat these tensors as 2×3 arrays where each cell contains a matrix, and it will multiply the corresponding matrices: the matrix at the ith row and j^{th} column in `A` will be multiplied by the matrix at the ith row and j^{th} column in `B`. Since the product of a 4×5 matrix with a 5×6 matrix is a 4×6 matrix, `tf.matmul(A, B)` will return an array of shape [2, 3, 4, 6].

If we ignore the skip connections, the layer normalization layers, the Feed Forward blocks, and the fact that this is Scaled Dot-Product Attention, not exactly Multi-Head Attention, then the rest of the Transformer model can be implemented like this:

```
Z = encoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True)([Z, Z])

encoder_outputs = Z
Z = decoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True, causal=True)([Z, Z])
    Z = keras.layers.Attention(use_scale=True)([Z, encoder_outputs])

outputs = keras.layers.TimeDistributed(
    keras.layers.Dense(vocab_size, activation="softmax"))(Z)
```

The `use_scale=True` argument creates an additional parameter that lets the layer learn how to properly downscale the similarity scores. This is a bit different from the Transformer model, which always downscales the similarity scores by the same factor ($\sqrt{d_{\text{keys}}}$). The `causal=True` argument when creating the second attention layer ensures that each output token only attends to previous output tokens, not future ones.

Now it's time to look at the final piece of the puzzle: what is a Multi-Head Attention layer? Its architecture is shown in Figure 16-10.

Figure 16-10. Multi-Head Attention layer architecture[23]

As you can see, it is just a bunch of Scaled Dot-Product Attention layers, each preceded by a linear transformation of the values, keys, and queries (i.e., a time-distributed `Dense` layer with no activation function). All the outputs are simply concatenated, and they go through a final linear transformation (again, time-distributed). But why? What is the intuition behind this architecture? Well, consider the word "played" we discussed earlier (in the sentence "They played chess"). The encoder was smart enough to encode the fact that it is a verb. But the word representation also includes its position in the text, thanks to the positional encodings, and it probably includes many other features that are useful for its translation, such as the fact that it is in the past tense. In short, the word representation encodes many different characteristics of the word. If we just used a single Scaled Dot-Product Attention layer, we would only be able to query all of these characteristics in one shot. This is why the Multi-Head Attention layer applies multiple different linear transformations of the values, keys, and queries: this allows the model to apply many different projections of the word representation into different subspaces, each focusing on a subset of the word's characteristics. Perhaps one of the linear layers will project the word representation into a subspace where all that remains is the information that the word is a

23 This is the right part of figure 2 from the paper, reproduced with the kind authorization of the authors.

verb, another linear layer will extract just the fact that it is past tense, and so on. Then the Scaled Dot-Product Attention layers implement the lookup phase, and finally we concatenate all the results and project them back to the original space.

At the time of this writing, there is no `Transformer` class or `MultiHeadAttention` class available for TensorFlow 2. However, you can check out TensorFlow's great tutorial for building a Transformer model for language understanding. Moreover, the TF Hub team is currently porting several Transformer-based modules to TensorFlow 2, and they should be available soon. In the meantime, I hope I have demonstrated that it is not that hard to implement a Transformer yourself, and it is certainly a great exercise!

Recent Innovations in Language Models

The year 2018 has been called the "ImageNet moment for NLP": progress was astounding, with larger and larger LSTM and Transformer-based architectures trained on immense datasets. I highly recommend you check out the following papers, all published in 2018:

- The ELMo paper[24] by Matthew Peters introduced *Embeddings from Language Models* (ELMo): these are contextualized word embeddings learned from the internal states of a deep bidirectional language model. For example, the word "queen" will not have the same embedding in "Queen of the United Kingdom" and in "queen bee."
- The ULMFiT paper[25] by Jeremy Howard and Sebastian Ruder demonstrated the effectiveness of unsupervised pretraining for NLP tasks: the authors trained an LSTM language model using self-supervised learning (i.e., generating the labels automatically from the data) on a huge text corpus, then they fine-tuned it on various tasks. Their model outperformed the state of the art on six text classification tasks by a large margin (reducing the error rate by 18–24% in most cases). Moreover, they showed that by fine-tuning the pretrained model on just 100 labeled examples, they could achieve the same performance as a model trained from scratch on 10,000 examples.
- The GPT paper[26] by Alec Radford and other OpenAI researchers also demonstrated the effectiveness of unsupervised pretraining, but this time using a

24 Matthew Peters et al., "Deep Contextualized Word Representations," *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2018): 2227–2237.

25 Jeremy Howard and Sebastian Ruder, "Universal Language Model Fine-Tuning for Text Classification," *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics* 1 (2018): 328–339.

26 Alec Radford et al., "Improving Language Understanding by Generative Pre-Training" (2018).

Transformer-like architecture. The authors pretrained a large but fairly simple architecture composed of a stack of 12 Transformer modules (using only Masked Multi-Head Attention layers) on a large dataset, once again trained using self-supervised learning. Then they fine-tuned it on various language tasks, using only minor adaptations for each task. The tasks were quite diverse: they included text classification, *entailment* (whether sentence A entails sentence B),[27] similarity (e.g., "Nice weather today" is very similar to "It is sunny"), and question answering (given a few paragraphs of text giving some context, the model must answer some multiple-choice questions). Just a few months later, in February 2019, Alec Radford, Jeffrey Wu, and other OpenAI researchers published the GPT-2 paper,[28] which proposed a very similar architecture, but larger still (with over 1.5 billion parameters!) and they showed that it could achieve good performance on many tasks without any fine-tuning. This is called *zero-shot learning* (ZSL). A smaller version of the GPT-2 model (with "just" 117 million parameters) is available at *https://github.com/openai/gpt-2*, along with its pretrained weights.

- The BERT paper[29] by Jacob Devlin and other Google researchers also demonstrates the effectiveness of self-supervised pretraining on a large corpus, using a similar architecture to GPT but non-masked Multi-Head Attention layers (like in the Transformer's encoder). This means that the model is naturally bidirectional; hence the B in BERT (*Bidirectional Encoder Representations from Transformers*). Most importantly, the authors proposed two pretraining tasks that explain most of the model's strength:

  Masked language model (MLM)

  Each word in a sentence has a 15% probability of being masked, and the model is trained to predict the masked words. For example, if the original sentence is "She had fun at the birthday party," then the model may be given the sentence "She <mask> fun at the <mask> party" and it must predict the words "had" and "birthday" (the other outputs will be ignored). To be more precise, each selected word has an 80% chance of being masked, a 10% chance of being replaced by a random word (to reduce the discrepancy between pretraining and fine-tuning, since the model will not see <mask> tokens during fine-tuning), and a 10% chance of being left alone (to bias the model toward the correct answer).

27 For example, the sentence "Jane had a lot of fun at her friend's birthday party" entails "Jane enjoyed the party," but it is contradicted by "Everyone hated the party" and it is unrelated to "The Earth is flat."

28 Alec Radford et al., "Language Models Are Unsupervised Multitask Learners" (2019).

29 Jacob Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2019).

Next sentence prediction (NSP)

The model is trained to predict whether two sentences are consecutive or not. For example, it should predict that "The dog sleeps" and "It snores loudly" are consecutive sentences, while "The dog sleeps" and "The Earth orbits the Sun" are not consecutive. This is a challenging task, and it significantly improves the performance of the model when it is fine-tuned on tasks such as question answering or entailment.

As you can see, the main innovations in 2018 and 2019 have been better subword tokenization, shifting from LSTMs to Transformers, and pretraining universal language models using self-supervised learning, then fine-tuning them with very few architectural changes (or none at all). Things are moving fast; no one can say what architectures will prevail next year. Today, it's clearly Transformers, but tomorrow it might be CNNs (e.g., check out the 2018 paper[30] by Maha Elbayad et al., where the researchers use masked 2D convolutional layers for sequence-to-sequence tasks). Or it might even be RNNs, if they make a surprise comeback (e.g., check out the 2018 paper[31] by Shuai Li et al. that shows that by making neurons independent of each other in a given RNN layer, it is possible to train much deeper RNNs capable of learning much longer sequences).

In the next chapter we will discuss how to learn deep representations in an unsupervised way using autoencoders, and we will use generative adversarial networks (GANs) to produce images and more!

Exercises

1. What are the pros and cons of using a stateful RNN versus a stateless RNN?
2. Why do people use Encoder–Decoder RNNs rather than plain sequence-to-sequence RNNs for automatic translation?
3. How can you deal with variable-length input sequences? What about variable-length output sequences?
4. What is beam search and why would you use it? What tool can you use to implement it?
5. What is an attention mechanism? How does it help?

30 Maha Elbayad et al., "Pervasive Attention: 2D Convolutional Neural Networks for Sequence-to-Sequence Prediction," arXiv preprint arXiv:1808.03867 (2018).

31 Shuai Li et al., "Independently Recurrent Neural Network (IndRNN): Building a Longer and Deeper RNN," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018): 5457–5466.

6. What is the most important layer in the Transformer architecture? What is its purpose?
7. When would you need to use sampled softmax?
8. *Embedded Reber grammars* were used by Hochreiter and Schmidhuber in their paper about LSTMs. They are artificial grammars that produce strings such as "BPBTSXXVPSEPE." Check out Jenny Orr's nice introduction to this topic. Choose a particular embedded Reber grammar (such as the one represented on Jenny Orr's page), then train an RNN to identify whether a string respects that grammar or not. You will first need to write a function capable of generating a training batch containing about 50% strings that respect the grammar, and 50% that don't.
9. Train an Encoder–Decoder model that can convert a date string from one format to another (e.g., from "April 22, 2019" to "2019-04-22").
10. Go through TensorFlow's Neural Machine Translation with Attention tutorial.
11. Use one of the recent language models (e.g., BERT) to generate more convincing Shakespearean text.

Solutions to these exercises are available in Appendix A.

CHAPTER 17

Representation Learning and Generative Learning Using Autoencoders and GANs

Autoencoders are artificial neural networks capable of learning dense representations of the input data, called *latent representations* or *codings*, without any supervision (i.e., the training set is unlabeled). These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction (see Chapter 8), especially for visualization purposes. Autoencoders also act as feature detectors, and they can be used for unsupervised pretraining of deep neural networks (as we discussed in Chapter 11). Lastly, some autoencoders are *generative models*: they are capable of randomly generating new data that looks very similar to the training data. For example, you could train an autoencoder on pictures of faces, and it would then be able to generate new faces. However, the generated images are usually fuzzy and not entirely realistic.

In contrast, faces generated by generative adversarial networks (GANs) are now so convincing that it is hard to believe that the people they represent do not exist. You can judge so for yourself by visiting *https://thispersondoesnotexist.com/*, a website that shows faces generated by a recent GAN architecture called *StyleGAN* (you can also check out *https://thisrentaldoesnotexist.com/* to see some generated Airbnb bedrooms). GANs are now widely used for super resolution (increasing the resolution of an image), colorization, powerful image editing (e.g., replacing photo bombers with realistic background), turning a simple sketch into a photorealistic image, predicting the next frames in a video, augmenting a dataset (to train other models), generating other types of data (such as text, audio, and time series), identifying the weaknesses in other models and strengthening them, and more.

Autoencoders and GANs are both unsupervised, they both learn dense representations, they can both be used as generative models, and they have many similar applications. However, they work very differently:

- Autoencoders simply learn to copy their inputs to their outputs. This may sound like a trivial task, but we will see that constraining the network in various ways can make it rather difficult. For example, you can limit the size of the latent representations, or you can add noise to the inputs and train the network to recover the original inputs. These constraints prevent the autoencoder from trivially copying the inputs directly to the outputs, which forces it to learn efficient ways of representing the data. In short, the codings are byproducts of the autoencoder learning the identity function under some constraints.
- GANs are composed of two neural networks: a *generator* that tries to generate data that looks similar to the training data, and a *discriminator* that tries to tell real data from fake data. This architecture is very original in Deep Learning in that the generator and the discriminator compete against each other during training: the generator is often compared to a criminal trying to make realistic counterfeit money, while the discriminator is like the police investigator trying to tell real money from fake. *Adversarial training* (training competing neural networks) is widely considered as one of the most important ideas in recent years. In 2016, Yann LeCun even said that it was "the most interesting idea in the last 10 years in Machine Learning."

In this chapter we will start by exploring in more depth how autoencoders work and how to use them for dimensionality reduction, feature extraction, unsupervised pretraining, or as generative models. This will naturally lead us to GANs. We will start by building a simple GAN to generate fake images, but we will see that training is often quite difficult. We will discuss the main difficulties you will encounter with adversarial training, as well as some of the main techniques to work around these difficulties. Let's start with autoencoders!

Efficient Data Representations

Which of the following number sequences do you find the easiest to memorize?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14

At first glance, it would seem that the first sequence should be easier, since it is much shorter. However, if you look carefully at the second sequence, you will notice that it is just the list of even numbers from 50 down to 14. Once you notice this pattern, the second sequence becomes much easier to memorize than the first because you only need to remember the pattern (i.e., decreasing even numbers) and the starting and ending numbers (i.e., 50 and 14). Note that if you could quickly and easily memorize very long sequences, you would not care much about the existence of a pattern in the second sequence. You would just learn every number by heart, and that would be that. The fact that it is hard to memorize long sequences is what makes it useful to recognize patterns, and hopefully this clarifies why constraining an autoencoder during training pushes it to discover and exploit patterns in the data.

The relationship between memory, perception, and pattern matching was famously studied by William Chase and Herbert Simon in the early 1970s.[1] They observed that expert chess players were able to memorize the positions of all the pieces in a game by looking at the board for just five seconds, a task that most people would find impossible. However, this was only the case when the pieces were placed in realistic positions (from actual games), not when the pieces were placed randomly. Chess experts don't have a much better memory than you and I; they just see chess patterns more easily, thanks to their experience with the game. Noticing patterns helps them store information efficiently.

Just like the chess players in this memory experiment, an autoencoder looks at the inputs, converts them to an efficient latent representation, and then spits out something that (hopefully) looks very close to the inputs. An autoencoder is always composed of two parts: an *encoder* (or *recognition network*) that converts the inputs to a latent representation, followed by a *decoder* (or *generative network*) that converts the internal representation to the outputs (see Figure 17-1).

1 William G. Chase and Herbert A. Simon, "Perception in Chess," *Cognitive Psychology* 4, no. 1 (1973): 55–81.

Figure 17-1. The chess memory experiment (left) and a simple autoencoder (right)

As you can see, an autoencoder typically has the same architecture as a Multi-Layer Perceptron (MLP; see Chapter 10), except that the number of neurons in the output layer must be equal to the number of inputs. In this example, there is just one hidden layer composed of two neurons (the encoder), and one output layer composed of three neurons (the decoder). The outputs are often called the *reconstructions* because the autoencoder tries to reconstruct the inputs, and the cost function contains a *reconstruction loss* that penalizes the model when the reconstructions are different from the inputs.

Because the internal representation has a lower dimensionality than the input data (it is 2D instead of 3D), the autoencoder is said to be *undercomplete*. An undercomplete autoencoder cannot trivially copy its inputs to the codings, yet it must find a way to output a copy of its inputs. It is forced to learn the most important features in the input data (and drop the unimportant ones).

Let's see how to implement a very simple undercomplete autoencoder for dimensionality reduction.

Performing PCA with an Undercomplete Linear Autoencoder

If the autoencoder uses only linear activations and the cost function is the mean squared error (MSE), then it ends up performing Principal Component Analysis (PCA; see Chapter 8).

The following code builds a simple linear autoencoder to perform PCA on a 3D dataset, projecting it to 2D:

```
from tensorflow import keras

encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])])
decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])])
autoencoder = keras.models.Sequential([encoder, decoder])

autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=0.1))
```

This code is really not very different from all the MLPs we built in past chapters, but there are a few things to note:

- We organized the autoencoder into two subcomponents: the encoder and the decoder. Both are regular `Sequential` models with a single `Dense` layer each, and the autoencoder is a `Sequential` model containing the encoder followed by the decoder (remember that a model can be used as a layer in another model).
- The autoencoder's number of outputs is equal to the number of inputs (i.e., 3).
- To perform simple PCA, we do not use any activation function (i.e., all neurons are linear), and the cost function is the MSE. We will see more complex autoencoders shortly.

Now let's train the model on a simple generated 3D dataset and use it to encode that same dataset (i.e., project it to 2D):

```
history = autoencoder.fit(X_train, X_train, epochs=20)
codings = encoder.predict(X_train)
```

Note that the same dataset, `X_train`, is used as both the inputs and the targets. Figure 17-2 shows the original 3D dataset (on the left) and the output of the autoencoder's hidden layer (i.e., the coding layer, on the right). As you can see, the autoencoder found the best 2D plane to project the data onto, preserving as much variance in the data as it could (just like PCA).

Figure 17-2. PCA performed by an undercomplete linear autoencoder

You can think of autoencoders as a form of self-supervised learning (i.e., using a supervised learning technique with automatically generated labels, in this case simply equal to the inputs).

Stacked Autoencoders

Just like other neural networks we have discussed, autoencoders can have multiple hidden layers. In this case they are called *stacked autoencoders* (or *deep autoencoders*). Adding more layers helps the autoencoder learn more complex codings. That said, one must be careful not to make the autoencoder too powerful. Imagine an encoder so powerful that it just learns to map each input to a single arbitrary number (and the decoder learns the reverse mapping). Obviously such an autoencoder will reconstruct the training data perfectly, but it will not have learned any useful data representation in the process (and it is unlikely to generalize well to new instances).

The architecture of a stacked autoencoder is typically symmetrical with regard to the central hidden layer (the coding layer). To put it simply, it looks like a sandwich. For example, an autoencoder for MNIST (introduced in Chapter 3) may have 784 inputs, followed by a hidden layer with 100 neurons, then a central hidden layer of 30 neurons, then another hidden layer with 100 neurons, and an output layer with 784 neurons. This stacked autoencoder is represented in Figure 17-3.

Figure 17-3. Stacked autoencoder

Implementing a Stacked Autoencoder Using Keras

You can implement a stacked autoencoder very much like a regular deep MLP. In particular, the same techniques we used in Chapter 11 for training deep nets can be applied. For example, the following code builds a stacked autoencoder for Fashion

MNIST (loaded and normalized as in Chapter 10), using the SELU activation function:

```
stacked_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu"),
])
stacked_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
stacked_ae = keras.models.Sequential([stacked_encoder, stacked_decoder])
stacked_ae.compile(loss="binary_crossentropy",
                   optimizer=keras.optimizers.SGD(lr=1.5))
history = stacked_ae.fit(X_train, X_train, epochs=10,
                         validation_data=[X_valid, X_valid])
```

Let's go through this code:

- Just like earlier, we split the autoencoder model into two submodels: the encoder and the decoder.
- The encoder takes 28 × 28–pixel grayscale images, flattens them so that each image is represented as a vector of size 784, then processes these vectors through two `Dense` layers of diminishing sizes (100 units then 30 units), both using the SELU activation function (you may want to add LeCun normal initialization as well, but the network is not very deep so it won't make a big difference). For each input image, the encoder outputs a vector of size 30.
- The decoder takes codings of size 30 (output by the encoder) and processes them through two `Dense` layers of increasing sizes (100 units then 784 units), and it reshapes the final vectors into 28 × 28 arrays so the decoder's outputs have the same shape as the encoder's inputs.
- When compiling the stacked autoencoder, we use the binary cross-entropy loss instead of the mean squared error. We are treating the reconstruction task as a multilabel binary classification problem: each pixel intensity represents the probability that the pixel should be black. Framing it this way (rather than as a regression problem) tends to make the model converge faster.[2]
- Finally, we train the model using `X_train` as both the inputs and the targets (and similarly, we use `X_valid` as both the validation inputs and targets).

2 You might be tempted to use the accuracy metric, but it would not work properly, since this metric expects the labels to be either 0 or 1 for each pixel. You can easily work around this problem by creating a custom metric that computes the accuracy after rounding the targets and predictions to 0 or 1.

Visualizing the Reconstructions

One way to ensure that an autoencoder is properly trained is to compare the inputs and the outputs: the differences should not be too significant. Let's plot a few images from the validation set, as well as their reconstructions:

```
def plot_image(image):
    plt.imshow(image, cmap="binary")
    plt.axis("off")

def show_reconstructions(model, n_images=5):
    reconstructions = model.predict(X_valid[:n_images])
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plot_image(X_valid[image_index])
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plot_image(reconstructions[image_index])

show_reconstructions(stacked_ae)
```

Figure 17-4 shows the resulting images.

Figure 17-4. Original images (top) and their reconstructions (bottom)

The reconstructions are recognizable, but a bit too lossy. We may need to train the model for longer, or make the encoder and decoder deeper, or make the codings larger. But if we make the network too powerful, it will manage to make perfect reconstructions without having learned any useful patterns in the data. For now, let's go with this model.

Visualizing the Fashion MNIST Dataset

Now that we have trained a stacked autoencoder, we can use it to reduce the dataset's dimensionality. For visualization, this does not give great results compared to other dimensionality reduction algorithms (such as those we discussed in Chapter 8), but one big advantage of autoencoders is that they can handle large datasets, with many instances and many features. So one strategy is to use an autoencoder to reduce the dimensionality down to a reasonable level, then use another dimensionality

reduction algorithm for visualization. Let's use this strategy to visualize Fashion MNIST. First, we use the encoder from our stacked autoencoder to reduce the dimensionality down to 30, then we use Scikit-Learn's implementation of the t-SNE algorithm to reduce the dimensionality down to 2 for visualization:

```
from sklearn.manifold import TSNE

X_valid_compressed = stacked_encoder.predict(X_valid)
tsne = TSNE()
X_valid_2D = tsne.fit_transform(X_valid_compressed)
```

Now we can plot the dataset:

```
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap="tab10")
```

Figure 17-5 shows the resulting scatterplot (beautified a bit by displaying some of the images). The t-SNE algorithm identified several clusters which match the classes reasonably well (each class is represented with a different color).

Figure 17-5. Fashion MNIST visualization using an autoencoder followed by t-SNE

So, autoencoders can be used for dimensionality reduction. Another application is for unsupervised pretraining.

Unsupervised Pretraining Using Stacked Autoencoders

As we discussed in Chapter 11, if you are tackling a complex supervised task but you do not have a lot of labeled training data, one solution is to find a neural network that performs a similar task and reuse its lower layers. This makes it possible to train a high-performance model using little training data because your neural network won't have to learn all the low-level features; it will just reuse the feature detectors learned by the existing network.

Similarly, if you have a large dataset but most of it is unlabeled, you can first train a stacked autoencoder using all the data, then reuse the lower layers to create a neural network for your actual task and train it using the labeled data. For example, Figure 17-6 shows how to use a stacked autoencoder to perform unsupervised pretraining for a classification neural network. When training the classifier, if you really don't have much labeled training data, you may want to freeze the pretrained layers (at least the lower ones).

Figure 17-6. Unsupervised pretraining using autoencoders

Having plenty of unlabeled data and little labeled data is common. Building a large unlabeled dataset is often cheap (e.g., a simple script can download millions of images off the internet), but labeling those images (e.g., classifying them as cute or not) can usually be done reliably only by humans. Labeling instances is time-consuming and costly, so it's normal to have only a few thousand human-labeled instances.

There is nothing special about the implementation: just train an autoencoder using all the training data (labeled plus unlabeled), then reuse its encoder layers to create a new neural network (see the exercises at the end of this chapter for an example).

Next, let's look at a few techniques for training stacked autoencoders.

Tying Weights

When an autoencoder is neatly symmetrical, like the one we just built, a common technique is to *tie* the weights of the decoder layers to the weights of the encoder layers. This halves the number of weights in the model, speeding up training and limiting the risk of overfitting. Specifically, if the autoencoder has a total of N layers (not counting the input layer), and $\mathbf{W}_L$ represents the connection weights of the Lth layer (e.g., layer 1 is the first hidden layer, layer $N/2$ is the coding layer, and layer N is the output layer), then the decoder layer weights can be defined simply as: $\mathbf{W}_{N-L+1} = \mathbf{W}_L^{\intercal}$ (with $L = 1, 2, \ldots, N/2$).

To tie weights between layers using Keras, let's define a custom layer:

```
class DenseTranspose(keras.layers.Layer):
    def __init__(self, dense, activation=None, **kwargs):
        self.dense = dense
        self.activation = keras.activations.get(activation)
        super().__init__(**kwargs)
    def build(self, batch_input_shape):
        self.biases = self.add_weight(name="bias", initializer="zeros",
                                      shape=[self.dense.input_shape[-1]])
        super().build(batch_input_shape)
    def call(self, inputs):
        z = tf.matmul(inputs, self.dense.weights[0], transpose_b=True)
        return self.activation(z + self.biases)
```

This custom layer acts like a regular `Dense` layer, but it uses another `Dense` layer's weights, transposed (setting `transpose_b=True` is equivalent to transposing the second argument, but it's more efficient as it performs the transposition on the fly within the `matmul()` operation). However, it uses its own bias vector. Next, we can build a new stacked autoencoder, much like the previous one, but with the decoder's `Dense` layers tied to the encoder's `Dense` layers:

```
dense_1 = keras.layers.Dense(100, activation="selu")
dense_2 = keras.layers.Dense(30, activation="selu")

tied_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    dense_1,
    dense_2
])
```

```
tied_decoder = keras.models.Sequential([
    DenseTranspose(dense_2, activation="selu"),
    DenseTranspose(dense_1, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

tied_ae = keras.models.Sequential([tied_encoder, tied_decoder])
```

This model achieves a very slightly lower reconstruction error than the previous model, with almost half the number of parameters.

Training One Autoencoder at a Time

Rather than training the whole stacked autoencoder in one go like we just did, it is possible to train one shallow autoencoder at a time, then stack all of them into a single stacked autoencoder (hence the name), as shown in Figure 17-7. This technique is not used as much these days, but you may still run into papers that talk about "greedy layerwise training," so it's good to know what it means.

Figure 17-7. Training one autoencoder at a time

During the first phase of training, the first autoencoder learns to reconstruct the inputs. Then we encode the whole training set using this first autoencoder, and this gives us a new (compressed) training set. We then train a second autoencoder on this new dataset. This is the second phase of training. Finally, we build a big sandwich using all these autoencoders, as shown in Figure 17-7 (i.e., we first stack the hidden layers of each autoencoder, then the output layers in reverse order). This gives us the final stacked autoencoder (see the "Training One Autoencoder at a Time" section in the notebook for an implementation). We could easily train more autoencoders this way, building a very deep stacked autoencoder.

As we discussed earlier, one of the triggers of the current tsunami of interest in Deep Learning was the discovery in 2006 by Geoffrey Hinton et al. that deep neural networks can be pretrained in an unsupervised fashion, using this greedy layerwise approach. They used restricted Boltzmann machines (RBMs; see Appendix E) for this purpose, but in 2007 Yoshua Bengio et al. showed[3] that autoencoders worked just as well. For several years this was the only efficient way to train deep nets, until many of the techniques introduced in Chapter 11 made it possible to just train a deep net in one shot.

Autoencoders are not limited to dense networks: you can also build convolutional autoencoders, or even recurrent autoencoders. Let's look at these now.

Convolutional Autoencoders

If you are dealing with images, then the autoencoders we have seen so far will not work well (unless the images are very small): as we saw in Chapter 14, convolutional neural networks are far better suited than dense networks to work with images. So if you want to build an autoencoder for images (e.g., for unsupervised pretraining or dimensionality reduction), you will need to build a *convolutional autoencoder*.[4] The encoder is a regular CNN composed of convolutional layers and pooling layers. It typically reduces the spatial dimensionality of the inputs (i.e., height and width) while increasing the depth (i.e., the number of feature maps). The decoder must do the reverse (upscale the image and reduce its depth back to the original dimensions), and for this you can use transpose convolutional layers (alternatively, you could combine upsampling layers with convolutional layers). Here is a simple convolutional autoencoder for Fashion MNIST:

```
conv_encoder = keras.models.Sequential([
    keras.layers.Reshape([28, 28, 1], input_shape=[28, 28]),
    keras.layers.Conv2D(16, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(64, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2)
])
conv_decoder = keras.models.Sequential([
    keras.layers.Conv2DTranspose(32, kernel_size=3, strides=2, padding="valid",
                                 activation="selu",
                                 input_shape=[3, 3, 64]),
```

3 Yoshua Bengio et al., "Greedy Layer-Wise Training of Deep Networks," *Proceedings of the 19th International Conference on Neural Information Processing Systems* (2006): 153–160.

4 Jonathan Masci et al., "Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction," *Proceedings of the 21st International Conference on Artificial Neural Networks* 1 (2011): 52–59.

```
    keras.layers.Conv2DTranspose(16, kernel_size=3, strides=2, padding="same",
                                 activation="selu"),
    keras.layers.Conv2DTranspose(1, kernel_size=3, strides=2, padding="same",
                                 activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
conv_ae = keras.models.Sequential([conv_encoder, conv_decoder])
```

Recurrent Autoencoders

If you want to build an autoencoder for sequences, such as time series or text (e.g., for unsupervised learning or dimensionality reduction), then recurrent neural networks (see Chapter 15) may be better suited than dense networks. Building a *recurrent autoencoder* is straightforward: the encoder is typically a sequence-to-vector RNN which compresses the input sequence down to a single vector. The decoder is a vector-to-sequence RNN that does the reverse:

```
recurrent_encoder = keras.models.Sequential([
    keras.layers.LSTM(100, return_sequences=True, input_shape=[None, 28]),
    keras.layers.LSTM(30)
])
recurrent_decoder = keras.models.Sequential([
    keras.layers.RepeatVector(28, input_shape=[30]),
    keras.layers.LSTM(100, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(28, activation="sigmoid"))
])
recurrent_ae = keras.models.Sequential([recurrent_encoder, recurrent_decoder])
```

This recurrent autoencoder can process sequences of any length, with 28 dimensions per time step. Conveniently, this means it can process Fashion MNIST images by treating each image as a sequence of rows: at each time step, the RNN will process a single row of 28 pixels. Obviously, you could use a recurrent autoencoder for any kind of sequence. Note that we use a `RepeatVector` layer as the first layer of the decoder, to ensure that its input vector gets fed to the decoder at each time step.

OK, let's step back for a second. So far we have seen various kinds of autoencoders (basic, stacked, convolutional, and recurrent), and we have looked at how to train them (either in one shot or layer by layer). We also looked at a couple applications: data visualization and unsupervised pretraining.

Up to now, in order to force the autoencoder to learn interesting features, we have limited the size of the coding layer, making it undercomplete. There are actually many other kinds of constraints that can be used, including ones that allow the coding layer to be just as large as the inputs, or even larger, resulting in an *overcomplete autoencoder*. Let's look at some of those approaches now.

Denoising Autoencoders

Another way to force the autoencoder to learn useful features is to add noise to its inputs, training it to recover the original, noise-free inputs. This idea has been around since the 1980s (e.g., it is mentioned in Yann LeCun's 1987 master's thesis). In a 2008 paper,[5] Pascal Vincent et al. showed that autoencoders could also be used for feature extraction. In a 2010 paper,[6] Vincent et al. introduced *stacked denoising autoencoders*.

The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched-off inputs, just like in dropout (introduced in Chapter 11). Figure 17-8 shows both options.

Figure 17-8. Denoising autoencoders, with Gaussian noise (left) or dropout (right)

The implementation is straightforward: it is a regular stacked autoencoder with an additional `Dropout` layer applied to the encoder's inputs (or you could use a `GaussianNoise` layer instead). Recall that the `Dropout` layer is only active during training (and so is the `GaussianNoise` layer):

5 Pascal Vincent et al., "Extracting and Composing Robust Features with Denoising Autoencoders," *Proceedings of the 25th International Conference on Machine Learning* (2008): 1096–1103.

6 Pascal Vincent et al., "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion," *Journal of Machine Learning Research* 11 (2010): 3371–3408.

```
dropout_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu")
])
dropout_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
dropout_ae = keras.models.Sequential([dropout_encoder, dropout_decoder])
```

Figure 17-9 shows a few noisy images (with half the pixels turned off), and the images reconstructed by the dropout-based denoising autoencoder. Notice how the autoencoder guesses details that are actually not in the input, such as the top of the white shirt (bottom row, fourth image). As you can see, not only can denoising autoencoders be used for data visualization or unsupervised pretraining, like the other autoencoders we've discussed so far, but they can also be used quite simply and efficiently to remove noise from images.

Figure 17-9. Noisy images (top) and their reconstructions (bottom)

Sparse Autoencoders

Another kind of constraint that often leads to good feature extraction is *sparsity*: by adding an appropriate term to the cost function, the autoencoder is pushed to reduce the number of active neurons in the coding layer. For example, it may be pushed to have on average only 5% significantly active neurons in the coding layer. This forces the autoencoder to represent each input as a combination of a small number of activations. As a result, each neuron in the coding layer typically ends up representing a useful feature (if you could speak only a few words per month, you would probably try to make them worth listening to).

A simple approach is to use the sigmoid activation function in the coding layer (to constrain the codings to values between 0 and 1), use a large coding layer (e.g., with

300 units), and add some ℓ_1 regularization to the coding layer's activations (the decoder is just a regular decoder):

```
sparse_l1_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid"),
    keras.layers.ActivityRegularization(l1=1e-3)
])
sparse_l1_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
sparse_l1_ae = keras.models.Sequential([sparse_l1_encoder, sparse_l1_decoder])
```

This `ActivityRegularization` layer just returns its inputs, but as a side effect it adds a training loss equal to the sum of absolute values of its inputs (this layer only has an effect during training). Equivalently, you could remove the `ActivityRegularization` layer and set `activity_regularizer=keras.regularizers.l1(1e-3)` in the previous layer. This penalty will encourage the neural network to produce codings close to 0, but since it will also be penalized if it does not reconstruct the inputs correctly, it will have to output at least a few nonzero values. Using the ℓ_1 norm rather than the ℓ_2 norm will push the neural network to preserve the most important codings while eliminating the ones that are not needed for the input image (rather than just reducing all codings).

Another approach, which often yields better results, is to measure the actual sparsity of the coding layer at each training iteration, and penalize the model when the measured sparsity differs from a target sparsity. We do so by computing the average activation of each neuron in the coding layer, over the whole training batch. The batch size must not be too small, or else the mean will not be accurate.

Once we have the mean activation per neuron, we want to penalize the neurons that are too active, or not active enough, by adding a *sparsity loss* to the cost function. For example, if we measure that a neuron has an average activation of 0.3, but the target sparsity is 0.1, it must be penalized to activate less. One approach could be simply adding the squared error $(0.3 - 0.1)^2$ to the cost function, but in practice a better approach is to use the Kullback–Leibler (KL) divergence (briefly discussed in Chapter 4), which has much stronger gradients than the mean squared error, as you can see in Figure 17-10.

Figure 17-10. Sparsity loss

Given two discrete probability distributions P and Q, the KL divergence between these distributions, noted $D_{\mathrm{KL}}(P \parallel Q)$, can be computed using Equation 17-1.

Equation 17-1. Kullback–Leibler divergence

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

In our case, we want to measure the divergence between the target probability p that a neuron in the coding layer will activate and the actual probability q (i.e., the mean activation over the training batch). So the KL divergence simplifies to Equation 17-2.

Equation 17-2. KL divergence between the target sparsity p and the actual sparsity q

$$D_{\mathrm{KL}}(p \parallel q) = p \log \frac{p}{q} + (1 - p) \log \frac{1 - p}{1 - q}$$

Once we have computed the sparsity loss for each neuron in the coding layer, we sum up these losses and add the result to the cost function. In order to control the relative importance of the sparsity loss and the reconstruction loss, we can multiply the sparsity loss by a sparsity weight hyperparameter. If this weight is too high, the model will stick closely to the target sparsity, but it may not reconstruct the inputs properly, making the model useless. Conversely, if it is too low, the model will mostly ignore the sparsity objective and will not learn any interesting features.

We now have all we need to implement a sparse autoencoder based on the KL divergence. First, let's create a custom regularizer to apply KL divergence regularization:

```
K = keras.backend
kl_divergence = keras.losses.kullback_leibler_divergence

class KLDivergenceRegularizer(keras.regularizers.Regularizer):
    def __init__(self, weight, target=0.1):
        self.weight = weight
        self.target = target
    def __call__(self, inputs):
        mean_activities = K.mean(inputs, axis=0)
        return self.weight * (
            kl_divergence(self.target, mean_activities) +
            kl_divergence(1. - self.target, 1. - mean_activities))
```

Now we can build the sparse autoencoder, using the `KLDivergenceRegularizer` for the coding layer's activations:

```
kld_reg = KLDivergenceRegularizer(weight=0.05, target=0.1)
sparse_kl_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid", activity_regularizer=kld_reg)
])
sparse_kl_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
sparse_kl_ae = keras.models.Sequential([sparse_kl_encoder, sparse_kl_decoder])
```

After training this sparse autoencoder on Fashion MNIST, the activations of the neurons in the coding layer are mostly close to 0 (about 70% of all activations are lower than 0.1), and all neurons have a mean activation around 0.1 (about 90% of all neurons have a mean activation between 0.1 and 0.2), as shown in Figure 17-11.

Figure 17-11. Distribution of all the activations in the coding layer (left) and distribution of the mean activation per neuron (right)

Variational Autoencoders

Another important category of autoencoders was introduced in 2013 by Diederik Kingma and Max Welling and quickly became one of the most popular types of autoencoders: *variational autoencoders*.[7]

They are quite different from all the autoencoders we have discussed so far, in these particular ways:

- They are *probabilistic autoencoders*, meaning that their outputs are partly determined by chance, even after training (as opposed to denoising autoencoders, which use randomness only during training).
- Most importantly, they are *generative autoencoders*, meaning that they can generate new instances that look like they were sampled from the training set.

Both these properties make them rather similar to RBMs, but they are easier to train, and the sampling process is much faster (with RBMs you need to wait for the network to stabilize into a "thermal equilibrium" before you can sample a new instance). Indeed, as their name suggests, variational autoencoders perform variational Bayesian inference (introduced in Chapter 9), which is an efficient way to perform approximate Bayesian inference.

Let's take a look at how they work. Figure 17-12 (left) shows a variational autoencoder. You can recognize the basic structure of all autoencoders, with an encoder followed by a decoder (in this example, they both have two hidden layers), but there is a twist: instead of directly producing a coding for a given input, the encoder produces a *mean coding* $\boldsymbol{\mu}$ and a standard deviation $\boldsymbol{\sigma}$. The actual coding is then sampled randomly from a Gaussian distribution with mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$. After that the decoder decodes the sampled coding normally. The right part of the diagram shows a training instance going through this autoencoder. First, the encoder produces $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$, then a coding is sampled randomly (notice that it is not exactly located at $\boldsymbol{\mu}$), and finally this coding is decoded; the final output resembles the training instance.

7 Diederik Kingma and Max Welling, "Auto-Encoding Variational Bayes," arXiv preprint arXiv:1312.6114 (2013).

Figure 17-12. Variational autoencoder (left) and an instance going through it (right)

As you can see in the diagram, although the inputs may have a very convoluted distribution, a variational autoencoder tends to produce codings that look as though they were sampled from a simple Gaussian distribution:[8] during training, the cost function (discussed next) pushes the codings to gradually migrate within the coding space (also called the *latent space*) to end up looking like a cloud of Gaussian points. One great consequence is that after training a variational autoencoder, you can very easily generate a new instance: just sample a random coding from the Gaussian distribution, decode it, and voilà!

Now, let's look at the cost function. It is composed of two parts. The first is the usual reconstruction loss that pushes the autoencoder to reproduce its inputs (we can use cross entropy for this, as discussed earlier). The second is the *latent loss* that pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution: it is the KL divergence between the target distribution (i.e., the Gaussian distribution) and the actual distribution of the codings. The math is a bit more complex than with the sparse autoencoder, in particular because of the Gaussian noise, which limits the amount of information that can be transmitted to the coding layer (thus pushing the autoencoder to learn useful features). Luckily, the

8 Variational autoencoders are actually more general; the codings are not limited to Gaussian distributions.

equations simplify, so the latent loss can be computed quite simply using Equation 17-3:[9]

Equation 17-3. Variational autoencoder's latent loss

$$\mathscr{L} = -\frac{1}{2}\sum_{i=1}^{K} 1 + \log\left(\sigma_i^2\right) - \sigma_i^2 - \mu_i^2$$

In this equation, $\mathscr{L}$ is the latent loss, n is the codings' dimensionality, and μ_i and σ_i are the mean and standard deviation of the ith component of the codings. The vectors $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ (which contain all the μ_i and σ_i) are output by the encoder, as shown in Figure 17-12 (left).

A common tweak to the variational autoencoder's architecture is to make the encoder output $\boldsymbol{\gamma} = \log(\boldsymbol{\sigma}^2)$ rather than $\boldsymbol{\sigma}$. The latent loss can then be computed as shown in Equation 17-4. This approach is more numerically stable and speeds up training.

Equation 17-4. Variational autoencoder's latent loss, rewritten using $\boldsymbol{\gamma} = \log(\boldsymbol{\sigma}^2)$

$$\mathscr{L} = -\frac{1}{2}\sum_{i=1}^{K} 1 + \gamma_i - \exp\left(\gamma_i\right) - \mu_i^2$$

Let's start building a variational autoencoder for Fashion MNIST (as shown in Figure 17-12, but using the $\boldsymbol{\gamma}$ tweak). First, we will need a custom layer to sample the codings, given $\boldsymbol{\mu}$ and $\boldsymbol{\gamma}$:

```
class Sampling(keras.layers.Layer):
    def call(self, inputs):
        mean, log_var = inputs
        return K.random_normal(tf.shape(log_var)) * K.exp(log_var / 2) + mean
```

This `Sampling` layer takes two inputs: `mean` ($\boldsymbol{\mu}$) and `log_var` ($\boldsymbol{\gamma}$). It uses the function `K.random_normal()` to sample a random vector (of the same shape as $\boldsymbol{\gamma}$) from the Normal distribution, with mean 0 and standard deviation 1. Then it multiplies it by $\exp(\boldsymbol{\gamma} / 2)$ (which is equal to $\boldsymbol{\sigma}$, as you can verify), and finally it adds $\boldsymbol{\mu}$ and returns the result. This samples a codings vector from the Normal distribution with mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$.

Next, we can create the encoder, using the Functional API because the model is not entirely sequential:

9 For more mathematical details, check out the original paper on variational autoencoders, or Carl Doersch's great tutorial (2016).

```
codings_size = 10

inputs = keras.layers.Input(shape=[28, 28])
z = keras.layers.Flatten()(inputs)
z = keras.layers.Dense(150, activation="selu")(z)
z = keras.layers.Dense(100, activation="selu")(z)
codings_mean = keras.layers.Dense(codings_size)(z)  # μ
codings_log_var = keras.layers.Dense(codings_size)(z)  # γ
codings = Sampling()([codings_mean, codings_log_var])
variational_encoder = keras.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])
```

Note that the `Dense` layers that output `codings_mean` (**μ**) and `codings_log_var` (**γ**) have the same inputs (i.e., the outputs of the second `Dense` layer). We then pass both `codings_mean` and `codings_log_var` to the `Sampling` layer. Finally, the `variational_encoder` model has three outputs, in case you want to inspect the values of `codings_mean` and `codings_log_var`. The only output we will use is the last one (`codings`). Now let's build the decoder:

```
decoder_inputs = keras.layers.Input(shape=[codings_size])
x = keras.layers.Dense(100, activation="selu")(decoder_inputs)
x = keras.layers.Dense(150, activation="selu")(x)
x = keras.layers.Dense(28 * 28, activation="sigmoid")(x)
outputs = keras.layers.Reshape([28, 28])(x)
variational_decoder = keras.Model(inputs=[decoder_inputs], outputs=[outputs])
```

For this decoder, we could have used the Sequential API instead of the Functional API, since it is really just a simple stack of layers, virtually identical to many of the decoders we have built so far. Finally, let's build the variational autoencoder model:

```
_, _, codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
variational_ae = keras.Model(inputs=[inputs], outputs=[reconstructions])
```

Note that we ignore the first two outputs of the encoder (we only want to feed the codings to the decoder). Lastly, we must add the latent loss and the reconstruction loss:

```
latent_loss = -0.5 * K.sum(
    1 + codings_log_var - K.exp(codings_log_var) - K.square(codings_mean),
    axis=-1)
variational_ae.add_loss(K.mean(latent_loss) / 784.)
variational_ae.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

We first apply Equation 17-4 to compute the latent loss for each instance in the batch (we sum over the last axis). Then we compute the mean loss over all the instances in the batch, and we divide the result by 784 to ensure it has the appropriate scale compared to the reconstruction loss. Indeed, the variational autoencoder's reconstruction loss is supposed to be the sum of the pixel reconstruction errors, but when Keras computes the `"binary_crossentropy"` loss, it computes the mean over all 784 pixels,

rather than the sum. So, the reconstruction loss is 784 times smaller than we need it to be. We could define a custom loss to compute the sum rather than the mean, but it is simpler to divide the latent loss by 784 (the final loss will be 784 times smaller than it should be, but this just means that we should use a larger learning rate).

Note that we use the `RMSprop` optimizer, which works well in this case. And finally we can train the autoencoder!

```
history = variational_ae.fit(X_train, X_train, epochs=50, batch_size=128,
                             validation_data=[X_valid, X_valid])
```

Generating Fashion MNIST Images

Now let's use this variational autoencoder to generate images that look like fashion items. All we need to do is sample random codings from a Gaussian distribution and decode them:

```
codings = tf.random.normal(shape=[12, codings_size])
images = variational_decoder(codings).numpy()
```

Figure 17-13 shows the 12 generated images.

Figure 17-13. Fashion MNIST images generated by the variational autoencoder

The majority of these images look fairly convincing, if a bit too fuzzy. The rest are not great, but don't be too harsh on the autoencoder—it only had a few minutes to learn! Give it a bit more fine-tuning and training time, and those images should look better.

Variational autoencoders make it possible to perform *semantic interpolation*: instead of interpolating two images at the pixel level (which would look as if the two images were overlaid), we can interpolate at the codings level. We first run both images through the encoder, then we interpolate the two codings we get, and finally we decode the interpolated codings to get the final image. It will look like a regular Fashion MNIST image, but it will be an intermediate between the original images. In the following code example, we take the 12 codings we just generated, we organize them

in a 3 × 4 grid, and we use TensorFlow's `tf.image.resize()` function to resize this grid to 5 × 7. By default, the `resize()` function will perform bilinear interpolation, so every other row and column will contain interpolated codings. We then use the decoder to produce all the images:

```
codings_grid = tf.reshape(codings, [1, 3, 4, codings_size])
larger_grid = tf.image.resize(codings_grid, size=[5, 7])
interpolated_codings = tf.reshape(larger_grid, [-1, codings_size])
images = variational_decoder(interpolated_codings).numpy()
```

Figure 17-14 shows the resulting images. The original images are framed, and the rest are the result of semantic interpolation between the nearby images. Notice, for example, how the shoe in the fourth row and fifth column is a nice interpolation between the two shoes located above and below it.

Figure 17-14. Semantic interpolation

For several years, variational autoencoders were quite popular, but GANs eventually took the lead, in particular because they are capable of generating much more realistic and crisp images. So let's turn our attention to GANs.

Generative Adversarial Networks

Generative adversarial networks were proposed in a 2014 paper[10] by Ian Goodfellow et al., and although the idea got researchers excited almost instantly, it took a few years to overcome some of the difficulties of training GANs. Like many great ideas, it seems simple in hindsight: make neural networks compete against each other in the hope that this competition will push them to excel. As shown in Figure 17-15, a GAN is composed of two neural networks:

Generator
: Takes a random distribution as input (typically Gaussian) and outputs some data—typically, an image. You can think of the random inputs as the latent representations (i.e., codings) of the image to be generated. So, as you can see, the generator offers the same functionality as a decoder in a variational autoencoder, and it can be used in the same way to generate new images (just feed it some Gaussian noise, and it outputs a brand-new image). However, it is trained very differently, as we will soon see.

Discriminator
: Takes either a fake image from the generator or a real image from the training set as input, and must guess whether the input image is fake or real.

Figure 17-15. A generative adversarial network

10 Ian Goodfellow et al., "Generative Adversarial Nets," *Proceedings of the 27th International Conference on Neural Information Processing Systems* 2 (2014): 2672–2680.

During training, the generator and the discriminator have opposite goals: the discriminator tries to tell fake images from real images, while the generator tries to produce images that look real enough to trick the discriminator. Because the GAN is composed of two networks with different objectives, it cannot be trained like a regular neural network. Each training iteration is divided into two phases:

- In the first phase, we train the discriminator. A batch of real images is sampled from the training set and is completed with an equal number of fake images produced by the generator. The labels are set to 0 for fake images and 1 for real images, and the discriminator is trained on this labeled batch for one step, using the binary cross-entropy loss. Importantly, backpropagation only optimizes the weights of the discriminator during this phase.
- In the second phase, we train the generator. We first use it to produce another batch of fake images, and once again the discriminator is used to tell whether the images are fake or real. This time we do not add real images in the batch, and all the labels are set to 1 (real): in other words, we want the generator to produce images that the discriminator will (wrongly) believe to be real! Crucially, the weights of the discriminator are frozen during this step, so backpropagation only affects the weights of the generator.

The generator never actually sees any real images, yet it gradually learns to produce convincing fake images! All it gets is the gradients flowing back through the discriminator. Fortunately, the better the discriminator gets, the more information about the real images is contained in these secondhand gradients, so the generator can make significant progress.

Let's go ahead and build a simple GAN for Fashion MNIST.

First, we need to build the generator and the discriminator. The generator is similar to an autoencoder's decoder, and the discriminator is a regular binary classifier (it takes an image as input and ends with a `Dense` layer containing a single unit and using the sigmoid activation function). For the second phase of each training iteration, we also need the full GAN model containing the generator followed by the discriminator:

```
codings_size = 30

generator = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[codings_size]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
```

```
discriminator = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(1, activation="sigmoid")
])
gan = keras.models.Sequential([generator, discriminator])
```

Next, we need to compile these models. As the discriminator is a binary classifier, we can naturally use the binary cross-entropy loss. The generator will only be trained through the `gan` model, so we do not need to compile it at all. The `gan` model is also a binary classifier, so it can use the binary cross-entropy loss. Importantly, the discriminator should not be trained during the second phase, so we make it non-trainable before compiling the `gan` model:

```
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
discriminator.trainable = False
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

The `trainable` attribute is taken into account by Keras only when compiling a model, so after running this code, the `discriminator` *is* trainable if we call its `fit()` method or its `train_on_batch()` method (which we will be using), while it is *not* trainable when we call these methods on the `gan` model.

Since the training loop is unusual, we cannot use the regular `fit()` method. Instead, we will write a custom training loop. For this, we first need to create a `Dataset` to iterate through the images:

```
batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices(X_train).shuffle(1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
```

We are now ready to write the training loop. Let's wrap it in a `train_gan()` function:

```
def train_gan(gan, dataset, batch_size, codings_size, n_epochs=50):
    generator, discriminator = gan.layers
    for epoch in range(n_epochs):
        for X_batch in dataset:
            # phase 1 - training the discriminator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            generated_images = generator(noise)
            X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)
            y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
            discriminator.trainable = True
            discriminator.train_on_batch(X_fake_and_real, y1)
            # phase 2 - training the generator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            y2 = tf.constant([[1.]] * batch_size)
            discriminator.trainable = False
            gan.train_on_batch(noise, y2)

train_gan(gan, dataset, batch_size, codings_size)
```

As discussed earlier, you can see the two phases at each iteration:

- In phase one we feed Gaussian noise to the generator to produce fake images, and we complete this batch by concatenating an equal number of real images. The targets `y1` are set to 0 for fake images and 1 for real images. Then we train the discriminator on this batch. Note that we set the discriminator's `trainable` attribute to `True`: this is only to get rid of a warning that Keras displays when it notices that `trainable` is now `False` but was `True` when the model was compiled (or vice versa).
- In phase two, we feed the GAN some Gaussian noise. Its generator will start by producing fake images, then the discriminator will try to guess whether these images are fake or real. We want the discriminator to believe that the fake images are real, so the targets `y2` are set to 1. Note that we set the `trainable` attribute to `False`, once again to avoid a warning.

That's it! If you display the generated images (see Figure 17-16), you will see that at the end of the first epoch, they already start to look like (very noisy) Fashion MNIST images.

Unfortunately, the images never really get much better than that, and you may even find epochs where the GAN seems to be forgetting what it learned. Why is that? Well, it turns out that training a GAN can be challenging. Let's see why.

Figure 17-16. Images generated by the GAN after one epoch of training

The Difficulties of Training GANs

During training, the generator and the discriminator constantly try to outsmart each other, in a zero-sum game. As training advances, the game may end up in a state that game theorists call a *Nash equilibrium*, named after the mathematician John Nash: this is when no player would be better off changing their own strategy, assuming the other players do not change theirs. For example, a Nash equilibrium is reached when everyone drives on the left side of the road: no driver would be better off being the only one to switch sides. Of course, there is a second possible Nash equilibrium: when everyone drives on the *right* side of the road. Different initial states and dynamics may lead to one equilibrium or the other. In this example, there is a single optimal strategy once an equilibrium is reached (i.e., driving on the same side as everyone else), but a Nash equilibrium can involve multiple competing strategies (e.g., a predator chases its prey, the prey tries to escape, and neither would be better off changing their strategy).

So how does this apply to GANs? Well, the authors of the paper demonstrated that a GAN can only reach a single Nash equilibrium: that's when the generator produces perfectly realistic images, and the discriminator is forced to guess (50% real, 50% fake). This fact is very encouraging: it would seem that you just need to train the GAN for long enough, and it will eventually reach this equilibrium, giving you a perfect generator. Unfortunately, it's not that simple: nothing guarantees that the equilibrium will ever be reached.

The biggest difficulty is called *mode collapse*: this is when the generator's outputs gradually become less diverse. How can this happen? Suppose that the generator gets better at producing convincing shoes than any other class. It will fool the discriminator a bit more with shoes, and this will encourage it to produce even more images of shoes. Gradually, it will forget how to produce anything else. Meanwhile, the only fake images that the discriminator will see will be shoes, so it will also forget how to discriminate fake images of other classes. Eventually, when the discriminator manages to discriminate the fake shoes from the real ones, the generator will be forced to move to another class. It may then become good at shirts, forgetting about shoes, and the discriminator will follow. The GAN may gradually cycle across a few classes, never really becoming very good at any of them.

Moreover, because the generator and the discriminator are constantly pushing against each other, their parameters may end up oscillating and becoming unstable. Training may begin properly, then suddenly diverge for no apparent reason, due to these instabilities. And since many factors affect these complex dynamics, GANs are very sensitive to the hyperparameters: you may have to spend a lot of effort fine-tuning them.

These problems have kept researchers very busy since 2014: many papers were published on this topic, some proposing new cost functions[11] (though a 2018 paper[12] by Google researchers questions their efficiency) or techniques to stabilize training or to avoid the mode collapse issue. For example, a popular technique called *experience replay* consists in storing the images produced by the generator at each iteration in a replay buffer (gradually dropping older generated images) and training the discriminator using real images plus fake images drawn from this buffer (rather than just fake images produced by the current generator). This reduces the chances that the discriminator will overfit the latest generator's outputs. Another common technique is called *mini-batch discrimination*: it measures how similar images are across the batch and provides this statistic to the discriminator, so it can easily reject a whole batch of fake images that lack diversity. This encourages the generator to produce a greater variety of images, reducing the chance of mode collapse. Other papers simply propose specific architectures that happen to perform well.

In short, this is still a very active field of research, and the dynamics of GANs are still not perfectly understood. But the good news is that great progress has been made, and some of the results are truly astounding! So let's look at some of the most successful architectures, starting with deep convolutional GANs, which were the state of the art just a few years ago. Then we will look at two more recent (and more complex) architectures.

11 For a nice comparison of the main GAN losses, check out this great GitHub project by Hwalsuk Lee.

12 Mario Lucic et al., "Are GANs Created Equal? A Large-Scale Study," *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (2018): 698–707.

Deep Convolutional GANs

The original GAN paper in 2014 experimented with convolutional layers, but only tried to generate small images. Soon after, many researchers tried to build GANs based on deeper convolutional nets for larger images. This proved to be tricky, as training was very unstable, but Alec Radford et al. finally succeeded in late 2015, after experimenting with many different architectures and hyperparameters. They called their architecture *deep convolutional GANs* (DCGANs).[13] Here are the main guidelines they proposed for building stable convolutional GANs:

- Replace any pooling layers with strided convolutions (in the discriminator) and transposed convolutions (in the generator).
- Use Batch Normalization in both the generator and the discriminator, except in the generator's output layer and the discriminator's input layer.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in the generator for all layers except the output layer, which should use tanh.
- Use leaky ReLU activation in the discriminator for all layers.

These guidelines will work in many cases, but not always, so you may still need to experiment with different hyperparameters (in fact, just changing the random seed and training the same model again will sometimes work). For example, here is a small DCGAN that works reasonably well with Fashion MNIST:

13 Alec Radford et al., "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," arXiv preprint arXiv:1511.06434 (2015).

```
codings_size = 100

generator = keras.models.Sequential([
    keras.layers.Dense(7 * 7 * 128, input_shape=[codings_size]),
    keras.layers.Reshape([7, 7, 128]),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(64, kernel_size=5, strides=2, padding="same",
                                 activation="selu"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2, padding="same",
                                 activation="tanh")
])
discriminator = keras.models.Sequential([
    keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="same",
                        activation=keras.layers.LeakyReLU(0.2),
                        input_shape=[28, 28, 1]),
    keras.layers.Dropout(0.4),
    keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="same",
                        activation=keras.layers.LeakyReLU(0.2)),
    keras.layers.Dropout(0.4),
    keras.layers.Flatten(),
    keras.layers.Dense(1, activation="sigmoid")
])
gan = keras.models.Sequential([generator, discriminator])
```

The generator takes codings of size 100, and it projects them to 6272 dimensions (7 * 7 * 128), and reshapes the result to get a 7 × 7 × 128 tensor. This tensor is batch normalized and fed to a transposed convolutional layer with a stride of 2, which upsamples it from 7 × 7 to 14 × 14 and reduces its depth from 128 to 64. The result is batch normalized again and fed to another transposed convolutional layer with a stride of 2, which upsamples it from 14 × 14 to 28 × 28 and reduces the depth from 64 to 1. This layer uses the tanh activation function, so the outputs will range from –1 to 1. For this reason, before training the GAN, we need to rescale the training set to that same range. We also need to reshape it to add the channel dimension:

```
X_train = X_train.reshape(-1, 28, 28, 1) * 2. - 1. # reshape and rescale
```

The discriminator looks much like a regular CNN for binary classification, except instead of using max pooling layers to downsample the image, we use strided convolutions (`strides=2`). Also note that we use the leaky ReLU activation function.

Overall, we respected the DCGAN guidelines, except we replaced the `BatchNormalization` layers in the discriminator with `Dropout` layers (otherwise training was unstable in this case) and we replaced ReLU with SELU in the generator. Feel free to tweak this architecture: you will see how sensitive it is to the hyperparameters (especially the relative learning rates of the two networks).

Lastly, to build the dataset, then compile and train this model, we use the exact same code as earlier. After 50 epochs of training, the generator produces images like those

shown in Figure 17-17. It's still not perfect, but many of these images are pretty convincing.

Figure 17-17. Images generated by the DCGAN after 50 epochs of training

If you scale up this architecture and train it on a large dataset of faces, you can get fairly realistic images. In fact, DCGANs can learn quite meaningful latent representations, as you can see in Figure 17-18: many images were generated, and nine of them were picked manually (top left), including three representing men with glasses, three men without glasses, and three women without glasses. For each of these categories, the codings that were used to generate the images were averaged, and an image was generated based on the resulting mean codings (lower left). In short, each of the three lower-left images represents the mean of the three images located above it. But this is not a simple mean computed at the pixel level (this would result in three overlapping faces), it is a mean computed in the latent space, so the images still look like normal faces. Amazingly, if you compute men with glasses, minus men without glasses, plus women without glasses—where each term corresponds to one of the mean codings—and you generate the image that corresponds to this coding, you get the image at the center of the 3×3 grid of faces on the right: a woman with glasses! The eight other images around it were generated based on the same vector plus a bit of noise, to illustrate the semantic interpolation capabilities of DCGANs. Being able to do arithmetic on faces feels like science fiction!

Figure 17-18. Vector arithmetic for visual concepts (part of figure 7 from the DCGAN paper)[14]

If you add each image's class as an extra input to both the generator and the discriminator, they will both learn what each class looks like, and thus you will be able to control the class of each image produced by the generator. This is called a *conditional GAN*[15] (CGAN).

DCGANs aren't perfect, though. For example, when you try to generate very large images using DCGANs, you often end up with locally convincing features but overall inconsistencies (such as shirts with one sleeve much longer than the other). How can you fix this?

Progressive Growing of GANs

An important technique was proposed in a 2018 paper[16] by Nvidia researchers Tero Karras et al.: they suggested generating small images at the beginning of training, then gradually adding convolutional layers to both the generator and the discriminator to produce larger and larger images (4×4, 8×8, 16×16, …, 512×512, $1{,}024 \times 1{,}024$). This approach resembles greedy layer-wise training of stacked autoencoders.

14 Reproduced with the kind authorization of the authors.

15 Mehdi Mirza and Simon Osindero, "Conditional Generative Adversarial Nets," arXiv preprint arXiv: 1411.1784 (2014).

16 Tero Karras et al., "Progressive Growing of GANs for Improved Quality, Stability, and Variation," *Proceedings of the International Conference on Learning Representations* (2018).

The extra layers get added at the end of the generator and at the beginning of the discriminator, and previously trained layers remain trainable.

For example, when growing the generator's outputs from 4 × 4 to 8 × 8 (see Figure 17-19), an upsampling layer (using nearest neighbor filtering) is added to the existing convolutional layer, so it outputs 8 × 8 feature maps, which are then fed to the new convolutional layer (which uses `"same"` padding and strides of 1, so its outputs are also 8 × 8). This new layer is followed by a new output convolutional layer: this is a regular convolutional layer with kernel size 1 that projects the outputs down to the desired number of color channels (e.g., 3). To avoid breaking the trained weights of the first convolutional layer when the new convolutional layer is added, the final output is a weighted sum of the original output layer (which now outputs 8 × 8 feature maps) and the new output layer. The weight of the new outputs is α, while the weight of the original outputs is $1 - \alpha$, and α is slowly increased from 0 to 1. In other words, the new convolutional layers (represented with dashed lines in Figure 17-19) are gradually faded in, while the original output layer is gradually faded out. A similar fade-in/fade-out technique is used when a new convolutional layer is added to the discriminator (followed by an average pooling layer for downsampling).

Figure 17-19. Progressively growing GAN: a GAN generator outputs 4 × 4 color images (left); we extend it to output 8 × 8 images (right)

The paper also introduced several other techniques aimed at increasing the diversity of the outputs (to avoid mode collapse) and making training more stable:

Minibatch standard deviation layer

Added near the end of the discriminator. For each position in the inputs, it computes the standard deviation across all channels and all instances in the batch (`S = tf.math.reduce_std(inputs, axis=[0, -1])`). These standard deviations are then averaged across all points to get a single value (`v = tf.reduce_mean(S)`). Finally, an extra feature map is added to each instance in the batch and filled with the computed value (`tf.concat([inputs, tf.fill([batch_size, height, width, 1], v)], axis=-1)`). How does this help? Well, if the generator produces images with little variety, then there will be a small standard deviation across feature maps in the discriminator. Thanks to this layer, the discriminator will have easy access to this statistic, making it less likely to be fooled by a generator that produces too little diversity. This will encourage the generator to produce more diverse outputs, reducing the risk of mode collapse.

Equalized learning rate

Initializes all weights using a simple Gaussian distribution with mean 0 and standard deviation 1 rather than using He initialization. However, the weights are scaled down at runtime (i.e., every time the layer is executed) by the same factor as in He initialization: they are divided by $\sqrt{2/n_{\text{inputs}}}$, where n_{inputs} is the number of inputs to the layer. The paper demonstrated that this technique significantly improved the GAN's performance when using RMSProp, Adam, or other adaptive gradient optimizers. Indeed, these optimizers normalize the gradient updates by their estimated standard deviation (see Chapter 11), so parameters that have a larger dynamic range[17] will take longer to train, while parameters with a small dynamic range may be updated too quickly, leading to instabilities. By rescaling the weights as part of the model itself rather than just rescaling them upon initialization, this approach ensures that the dynamic range is the same for all parameters, throughout training, so they all learn at the same speed. This both speeds up and stabilizes training.

Pixelwise normalization layer

Added after each convolutional layer in the generator. It normalizes each activation based on all the activations in the same image and at the same location, but across all channels (dividing by the square root of the mean squared activation). In TensorFlow code, this is `inputs / tf.sqrt(tf.reduce_mean(tf.square(X), axis=-1, keepdims=True) + 1e-8)` (the smoothing term `1e-8` is needed to

17 The dynamic range of a variable is the ratio between the highest and the lowest value it may take.

avoid division by zero). This technique avoids explosions in the activations due to excessive competition between the generator and the discriminator.

The combination of all these techniques allowed the authors to generate extremely convincing high-definition images of faces. But what exactly do we call "convincing"? Evaluation is one of the big challenges when working with GANs: although it is possible to automatically evaluate the diversity of the generated images, judging their quality is a much trickier and subjective task. One technique is to use human raters, but this is costly and time-consuming. So the authors proposed to measure the similarity between the local image structure of the generated images and the training images, considering every scale. This idea led them to another groundbreaking innovation: StyleGANs.

StyleGANs

The state of the art in high-resolution image generation was advanced once again by the same Nvidia team in a 2018 paper[18] that introduced the popular StyleGAN architecture. The authors used *style transfer* techniques in the generator to ensure that the generated images have the same local structure as the training images, at every scale, greatly improving the quality of the generated images. The discriminator and the loss function were not modified, only the generator. Let's take a look at the StyleGAN. It is composed of two networks (see Figure 17-20):

Mapping network

An eight-layer MLP that maps the latent representations **z** (i.e., the codings) to a vector **w**. This vector is then sent through multiple *affine transformations* (i.e., `Dense` layers with no activation functions, represented by the "A" boxes in Figure 17-20), which produces multiple vectors. These vectors control the style of the generated image at different levels, from fine-grained texture (e.g., hair color) to high-level features (e.g., adult or child). In short, the mapping network maps the codings to multiple style vectors.

Synthesis network

Responsible for generating the images. It has a constant learned input (to be clear, this input will be constant *after* training, but *during* training it keeps getting tweaked by backpropagation). It processes this input through multiple convolutional and upsampling layers, as earlier, but there are two twists: first, some noise is added to the input and to all the outputs of the convolutional layers (before the activation function). Second, each noise layer is followed by an *Adaptive Instance Normalization* (AdaIN) layer: it standardizes each feature map independently (by

18 Tero Karras et al., "A Style-Based Generator Architecture for Generative Adversarial Networks," arXiv preprint arXiv:1812.04948 (2018).

subtracting the feature map's mean and dividing by its standard deviation), then it uses the style vector to determine the scale and offset of each feature map (the style vector contains one scale and one bias term for each feature map).

Figure 17-20. StyleGAN's generator architecture (part of figure 1 from the StyleGAN paper)[19]

The idea of adding noise independently from the codings is very important. Some parts of an image are quite random, such as the exact position of each freckle or hair. In earlier GANs, this randomness had to either come from the codings or be some pseudorandom noise produced by the generator itself. If it came from the codings, it meant that the generator had to dedicate a significant portion of the codings' representational power to store noise: this is quite wasteful. Moreover, the noise had to be

19 Reproduced with the kind authorization of the authors.

able to flow through the network and reach the final layers of the generator: this seems like an unnecessary constraint that probably slowed down training. And finally, some visual artifacts may appear because the same noise was used at different levels. If instead the generator tried to produce its own pseudorandom noise, this noise might not look very convincing, leading to more visual artifacts. Plus, part of the generator's weights would be dedicated to generating pseudorandom noise, which again seems wasteful. By adding extra noise inputs, all these issues are avoided; the GAN is able to use the provided noise to add the right amount of stochasticity to each part of the image.

The added noise is different for each level. Each noise input consists of a single feature map full of Gaussian noise, which is broadcast to all feature maps (of the given level) and scaled using learned per-feature scaling factors (this is represented by the "B" boxes in Figure 17-20) before it is added.

Finally, StyleGAN uses a technique called *mixing regularization* (or *style mixing*), where a percentage of the generated images are produced using two different codings. Specifically, the codings $\mathbf{c}_1$ and $\mathbf{c}_2$ are sent through the mapping network, giving two style vectors $\mathbf{w}_1$ and $\mathbf{w}_2$. Then the synthesis network generates an image based on the styles $\mathbf{w}_1$ for the first levels and the styles $\mathbf{w}_2$ for the remaining levels. The cutoff level is picked randomly. This prevents the network from assuming that styles at adjacent levels are correlated, which in turn encourages locality in the GAN, meaning that each style vector only affects a limited number of traits in the generated image.

There is such a wide variety of GANs out there that it would require a whole book to cover them all. Hopefully this introduction has given you the main ideas, and most importantly the desire to learn more. If you're struggling with a mathematical concept, there are probably blog posts out there that will help you understand it better. Then go ahead and implement your own GAN, and do not get discouraged if it has trouble learning at first: unfortunately, this is normal, and it will require quite a bit of patience before it works, but the result is worth it. If you're struggling with an implementation detail, there are plenty of Keras or TensorFlow implementations that you can look at. In fact, if all you want is to get some amazing results quickly, then you can just use a pretrained model (e.g., there are pretrained StyleGAN models available for Keras).

In the next chapter we will move to an entirely different branch of Deep Learning: Deep Reinforcement Learning.

Exercises

1. What are the main tasks that autoencoders are used for?
2. Suppose you want to train a classifier, and you have plenty of unlabeled training data but only a few thousand labeled instances. How can autoencoders help? How would you proceed?
3. If an autoencoder perfectly reconstructs the inputs, is it necessarily a good autoencoder? How can you evaluate the performance of an autoencoder?
4. What are undercomplete and overcomplete autoencoders? What is the main risk of an excessively undercomplete autoencoder? What about the main risk of an overcomplete autoencoder?
5. How do you tie weights in a stacked autoencoder? What is the point of doing so?
6. What is a generative model? Can you name a type of generative autoencoder?
7. What is a GAN? Can you name a few tasks where GANs can shine?
8. What are the main difficulties when training GANs?
9. Try using a denoising autoencoder to pretrain an image classifier. You can use MNIST (the simplest option), or a more complex image dataset such as CIFAR10 if you want a bigger challenge. Regardless of the dataset you're using, follow these steps:
   - Split the dataset into a training set and a test set. Train a deep denoising autoencoder on the full training set.
   - Check that the images are fairly well reconstructed. Visualize the images that most activate each neuron in the coding layer.
   - Build a classification DNN, reusing the lower layers of the autoencoder. Train it using only 500 images from the training set. Does it perform better with or without pretraining?
10. Train a variational autoencoder on the image dataset of your choice, and use it to generate images. Alternatively, you can try to find an unlabeled dataset that you are interested in and see if you can generate new samples.
11. Train a DCGAN to tackle the image dataset of your choice, and use it to generate images. Add experience replay and see if this helps. Turn it into a conditional GAN where you can control the generated class.

Solutions to these exercises are available in Appendix A.

CHAPTER 18
Reinforcement Learning

Reinforcement Learning (RL) is one of the most exciting fields of Machine Learning today, and also one of the oldest. It has been around since the 1950s, producing many interesting applications over the years,[1] particularly in games (e.g., *TD-Gammon*, a Backgammon-playing program) and in machine control, but seldom making the headline news. But a revolution took place in 2013, when researchers from a British startup called DeepMind demonstrated a system that could learn to play just about any Atari game from scratch,[2] eventually outperforming humans[3] in most of them, using only raw pixels as inputs and without any prior knowledge of the rules of the games.[4] This was the first of a series of amazing feats, culminating in March 2016 with the victory of their system AlphaGo against Lee Sedol, a legendary professional player of the game of Go, and in May 2017 against Ke Jie, the world champion. No program had ever come close to beating a master of this game, let alone the world champion. Today the whole field of RL is boiling with new ideas, with a wide range of applications. DeepMind was bought by Google for over $500 million in 2014.

So how did DeepMind achieve all this? With hindsight it seems rather simple: they applied the power of Deep Learning to the field of Reinforcement Learning, and it worked beyond their wildest dreams. In this chapter we will first explain what

1 For more details, be sure to check out Richard Sutton and Andrew Barto's book on RL, *Reinforcement Learning: An Introduction* (MIT Press).

2 Volodymyr Mnih et al., "Playing Atari with Deep Reinforcement Learning," arXiv preprint arXiv:1312.5602 (2013).

3 Volodymyr Mnih et al., "Human-Level Control Through Deep Reinforcement Learning," *Nature* 518 (2015): 529–533.

4 Check out the videos of DeepMind's system learning to play *Space Invaders*, *Breakout*, and other video games at *https://homl.info/dqn3*.

Reinforcement Learning is and what it's good at, then present two of the most important techniques in Deep Reinforcement Learning: *policy gradients* and *deep Q-networks* (DQNs), including a discussion of *Markov decision processes* (MDPs). We will use these techniques to train models to balance a pole on a moving cart; then I'll introduce the TF-Agents library, which uses state-of-the-art algorithms that greatly simplify building powerful RL systems, and we will use the library to train an agent to play *Breakout*, the famous Atari game. I'll close the chapter by taking a look at some of the latest advances in the field.

Learning to Optimize Rewards

In Reinforcement Learning, a software *agent* makes *observations* and takes *actions* within an *environment*, and in return it receives *rewards*. Its objective is to learn to act in a way that will maximize its expected rewards over time. If you don't mind a bit of anthropomorphism, you can think of positive rewards as pleasure, and negative rewards as pain (the term "reward" is a bit misleading in this case). In short, the agent acts in the environment and learns by trial and error to maximize its pleasure and minimize its pain.

This is quite a broad setting, which can apply to a wide variety of tasks. Here are a few examples (see Figure 18-1):

a. The agent can be the program controlling a robot. In this case, the environment is the real world, the agent observes the environment through a set of *sensors* such as cameras and touch sensors, and its actions consist of sending signals to activate motors. It may be programmed to get positive rewards whenever it approaches the target destination, and negative rewards whenever it wastes time or goes in the wrong direction.

b. The agent can be the program controlling *Ms. Pac-Man*. In this case, the environment is a simulation of the Atari game, the actions are the nine possible joystick positions (upper left, down, center, and so on), the observations are screenshots, and the rewards are just the game points.

c. Similarly, the agent can be the program playing a board game such as Go.

d. The agent does not have to control a physically (or virtually) moving thing. For example, it can be a smart thermostat, getting positive rewards whenever it is close to the target temperature and saves energy, and negative rewards when humans need to tweak the temperature, so the agent must learn to anticipate human needs.

e. The agent can observe stock market prices and decide how much to buy or sell every second. Rewards are obviously the monetary gains and losses.

Figure 18-1. Reinforcement Learning examples: (a) robotics, (b) Ms. Pac-Man, (c) Go player, (d) thermostat, (e) automatic trader[5]

Note that there may not be any positive rewards at all; for example, the agent may move around in a maze, getting a negative reward at every time step, so it had better find the exit as quickly as possible! There are many other examples of tasks to which Reinforcement Learning is well suited, such as self-driving cars, recommender systems, placing ads on a web page, or controlling where an image classification system should focus its attention.

5 Image (a) is from NASA (public domain). (b) is a screenshot from the *Ms. Pac-Man* game, copyright Atari (fair use in this chapter). Images (c) and (d) are reproduced from Wikipedia. (c) was created by user Stevertigo and released under Creative Commons BY-SA 2.0. (d) is in the public domain. (e) was reproduced from Pixabay, released under Creative Commons CC0.

Policy Search

The algorithm a software agent uses to determine its actions is called its *policy*. The policy could be a neural network taking observations as inputs and outputting the action to take (see Figure 18-2).

Figure 18-2. Reinforcement Learning using a neural network policy

The policy can be any algorithm you can think of, and it does not have to be deterministic. In fact, in some cases it does not even have to observe the environment! For example, consider a robotic vacuum cleaner whose reward is the amount of dust it picks up in 30 minutes. Its policy could be to move forward with some probability p every second, or randomly rotate left or right with probability $1 - p$. The rotation angle would be a random angle between $-r$ and $+r$. Since this policy involves some randomness, it is called a *stochastic policy*. The robot will have an erratic trajectory, which guarantees that it will eventually get to any place it can reach and pick up all the dust. The question is, how much dust will it pick up in 30 minutes?

How would you train such a robot? There are just two *policy parameters* you can tweak: the probability p and the angle range r. One possible learning algorithm could be to try out many different values for these parameters, and pick the combination that performs best (see Figure 18-3). This is an example of *policy search*, in this case using a brute force approach. When the *policy space* is too large (which is generally the case), finding a good set of parameters this way is like searching for a needle in a gigantic haystack.

Another way to explore the policy space is to use *genetic algorithms*. For example, you could randomly create a first generation of 100 policies and try them out, then "kill" the 80 worst policies[6] and make the 20 survivors produce 4 offspring each. An

6 It is often better to give the poor performers a slight chance of survival, to preserve some diversity in the "gene pool."

offspring is a copy of its parent[7] plus some random variation. The surviving policies plus their offspring together constitute the second generation. You can continue to iterate through generations this way until you find a good policy.[8]

Figure 18-3. Four points in policy space (left) and the agent's corresponding behavior (right)

Yet another approach is to use optimization techniques, by evaluating the gradients of the rewards with regard to the policy parameters, then tweaking these parameters by following the gradients toward higher rewards.[9] We will discuss this approach, is called *policy gradients* (PG), in more detail later in this chapter. Going back to the vacuum cleaner robot, you could slightly increase p and evaluate whether doing so increases the amount of dust picked up by the robot in 30 minutes; if it does, then increase p some more, or else reduce p. We will implement a popular PG algorithm using TensorFlow, but before we do, we need to create an environment for the agent to live in—so it's time to introduce OpenAI Gym.

Introduction to OpenAI Gym

One of the challenges of Reinforcement Learning is that in order to train an agent, you first need to have a working environment. If you want to program an agent that

7 If there is a single parent, this is called *asexual reproduction*. With two (or more) parents, it is called *sexual reproduction*. An offspring's genome (in this case a set of policy parameters) is randomly composed of parts of its parents' genomes.

8 One interesting example of a genetic algorithm used for Reinforcement Learning is the *NeuroEvolution of Augmenting Topologies* (NEAT) algorithm.

9 This is called *Gradient Ascent*. It's just like Gradient Descent but in the opposite direction: maximizing instead of minimizing.

will learn to play an Atari game, you will need an Atari game simulator. If you want to program a walking robot, then the environment is the real world, and you can directly train your robot in that environment, but this has its limits: if the robot falls off a cliff, you can't just click Undo. You can't speed up time either; adding more computing power won't make the robot move any faster. And it's generally too expensive to train 1,000 robots in parallel. In short, training is hard and slow in the real world, so you generally need a *simulated environment* at least for bootstrap training. For example, you may use a library like PyBullet or MuJoCo for 3D physics simulation.

OpenAI Gym[10] is a toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, and so on), so you can train agents, compare them, or develop new RL algorithms.

Before installing the toolkit, if you created an isolated environment using virtualenv, you first need to activate it:

```
$ cd $ML_PATH                # Your ML working directory (e.g., $HOME/ml)
$ source my_env/bin/activate # on Linux or MacOS
$ .\my_env\Scripts\activate  # on Windows
```

Next, install OpenAI Gym (if you are not using a virtual environment, you will need to add the `--user` option, or have administrator rights):

```
$ python3 -m pip install -U gym
```

Depending on your system, you may also need to install the Mesa OpenGL Utility (GLU) library (e.g., on Ubuntu 18.04 you need to run `apt install libglu1-mesa`). This library will be needed to render the first environment. Next, open up a Python shell or a Jupyter notebook and create an environment with `make()`:

```
>>> import gym
>>> env = gym.make("CartPole-v1")
>>> obs = env.reset()
>>> obs
array([-0.01258566, -0.00156614,  0.04207708, -0.00180545])
```

Here, we've created a CartPole environment. This is a 2D simulation in which a cart can be accelerated left or right in order to balance a pole placed on top of it (see Figure 18-4). You can get the list of all available environments by running `gym.envs.registry.all()`. After the environment is created, you must initialize it using the `reset()` method. This returns the first observation. Observations depend on the type of environment. For the CartPole environment, each observation is a 1D NumPy array containing four floats: these floats represent the cart's horizontal

10 OpenAI is an artificial intelligence research company, funded in part by Elon Musk. Its stated goal is to promote and develop friendly AIs that will benefit humanity (rather than exterminate it).

position (`0.0` = center), its velocity (positive means right), the angle of the pole (`0.0` = vertical), and its angular velocity (positive means clockwise).

Now let's display this environment by calling its `render()` method (see Figure 18-4). On Windows, this requires first installing an X Server, such as VcXsrv or Xming:

```
>>> env.render()
True
```

Figure 18-4. The CartPole environment

If you are using a headless server (i.e., without a screen), such as a virtual machine on the cloud, rendering will fail. The only way to avoid this is to use a fake X server such as Xvfb or Xdummy. For example, you can install Xvfb (`apt install xvfb` on Ubuntu or Debian) and start Python using the following command: `xvfb-run -s "-screen 0 1400x900x24" python3`. Alternatively, install Xvfb and the `pyvirtualdisplay` library (which wraps Xvfb) and run `pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()` at the beginning of your program.

If you want `render()` to return the rendered image as a NumPy array, you can set `mode="rgb_array"` (oddly, this environment will render the environment to screen as well):

```
>>> img = env.render(mode="rgb_array")
>>> img.shape  # height, width, channels (3 = Red, Green, Blue)
(800, 1200, 3)
```

Let's ask the environment what actions are possible:

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` means that the possible actions are integers 0 and 1, which represent accelerating left (0) or right (1). Other environments may have additional discrete

actions, or other kinds of actions (e.g., continuous). Since the pole is leaning toward the right (`obs[2] > 0`), let's accelerate the cart toward the right:

```
>>> action = 1  # accelerate right
>>> obs, reward, done, info = env.step(action)
>>> obs
array([-0.01261699,  0.19292789,  0.04204097, -0.28092127])
>>> reward
1.0
>>> done
False
>>> info
{}
```

The `step()` method executes the given action and returns four values:

`obs`
: This is the new observation. The cart is now moving toward the right (`obs[1] > 0`). The pole is still tilted toward the right (`obs[2] > 0`), but its angular velocity is now negative (`obs[3] < 0`), so it will likely be tilted toward the left after the next step.

`reward`
: In this environment, you get a reward of 1.0 at every step, no matter what you do, so the goal is to keep the episode running as long as possible.

`done`
: This value will be `True` when the episode is over. This will happen when the pole tilts too much, or goes off the screen, or after 200 steps (in this last case, you have won). After that, the environment must be reset before it can be used again.

`info`
: This environment-specific dictionary can provide some extra information that you may find useful for debugging or for training. For example, in some games it may indicate how many lives the agent has.

Once you have finished using an environment, you should call its `close()` method to free resources.

Let's hardcode a simple policy that accelerates left when the pole is leaning toward the left and accelerates right when the pole is leaning toward the right. We will run this policy to see the average rewards it gets over 500 episodes:

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset()
    for step in range(200):
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

This code is hopefully self-explanatory. Let's look at the result:

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
(41.718, 8.858356280936096, 24.0, 68.0)
```

Even with 500 tries, this policy never managed to keep the pole upright for more than 68 consecutive steps. Not great. If you look at the simulation in the Jupyter notebooks, you will see that the cart oscillates left and right more and more strongly until the pole tilts too much. Let's see if a neural network can come up with a better policy.

Neural Network Policies

Let's create a neural network policy. Just like with the policy we hardcoded earlier, this neural network will take an observation as input, and it will output the action to be executed. More precisely, it will estimate a probability for each action, and then we will select an action randomly, according to the estimated probabilities (see Figure 18-5). In the case of the CartPole environment, there are just two possible actions (left or right), so we only need one output neuron. It will output the probability p of action 0 (left), and of course the probability of action 1 (right) will be $1 - p$. For example, if it outputs 0.7, then we will pick action 0 with 70% probability, or action 1 with 30% probability.

Figure 18-5. Neural network policy

You may wonder why we are picking a random action based on the probabilities given by the neural network, rather than just picking the action with the highest score. This approach lets the agent find the right balance between *exploring* new actions and *exploiting* the actions that are known to work well. Here's an analogy: suppose you go to a restaurant for the first time, and all the dishes look equally appealing, so you randomly pick one. If it turns out to be good, you can increase the probability that you'll order it next time, but you shouldn't increase that probability up to 100%, or else you will never try out the other dishes, some of which may be even better than the one you tried.

Also note that in this particular environment, the past actions and observations can safely be ignored, since each observation contains the environment's full state. If there were some hidden state, then you might need to consider past actions and observations as well. For example, if the environment only revealed the position of the cart but not its velocity, you would have to consider not only the current observation but also the previous observation in order to estimate the current velocity. Another example is when the observations are noisy; in that case, you generally want to use the past few observations to estimate the most likely current state. The CartPole problem is thus as simple as can be; the observations are noise-free, and they contain the environment's full state.

Here is the code to build this neural network policy using tf.keras:

```
import tensorflow as tf
from tensorflow import keras

n_inputs = 4 # == env.observation_space.shape[0]

model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu", input_shape=[n_inputs]),
    keras.layers.Dense(1, activation="sigmoid"),
])
```

After the imports, we use a simple `Sequential` model to define the policy network. The number of inputs is the size of the observation space (which in the case of CartPole is 4), and we have just five hidden units because it's a simple problem. Finally, we want to output a single probability (the probability of going left), so we have a single output neuron using the sigmoid activation function. If there were more than two possible actions, there would be one output neuron per action, and we would use the softmax activation function instead.

OK, we now have a neural network policy that will take observations and output action probabilities. But how do we train it?

Evaluating Actions: The Credit Assignment Problem

If we knew what the best action was at each step, we could train the neural network as usual, by minimizing the cross entropy between the estimated probability distribution and the target probability distribution. It would just be regular supervised learning. However, in Reinforcement Learning the only guidance the agent gets is through rewards, and rewards are typically sparse and delayed. For example, if the agent manages to balance the pole for 100 steps, how can it know which of the 100 actions it took were good, and which of them were bad? All it knows is that the pole fell after the last action, but surely this last action is not entirely responsible. This is called the *credit assignment problem*: when the agent gets a reward, it is hard for it to know which actions should get credited (or blamed) for it. Think of a dog that gets rewarded hours after it behaved well; will it understand what it is being rewarded for?

To tackle this problem, a common strategy is to evaluate an action based on the sum of all the rewards that come after it, usually applying a *discount factor* γ (gamma) at each step. This sum of discounted rewards is called the action's *return*. Consider the example in Figure 18-6). If an agent decides to go right three times in a row and gets +10 reward after the first step, 0 after the second step, and finally −50 after the third step, then assuming we use a discount factor $\gamma = 0.8$, the first action will have a return of $10 + \gamma \times 0 + \gamma^2 \times (-50) = -22$. If the discount factor is close to 0, then future rewards won't count for much compared to immediate rewards. Conversely, if the discount factor is close to 1, then rewards far into the future will count almost as

much as immediate rewards. Typical discount factors vary from 0.9 to 0.99. With a discount factor of 0.95, rewards 13 steps into the future count roughly for half as much as immediate rewards (since $0.95^{13} \approx 0.5$), while with a discount factor of 0.99, rewards 69 steps into the future count for half as much as immediate rewards. In the CartPole environment, actions have fairly short-term effects, so choosing a discount factor of 0.95 seems reasonable.

Figure 18-6. Computing an action's return: the sum of discounted future rewards

Of course, a good action may be followed by several bad actions that cause the pole to fall quickly, resulting in the good action getting a low return (similarly, a good actor may sometimes star in a terrible movie). However, if we play the game enough times, on average good actions will get a higher return than bad ones. We want to estimate how much better or worse an action is, compared to the other possible actions, on average. This is called the *action advantage*. For this, we must run many episodes and normalize all the action returns (by subtracting the mean and dividing by the standard deviation). After that, we can reasonably assume that actions with a negative advantage were bad while actions with a positive advantage were good. Perfect—now that we have a way to evaluate each action, we are ready to train our first agent using policy gradients. Let's see how.

Policy Gradients

As discussed earlier, PG algorithms optimize the parameters of a policy by following the gradients toward higher rewards. One popular class of PG algorithms, called

REINFORCE algorithms, was introduced back in 1992[11] by Ronald Williams. Here is one common variant:

1. First, let the neural network policy play the game several times, and at each step, compute the gradients that would make the chosen action even more likely—but don't apply these gradients yet.
2. Once you have run several episodes, compute each action's advantage (using the method described in the previous section).
3. If an action's advantage is positive, it means that the action was probably good, and you want to apply the gradients computed earlier to make the action even more likely to be chosen in the future. However, if the action's advantage is negative, it means the action was probably bad, and you want to apply the opposite gradients to make this action slightly *less* likely in the future. The solution is simply to multiply each gradient vector by the corresponding action's advantage.
4. Finally, compute the mean of all the resulting gradient vectors, and use it to perform a Gradient Descent step.

Let's use tf.keras to implement this algorithm. We will train the neural network policy we built earlier so that it learns to balance the pole on the cart. First, we need a function that will play one step. We will pretend for now that whatever action it takes is the right one so that we can compute the loss and its gradients (these gradients will just be saved for a while, and we will modify them later depending on how good or bad the action turned out to be):

```
def play_one_step(env, obs, model, loss_fn):
    with tf.GradientTape() as tape:
        left_proba = model(obs[np.newaxis])
        action = (tf.random.uniform([1, 1]) > left_proba)
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))
    grads = tape.gradient(loss, model.trainable_variables)
    obs, reward, done, info = env.step(int(action[0, 0].numpy()))
    return obs, reward, done, grads
```

Let's walk though this function:

- Within the `GradientTape` block (see Chapter 12), we start by calling the model, giving it a single observation (we reshape the observation so it becomes a batch containing a single instance, as the model expects a batch). This outputs the probability of going left.

11 Ronald J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Leaning," *Machine Learning* 8 (1992) : 229–256.

- Next, we sample a random float between 0 and 1, and we check whether it is greater than `left_proba`. The `action` will be `False` with probability `left_proba`, or `True` with probability `1 - left_proba`. Once we cast this Boolean to a number, the action will be 0 (left) or 1 (right) with the appropriate probabilities.
- Next, we define the target probability of going left: it is 1 minus the action (cast to a float). If the action is 0 (left), then the target probability of going left will be 1. If the action is 1 (right), then the target probability will be 0.
- Then we compute the loss using the given loss function, and we use the tape to compute the gradient of the loss with regard to the model's trainable variables. Again, these gradients will be tweaked later, before we apply them, depending on how good or bad the action turned out to be.
- Finally, we play the selected action, and we return the new observation, the reward, whether the episode is ended or not, and of course the gradients that we just computed.

Now let's create another function that will rely on the `play_one_step()` function to play multiple episodes, returning all the rewards and gradients for each episode and each step:

```
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):
    all_rewards = []
    all_grads = []
    for episode in range(n_episodes):
        current_rewards = []
        current_grads = []
        obs = env.reset()
        for step in range(n_max_steps):
            obs, reward, done, grads = play_one_step(env, obs, model, loss_fn)
            current_rewards.append(reward)
            current_grads.append(grads)
            if done:
                break
        all_rewards.append(current_rewards)
        all_grads.append(current_grads)
    return all_rewards, all_grads
```

This code returns a list of reward lists (one reward list per episode, containing one reward per step), as well as a list of gradient lists (one gradient list per episode, each containing one tuple of gradients per step and each tuple containing one gradient tensor per trainable variable).

The algorithm will use the `play_multiple_episodes()` function to play the game several times (e.g., 10 times), then it will go back and look at all the rewards, discount them, and normalize them. To do that, we need a couple more functions: the first will compute the sum of future discounted rewards at each step, and the second will

normalize all these discounted rewards (returns) across many episodes by subtracting the mean and dividing by the standard deviation:

```
def discount_rewards(rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_factor
    return discounted

def discount_and_normalize_rewards(all_rewards, discount_factor):
    all_discounted_rewards = [discount_rewards(rewards, discount_factor)
                              for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean) / reward_std
            for discounted_rewards in all_discounted_rewards]
```

Let's check that this works:

```
>>> discount_rewards([10, 0, -50], discount_factor=0.8)
array([-22, -40, -50])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]],
...                                discount_factor=0.8)
...
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([1.26665318, 1.0727777 ])]
```

The call to `discount_rewards()` returns exactly what we expect (see Figure 18-6). You can verify that the function `discount_and_normalize_rewards()` does indeed return the normalized action advantages for each action in both episodes. Notice that the first episode was much worse than the second, so its normalized advantages are all negative; all actions from the first episode would be considered bad, and conversely all actions from the second episode would be considered good.

We are almost ready to run the algorithm! Now let's define the hyperparameters. We will run 150 training iterations, playing 10 episodes per iteration, and each episode will last at most 200 steps. We will use a discount factor of 0.95:

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95
```

We also need an optimizer and the loss function. A regular Adam optimizer with learning rate 0.01 will do just fine, and we will use the binary cross-entropy loss function because we are training a binary classifier (there are two possible actions: left or right):

```
optimizer = keras.optimizers.Adam(lr=0.01)
loss_fn = keras.losses.binary_crossentropy
```

We are now ready to build and run the training loop!

```
for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)
    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_factor)
    all_mean_grads = []
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean(
            [final_reward * all_grads[episode_index][step][var_index]
             for episode_index, final_rewards in enumerate(all_final_rewards)
                 for step, final_reward in enumerate(final_rewards)], axis=0)
        all_mean_grads.append(mean_grads)
    optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))
```

Let's walk through this code:

- At each training iteration, this loop calls the `play_multiple_episodes()` function, which plays the game 10 times and returns all the rewards and gradients for every episode and step.
- Then we call the `discount_and_normalize_rewards()` to compute each action's normalized advantage (which in the code we call the `final_reward`). This provides a measure of how good or bad each action actually was, in hindsight.
- Next, we go through each trainable variable, and for each of them we compute the weighted mean of the gradients for that variable over all episodes and all steps, weighted by the `final_reward`.
- Finally, we apply these mean gradients using the optimizer: the model's trainable variables will be tweaked, and hopefully the policy will be a bit better.

And we're done! This code will train the neural network policy, and it will successfully learn to balance the pole on the cart (you can try it out in the "Policy Gradients" section of the Jupyter notebook). The mean reward per episode will get very close to 200 (which is the maximum by default with this environment). Success!

Researchers try to find algorithms that work well even when the agent initially knows nothing about the environment. However, unless you are writing a paper, you should not hesitate to inject prior knowledge into the agent, as it will speed up training dramatically. For example, since you know that the pole should be as vertical as possible, you could add negative rewards proportional to the pole's angle. This will make the rewards much less sparse and speed up training. Also, if you already have a reasonably good policy (e.g., hardcoded), you may want to train the neural network to imitate it before using policy gradients to improve it.

The simple policy gradients algorithm we just trained solved the CartPole task, but it would not scale well to larger and more complex tasks. Indeed, it is highly *sample inefficient*, meaning it needs to explore the game for a very long time before it can make significant progress. This is due to the fact that it must run multiple episodes to estimate the advantage of each action, as we have seen. However, it is the foundation of more powerful algorithms, such as *Actor-Critic* algorithms (which we will discuss briefly at the end of this chapter).

We will now look at another popular family of algorithms. Whereas PG algorithms directly try to optimize the policy to increase rewards, the algorithms we will look at now are less direct: the agent learns to estimate the expected return for each state, or for each action in each state, then it uses this knowledge to decide how to act. To understand these algorithms, we must first introduce *Markov decision processes*.

Markov Decision Processes

In the early 20th century, the mathematician Andrey Markov studied stochastic processes with no memory, called *Markov chains*. Such a process has a fixed number of states, and it randomly evolves from one state to another at each step. The probability for it to evolve from a state s to a state s' is fixed, and it depends only on the pair (s, s'), not on past states (this is why we say that the system has no memory).

Figure 18-7 shows an example of a Markov chain with four states.

Figure 18-7. Example of a Markov chain

Suppose that the process starts in state s_0, and there is a 70% chance that it will remain in that state at the next step. Eventually it is bound to leave that state and never come back because no other state points back to s_0. If it goes to state s_1, it will then most likely go to state s_2 (90% probability), then immediately back to state s_1

(with 100% probability). It may alternate a number of times between these two states, but eventually it will fall into state s_3 and remain there forever (this is a *terminal state*). Markov chains can have very different dynamics, and they are heavily used in thermodynamics, chemistry, statistics, and much more.

Markov decision processes were first described in the 1950s by Richard Bellman.[12] They resemble Markov chains but with a twist: at each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action. Moreover, some state transitions return some reward (positive or negative), and the agent's goal is to find a policy that will maximize reward over time.

For example, the MDP represented in Figure 18-8 has three states (represented by circles) and up to three possible discrete actions at each step (represented by diamonds).

Figure 18-8. Example of a Markov decision process

If it starts in state s_0, the agent can choose between actions a_0, a_1, or a_2. If it chooses action a_1, it just remains in state s_0 with certainty, and without any reward. It can thus decide to stay there forever if it wants to. But if it chooses action a_0, it has a 70% probability of gaining a reward of +10 and remaining in state s_0. It can then try again and again to gain as much reward as possible, but at one point it is going to end up instead in state s_1. In state s_1 it has only two possible actions: a_0 or a_2. It can choose to stay put by repeatedly choosing action a_0, or it can choose to move on to state s_2 and get a negative reward of −50 (ouch). In state s_2 it has no other choice than to take action a_1, which will most likely lead it back to state s_0, gaining a reward of +40 on the

12 Richard Bellman, "A Markovian Decision Process," *Journal of Mathematics and Mechanics* 6, no. 5 (1957): 679–684.

way. You get the picture. By looking at this MDP, can you guess which strategy will gain the most reward over time? In state s_0 it is clear that action a_0 is the best option, and in state s_2 the agent has no choice but to take action a_1, but in state s_1 it is not obvious whether the agent should stay put (a_0) or go through the fire (a_2).

Bellman found a way to estimate the *optimal state value* of any state s, noted $V^*(s)$, which is the sum of all discounted future rewards the agent can expect on average after it reaches a state s, assuming it acts optimally. He showed that if the agent acts optimally, then the *Bellman Optimality Equation* applies (see Equation 18-1). This recursive equation says that if the agent acts optimally, then the optimal value of the current state is equal to the reward it will get on average after taking one optimal action, plus the expected optimal value of all possible next states that this action can lead to.

Equation 18-1. Bellman Optimality Equation

$$V^*(s) = \max_a \sum_s T(s, a, s')\left[R(s, a, s') + \gamma \cdot V^*(s')\right] \quad \text{for all } s$$

In this equation:

- $T(s, a, s')$ is the transition probability from state s to state s', given that the agent chose action a. For example, in Figure 18-8, $T(s_2, a_1, s_0) = 0.8$.
- $R(s, a, s')$ is the reward that the agent gets when it goes from state s to state s', given that the agent chose action a. For example, in Figure 18-8, $R(s_2, a_1, s_0) = +40$.
- γ is the discount factor.

This equation leads directly to an algorithm that can precisely estimate the optimal state value of every possible state: you first initialize all the state value estimates to zero, and then you iteratively update them using the *Value Iteration* algorithm (see Equation 18-2). A remarkable result is that, given enough time, these estimates are guaranteed to converge to the optimal state values, corresponding to the optimal policy.

Equation 18-2. Value Iteration algorithm

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')\left[R(s, a, s') + \gamma \cdot V_k(s')\right] \quad \text{for all } s$$

In this equation, $V_k(s)$ is the estimated value of state s at the kth iteration of the algorithm.

This algorithm is an example of *Dynamic Programming*, which breaks down a complex problem into tractable subproblems that can be tackled iteratively.

Knowing the optimal state values can be useful, in particular to evaluate a policy, but it does not give us the optimal policy for the agent. Luckily, Bellman found a very similar algorithm to estimate the optimal *state-action values*, generally called *Q-Values* (Quality Values). The optimal Q-Value of the state-action pair (s, a), noted $Q^\star(s, a)$, is the sum of discounted future rewards the agent can expect on average after it reaches the state s and chooses action a, but before it sees the outcome of this action, assuming it acts optimally after that action.

Here is how it works: once again, you start by initializing all the Q-Value estimates to zero, then you update them using the *Q-Value Iteration* algorithm (see Equation 18-3).

Equation 18-3. Q-Value Iteration algorithm

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \cdot \max_{a'} Q_k(s',a')\right] \quad \text{for all } (s'a)$$

Once you have the optimal Q-Values, defining the optimal policy, noted $\pi^\star(s)$, is trivial: when the agent is in state s, it should choose the action with the highest Q-Value for that state: $\pi^\star(s) = \underset{a}{\operatorname{argmax}}\ Q^\star(s,a)$.

Let's apply this algorithm to the MDP represented in Figure 18-8. First, we need to define the MDP:

```
transition_probabilities = [ # shape=[s, a, s']
        [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
        [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
        [None, [0.8, 0.1, 0.1], None]]
rewards = [ # shape=[s, a, s']
        [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
        [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
        [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]]
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

For example, to know the transition probability from s_2 to s_0 after playing action a_1, we will look up `transition_probabilities[2][1][0]` (which is 0.8). Similarly, to get the corresponding reward, we will look up `rewards[2][1][0]` (which is +40). And to get the list of possible actions in s_2, we will look up `possible_actions[2]` (in this case, only action a_1 is possible). Next, we must initialize all the Q-Values to 0 (except for the the impossible actions, for which we set the Q-Values to $-\infty$):

```
Q_values = np.full((3, 3), -np.inf) # -np.inf for impossible actions
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0  # for all possible actions
```

Now let's run the Q-Value Iteration algorithm. It applies Equation 18-3 repeatedly, to all Q-Values, for every state and every possible action:

```
gamma = 0.90 # the discount factor

for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                    transition_probabilities[s][a][sp]
                    * (rewards[s][a][sp] + gamma * np.max(Q_prev[sp]))
                for sp in range(3)])
```

That's it! The resulting Q-Values look like this:

```
>>> Q_values
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.        ,        -inf, -4.87971488],
       [       -inf, 50.13365013,        -inf]])
```

For example, when the agent is in state s_0 and it chooses action a_1, the expected sum of discounted future rewards is approximately 17.0.

For each state, let's look at the action that has the highest Q-Value:

```
>>> np.argmax(Q_values, axis=1) # optimal action for each state
array([0, 0, 1])
```

This gives us the optimal policy for this MDP, when using a discount factor of 0.90: in state s_0 choose action a_0; in state s_1 choose action a_0 (i.e., stay put); and in state s_2 choose action a_1 (the only possible action). Interestingly, if we increase the discount factor to 0.95, the optimal policy changes: in state s_1 the best action becomes a_2 (go through the fire!). This makes sense because the more you value future rewards, the more you are willing to put up with some pain now for the promise of future bliss.

Temporal Difference Learning

Reinforcement Learning problems with discrete actions can often be modeled as Markov decision processes, but the agent initially has no idea what the transition probabilities are (it does not know $T(s, a, s')$), and it does not know what the rewards are going to be either (it does not know $R(s, a, s')$). It must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities.

The *Temporal Difference Learning* (TD Learning) algorithm is very similar to the Value Iteration algorithm, but tweaked to take into account the fact that the agent has

only partial knowledge of the MDP. In general we assume that the agent initially knows only the possible states and actions, and nothing more. The agent uses an *exploration policy*—for example, a purely random policy—to explore the MDP, and as it progresses, the TD Learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed (see Equation 18-4).

Equation 18-4. TD Learning algorithm

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

or, equivalently:

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s')$$

with $\delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k(s)$

In this equation:

- α is the learning rate (e.g., 0.01).
- $r + \gamma \cdot V_k(s')$ is called the *TD target*.
- $\delta_k(s, r, s')$ is called the *TD error*.

A more concise way of writing the first form of this equation is to use the notation $a \underset{\alpha}{\leftarrow} b$, which means $a_{k+1} \leftarrow (1 - \alpha) \cdot a_k + \alpha \cdot b_k$. So, the first line of Equation 18-4 can be rewritten like this: $V(s) \underset{\alpha}{\leftarrow} r + \gamma \cdot V(s')$.

TD Learning has many similarities with Stochastic Gradient Descent, in particular the fact that it handles one sample at a time. Moreover, just like Stochastic GD, it can only truly converge if you gradually reduce the learning rate (otherwise it will keep bouncing around the optimum Q-Values).

For each state s, this algorithm simply keeps track of a running average of the immediate rewards the agent gets upon leaving that state, plus the rewards it expects to get later (assuming it acts optimally).

Q-Learning

Similarly, the Q-Learning algorithm is an adaptation of the Q-Value Iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown (see Equation 18-5). Q-Learning works by watching an agent play (e.g., randomly) and gradually improving its estimates of the Q-Values. Once it has

accurate Q-Value estimates (or close enough), then the optimal policy is choosing the action that has the highest Q-Value (i.e., the greedy policy).

Equation 18-5. Q-Learning algorithm

$$Q\left(s, a\right) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} \quad Q\left(s', a'\right)$$

For each state-action pair (s, a), this algorithm keeps track of a running average of the rewards r the agent gets upon leaving the state s with action a, plus the sum of discounted future rewards it expects to get. To estimate this sum, we take the maximum of the Q-Value estimates for the next state s', since we assume that the target policy would act optimally from then on.

Let's implement the Q-Learning algorithm. First, we will need to make an agent explore the environment. For this, we need a step function so that the agent can execute one action and get the resulting state and reward:

```
def step(state, action):
    probas = transition_probabilities[state][action]
    next_state = np.random.choice([0, 1, 2], p=probas)
    reward = rewards[state][action][next_state]
    return next_state, reward
```

Now let's implement the agent's exploration policy. Since the state space is pretty small, a simple random policy will be sufficient. If we run the algorithm for long enough, the agent will visit every state many times, and it will also try every possible action many times:

```
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

Next, after we initialize the Q-Values just like earlier, we are ready to run the Q-Learning algorithm with learning rate decay (using power scheduling, introduced in Chapter 11):

```
alpha0 = 0.05 # initial learning rate
decay = 0.005 # learning rate decay
gamma = 0.90 # discount factor
state = 0 # initial state

for iteration in range(10000):
    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = np.max(Q_values[next_state])
    alpha = alpha0 / (1 + iteration * decay)
    Q_values[state, action] *= 1 - alpha
    Q_values[state, action] += alpha * (reward + gamma * next_value)
    state = next_state
```

This algorithm will converge to the optimal Q-Values, but it will take many iterations, and possibly quite a lot of hyperparameter tuning. As you can see in Figure 18-9, the Q-Value Iteration algorithm (left) converges very quickly, in fewer than 20 iterations, while the Q-Learning algorithm (right) takes about 8,000 iterations to converge. Obviously, not knowing the transition probabilities or the rewards makes finding the optimal policy significantly harder!

Figure 18-9. The Q-Value Iteration algorithm (left) versus the Q-Learning algorithm (right)

The Q-Learning algorithm is called an *off-policy* algorithm because the policy being trained is not necessarily the one being executed: in the previous code example, the policy being executed (the exploration policy) is completely random, while the policy being trained will always choose the actions with the highest Q-Values. Conversely, the Policy Gradients algorithm is an *on-policy* algorithm: it explores the world using the policy being trained. It is somewhat surprising that Q-Learning is capable of learning the optimal policy by just watching an agent act randomly (imagine learning to play golf when your teacher is a drunk monkey). Can we do better?

Exploration Policies

Of course, Q-Learning can work only if the exploration policy explores the MDP thoroughly enough. Although a purely random policy is guaranteed to eventually visit every state and every transition many times, it may take an extremely long time to do so. Therefore, a better option is to use the *ε-greedy policy* (ε is epsilon): at each step it acts randomly with probability ε, or greedily with probability $1-\varepsilon$ (i.e., choosing the action with the highest Q-Value). The advantage of the ε-greedy policy (compared to a completely random policy) is that it will spend more and more time exploring the interesting parts of the environment, as the Q-Value estimates get better and better, while still spending some time visiting unknown regions of the MDP. It is quite common to start with a high value for ε (e.g., 1.0) and then gradually reduce it (e.g., down to 0.05).

Alternatively, rather than relying only on chance for exploration, another approach is to encourage the exploration policy to try actions that it has not tried much before. This can be implemented as a bonus added to the Q-Value estimates, as shown in Equation 18-6.

Equation 18-6. Q-Learning using an exploration function

$$Q\left(s, a\right) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} \quad f(Q(s', a'), N(s', a'))$$

In this equation:

- $N(s', a')$ counts the number of times the action a' was chosen in state s'.
- $f(Q, N)$ is an *exploration function*, such as $f(Q, N) = Q + \kappa/(1 + N)$, where κ is a curiosity hyperparameter that measures how much the agent is attracted to the unknown.

Approximate Q-Learning and Deep Q-Learning

The main problem with Q-Learning is that it does not scale well to large (or even medium) MDPs with many states and actions. For example, suppose you wanted to use Q-Learning to train an agent to play *Ms. Pac-Man* (see Figure 18-1). There are about 150 pellets that Ms. Pac-Man can eat, each of which can be present or absent (i.e., already eaten). So, the number of possible states is greater than $2^{150} \approx 10^{45}$. And if you add all the possible combinations of positions for all the ghosts and Ms. Pac-Man, the number of possible states becomes larger than the number of atoms in our planet, so there's absolutely no way you can keep track of an estimate for every single Q-Value.

The solution is to find a function $Q_{\boldsymbol{\theta}}(s, a)$ that approximates the Q-Value of any state-action pair (s, a) using a manageable number of parameters (given by the parameter vector $\boldsymbol{\theta}$). This is called *Approximate Q-Learning*. For years it was recommended to use linear combinations of handcrafted features extracted from the state (e.g., distance of the closest ghosts, their directions, and so on) to estimate Q-Values, but in 2013, DeepMind showed that using deep neural networks can work much better, especially for complex problems, and it does not require any feature engineering. A DNN used to estimate Q-Values is called a *Deep Q-Network* (DQN), and using a DQN for Approximate Q-Learning is called *Deep Q-Learning*.

Now, how can we train a DQN? Well, consider the approximate Q-Value computed by the DQN for a given state-action pair (s, a). Thanks to Bellman, we know we want this approximate Q-Value to be as close as possible to the reward r that we actually observe after playing action a in state s, plus the discounted value of playing optimally

from then on. To estimate this sum of future discounted rewards, we can simply execute the DQN on the next state s' and for all possible actions a'. We get an approximate future Q-Value for each possible action. We then pick the highest (since we assume we will be playing optimally) and discount it, and this gives us an estimate of the sum of future discounted rewards. By summing the reward r and the future discounted value estimate, we get a target Q-Value $y(s, a)$ for the state-action pair (s, a), as shown in Equation 18-7.

Equation 18-7. Target Q-Value

$$Q_{\text{target}}\left(s, a\right) = r + \gamma \cdot \max_{a'} \quad Q_{\boldsymbol{\theta}}\left(s', a'\right)$$

With this target Q-Value, we can run a training step using any Gradient Descent algorithm. Specifically, we generally try to minimize the squared error between the estimated Q-Value $Q(s, a)$ and the target Q-Value (or the Huber loss to reduce the algorithm's sensitivity to large errors). And that's all for the basic Deep Q-Learning algorithm! Let's see how to implement it to solve the CartPole environment.

Implementing Deep Q-Learning

The first thing we need is a Deep Q-Network. In theory, you need a neural net that takes a state-action pair and outputs an approximate Q-Value, but in practice it's much more efficient to use a neural net that takes a state and outputs one approximate Q-Value for each possible action. To solve the CartPole environment, we do not need a very complicated neural net; a couple of hidden layers will do:

```
env = gym.make("CartPole-v0")
input_shape = [4] # == env.observation_space.shape
n_outputs = 2 # == env.action_space.n

model = keras.models.Sequential([
    keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    keras.layers.Dense(32, activation="elu"),
    keras.layers.Dense(n_outputs)
])
```

To select an action using this DQN, we pick the action with the largest predicted Q-Value. To ensure that the agent explores the environment, we will use an ε-greedy policy (i.e., we will choose a random action with probability ε):

```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(2)
    else:
        Q_values = model.predict(state[np.newaxis])
        return np.argmax(Q_values[0])
```

Instead of training the DQN based only on the latest experiences, we will store all experiences in a *replay buffer* (or *replay memory*), and we will sample a random training batch from it at each training iteration. This helps reduce the correlations between the experiences in a training batch, which tremendously helps training. For this, we will just use a deque list:

```
from collections import deque

replay_buffer = deque(maxlen=2000)
```

A *deque* is a linked list, where each element points to the next one and to the previous one. It makes inserting and deleting items very fast, but the longer the deque is, the slower random access will be. If you need a very large replay buffer, use a circular buffer; see the "Deque vs Rotating List" section of the notebook for an implementation.

Each experience will be composed of five elements: a state, the action the agent took, the resulting reward, the next state it reached, and finally a Boolean indicating whether the episode ended at that point (`done`). We will need a small function to sample a random batch of experiences from the replay buffer. It will return five NumPy arrays corresponding to the five experience elements:

```
def sample_experiences(batch_size):
    indices = np.random.randint(len(replay_buffer), size=batch_size)
    batch = [replay_buffer[index] for index in indices]
    states, actions, rewards, next_states, dones = [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(5)]
    return states, actions, rewards, next_states, dones
```

Let's also create a function that will play a single step using the ε-greedy policy, then store the resulting experience in the replay buffer:

```
def play_one_step(env, state, epsilon):
    action = epsilon_greedy_policy(state, epsilon)
    next_state, reward, done, info = env.step(action)
    replay_buffer.append((state, action, reward, next_state, done))
    return next_state, reward, done, info
```

Finally, let's create one last function that will sample a batch of experiences from the replay buffer and train the DQN by performing a single Gradient Descent step on this batch:

```
batch_size = 32
discount_factor = 0.95
optimizer = keras.optimizers.Adam(lr=1e-3)
loss_fn = keras.losses.mean_squared_error
```

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    max_next_Q_values = np.max(next_Q_values, axis=1)
    target_Q_values = (rewards +
                       (1 - dones) * discount_factor * max_next_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Let's go through this code:

- First we define some hyperparameters, and we create the optimizer and the loss function.
- Then we create the `training_step()` function. It starts by sampling a batch of experiences, then it uses the DQN to predict the Q-Value for each possible action in each experience's next state. Since we assume that the agent will be playing optimally, we only keep the maximum Q-Value for each next state. Next, we use Equation 18-7 to compute the target Q-Value for each experience's state-action pair.
- Next, we want to use the DQN to compute the Q-Value for each experienced state-action pair. However, the DQN will also output the Q-Values for the other possible actions, not just for the action that was actually chosen by the agent. So we need to mask out all the Q-Values we do not need. The `tf.one_hot()` function makes it easy to convert an array of action indices into such a mask. For example, if the first three experiences contain actions 1, 1, 0, respectively, then the mask will start with `[[0, 1], [0, 1], [1, 0],...]`. We can then multiply the DQN's output with this mask, and this will zero out all the Q-Values we do not want. We then sum over axis 1 to get rid of all the zeros, keeping only the Q-Values of the experienced state-action pairs. This gives us the `Q_values` tensor, containing one predicted Q-Value for each experience in the batch.
- Then we compute the loss: it is the mean squared error between the target and predicted Q-Values for the experienced state-action pairs.
- Finally, we perform a Gradient Descent step to minimize the loss with regard to the model's trainable variables.

This was the hardest part. Now training the model is straightforward:

```
for episode in range(600):
    obs = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, info = play_one_step(env, obs, epsilon)
        if done:
            break
    if episode > 50:
        training_step(batch_size)
```

We run 600 episodes, each for a maximum of 200 steps. At each step, we first compute the `epsilon` value for the ε-greedy policy: it will go from 1 down to 0.01, linearly, in a bit under 500 episodes. Then we call the `play_one_step()` function, which will use the ε-greedy policy to pick an action, then execute it and record the experience in the replay buffer. If the episode is done, we exit the loop. Finally, if we are past the 50th episode, we call the `training_step()` function to train the model on one batch sampled from the replay buffer. The reason we play 50 episodes without training is to give the replay buffer some time to fill up (if we don't wait enough, then there will not be enough diversity in the replay buffer). And that's it, we just implemented the Deep Q-Learning algorithm!

Figure 18-10 shows the total rewards the agent got during each episode.

Figure 18-10. Learning curve of the Deep Q-Learning algorithm

As you can see, the algorithm made no apparent progress at all for almost 300 episodes (in part because ε was very high at the beginning), then its performance suddenly skyrocketed up to 200 (which is the maximum possible performance in this environment). That's great news: the algorithm worked fine, and it actually ran much faster than the Policy Gradient algorithm! But wait… just a few episodes later, it forgot everything it knew, and its performance dropped below 25! This is called

catastrophic forgetting, and it is one of the big problems facing virtually all RL algorithms: as the agent explores the environment, it updates its policy, but what it learns in one part of the environment may break what it learned earlier in other parts of the environment. The experiences are quite correlated, and the learning environment keeps changing—this is not ideal for Gradient Descent! If you increase the size of the replay buffer, the algorithm will be less subject to this problem. Reducing the learning rate may also help. But the truth is, Reinforcement Learning is hard: training is often unstable, and you may need to try many hyperparameter values and random seeds before you find a combination that works well. For example, if you try changing the number of neurons per layer in the preceding from 32 to 30 or 34, the performance will never go above 100 (the DQN may be more stable with one hidden layer instead of two).

Reinforcement Learning is notoriously difficult, largely because of the training instabilities and the huge sensitivity to the choice of hyperparameter values and random seeds.[13] As the researcher Andrej Karpathy put it: "[Supervised learning] wants to work. […] RL must be forced to work." You will need time, patience, perseverance, and perhaps a bit of luck too. This is a major reason RL is not as widely adopted as regular Deep Learning (e.g., convolutional nets). But there are a few real-world applications, beyond AlphaGo and Atari games: for example, Google uses RL to optimize its data-center costs, and it is used in some robotics applications, for hyperparameter tuning, and in recommender systems.

You might wonder why we didn't plot the loss. It turns out that loss is a poor indicator of the model's performance. The loss might go down, yet the agent might perform worse (e.g., this can happen when the agent gets stuck in one small region of the environment, and the DQN starts overfitting this region). Conversely, the loss could go up, yet the agent might perform better (e.g., if the DQN was underestimating the Q-Values, and it starts correctly increasing its predictions, the agent will likely perform better, getting more rewards, but the loss might increase because the DQN also sets the targets, which will be larger too).

The basic Deep Q-Learning algorithm we've been using so far would be too unstable to learn to play Atari games. So how did DeepMind do it? Well, they tweaked the algorithm!

13 A great 2018 post by Alex Irpan nicely lays out RL's biggest difficulties and limitations.

Deep Q-Learning Variants

Let's look at a few variants of the Deep Q-Learning algorithm that can stabilize and speed up training.

Fixed Q-Value Targets

In the basic Deep Q-Learning algorithm, the model is used both to make predictions and to set its own targets. This can lead to a situation analogous to a dog chasing its own tail. This feedback loop can make the network unstable: it can diverge, oscillate, freeze, and so on. To solve this problem, in their 2013 paper the DeepMind researchers used two DQNs instead of one: the first is the *online model*, which learns at each step and is used to move the agent around, and the other is the *target model* used only to define the targets. The target model is just a clone of the online model:

```
target = keras.models.clone_model(model)
target.set_weights(model.get_weights())
```

Then, in the `training_step()` function, we just need to change one line to use the target model instead of the online model when computing the Q-Values of the next states:

```
next_Q_values = target.predict(next_states)
```

Finally, in the training loop, we must copy the weights of the online model to the target model, at regular intervals (e.g., every 50 episodes):

```
if episode % 50 == 0:
    target.set_weights(model.get_weights())
```

Since the target model is updated much less often than the online model, the Q-Value targets are more stable, the feedback loop we discussed earlier is dampened, and its effects are less severe. This approach was one of the DeepMind researchers' main contributions in their 2013 paper, allowing agents to learn to play Atari games from raw pixels. To stabilize training, they used a tiny learning rate of 0.00025, they updated the target model only every 10,000 steps (instead of the 50 in the previous code example), and they used a very large replay buffer of 1 million experiences. They decreased `epsilon` very slowly, from 1 to 0.1 in 1 million steps, and they let the algorithm run for 50 million steps.

Later in this chapter, we will use the TF-Agents library to train a DQN agent to play *Breakout* using these hyperparameters, but before we get there, let's take a look at another DQN variant that managed to beat the state of the art once more.

Double DQN

In a 2015 paper,[14] DeepMind researchers tweaked their DQN algorithm, increasing its performance and somewhat stabilizing training. They called this variant *Double DQN*. The update was based on the observation that the target network is prone to overestimating Q-Values. Indeed, suppose all actions are equally good: the Q-Values estimated by the target model should be identical, but since they are approximations, some may be slightly greater than others, by pure chance. The target model will always select the largest Q-Value, which will be slightly greater than the mean Q-Value, most likely overestimating the true Q-Value (a bit like counting the height of the tallest random wave when measuring the depth of a pool). To fix this, they proposed using the online model instead of the target model when selecting the best actions for the next states, and using the target model only to estimate the Q-Values for these best actions. Here is the updated `training_step()` function:

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    best_next_actions = np.argmax(next_Q_values, axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    next_best_Q_values = (target.predict(next_states) * next_mask).sum(axis=1)
    target_Q_values = (rewards +
                       (1 - dones) * discount_factor * next_best_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    [...] # the rest is the same as earlier
```

Just a few months later, another improvement to the DQN algorithm was proposed.

Prioritized Experience Replay

Instead of sampling experiences *uniformly* from the replay buffer, why not sample important experiences more frequently? This idea is called *importance sampling* (IS) or *prioritized experience replay* (PER), and it was introduced in a 2015 paper[15] by DeepMind researchers (once again!).

More specifically, experiences are considered "important" if they are likely to lead to fast learning progress. But how can we estimate this? One reasonable approach is to measure the magnitude of the TD error $\delta = r + \gamma \cdot V(s') - V(s)$. A large TD error indicates that a transition (s, r, s') is very surprising, and thus probably worth learning

14 Hado van Hasselt et al., "Deep Reinforcement Learning with Double Q-Learning," *Proceedings of the 30th AAAI Conference on Artificial Intelligence* (2015): 2094–2100.

15 Tom Schaul et al., "Prioritized Experience Replay," arXiv preprint arXiv:1511.05952 (2015).

from.[16] When an experience is recorded in the replay buffer, its priority is set to a very large value, to ensure that it gets sampled at least once. However, once it is sampled (and every time it is sampled), the TD error δ is computed, and this experience's priority is set to $p = |\delta|$ (plus a small constant to ensure that every experience has a nonzero probability of being sampled). The probability P of sampling an experience with priority p is proportional to p^{ζ}, where ζ is a hyperparameter that controls how greedy we want importance sampling to be: when $\zeta = 0$, we just get uniform sampling, and when $\zeta = 1$, we get full-blown importance sampling. In the paper, the authors used $\zeta = 0.6$, but the optimal value will depend on the task.

There's one catch, though: since the samples will be biased toward important experiences, we must compensate for this bias during training by downweighting the experiences according to their importance, or else the model will just overfit the important experiences. To be clear, we want important experiences to be sampled more often, but this also means we must give them a lower weight during training. To do this, we define each experience's training weight as $w = (n\ P)^{-\beta}$, where n is the number of experiences in the replay buffer, and β is a hyperparameter that controls how much we want to compensate for the importance sampling bias (0 means not at all, while 1 means entirely). In the paper, the authors used $\beta = 0.4$ at the beginning of training and linearly increased it to $\beta = 1$ by the end of training. Again, the optimal value will depend on the task, but if you increase one, you will usually want to increase the other as well.

Now let's look at one last important variant of the DQN algorithm.

Dueling DQN

The *Dueling DQN* algorithm (DDQN, not to be confused with Double DQN, although both techniques can easily be combined) was introduced in yet another 2015 paper[17] by DeepMind researchers. To understand how it works, we must first note that the Q-Value of a state-action pair (s, a) can be expressed as $Q(s, a) = V(s) + A(s, a)$, where $V(s)$ is the value of state s and $A(s, a)$ is the *advantage* of taking the action a in state s, compared to all other possible actions in that state. Moreover, the value of a state is equal to the Q-Value of the best action a^* for that state (since we assume the optimal policy will pick the best action), so $V(s) = Q(s, a^*)$, which implies that $A(s, a^*) = 0$. In a Dueling DQN, the model estimates both the value of the state and the advantage of each possible action. Since the best action should have an advantage of 0, the model subtracts the maximum predicted advantage from all pre-

16 It could also just be that the rewards are noisy, in which case there are better methods for estimating an experience's importance (see the paper for some examples).

17 Ziyu Wang et al., "Dueling Network Architectures for Deep Reinforcement Learning," arXiv preprint arXiv: 1511.06581 (2015).

dicted advantages. Here is a simple Dueling DQN model, implemented using the Functional API:

```
K = keras.backend
input_states = keras.layers.Input(shape=[4])
hidden1 = keras.layers.Dense(32, activation="elu")(input_states)
hidden2 = keras.layers.Dense(32, activation="elu")(hidden1)
state_values = keras.layers.Dense(1)(hidden2)
raw_advantages = keras.layers.Dense(n_outputs)(hidden2)
advantages = raw_advantages - K.max(raw_advantages, axis=1, keepdims=True)
Q_values = state_values + advantages
model = keras.Model(inputs=[input_states], outputs=[Q_values])
```

The rest of the algorithm is just the same as earlier. In fact, you can build a Double Dueling DQN and combine it with prioritized experience replay! More generally, many RL techniques can be combined, as DeepMind demonstrated in a 2017 paper.[18] The paper's authors combined six different techniques into an agent called *Rainbow*, which largely outperformed the state of the art.

Unfortunately, implementing all of these techniques, debugging them, fine-tuning them, and of course training the models can require a huge amount of work. So instead of reinventing the wheel, it is often best to reuse scalable and well-tested libraries, such as TF-Agents.

The TF-Agents Library

The TF-Agents library is a Reinforcement Learning library based on TensorFlow, developed at Google and open sourced in 2018. Just like OpenAI Gym, it provides many off-the-shelf environments (including wrappers for all OpenAI Gym environments), plus it supports the PyBullet library (for 3D physics simulation), DeepMind's DM Control library (based on MuJoCo's physics engine), and Unity's ML-Agents library (simulating many 3D environments). It also implements many RL algorithms, including REINFORCE, DQN, and DDQN, as well as various RL components such as efficient replay buffers and metrics. It is fast, scalable, easy to use, and customizable: you can create your own environments and neural nets, and you can customize pretty much any component. In this section we will use TF-Agents to train an agent to play *Breakout*, the famous Atari game (see Figure 18-11[19]), using the DQN algorithm (you can easily switch to another algorithm if you prefer).

18 Matteo Hessel et al., "Rainbow: Combining Improvements in Deep Reinforcement Learning," arXiv preprint arXiv:1710.02298 (2017): 3215–3222.

19 If you don't know this game, it's simple: a ball bounces around and breaks bricks when it touches them. You control a paddle near the bottom of the screen. The paddle can go left or right, and you must get the ball to break every brick, while preventing it from touching the bottom of the screen.

Figure 18-11. The famous Breakout game

Installing TF-Agents

Let's start by installing TF-Agents. This can be done using pip (as always, if you are using a virtual environment, make sure to activate it first; if not, you will need to use the `--user` option, or have administrator rights):

```
$ python3 -m pip install -U tf-agents
```

At the time of this writing, TF-Agents is still quite new and improving every day, so the API may change a bit by the time you read this—but the big picture should remain the same, as well as most of the code. If anything breaks, I will update the Jupyter notebook accordingly, so make sure to check it out.

Next, let's create a TF-Agents environment that will just wrap OpenAI GGym's Breakout environment. For this, you must first install OpenAI Gym's Atari dependencies:

```
$ python3 -m pip install -U 'gym[atari]'
```

Among other libraries, this command will install `atari-py`, which is a Python interface for the Arcade Learning Environment (ALE), a framework built on top of the Atari 2600 emulator Stella.

TF-Agents Environments

If everything went well, you should be able to import TF-Agents and create a Breakout environment:

```
>>> from tf_agents.environments import suite_gym
>>> env = suite_gym.load("Breakout-v4")
>>> env
<tf_agents.environments.wrappers.TimeLimit at 0x10c523c18>
```

This is just a wrapper around an OpenAI Gym environment, which you can access through the `gym` attribute:

```
>>> env.gym
<gym.envs.atari.atari_env.AtariEnv at 0x24dcab940>
```

TF-Agents environments are very similar to OpenAI Gym environments, but there are a few differences. First, the `reset()` method does not return an observation; instead it returns a `TimeStep` object that wraps the observation, as well as some extra information:

```
>>> env.reset()
TimeStep(step_type=array(0, dtype=int32),
         reward=array(0., dtype=float32),
         discount=array(1., dtype=float32),
         observation=array([[[0., 0., 0.], [0., 0., 0.],...]]], dtype=float32))
```

The `step()` method returns a `TimeStep` object as well:

```
>>> env.step(1) # Fire
TimeStep(step_type=array(1, dtype=int32),
         reward=array(0., dtype=float32),
         discount=array(1., dtype=float32),
         observation=array([[[0., 0., 0.], [0., 0., 0.],...]]], dtype=float32))
```

The `reward` and `observation` attributes are self-explanatory, and they are the same as for OpenAI Gym (except the `reward` is represented as a NumPy array). The `step_type` attribute is equal to 0 for the first time step in the episode, 1 for intermediate time steps, and 2 for the final time step. You can call the time step's `is_last()` method to check whether it is the final one or not. Lastly, the `discount` attribute indicates the discount factor to use at this time step. In this example it is equal to 1, so there will be no discount at all. You can define the discount factor by setting the `discount` parameter when loading the environment.

At any time, you can access the environment's current time step by calling its `current_time_step()` method.

Environment Specifications

Conveniently, a TF-Agents environment provides the specifications of the observations, actions, and time steps, including their shapes, data types, and names, as well as their minimum and maximum values:

```
>>> env.observation_spec()
BoundedArraySpec(shape=(210, 160, 3), dtype=dtype('float32'), name=None,
                 minimum=[[[0. 0. 0.], [0. 0. 0.],...]],
                 maximum=[[[255., 255., 255.], [255., 255., 255.], ...]])
>>> env.action_spec()
BoundedArraySpec(shape=(), dtype=dtype('int64'), name=None,
                 minimum=0, maximum=3)
>>> env.time_step_spec()
TimeStep(step_type=ArraySpec(shape=(), dtype=dtype('int32'), name='step_type'),
         reward=ArraySpec(shape=(), dtype=dtype('float32'), name='reward'),
         discount=BoundedArraySpec(shape=(), ..., minimum=0.0, maximum=1.0),
         observation=BoundedArraySpec(shape=(210, 160, 3), ...))
```

As you can see, the observations are simply screenshots of the Atari screen, represented as NumPy arrays of shape [210, 160, 3]. To render an environment, you can call `env.render(mode="human")`, and if you want to get back the image in the form of a NumPy array, just call `env.render(mode="rgb_array")` (unlike in OpenAI Gym, this is the default mode).

There are four actions available. Gym's Atari environments have an extra method that you can call to know what each action corresponds to:

```
>>> env.gym.get_action_meanings()
['NOOP', 'FIRE', 'RIGHT', 'LEFT']
```

Specs can be instances of a specification class, nested lists, or dictionaries of specs. If the specification is nested, then the specified object must match the specification's nested structure. For example, if the observation spec is `{"sensors": ArraySpec(shape=[2]), "camera": ArraySpec(shape=[100, 100])}`, then a valid observation would be `{"sensors": np.array([1.5, 3.5]), "camera": np.array(...)}`. The `tf.nest` package provides tools to handle such nested structures (a.k.a. *nests*).

The observations are quite large, so we will downsample them and also convert them to grayscale. This will speed up training and use less RAM. For this, we can use an *environment wrapper*.

Environment Wrappers and Atari Preprocessing

TF-Agents provides several environment wrappers in the `tf_agents.environments.wrappers` package. As their name suggests, they wrap an environment, forwarding every call to it, but also adding some extra functionality. Here are some of the available wrappers:

`ActionClipWrapper`
: Clips the actions to the action spec.

`ActionDiscretizeWrapper`
: Quantizes a continuous action space to a discrete action space. For example, if the original environment's action space is the continuous range from –1.0 to +1.0, but you want to use an algorithm that only supports discrete action spaces, such as a DQN, then you can wrap the environment using `discrete_env = ActionDiscretizeWrapper(env, num_actions=5)`, and the new `discrete_env` will have a discrete action space with five possible actions: 0, 1, 2, 3, 4. These actions correspond to the actions –1.0, –0.5, 0.0, 0.5, and 1.0 in the original environment.

`ActionRepeat`
: Repeats each action over *n* steps, while accumulating the rewards. In many environments, this can speed up training significantly.

`RunStats`
: Records environment statistics such as the number of steps and the number of episodes.

`TimeLimit`
: Interrupts the environment if it runs for longer than a maximum number of steps.

`VideoWrapper`
: Records a video of the environment.

To create a wrapped environment, you must create a wrapper, passing the wrapped environment to the constructor. That's all! For example, the following code will wrap our environment in an `ActionRepeat` wrapper so that every action is repeated four times:

```
from tf_agents.environments.wrappers import ActionRepeat

repeating_env = ActionRepeat(env, times=4)
```

OpenAI Gym has some environment wrappers of its own in the `gym.wrappers` package. They are meant to wrap Gym environments, though, not TF-Agents environments, so to use them you must first wrap the Gym environment with a Gym wrapper, then wrap the resulting environment with a TF-Agents wrapper. The `suite_gym.wrap_env()` function will do this for you, provided you give it a Gym environment and a list of Gym wrappers and/or a list of TF-Agents wrappers. Alternatively, the `suite_gym.load()` function will both create the Gym environment and wrap it for you, if you give it some wrappers. Each wrapper will be created without any arguments, so if you want to set some arguments, you must pass a `lambda`. For example, the following code creates a Breakout environment that will run for a maximum of 10,000 steps during each episode, and each action will be repeated four times:

```
from gym.wrappers import TimeLimit

limited_repeating_env = suite_gym.load(
    "Breakout-v4",
    gym_env_wrappers=[lambda env: TimeLimit(env, max_episode_steps=10000)],
    env_wrappers=[lambda env: ActionRepeat(env, times=4)])
```

For Atari environments, some standard preprocessing steps are applied in most papers that use them, so TF-Agents provides a handy `AtariPreprocessing` wrapper that implements them. Here is the list of preprocessing steps it supports:

Grayscale and downsampling
: Observations are converted to grayscale and downsampled (by default to 84 × 84 pixels).

Max pooling
: The last two frames of the game are max-pooled using a 1 × 1 filter. This is to remove the flickering that occurs in some Atari games due to the limited number of sprites that the Atari 2600 could display in each frame.

Frame skipping
: The agent only gets to see every n frames of the game (by default $n = 4$), and its actions are repeated for each frame, collecting all the rewards. This effectively speeds up the game from the perspective of the agent, and it also speeds up training because rewards are less delayed.

End on life lost
: In some games, the rewards are just based on the score, so the agent gets no immediate penalty for losing a life. One solution is to end the game immediately whenever a life is lost. There is some debate over the actual benefits of this strategy, so it is off by default.

Since the default Atari environment already applies random frame skipping and max pooling, we will need to load the raw, nonskipping variant called `"BreakoutNoFrameskip-v4"`. Moreover, a single frame from the *Breakout* game is insufficient to know the direction and speed of the ball, which will make it very difficult for the agent to play the game properly (unless it is an RNN agent, which preserves some internal state between steps). One way to handle this is to use an environment wrapper that will output observations composed of multiple frames stacked on top of each other along the channels dimension. This strategy is implemented by the `FrameStack4` wrapper, which returns stacks of four frames. Let's create the wrapped Atari environment!

```
from tf_agents.environments import suite_atari
from tf_agents.environments.atari_preprocessing import AtariPreprocessing
from tf_agents.environments.atari_wrappers import FrameStack4

max_episode_steps = 27000 # <=> 108k ALE frames since 1 step = 4 frames
environment_name = "BreakoutNoFrameskip-v4"

env = suite_atari.load(
    environment_name,
    max_episode_steps=max_episode_steps,
    gym_env_wrappers=[AtariPreprocessing, FrameStack4])
```

The result of all this preprocessing is shown in Figure 18-12. You can see that the resolution is much lower, but sufficient to play the game. Moreover, frames are stacked along the channels dimension, so red represents the frame from three steps ago, green is two steps ago, blue is the previous frame, and pink is the current frame.[20] From this single observation, the agent can see that the ball is going toward the lower-left corner, and that it should continue to move the paddle to the left (as it did in the previous steps).

Figure 18-12. Preprocessed Breakout observation

Lastly, we can wrap the environment inside a `TFPyEnvironment`:

```
from tf_agents.environments.tf_py_environment import TFPyEnvironment

tf_env = TFPyEnvironment(env)
```

This will make the environment usable from within a TensorFlow graph (under the hood, this class relies on `tf.py_function()`, which allows a graph to call arbitrary

20 Since there are only three primary colors, you cannot just display an image with four color channels. For this reason, I combined the last channel with the first three to get the RGB image represented here. Pink is actually a mix of blue and red, but the agent sees four independent channels.

Python code). Thanks to the `TFPyEnvironment` class, TF-Agents supports both pure Python environments and TensorFlow-based environments. More generally, TF-Agents supports and provides both pure Python and TensorFlow-based components (agents, replay buffers, metrics, and so on).

Now that we have a nice Breakout environment, with all the appropriate preprocessing and TensorFlow support, we must create the DQN agent and the other components we will need to train it. Let's look at the architecture of the system we will build.

Training Architecture

A TF-Agents training program is usually split into two parts that run in parallel, as you can see in Figure 18-13: on the left, a *driver* explores the *environment* using a *collect policy* to choose actions, and it collects *trajectories* (i.e., experiences), sending them to an *observer*, which saves them to a *replay buffer*; on the right, an *agent* pulls batches of trajectories from the replay buffer and trains some *networks*, which the collect policy uses. In short, the left part explores the environment and collects trajectories, while the right part learns and updates the collect policy.

Figure 18-13. A typical TF-Agents training architecture

This figure begs a few questions, which I'll attempt to answer here:

- Why are there multiple environments? Instead of exploring a single environment, you generally want the driver to explore multiple copies of the environment in parallel, taking advantage of the power of all your CPU cores, keeping

the training GPUs busy, and providing less-correlated trajectories to the training algorithm.

- What is a *trajectory*? It is a concise representation of a *transition* from one time step to the next, or a sequence of consecutive transitions from time step n to time step $n + t$. The trajectories collected by the driver are passed to the observer, which saves them in the replay buffer, and they are later sampled by the agent and used for training.
- Why do we need an observer? Can't the driver save the trajectories directly? Indeed, it could, but this would make the architecture less flexible. For example, what if you don't want to use a replay buffer? What if you want to use the trajectories for something else, like computing metrics? In fact, an observer is just any function that takes a trajectory as an argument. You can use an observer to save the trajectories to a replay buffer, or to save them to a TFRecord file (see Chapter 13), or to compute metrics, or for anything else. Moreover, you can pass multiple observers to the driver, and it will broadcast the trajectories to all of them.

Although this architecture is the most common, you can customize it as you please, and even replace some components with your own. In fact, unless you are researching new RL algorithms, you will most likely want to use a custom environment for your task. For this, you just need to create a custom class that inherits from the `PyEnvironment` class in the `tf_agents.environments.py_environment` package and overrides the appropriate methods, such as `action_spec()`, `observation_spec()`, `_reset()`, and `_step()` (see the "Creating a Custom TF_Agents Environment" section of the notebook for an example).

Now we will create all these components: first the Deep Q-Network, then the DQN agent (which will take care of creating the collect policy), then the replay buffer and the observer to write to it, then a few training metrics, then the driver, and finally the dataset. Once we have all the components in place, we will populate the replay buffer with some initial trajectories, then we will run the main training loop. So, let's start by creating the Deep Q-Network.

Creating the Deep Q-Network

The TF-Agents library provides many networks in the `tf_agents.networks` package and its subpackages. We will use the `tf_agents.networks.q_network.QNetwork` class:

```
from tf_agents.networks.q_network import QNetwork

preprocessing_layer = keras.layers.Lambda(
                          lambda obs: tf.cast(obs, np.float32) / 255.)
conv_layer_params=[(32, (8, 8), 4), (64, (4, 4), 2), (64, (3, 3), 1)]
fc_layer_params=[512]

q_net = QNetwork(
    tf_env.observation_spec(),
    tf_env.action_spec(),
    preprocessing_layers=preprocessing_layer,
    conv_layer_params=conv_layer_params,
    fc_layer_params=fc_layer_params)
```

This `QNetwork` takes an observation as input and outputs one Q-Value per action, so we must give it the specifications of the observations and the actions. It starts with a preprocessing layer: a simple `Lambda` layer that casts the observations to 32-bit floats and normalizes them (the values will range from 0.0 to 1.0). The observations contain unsigned bytes, which use 4 times less space than 32-bit floats, which is why we did not cast the observations to 32-bit floats earlier; we want to save RAM in the replay buffer. Next, the network applies three convolutional layers: the first has 32 8×8 filters and uses a stride of 4, the second has 64 4×4 filters and a stride of 2, and the third has 64 3×3 filters and a stride of 1. Lastly, it applies a dense layer with 512 units, followed by a dense output layer with 4 units, one per Q-Value to output (i.e., one per action). All convolutional layers and all dense layers except the output layer use the ReLU activation function by default (you can change this by setting the `activation_fn` argument). The output layer does not use any activation function.

Under the hood, a `QNetwork` is composed of two parts: an encoding network that processes the observations, followed by a dense output layer that outputs one Q-Value per action. TF-Agent's `EncodingNetwork` class implements a neural network architecture found in various agents (see Figure 18-14).

It may have one or more inputs. For example, if each observation is composed of some sensor data plus an image from a camera, you will have two inputs. Each input may require some preprocessing steps, in which case you can specify a list of Keras layers via the `preprocessing_layers` argument, with one preprocessing layer per input, and the network will apply each layer to the corresponding input (if an input requires multiple layers of preprocessing, you can pass a whole model, since a Keras model can always be used as a layer). If there are two inputs or more, you must also pass an extra layer via the `preprocessing_combiner` argument, to combine the outputs from the preprocessing layers into a single output.

Next, the encoding network will optionally apply a list of convolutions sequentially, provided you specify their parameters via the `conv_layer_params` argument. This must be a list composed of 3-tuples (one per convolutional layer) indicating the

number of filters, the kernel size, and the stride. After these convolutional layers, the encoding network will optionally apply a sequence of dense layers, if you set the `fc_layer_params` argument: it must be a list containing the number of neurons for each dense layer. Optionally, you can also pass a list of dropout rates (one per dense layer) via the `dropout_layer_params` argument if you want to apply dropout after each dense layer. The `QNetwork` takes the output of this encoding network and passes it to the dense output layer (with one unit per action).

Figure 18-14. Architecture of an encoding network

The `QNetwork` class is flexible enough to build many different architectures, but you can always build your own network class if you need extra flexibility: extend the `tf_agents.networks.Network` class and implement it like a regular custom Keras layer. The `tf_agents.networks.Network` class is a subclass of the `keras.layers.Layer` class that adds some functionality required by some agents, such as the possibility to easily create shallow copies of the network (i.e., copying the network's architecture, but not its weights). For example, the `DQNAgent` uses this to create a copy of the online model.

Now that we have the DQN, we are ready to build the DQN agent.

Creating the DQN Agent

The TF-Agents library implements many types of agents, located in the `tf_agents.agents` package and its subpackages. We will use the `tf_agents.agents.dqn.dqn_agent.DqnAgent` class:

```
from tf_agents.agents.dqn.dqn_agent import DqnAgent

train_step = tf.Variable(0)
update_period = 4 # train the model every 4 steps
optimizer = keras.optimizers.RMSprop(lr=2.5e-4, rho=0.95, momentum=0.0,
                                     epsilon=0.00001, centered=True)
epsilon_fn = keras.optimizers.schedules.PolynomialDecay(
    initial_learning_rate=1.0, # initial ε
    decay_steps=250000 // update_period, # <=> 1,000,000 ALE frames
    end_learning_rate=0.01) # final ε
agent = DqnAgent(tf_env.time_step_spec(),
                 tf_env.action_spec(),
                 q_network=q_net,
                 optimizer=optimizer,
                 target_update_period=2000, # <=> 32,000 ALE frames
                 td_errors_loss_fn=keras.losses.Huber(reduction="none"),
                 gamma=0.99, # discount factor
                 train_step_counter=train_step,
                 epsilon_greedy=lambda: epsilon_fn(train_step))
agent.initialize()
```

Let's walk through this code:

- We first create a variable that will count the number of training steps.
- Then we build the optimizer, using the same hyperparameters as in the 2015 DQN paper.
- Next, we create a `PolynomialDecay` object that will compute the ε value for the ε-greedy collect policy, given the current training step (it is normally used to decay the learning rate, hence the names of the arguments, but it will work just fine to decay any other value). It will go from 1.0 down to 0.01 (the value used during in the 2015 DQN paper) in 1 million ALE frames, which corresponds to 250,000 steps, since we use frame skipping with a period of 4. Moreover, we will train the agent every 4 steps (i.e., 16 ALE frames), so ε will actually decay over 62,500 *training* steps.
- We then build the `DQNAgent`, passing it the time step and action specs, the `QNetwork` to train, the optimizer, the number of training steps between target model updates, the loss function to use, the discount factor, the `train_step` variable, and a function that returns the ε value (it must take no argument, which is why we need a lambda to pass the `train_step`).

  Note that the loss function must return an error per instance, not the mean error, which is why we set `reduction="none"`.
- Lastly, we initialize the agent.

Next, let's build the replay buffer and the observer that will write to it.

Creating the Replay Buffer and the Corresponding Observer

The TF-Agents library provides various replay buffer implementations in the `tf_agents.replay_buffers` package. Some are purely written in Python (their module names start with `py_`), and others are written based on TensorFlow (their module names start with `tf_`). We will use the `TFUniformReplayBuffer` class in the `tf_agents.replay_buffers.tf_uniform_replay_buffer` package. It provides a high-performance implementation of a replay buffer with uniform sampling:[21]

```
from tf_agents.replay_buffers import tf_uniform_replay_buffer

replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=agent.collect_data_spec,
    batch_size=tf_env.batch_size,
    max_length=1000000)
```

Let's look at each of these arguments:

`data_spec`
: The specification of the data that will be saved in the replay buffer. The DQN agent knowns what the collected data will look like, and it makes the data spec available via its `collect_data_spec` attribute, so that's what we give the replay buffer.

`batch_size`
: The number of trajectories that will be added at each step. In our case, it will be one, since the driver will just execute one action per step and collect one trajectory. If the environment were a *batched environment*, meaning an environment that takes a batch of actions at each step and returns a batch of observations, then the driver would have to save a batch of trajectories at each step. Since we are using a TensorFlow replay buffer, it needs to know the size of the batches it will handle (to build the computation graph). An example of a batched environment is the `ParallelPyEnvironment` (from the `tf_agents.environments.parallel_py_environment` package): it runs multiple environments in parallel in separate processes (they can be different as long as they have the same action and observation specs), and at each step it takes a batch of actions and executes them in the environments (one action per environment), then it returns all the resulting observations.

21 At the time of this writing, there is no prioritized experience replay buffer yet, but one will likely be open sourced soon.

`max_length`
: The maximum size of the replay buffer. We created a large replay buffer that can store one million trajectories (as was done in the 2015 DQN paper). This will require a lot of RAM.

When we store two consecutive trajectories, they contain two consecutive observations with four frames each (since we used the `FrameStack4` wrapper), and unfortunately three of the four frames in the second observation are redundant (they are already present in the first observation). In other words, we are using about four times more RAM than necessary. To avoid this, you can instead use a `PyHashedReplayBuffer` from the `tf_agents.replay_buffers.py_hashed_replay_buffer` package: it deduplicates data in the stored trajectories along the last axis of the observations.

Now we can create the observer that will write the trajectories to the replay buffer. An observer is just a function (or a callable object) that takes a trajectory argument, so we can directly use the `add_method()` method (bound to the `replay_buffer` object) as our observer:

```
replay_buffer_observer = replay_buffer.add_batch
```

If you wanted to create your own observer, you could write any function with a `trajectory` argument. If it must have a state, you can write a class with a `__call__(self, trajectory)` method. For example, here is a simple observer that will increment a counter every time it is called (except when the trajectory represents a boundary between two episodes, which does not count as a step), and every 100 increments it displays the progress up to a given total (the carriage return `\r` along with `end=""` ensures that the displayed counter remains on the same line):

```
class ShowProgress:
    def __init__(self, total):
        self.counter = 0
        self.total = total
    def __call__(self, trajectory):
        if not trajectory.is_boundary():
            self.counter += 1
        if self.counter % 100 == 0:
            print("\r{}/{}".format(self.counter, self.total), end="")
```

Now let's create a few training metrics.

Creating Training Metrics

TF-Agents implements several RL metrics in the `tf_agents.metrics` package, some purely in Python and some based on TensorFlow. Let's create a few of them in order

to count the number of episodes, the number of steps taken, and most importantly the average return per episode and the average episode length:

```
from tf_agents.metrics import tf_metrics

train_metrics = [
    tf_metrics.NumberOfEpisodes(),
    tf_metrics.EnvironmentSteps(),
    tf_metrics.AverageReturnMetric(),
    tf_metrics.AverageEpisodeLengthMetric(),
]
```

Discounting the rewards makes sense for training or to implement a policy, as it makes it possible to balance the importance of immediate rewards with future rewards. However, once an episode is over, we can evaluate how good it was overalls by summing the *undiscounted* rewards. For this reason, the `AverageReturnMetric` computes the sum of undiscounted rewards for each episode, and it keeps track of the streaming mean of these sums over all the episodes it encounters.

At any time, you can get the value of each of these metrics by calling its `result()` method (e.g., `train_metrics[0].result()`). Alternatively, you can log all metrics by calling `log_metrics(train_metrics)` (this function is located in the `tf_agents.eval.metric_utils` package):

```
>>> from tf_agents.eval.metric_utils import log_metrics
>>> import logging
>>> logging.get_logger().set_level(logging.INFO)
>>> log_metrics(train_metrics)
[...]
NumberOfEpisodes = 0
EnvironmentSteps = 0
AverageReturn = 0.0
AverageEpisodeLength = 0.0
```

Next, let's create the collect driver.

Creating the Collect Driver

As we explored in Figure 18-13, a driver is an object that explores an environment using a given policy, collects experiences, and broadcasts them to some observers. At each step, the following things happen:

- The driver passes the current time step to the collect policy, which uses this time step to choose an action and returns an *action step* object containing the action.

- The driver then passes the action to the environment, which returns the next time step.
- Finally, the driver creates a trajectory object to represent this transition and broadcasts it to all the observers.

Some policies, such as RNN policies, are stateful: they choose an action based on both the given time step and their own internal state. Stateful policies return their own state in the action step, along with the chosen action. The driver will then pass this state back to the policy at the next time step. Moreover, the driver saves the policy state to the trajectory (in the `policy_info` field), so it ends up in the replay buffer. This is essential when training a stateful policy: when the agent samples a trajectory, it must set the policy's state to the state it was in at the time of the sampled time step.

Also, as discussed earlier, the environment may be a batched environment, in which case the driver passes a *batched time step* to the policy (i.e., a time step object containing a batch of observations, a batch of step types, a batch of rewards, and a batch of discounts, all four batches of the same size). The driver also passes a batch of previous policy states. The policy then returns a *batched action step* containing a batch of actions and a batch of policy states. Finally, the driver creates a *batched trajectory* (i.e., a trajectory containing a batch of step types, a batch of observations, a batch of actions, a batch of rewards, and more generally a batch for each trajectory attribute, with all batches of the same size).

There are two main driver classes: `DynamicStepDriver` and `DynamicEpisodeDriver`. The first one collects experiences for a given number of steps, while the second collects experiences for a given number of episodes. We want to collect experiences for four steps for each training iteration (as was done in the 2015 DQN paper), so let's create a `DynamicStepDriver`:

```
from tf_agents.drivers.dynamic_step_driver import DynamicStepDriver

collect_driver = DynamicStepDriver(
    tf_env,
    agent.collect_policy,
    observers=[replay_buffer_observer] + training_metrics,
    num_steps=update_period) # collect 4 steps for each training iteration
```

We give it the environment to play with, the agent's collect policy, a list of observers (including the replay buffer observer and the training metrics), and finally the number of steps to run (in this case, four). We could now run it by calling its `run()` method, but it's best to warm up the replay buffer with experiences collected using a purely random policy. For this, we can use the `RandomTFPolicy` class and create a second driver that will run this policy for 20,000 steps (which is equivalent to 80,000 simulator frames, as was done in the 2015 DQN paper). We can use our `ShowProgress` observer to display the progress:

```
from tf_agents.policies.random_tf_policy import RandomTFPolicy

initial_collect_policy = RandomTFPolicy(tf_env.time_step_spec(),
                                        tf_env.action_spec())
init_driver = DynamicStepDriver(
    tf_env,
    initial_collect_policy,
    observers=[replay_buffer.add_batch, ShowProgress(20000)],
    num_steps=20000) # <=> 80,000 ALE frames
final_time_step, final_policy_state = init_driver.run()
```

We're almost ready to run the training loop! We just need one last component: the dataset.

Creating the Dataset

To sample a batch of trajectories from the replay buffer, call its `get_next()` method. This returns the batch of trajectories plus a `BufferInfo` object that contains the sample identifiers and their sampling probabilities (this may be useful for some algorithms, such as PER). For example, the following code will sample a small batch of two trajectories (subepisodes), each containing three consecutive steps. These subepisodes are shown in Figure 18-15 (each row contains three consecutive steps from an episode):

```
>>> trajectories, buffer_info = replay_buffer.get_next(
...     sample_batch_size=2, num_steps=3)
...
>>> trajectories._fields
('step_type', 'observation', 'action', 'policy_info',
 'next_step_type', 'reward', 'discount')
>>> trajectories.observation.shape
TensorShape([2, 3, 84, 84, 4])
>>> trajectories.step_type.numpy()
array([[1, 1, 1],
       [1, 1, 1]], dtype=int32)
```

The `trajectories` object is a named tuple, with seven fields. Each field contains a tensor whose first two dimensions are 2 and 3 (since there are two trajectories, each with three steps). This explains why the shape of the `observation` field is [2, 3, 84, 84, 4]: that's two trajectories, each with three steps, and each step's observation is 84 × 84 × 4. Similarly, the `step_type` tensor has a shape of [2, 3]: in this example, both trajectories contain three consecutive steps in the middle on an episode (types 1, 1, 1). In the second trajectory, you can barely see the ball at the lower left of the first observation, and it disappears in the next two observations, so the agent is about to lose a life, but the episode will not end immediately because it still has several lives left.

Figure 18-15. Two trajectories containing three consecutive steps each

Each trajectory is a concise representation of a sequence of consecutive time steps and action steps, designed to avoid redundancy. How so? Well, as you can see in Figure 18-16, transition n is composed of time step n, action step n, and time step $n + 1$, while transition $n + 1$ is composed of time step $n + 1$, action step $n + 1$, and time step $n + 2$. If we just stored these two transitions directly in the replay buffer, the time step $n + 1$ would be duplicated. To avoid this duplication, the nth trajectory step includes only the type and observation from time step n (not its reward and discount), and it does not contain the observation from time step $n + 1$ (however, it does contain a copy of the next time step's type; that's the only duplication).

Figure 18-16. Trajectories, transitions, time steps, and action steps

So if you have a batch of trajectories where each trajectory has $t + 1$ steps (from time step n to time step $n + t$), then it contains all the data from time step n to time step $n + t$, except for the reward and discount from time step n (but it contains the reward and discount of time step $n + t + 1$). This represents t transitions (n to $n + 1$, $n + 1$ to $n + 2$, …, $n + t - 1$ to $n + t$).

The `to_transition()` function in the `tf_agents.trajectories.trajectory` module converts a batched trajectory into a list containing a batched `time_step`, a batched `action_step`, and a batched `next_time_step`. Notice that the second dimension is 2 instead of 3, since there are t transitions between $t + 1$ time steps (don't worry if you're a bit confused; you'll get the hang of it):

```
>>> from tf_agents.trajectories.trajectory import to_transition
>>> time_steps, action_steps, next_time_steps = to_transition(trajectories)
>>> time_steps.observation.shape
TensorShape([2, 2, 84, 84, 4]) # 3 time steps = 2 transitions
```

A sampled trajectory may actually overlap two (or more) episodes! In this case, it will contain *boundary transitions*, meaning transitions with a `step_type` equal to 2 (end) and a `next_step_type` equal to 0 (start). Of course, TF-Agents properly handles such trajectories (e.g., by resetting the policy state when encountering a boundary). The trajectory's `is_boundary()` method returns a tensor indicating whether each step is a boundary or not.

For our main training loop, instead of calling the `get_next()` method, we will use a `tf.data.Dataset`. This way, we can benefit from the power of the Data API (e.g., parallelism and prefetching). For this, we call the replay buffer's `as_dataset()` method:

```
dataset = replay_buffer.as_dataset(
    sample_batch_size=64,
    num_steps=2,
    num_parallel_calls=3).prefetch(3)
```

We will sample batches of 64 trajectories at each training step (as in the 2015 DQN paper), each with 2 steps (i.e., 2 steps = 1 full transition, including the next step's observation). This dataset will process three elements in parallel, and prefetch three batches.

For on-policy algorithms such as Policy Gradients, each experience should be sampled once, used from training, and then discarded. In this case, you can still use a replay buffer, but instead of using a `Dataset`, you would call the replay buffer's `gather_all()` method at each training iteration to get a tensor containing all the trajectories recorded so far, then use them to perform a training step, and finally clear the replay buffer by calling its `clear()` method.

Now that we have all the components in place, we are ready to train the model!

Creating the Training Loop

To speed up training, we will convert the main functions to TensorFlow Functions. For this we will use the `tf_agents.utils.common.function()` function, which wraps `tf.function()`, with some extra experimental options:

```
from tf_agents.utils.common import function

collect_driver.run = function(collect_driver.run)
agent.train = function(agent.train)
```

Let's create a small function that will run the main training loop for `n_iterations`:

```
def train_agent(n_iterations):
    time_step = None
    policy_state = agent.collect_policy.get_initial_state(tf_env.batch_size)
    iterator = iter(dataset)
    for iteration in range(n_iterations):
        time_step, policy_state = collect_driver.run(time_step, policy_state)
        trajectories, buffer_info = next(iterator)
        train_loss = agent.train(trajectories)
        print("\r{} loss:{:.5f}".format(
            iteration, train_loss.loss.numpy()), end="")
        if iteration % 1000 == 0:
            log_metrics(train_metrics)
```

The function first asks the collect policy for its initial state (given the environment batch size, which is 1 in this case). Since the policy is stateless, this returns an empty tuple (so we could have written `policy_state = ()`). Next, we create an iterator over the dataset, and we run the training loop. At each iteration, we call the driver's `run()` method, passing it the current time step (initially `None`) and the current policy state. It will run the collect policy and collect experience for four steps (as we configured earlier), broadcasting the collected trajectories to the replay buffer and the metrics. Next, we sample one batch of trajectories from the dataset, and we pass it to the agent's `train()` method. It returns a `train_loss` object which may vary depending on the type of agent. Next, we display the iteration number and the training loss, and every 1,000 iterations we log all the metrics. Now you can just call `train_agent()` for some number of iterations, and see the agent gradually learn to play *Breakout*!

```
train_agent(10000000)
```

This will take a lot of computing power and a lot of patience (it may take hours, or even days, depending on your hardware), plus you may need to run the algorithm several times with different random seeds to get good results, but once it's done, the agent will be superhuman (at least at *Breakout*). You can also try training this DQN agent on other Atari games: it can achieve superhuman skill at most action games, but it is not so good at games with long-running storylines.[22]

Overview of Some Popular RL Algorithms

Before we finish this chapter, let's take a quick look at a few popular RL algorithms:

Actor-Critic algorithms

A family of RL algorithms that combine Policy Gradients with Deep Q-Networks. An Actor-Critic agent contains two neural networks: a policy net and a DQN. The DQN is trained normally, by learning from the agent's experiences. The policy net learns differently (and much faster) than in regular PG: instead of estimating the value of each action by going through multiple episodes, then summing the future discounted rewards for each action, and finally normalizing them, the agent (actor) relies on the action values estimated by the DQN (critic). It's a bit like an athlete (the agent) learning with the help of a coach (the DQN).

Asynchronous Advantage Actor-Critic[23] *(A3C)*

An important Actor-Critic variant introduced by DeepMind researchers in 2016, where multiple agents learn in parallel, exploring different copies of the environ-

22 For a comparison of this algorithm's performance on various Atari games, see figure 3 in DeepMind's 2015 paper.

23 Volodymyr Mnih et al., "Asynchonous Methods for Deep Reinforcement Learning," *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1928–1937.

ment. At regular intervals, but asynchronously (hence the name), each agent pushes some weight updates to a master network, then it pulls the latest weights from that network. Each agent thus contributes to improving the master network and benefits from what the other agents have learned. Moreover, instead of estimating the Q-Values, the DQN estimates the advantage of each action (hence the second A in the name), which stabilizes training.

Advantage Actor-Critic (A2C)

A variant of the A3C algorithm that removes the asynchronicity. All model updates are synchronous, so gradient updates are performed over larger batches, which allows the model to better utilize the power of the GPU.

Soft Actor-Critic[24] *(SAC)*

An Actor-Critic variant proposed in 2018 by Tuomas Haarnoja and other UC Berkeley researchers. It learns not only rewards, but also to maximize the entropy of its actions. In other words, it tries to be as unpredictable as possible while still getting as many rewards as possible. This encourages the agent to explore the environment, which speeds up training, and makes it less likely to repeatedly execute the same action when the DQN produces imperfect estimates. This algorithm has demonstrated an amazing sample efficiency (contrary to all the previous algorithms, which learn very slowly). SAC is available in TF-Agents.

Proximal Policy Optimization (PPO)[25]

An algorithm based on A2C that clips the loss function to avoid excessively large weight updates (which often lead to training instabilities). PPO is a simplification of the previous *Trust Region Policy Optimization*[26] (TRPO) algorithm, also by John Schulman and other OpenAI researchers. OpenAI made the news in April 2019 with their AI called OpenAI Five, based on the PPO algorithm, which defeated the world champions at the multiplayer game *Dota 2*. PPO is also available in TF-Agents.

24 Tuomas Haarnoja et al., "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," *Proceedings of the 35th International Conference on Machine Learning* (2018): 1856–1865.

25 John Schulman et al., "Proximal Policy Optimization Algorithms," arXiv preprint arXiv:1707.06347 (2017).

26 John Schulman et al., "Trust Region Policy Optimization," *Proceedings of the 32nd International Conference on Machine Learning* (2015): 1889–1897.

Curiosity-based exploration[27]

A recurring problem in RL is the sparsity of the rewards, which makes learning very slow and inefficient. Deepak Pathak and other UC Berkeley researchers have proposed an exciting way to tackle this issue: why not ignore the rewards, and just make the agent extremely curious to explore the environment? The rewards thus become intrinsic to the agent, rather than coming from the environment. Similarly, stimulating curiosity in a child is more likely to give good results than purely rewarding the child for getting good grades. How does this work? The agent continuously tries to predict the outcome of its actions, and it seeks situations where the outcome does not match its predictions. In other words, it wants to be surprised. If the outcome is predictable (boring), it goes elsewhere. However, if the outcome is unpredictable but the agent notices that it has no control over it, it also gets bored after a while. With only curiosity, the authors succeeded in training an agent at many video games: even though the agent gets no penalty for losing, the game starts over, which is boring so it learns to avoid it.

We covered many topics in this chapter: Policy Gradients, Markov chains, Markov decision processes, Q-Learning, Approximate Q-Learning, and Deep Q-Learning and its main variants (fixed Q-Value targets, Double DQN, Dueling DQN, and prioritized experience replay). We discussed how to use TF-Agents to train agents at scale, and finally we took a quick look at a few other popular algorithms. Reinforcement Learning is a huge and exciting field, with new ideas and algorithms popping out every day, so I hope this chapter sparked your curiosity: there is a whole world to explore!

Exercises

1. How would you define Reinforcement Learning? How is it different from regular supervised or unsupervised learning?
2. Can you think of three possible applications of RL that were not mentioned in this chapter? For each of them, what is the environment? What is the agent? What are some possible actions? What are the rewards?
3. What is the discount factor? Can the optimal policy change if you modify the discount factor?
4. How do you measure the performance of a Reinforcement Learning agent?
5. What is the credit assignment problem? When does it occur? How can you alleviate it?
6. What is the point of using a replay buffer?

27 Deepak Pathak et al., "Curiosity-Driven Exploration by Self-Supervised Prediction," *Proceedings of the 34th International Conference on Machine Learning* (2017): 2778–2787.

7. What is an off-policy RL algorithm?
8. Use policy gradients to solve OpenAI Gym's LunarLander-v2 environment. You will need to install the Box2D dependencies (`python3 -m pip install -U gym[box2d]`).
9. Use TF-Agents to train an agent that can achieve a superhuman level at SpaceInvaders-v4 using any of the available algorithms.
10. If you have about $100 to spare, you can purchase a Raspberry Pi 3 plus some cheap robotics components, install TensorFlow on the Pi, and go wild! For an example, check out this fun post by Lukas Biewald, or take a look at GoPiGo or BrickPi. Start with simple goals, like making the robot turn around to find the brightest angle (if it has a light sensor) or the closest object (if it has a sonar sensor), and move in that direction. Then you can start using Deep Learning: for example, if the robot has a camera, you can try to implement an object detection algorithm so it detects people and moves toward them. You can also try to use RL to make the agent learn on its own how to use the motors to achieve that goal. Have fun!

Solutions to these exercises are available in Appendix A.

CHAPTER 19

Training and Deploying TensorFlow Models at Scale

Once you have a beautiful model that makes amazing predictions, what do you do with it? Well, you need to put it in production! This could be as simple as running the model on a batch of data and perhaps writing a script that runs this model every night. However, it is often much more involved. Various parts of your infrastructure may need to use this model on live data, in which case you probably want to wrap your model in a web service: this way, any part of your infrastructure can query your model at any time using a simple REST API (or some other protocol), as we discussed in Chapter 2. But as time passes, you need to regularly retrain your model on fresh data and push the updated version to production. You must handle model versioning, gracefully transition from one model to the next, possibly roll back to the previous model in case of problems, and perhaps run multiple different models in parallel to perform *A/B experiments*.[1] If your product becomes successful, your service may start to get plenty of *queries per second* (QPS), and it must scale up to support the load. A great solution to scale up your service, as we will see in this chapter, is to use TF Serving, either on your own hardware infrastructure or via a cloud service such as Google Cloud AI Platform. It will take care of efficiently serving your model, handle graceful model transitions, and more. If you use the cloud platform, you will also get many extra features, such as powerful monitoring tools.

Moreover, if you have a lot of training data, and compute-intensive models, then training time may be prohibitively long. If your product needs to adapt to changes quickly, then a long training time can be a showstopper (e.g., think of a news

1 An A/B experiment consists in testing two different versions of your product on different subsets of users in order to check which version works best and get other insights.

recommendation system promoting news from last week). Perhaps even more importantly, a long training time will prevent you from experimenting with new ideas. In Machine Learning (as in many other fields), it is hard to know in advance which ideas will work, so you should try out as many as possible, as fast as possible. One way to speed up training is to use hardware accelerators such as GPUs or TPUs. To go even faster, you can train a model across multiple machines, each equipped with multiple hardware accelerators. TensorFlow's simple yet powerful Distribution Strategies API makes this easy, as we will see.

In this chapter we will look at how to deploy models, first to TF Serving, then to Google Cloud AI Platform. We will also take a quick look at deploying models to mobile apps, embedded devices, and web apps. Lastly, we will discuss how to speed up computations using GPUs and how to train models across multiple devices and servers using the Distribution Strategies API. That's a lot of topics to discuss, so let's get started!

Serving a TensorFlow Model

Once you have trained a TensorFlow model, you can easily use it in any Python code: if it's a tf.keras model, just call its `predict()` method! But as your infrastructure grows, there comes a point where it is preferable to wrap your model in a small service whose sole role is to make predictions and have the rest of the infrastructure query it (e.g., via a REST or gRPC API).[2] This decouples your model from the rest of the infrastructure, making it possible to easily switch model versions or scale the service up as needed (independently from the rest of your infrastructure), perform A/B experiments, and ensure that all your software components rely on the same model versions. It also simplifies testing and development, and more. You could create your own microservice using any technology you want (e.g., using the Flask library), but why reinvent the wheel when you can just use TF Serving?

Using TensorFlow Serving

TF Serving is a very efficient, battle-tested model server that's written in C++. It can sustain a high load, serve multiple versions of your models and watch a model repository to automatically deploy the latest versions, and more (see Figure 19-1).

2 A REST (or RESTful) API is an API that uses standard HTTP verbs, such as GET, POST, PUT, and DELETE, and uses JSON inputs and outputs. The gRPC protocol is more complex but more efficient. Data is exchanged using protocol buffers (see Chapter 13).

Figure 19-1. TF Serving can serve multiple models and automatically deploy the latest version of each model

So let's suppose you have trained an MNIST model using tf.keras, and you want to deploy it to TF Serving. The first thing you have to do is export this model to TensorFlow's *SavedModel format*.

Exporting SavedModels

TensorFlow provides a simple `tf.saved_model.save()` function to export models to the SavedModel format. All you need to do is give it the model, specifying its name and version number, and the function will save the model's computation graph and its weights:

```
model = keras.models.Sequential([...])
model.compile([...])
history = model.fit([...])

model_version = "0001"
model_name = "my_mnist_model"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)
```

Alternatively, you can just use the model's `save()` method (`model.save(model_path)`): as long as the file's extension is not `.h5`, the model will be saved using the SavedModel format instead of the HDF5 format.

It's usually a good idea to include all the preprocessing layers in the final model you export so that it can ingest data in its natural form once it is deployed to production. This avoids having to take care of preprocessing separately within the application that uses the model. Bundling the preprocessing steps within the model also makes it simpler to update them later on and limits the risk of mismatch between a model and the preprocessing steps it requires.

Since a SavedModel saves the computation graph, it can only be used with models that are based exclusively on TensorFlow operations, excluding the `tf.py_function()` operation (which wraps arbitrary Python code). It also excludes dynamic tf.keras models (see Appendix G), since these models cannot be converted to computation graphs. Dynamic models need to be served using other tools (e.g., Flask).

A SavedModel represents a version of your model. It is stored as a directory containing a *saved_model.pb* file, which defines the computation graph (represented as a serialized protocol buffer), and a *variables* subdirectory containing the variable values. For models containing a large number of weights, these variable values may be split across multiple files. A SavedModel also includes an *assets* subdirectory that may contain additional data, such as vocabulary files, class names, or some example instances for this model. The directory structure is as follows (in this example, we don't use assets):

```
my_mnist_model
└── 0001
    ├── assets
    ├── saved_model.pb
    └── variables
        ├── variables.data-00000-of-00001
        └── variables.index
```

As you might expect, you can load a SavedModel using the `tf.saved_model.load()` function. However, the returned object is not a Keras model: it represents the SavedModel, including its computation graph and variable values. You can use it like a function, and it will make predictions (make sure to pass the inputs as tensors of the appropriate type):

```
saved_model = tf.saved_model.load(model_path)
y_pred = saved_model(tf.constant(X_new, dtype=tf.float32))
```

Alternatively, you can load this SavedModel directly to a Keras model using the `keras.models.load_model()` function:

```
model = keras.models.load_model(model_path)
y_pred = model.predict(tf.constant(X_new, dtype=tf.float32))
```

TensorFlow also comes with a small `saved_model_cli` command-line tool to inspect SavedModels:

```
$ export ML_PATH="$HOME/ml" # point to this project, wherever it is
$ cd $ML_PATH
$ saved_model_cli show --dir my_mnist_model/0001 --all
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:
signature_def['__saved_model_init_op']:
  [...]
```

```
signature_def['serving_default']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['flatten_input'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 28, 28)
        name: serving_default_flatten_input:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['dense_1'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 10)
        name: StatefulPartitionedCall:0
  Method name is: tensorflow/serving/predict
```

A SavedModel contains one or more *metagraphs*. A metagraph is a computation graph plus some function signature definitions (including their input and output names, types, and shapes). Each metagraph is identified by a set of tags. For example, you may want to have a metagraph containing the full computation graph, including the training operations (this one may be tagged `"train"`, for example), and another metagraph containing a pruned computation graph with only the prediction operations, including some GPU-specific operations (this metagraph may be tagged `"serve", "gpu"`). However, when you pass a tf.keras model to the `tf.saved_model.save()` function, by default the function saves a much simpler SavedModel: it saves a single metagraph tagged `"serve"`, which contains two signature definitions, an initialization function (called `__saved_model_init_op`, which you do not need to worry about) and a default serving function (called `serving_default`). When saving a tf.keras model, the default serving function corresponds to the model's `call()` function, which of course makes predictions.

The `saved_model_cli` tool can also be used to make predictions (for testing, not really for production). Suppose you have a NumPy array (`X_new`) containing three images of handwritten digits that you want to make predictions for. You first need to export them to NumPy's `npy` format:

```
np.save("my_mnist_tests.npy", X_new)
```

Next, use the `saved_model_cli` command like this:

```
$ saved_model_cli run --dir my_mnist_model/0001 --tag_set serve \
                      --signature_def serving_default \
                      --inputs flatten_input=my_mnist_tests.npy
[...] Result for output key dense_1:
[[1.1739199e-04 1.1239604e-07 6.0210604e-04 [...] 3.9471846e-04]
 [1.2294615e-03 2.9207937e-05 9.8599273e-01 [...] 1.1113169e-07]
 [6.4066830e-05 9.6359509e-01 9.0598064e-03 [...] 4.2495009e-04]]
```

The tool's output contains the 10 class probabilities of each of the 3 instances. Great! Now that you have a working SavedModel, the next step is to install TF Serving.

Installing TensorFlow Serving

There are many ways to install TF Serving: using a Docker image,[3] using the system's package manager, installing from source, and more. Let's use the Docker option, which is highly recommended by the TensorFlow team as it is simple to install, it will not mess with your system, and it offers high performance. You first need to install Docker. Then download the official TF Serving Docker image:

```
$ docker pull tensorflow/serving
```

Now you can create a Docker container to run this image:

```
$ docker run -it --rm -p 8500:8500 -p 8501:8501 \
             -v "$ML_PATH/my_mnist_model:/models/my_mnist_model" \
             -e MODEL_NAME=my_mnist_model \
             tensorflow/serving
[...]
2019-06-01 [...] loaded servable version {name: my_mnist_model version: 1}
2019-06-01 [...] Running gRPC ModelServer at 0.0.0.0:8500 ...
2019-06-01 [...] Exporting HTTP/REST API at:localhost:8501 ...
[evhttp_server.cc : 237] RAW: Entering the event loop ...
```

That's it! TF Serving is running. It loaded our MNIST model (version 1), and it is serving it through both gRPC (on port 8500) and REST (on port 8501). Here is what all the command-line options mean:

`-it`
: Makes the container interactive (so you can press Ctrl-C to stop it) and displays the server's output.

`--rm`
: Deletes the container when you stop it (no need to clutter your machine with interrupted containers). However, it does not delete the image.

`-p 8500:8500`
: Makes the Docker engine forward the host's TCP port 8500 to the container's TCP port 8500. By default, TF Serving uses this port to serve the gRPC API.

`-p 8501:8501`
: Forwards the host's TCP port 8501 to the container's TCP port 8501. By default, TF Serving uses this port to serve the REST API.

3 If you are not familiar with Docker, it allows you to easily download a set of applications packaged in a *Docker image* (including all their dependencies and usually some good default configuration) and then run them on your system using a *Docker engine*. When you run an image, the engine creates a *Docker container* that keeps the applications well isolated from your own system (but you can give it some limited access if you want). It is similar to a virtual machine, but much faster and more lightweight, as the container relies directly on the host's kernel. This means that the image does not need to include or run its own kernel.

`-v "$ML_PATH/my_mnist_model:/models/my_mnist_model"`
: Makes the host's `$ML_PATH/my_mnist_model` directory available to the container at the path */models/mnist_model*. On Windows, you may need to replace `/` with `\` in the host path (but not in the container path).

`-e MODEL_NAME=my_mnist_model`
: Sets the container's `MODEL_NAME` environment variable, so TF Serving knows which model to serve. By default, it will look for models in the */models* directory, and it will automatically serve the latest version it finds.

`tensorflow/serving`
: This is the name of the image to run.

Now let's go back to Python and query this server, first using the REST API, then the gRPC API.

Querying TF Serving through the REST API

Let's start by creating the query. It must contain the name of the function signature you want to call, and of course the input data:

```
import json

input_data_json = json.dumps({
    "signature_name": "serving_default",
    "instances": X_new.tolist(),
})
```

Note that the JSON format is 100% text-based, so the `X_new` NumPy array had to be converted to a Python list and then formatted as JSON:

```
>>> input_data_json
'{"signature_name": "serving_default", "instances": [[[0.0, 0.0, 0.0, [...]
0.3294117647058824, 0.725490196078431, [...very long], 0.0, 0.0, 0.0, 0.0]]]}'
```

Now let's send the input data to TF Serving by sending an HTTP POST request. This can be done easily using the `requests` library (it is not part of Python's standard library, so you will need to install it first, e.g., using pip):

```
import requests

SERVER_URL = 'http://localhost:8501/v1/models/my_mnist_model:predict'
response = requests.post(SERVER_URL, data=input_data_json)
response.raise_for_status() # raise an exception in case of error
response = response.json()
```

The response is a dictionary containing a single `"predictions"` key. The corresponding value is the list of predictions. This list is a Python list, so let's convert it to a NumPy array and round the floats it contains to the second decimal:

```
>>> y_proba = np.array(response["predictions"])
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 0.  , 0.  ],
       [0.  , 0.  , 0.99, 0.01, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.96, 0.01, 0.  , 0.  , 0.  , 0.  , 0.01, 0.01, 0.  ]])
```

Hurray, we have the predictions! The model is close to 100% confident that the first image is a 7, 99% confident that the second image is a 2, and 96% confident that the third image is a 1.

The REST API is nice and simple, and it works well when the input and output data are not too large. Moreover, just about any client application can make REST queries without additional dependencies, whereas other protocols are not always so readily available. However, it is based on JSON, which is text-based and fairly verbose. For example, we had to convert the NumPy array to a Python list, and every float ended up represented as a string. This is very inefficient, both in terms of serialization/deserialization time (to convert all the floats to strings and back) and in terms of payload size: many floats end up being represented using over 15 characters, which translates to over 120 bits for 32-bit floats! This will result in high latency and bandwidth usage when transferring large NumPy arrays.[4] So let's use gRPC instead.

When transferring large amounts of data, it is much better to use the gRPC API (if the client supports it), as it is based on a compact binary format and an efficient communication protocol (based on HTTP/2 framing).

Querying TF Serving through the gRPC API

The gRPC API expects a serialized `PredictRequest` protocol buffer as input, and it outputs a serialized `PredictResponse` protocol buffer. These protobufs are part of the `tensorflow-serving-api` library, which you must install (e.g., using pip). First, let's create the request:

```
from tensorflow_serving.apis.predict_pb2 import PredictRequest

request = PredictRequest()
request.model_spec.name = model_name
request.model_spec.signature_name = "serving_default"
input_name = model.input_names[0]
request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new))
```

This code creates a `PredictRequest` protocol buffer and fills in the required fields, including the model name (defined earlier), the signature name of the function we

4 To be fair, this can be mitigated by serializing the data first and encoding it to Base64 before creating the REST request. Moreover, REST requests can be compressed using gzip, which reduces the payload size significantly.

want to call, and finally the input data, in the form of a `Tensor` protocol buffer. The `tf.make_tensor_proto()` function creates a `Tensor` protocol buffer based on the given tensor or NumPy array, in this case `X_new`.

Next, we'll send the request to the server and get its response (for this you will need the `grpcio` library, which you can install using pip):

```
import grpc
from tensorflow_serving.apis import prediction_service_pb2_grpc

channel = grpc.insecure_channel('localhost:8500')
predict_service = prediction_service_pb2_grpc.PredictionServiceStub(channel)
response = predict_service.Predict(request, timeout=10.0)
```

The code is quite straightforward: after the imports, we create a gRPC communication channel to *localhost* on TCP port 8500, then we create a gRPC service over this channel and use it to send a request, with a 10-second timeout (not that the call is synchronous: it will block until it receives the response or the timeout period expires). In this example the channel is insecure (no encryption, no authentication), but gRPC and TensorFlow Serving also support secure channels over SSL/TLS.

Next, let's convert the `PredictResponse` protocol buffer to a tensor:

```
output_name = model.output_names[0]
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto)
```

If you run this code and print `y_proba.numpy().round(2)`, you will get the exact same estimated class probabilities as earlier. And that's all there is to it: in just a few lines of code, you can now access your TensorFlow model remotely, using either REST or gRPC.

Deploying a new model version

Now let's create a new model version and export a SavedModel to the *my_mnist_model/0002* directory, just like earlier:

```
model = keras.models.Sequential([...])
model.compile([...])
history = model.fit([...])

model_version = "0002"
model_name = "my_mnist_model"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)
```

At regular intervals (the delay is configurable), TensorFlow Serving checks for new model versions. If it finds one, it will automatically handle the transition gracefully: by default, it will answer pending requests (if any) with the previous model version,

while handling new requests with the new version.[5] As soon as every pending request has been answered, the previous model version is unloaded. You can see this at work in the TensorFlow Serving logs:

```
[...]
reserved resources to load servable {name: my_mnist_model version: 2}
[...]
Reading SavedModel from: /models/my_mnist_model/0002
Reading meta graph with tags { serve }
Successfully loaded servable version {name: my_mnist_model version: 2}
Quiescing servable version {name: my_mnist_model version: 1}
Done quiescing servable version {name: my_mnist_model version: 1}
Unloading servable version {name: my_mnist_model version: 1}
```

This approach offers a smooth transition, but it may use too much RAM (especially GPU RAM, which is generally the most limited). In this case, you can configure TF Serving so that it handles all pending requests with the previous model version and unloads it before loading and using the new model version. This configuration will avoid having two model versions loaded at the same time, but the service will be unavailable for a short period.

As you can see, TF Serving makes it quite simple to deploy new models. Moreover, if you discover that version 2 does not work as well as you expected, then rolling back to version 1 is as simple as removing the *my_mnist_model/0002* directory.

Another great feature of TF Serving is its automatic batching capability, which you can activate using the `--enable_batching` option upon startup. When TF Serving receives multiple requests within a short period of time (the delay is configurable), it will automatically batch them together before using the model. This offers a significant performance boost by leveraging the power of the GPU. Once the model returns the predictions, TF Serving dispatches each prediction to the right client. You can trade a bit of latency for a greater throughput by increasing the batching delay (see the `--batching_parameters_file` option).

If you expect to get many queries per second, you will want to deploy TF Serving on multiple servers and load-balance the queries (see Figure 19-2). This will require deploying and managing many TF Serving containers across these servers. One way to handle that is to use a tool such as Kubernetes, which is an open source system for simplifying container orchestration across many servers. If you do not want to pur-

5 If the SavedModel contains some example instances in the *assets/extra* directory, you can configure TF Serving to execute the model on these instances before starting to serve new requests with it. This is called *model warmup*: it will ensure that everything is properly loaded, avoiding long response times for the first requests.

chase, maintain, and upgrade all the hardware infrastructure, you will want to use virtual machines on a cloud platform such as Amazon AWS, Microsoft Azure, Google Cloud Platform, IBM Cloud, Alibaba Cloud, Oracle Cloud, or some other Platform-as-a-Service (PaaS). Managing all the virtual machines, handling container orchestration (even with the help of Kubernetes), taking care of TF Serving configuration, tuning and monitoring—all of this can be a full-time job. Fortunately, some service providers can take care of all this for you. In this chapter we will use Google Cloud AI Platform because it's the only platform with TPUs today, it supports TensorFlow 2, it offers a nice suite of AI services (e.g., AutoML, Vision API, Natural Language API), and it is the one I have the most experience with. But there are several other providers in this space, such as Amazon AWS SageMaker and Microsoft AI Platform, which are also capable of serving TensorFlow models.

Figure 19-2. Scaling up TF Serving with load balancing

Now let's see how to serve our wonderful MNIST model on the cloud!

Creating a Prediction Service on GCP AI Platform

Before you can deploy a model, there's a little bit of setup to take care of:

1. Log in to your Google account, and then go to the Google Cloud Platform (GCP) console (see Figure 19-3). If you don't have a Google account, you'll have to create one.
2. If it is your first time using GCP, you will have to read and accept the terms and conditions. Click Tour Console if you want. At the time of this writing, new users are offered a free trial, including $300 worth of GCP credit that you can use over the course of 12 months. You will only need a small portion of that to pay for the services you will use in this chapter. Upon signing up for the free trial, you will still need to create a payment profile and enter your credit card number: it is used for verification purposes (probably to avoid people using the free trial multiple times), but you will not be billed. Activate and upgrade your account if requested.

Figure 19-3. Google Cloud Platform console

3. If you have used GCP before and your free trial has expired, then the services you will use in this chapter will cost you some money. It should not be too much, especially if you remember to turn off the services when you do not need them anymore. Make sure you understand and agree to the pricing conditions before you run any service. I hereby decline any responsibility if services end up costing more than you expected! Also make sure your billing account is active. To check, open the navigation menu on the left and click Billing, and make sure you have set up a payment method and that the billing account is active.
4. Every resource in GCP belongs to a project. This includes all the virtual machines you may use, the files you store, and the training jobs you run. When you create an account, GCP automatically creates a project for you, called "My First Project." If you want, you can change its display name by going to the project settings: in the navigation menu (on the left of the screen), select IAM & admin → Settings, change the project's display name, and click Save. Note that the project also has a unique ID and number. You can choose the project ID when you create a project, but you cannot change it later. The project number is automatically generated and cannot be changed. If you want to create a new project, click the project name at the top of the page, then click New Project and enter the project ID. Make sure billing is active for this new project.

Always set an alarm to remind yourself to turn services off when you know you will only need them for a few hours, or else you might leave them running for days or months, incurring potentially significant costs.

5. Now that you have a GCP account with billing activated, you can start using the services. The first one you will need is Google Cloud Storage (GCS): this is where you will put the SavedModels, the training data, and more. In the navigation menu, scroll down to the Storage section, and click Storage → Browser. All your files will go in one or more *buckets*. Click Create Bucket and choose the bucket name (you may need to activate the Storage API first). GCS uses a single worldwide namespace for buckets, so simple names like "machine-learning" will most likely not be available. Make sure the bucket name conforms to DNS naming conventions, as it may be used in DNS records. Moreover, bucket names are public, so do not put anything private in there. It is common to use your domain name or your company name as a prefix to ensure uniqueness, or simply use a random number as part of the name. Choose the location where you want the bucket to be hosted, and the rest of the options should be fine by default. Then click Create.
6. Upload the *my_mnist_model* folder you created earlier (including one or more versions) to your bucket. To do this, just go to the GCS Browser, click the bucket, then drag and drop the *my_mnist_model* folder from your system to the bucket (see Figure 19-4). Alternatively, you can click "Upload folder" and select the *my_mnist_model* folder to upload. By default, the maximum size for a SavedModel is 250 MB, but it is possible to request a higher quota.

Figure 19-4. Uploading a SavedModel to Google Cloud Storage

7. Now you need to configure AI Platform (formerly known as ML Engine) so that it knows which models and versions you want to use. In the navigation menu, scroll down to the Artificial Intelligence section, and click AI Platform → Models. Click Activate API (it takes a few minutes), then click "Create model." Fill in the model details (see Figure 19-5) and click Create.

Figure 19-5. Creating a new model on Google Cloud AI Platform

8. Now that you have a model on AI Platform, you need to create a model version. In the list of models, click the model you just created, then click "Create version" and fill in the version details (see Figure 19-6): set the name, description, Python version (3.5 or above), framework (TensorFlow), framework version (2.0 if available, or 1.13),[6] ML runtime version (2.0, if available or 1.13), machine type (choose "Single core CPU" for now), model path on GCS (this is the full path to the actual version folder, e.g., *gs://my-mnist-model-bucket/my_mnist_model/0002/*), scaling (choose automatic), and minimum number of TF Serving containers to have running at all times (leave this field empty). Then click Save.

6 At the time of this writing, TensorFlow version 2 is not available yet on AI Platform, but that's OK: you can use 1.13, and it will run your TF 2 SavedModels just fine.

Figure 19-6. Creating a new model version on Google Cloud AI Platform

Congratulations, you have deployed your first model on the cloud! Because you selected automatic scaling, AI Platform will start more TF Serving containers when the number of queries per second increases, and it will load-balance the queries between them. If the QPS goes down, it will stop containers automatically. The cost is therefore directly linked to the QPS (as well as the type of machine you choose and the amount of data you store on GCS). This pricing model is particularly useful for occasional users and for services with important usage spikes, as well as for startups: the price remains low until the startup actually starts up.

If you do not use the prediction service, AI Platform will stop all containers. This means you will only pay for the amount of storage you use (a few cents per gigabyte per month). Note that when you query the service, AI Platform will need to start up a TF Serving container, which will take a few seconds. If this delay is unacceptable, you will have to set the minimum number of TF Serving containers to 1 when creating the model version. Of course, this means at least one machine will run constantly, so the monthly fee will be higher.

Now let's query this prediction service!

Using the Prediction Service

Under the hood, AI Platform just runs TF Serving, so in principle you could use the same code as earlier, if you knew which URL to query. There's just one problem: GCP also takes care of encryption and authentication. Encryption is based on SSL/TLS, and authentication is token-based: a secret authentication token must be sent to the server in every request. So before your code can use the prediction service (or any other GCP service), it must obtain a token. We will see how to do this shortly, but first you need to configure authentication and give your application the appropriate access rights on GCP. You have two options for authentication:

- Your application (i.e., the client code that will query the prediction service) could authenticate using user credentials with your own Google login and password. Using user credentials would give your application the exact same rights as on GCP, which is certainly way more than it needs. Moreover, you would have to deploy your credentials in your application, so anyone with access could steal your credentials and fully access your GCP account. In short, do not choose this option; it is only needed in very rare cases (e.g., when your application needs to access its user's GCP account).
- The client code can authenticate with a *service account*. This is an account that represents an application, not a user. It is generally given very restricted access rights: strictly what it needs, and no more. This is the recommended option.

So, let's create a service account for your application: in the navigation menu, go to IAM & admin → Service accounts, then click Create Service Account, fill in the form (service account name, ID, description), and click Create (see Figure 19-7). Next, you must give this account some access rights. Select the ML Engine Developer role: this will allow the service account to make predictions, and not much more. Optionally, you can grant some users access to the service account (this is useful when your GCP user account is part of an organization, and you wish to authorize other users in the organization to deploy applications that will be based on this service account or to manage the service account itself). Next, click Create Key to export the service account's private key, choose JSON, and click Create. This will download the private key in the form of a JSON file. Make sure to keep it private!

Figure 19-7. Creating a new service account in Google IAM

Great! Now let's write a small script that will query the prediction service. Google provides several libraries to simplify access to its services:

Google API Client Library
: This is a fairly thin layer on top of *OAuth 2.0* (for the authentication) and REST. You can use it with all GCP services, including AI Platform. You can install it using pip: the library is called `google-api-python-client`.

Google Cloud Client Libraries
: These are a bit more high-level: each one is dedicated to a particular service, such as GCS, Google BigQuery, Google Cloud Natural Language, and Google Cloud Vision. All these libraries can be installed using pip (e.g., the GCS Client Library is called `google-cloud-storage`). When a client library is available for a given service, it is recommended to use it rather than the Google API Client Library, as it implements all the best practices and will often use gRPC rather than REST, for better performance.

At the time of this writing there is no client library for AI Platform, so we will use the Google API Client Library. It will need to use the service account's private key; you can tell it where it is by setting the `GOOGLE_APPLICATION_CREDENTIALS` environment variable, either before starting the script or within the script like this:

```
import os

os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "my_service_account_key.json"
```

If you deploy your application to a virtual machine on Google Cloud Engine (GCE), or within a container using Google Cloud Kubernetes Engine, or as a web application on Google Cloud App Engine, or as a microservice on Google Cloud Functions, and if the `GOOGLE_APPLICATION_CREDENTIALS` environment variable is not set, then the library will use the default service account for the host service (e.g., the default GCE service account, if your application runs on GCE).

Next, you must create a resource object that wraps access to the prediction service:[7]

```
import googleapiclient.discovery

project_id = "onyx-smoke-242003" # change this to your project ID
model_id = "my_mnist_model"
model_path = "projects/{}/models/{}".format(project_id, model_id)
ml_resource = googleapiclient.discovery.build("ml", "v1").projects()
```

Note that you can append `/versions/0001` (or any other version number) to the `model_path` to specify the version you want to query: this can be useful for A/B testing or for testing a new version on a small group of users before releasing it widely (this is called a *canary*). Next, let's write a small function that will use the resource object to call the prediction service and get the predictions back:

```
def predict(X):
    input_data_json = {"signature_name": "serving_default",
                       "instances": X.tolist()}
    request = ml_resource.predict(name=model_path, body=input_data_json)
    response = request.execute()
    if "error" in response:
        raise RuntimeError(response["error"])
    return np.array([pred[output_name] for pred in response["predictions"]])
```

The function takes a NumPy array containing the input images and prepares a dictionary that the client library will convert to the JSON format (as we did earlier). Then it prepares a prediction request, and executes it; it raises an exception if the response contains an error, or else it extracts the predictions for each instance and bundles them in a NumPy array. Let's see if it works:

```
>>> Y_probas = predict(X_new)
>>> np.round(Y_probas, 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 0.  , 0.  ],
       [0.  , 0.  , 0.99, 0.01, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.96, 0.01, 0.  , 0.  , 0.  , 0.  , 0.01, 0.01, 0.  ]])
```

7 If you get an error saying that module `google.appengine` was not found, set `cache_discovery=False` in the call to the `build()` method; see *https://stackoverflow.com/q/55561354*.

Yes! You now have a nice prediction service running on the cloud that can automatically scale up to any number of QPS, plus you can query it from anywhere securely. Moreover, it costs you close to nothing when you don't use it: you'll pay just a few cents per month per gigabyte used on GCS. And you can also get detailed logs and metrics using Google Stackdriver.

But what if you want to deploy your model to a mobile app? Or to an embedded device?

Deploying a Model to a Mobile or Embedded Device

If you need to deploy your model to a mobile or embedded device, a large model may simply take too long to download and use too much RAM and CPU, all of which will make your app unresponsive, heat the device, and drain its battery. To avoid this, you need to make a mobile-friendly, lightweight, and efficient model, without sacrificing too much of its accuracy. The TFLite library provides several tools[8] to help you deploy your models to mobile and embedded devices, with three main objectives:

- Reduce the model size, to shorten download time and reduce RAM usage.
- Reduce the amount of computations needed for each prediction, to reduce latency, battery usage, and heating.
- Adapt the model to device-specific constraints.

To reduce the model size, TFLite's model converter can take a SavedModel and compress it to a much lighter format based on FlatBuffers. This is an efficient cross-platform serialization library (a bit like protocol buffers) initially created by Google for gaming. It is designed so you can load FlatBuffers straight to RAM without any preprocessing: this reduces the loading time and memory footprint. Once the model is loaded into a mobile or embedded device, the TFLite interpreter will execute it to make predictions. Here is how you can convert a SavedModel to a FlatBuffer and save it to a *.tflite* file:

```
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_path)
tflite_model = converter.convert()
with open("converted_model.tflite", "wb") as f:
    f.write(tflite_model)
```

You can also save a tf.keras model directly to a FlatBuffer using `from_keras_model()`.

8 Also check out TensorFlow's Graph Transform Tools for modifying and optimizing computational graphs.

The converter also optimizes the model, both to shrink it and to reduce its latency. It prunes all the operations that are not needed to make predictions (such as training operations), and it optimizes computations whenever possible; for example, $3\times a + 4\times a + 5\times a$ will be converted to $(3 + 4 + 5)\times a$. It also tries to fuse operations whenever possible. For example, Batch Normalization layers end up folded into the previous layer's addition and multiplication operations, whenever possible. To get a good idea of how much TFLite can optimize a model, download one of the pretrained TFLite models, unzip the archive, then open the excellent Netron graph visualization tool and upload the *.pb* file to view the original model. It's a big, complex graph, right? Next, open the optimized *.tflite* model and marvel at its beauty!

Another way you can reduce the model size (other than simply using smaller neural network architectures) is by using smaller bit-widths: for example, if you use half-floats (16 bits) rather than regular floats (32 bits), the model size will shrink by a factor of 2, at the cost of a (generally small) accuracy drop. Moreover, training will be faster, and you will use roughly half the amount of GPU RAM.

TFLite's converter can go further than that, by quantizing the model weights down to fixed-point, 8-bit integers! This leads to a fourfold size reduction compared to using 32-bit floats. The simplest approach is called *post-training quantization*: it just quantizes the weights after training, using a fairly basic but efficient symmetrical quantization technique. It finds the maximum absolute weight value, m, then it maps the floating-point range $-m$ to $+m$ to the fixed-point (integer) range −127 to +127. For example (see Figure 19-8), if the weights range from −1.5 to +0.8, then the bytes −127, 0, and +127 will correspond to the floats −1.5, 0.0, and +1.5, respectively. Note that 0.0 always maps to 0 when using symmetrical quantization (also note that the byte values +68 to +127 will not be used, since they map to floats greater than +0.8).

Figure 19-8. From 32-bit floats to 8-bit integers, using symmetrical quantization

To perform this post-training quantization, simply add `OPTIMIZE_FOR_SIZE` to the list of converter optimizations before calling the `convert()` method:

```
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
```

This technique dramatically reduces the model's size, so it's much faster to download and store. However, at runtime the quantized weights get converted back to floats before they are used (these recovered floats are not perfectly identical to the original

floats, but not too far off, so the accuracy loss is usually acceptable). To avoid recomputing them all the time, the recovered floats are cached, so there is no reduction of RAM usage. And there is no reduction either in compute speed.

The most effective way to reduce latency and power consumption is to also quantize the activations so that the computations can be done entirely with integers, without the need for any floating-point operations. Even when using the same bit-width (e.g., 32-bit integers instead of 32-bit floats), integer computations use less CPU cycles, consume less energy, and produce less heat. And if you also reduce the bit-width (e.g., down to 8-bit integers), you can get huge speedups. Moreover, some neural network accelerator devices (such as the Edge TPU) can only process integers, so full quantization of both weights and activations is compulsory. This can be done post-training; it requires a calibration step to find the maximum absolute value of the activations, so you need to provide a representative sample of training data to TFLite (it does not need to be huge), and it will process the data through the model and measure the activation statistics required for quantization (this step is typically fast).

The main problem with quantization is that it loses a bit of accuracy: it is equivalent to adding noise to the weights and activations. If the accuracy drop is too severe, then you may need to use *quantization-aware training*. This means adding fake quantization operations to the model so it can learn to ignore the quantization noise during training; the final weights will then be more robust to quantization. Moreover, the calibration step can be taken care of automatically during training, which simplifies the whole process.

I have explained the core concepts of TFLite, but going all the way to coding a mobile app or an embedded program would require a whole other book. Fortunately, one exists: if you want to learn more about building TensorFlow applications for mobile and embedded devices, check out the O'Reilly book *TinyML: Machine Learning with TensorFlow on Arduino and Ultra-Low Power Micro-Controllers*, by Pete Warden (who leads the TFLite team) and Daniel Situnayake.

TensorFlow in the Browser

What if you want to use your model in a website, running directly in the user's browser? This can be useful in many scenarios, such as:

- When your web application is often used in situations where the user's connectivity is intermittent or slow (e.g., a website for hikers), so running the model directly on the client side is the only way to make your website reliable.
- When you need the model's responses to be as fast as possible (e.g., for an online game). Removing the need to query the server to make predictions will definitely reduce the latency and make the website much more responsive.
- When your web service makes predictions based on some private user data, and you want to protect the user's privacy by making the predictions on the client side so that the private data never has to leave the user's machine.[9]

For all these scenarios, you can export your model to a special format that can be loaded by the TensorFlow.js JavaScript library. This library can then use your model to make predictions directly in the user's browser. The TensorFlow.js project includes a `tensorflowjs_converter` tool that can convert a TensorFlow SavedModel or a Keras model file to the *TensorFlow.js Layers* format: this is a directory containing a set of sharded weight files in binary format and a *model.json* file that describes the model's architecture and links to the weight files. This format is optimized to be downloaded efficiently on the web. Users can then download the model and run predictions in the browser using the TensorFlow.js library. Here is a code snippet to give you an idea of what the JavaScript API looks like:

```
import * as tf from '@tensorflow/tfjs';
const model = await tf.loadLayersModel('https://example.com/tfjs/model.json');
const image = tf.fromPixels(webcamElement);
const prediction = model.predict(image);
```

Once again, doing justice to this topic would require a whole book. If you want to learn more about TensorFlow.js, check out the O'Reilly book *Practical Deep Learning for Cloud, Mobile, and Edge*, by Anirudh Koul, Siddha Ganju, and Meher Kasam.

Next, we will see how to use GPUs to speed up computations!

9 If you're interested in this topic, check out *federated learning*.

Using GPUs to Speed Up Computations

In Chapter 11 we discussed several techniques that can considerably speed up training: better weight initialization, Batch Normalization, sophisticated optimizers, and so on. But even with all of these techniques, training a large neural network on a single machine with a single CPU can take days or even weeks.

In this section we will look at how to speed up your models by using GPUs. We will also see how to split the computations across multiple devices, including the CPU and multiple GPU devices (see Figure 19-9). For now we will run everything on a single machine, but later in this chapter we will discuss how to distribute computations across multiple servers.

Figure 19-9. Executing a TensorFlow graph across multiple devices in parallel

Thanks to GPUs, instead of waiting for days or weeks for a training algorithm to complete, you may end up waiting for just a few minutes or hours. Not only does this save an enormous amount of time, but it also means that you can experiment with various models much more easily and frequently retrain your models on fresh data.

You can often get a major performance boost simply by adding GPU cards to a single machine. In fact, in many cases this will suffice; you won't need to use multiple machines at all. For example, you can typically train a neural network just as fast using four GPUs on a single machine rather than eight GPUs across multiple machines, due to the extra delay imposed by network communications in a distributed setup. Similarly, using a single powerful GPU is often preferable to using multiple slower GPUs.

The first step is to get your hands on a GPU. There are two options for this: you can either purchase your own GPU(s), or you can use GPU-equipped virtual machines on the cloud. Let's start with the first option.

Getting Your Own GPU

If you choose to purchase a GPU card, then take some time to make the right choice. Tim Dettmers wrote an excellent blog post to help you choose, and he updates it regularly: I encourage you to read it carefully. At the time of this writing, TensorFlow only supports Nvidia cards with CUDA Compute Capability 3.5+ (as well as Google's TPUs, of course), but it may extend its support to other manufacturers. Moreover, although TPUs are currently only available on GCP, it is highly likely that TPU-like cards will be available for sale in the near future, and TensorFlow may support them. In short, make sure to check TensorFlow's documentation to see what devices are supported at this point.

If you go for an Nvidia GPU card, you will need to install the appropriate Nvidia drivers and several Nvidia libraries.[10] These include the *Compute Unified Device Architecture* library (CUDA), which allows developers to use CUDA-enabled GPUs for all sorts of computations (not just graphics acceleration), and the *CUDA Deep Neural Network* library (cuDNN), a GPU-accelerated library of primitives for DNNs. cuDNN provides optimized implementations of common DNN computations such as activation layers, normalization, forward and backward convolutions, and pooling (see Chapter 14). It is part of Nvidia's Deep Learning SDK (note that you'll need to create an Nvidia developer account in order to download it). TensorFlow uses CUDA and cuDNN to control the GPU cards and accelerate computations (see Figure 19-10).

Figure 19-10. TensorFlow uses CUDA and cuDNN to control GPUs and boost DNNs

10 Please check the docs for detailed and up-to-date installation instructions, as they change quite often.

Once you have installed the GPU card(s) and all the required drivers and libraries, you can use the `nvidia-smi` command to check that CUDA is properly installed. It lists the available GPU cards, as well as processes running on each card:

```
$ nvidia-smi
Sun Jun  2 10:05:22 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 410.79       CUDA Version: 10.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   61C    P8    17W /  70W |      0MiB / 15079MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

At the time of this writing, you'll also need to install the GPU version of TensorFlow (i.e., the `tensorflow-gpu` library); however, there is ongoing work to have a unified installation procedure for both CPU-only and GPU machines, so please check the installation documentation to see which library you should install. In any case, since installing every required library correctly is a bit long and tricky (and all hell breaks loose if you do not install the correct library versions), TensorFlow provides a Docker image with everything you need inside. However, in order for the Docker container to have access to the GPU, you will still need to install the Nvidia drivers on the host machine.

To check that TensorFlow actually sees the GPUs, run the following tests:

```
>>> import tensorflow as tf
>>> tf.test.is_gpu_available()
True
>>> tf.test.gpu_device_name()
'/device:GPU:0'
>>> tf.config.experimental.list_physical_devices(device_type='GPU')
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

The `is_gpu_available()` function checks whether at least one GPU is available. The `gpu_device_name()` function gives the first GPU's name: by default, operations will

run on this GPU. The `list_physical_devices()` function returns the list of all available GPU devices (just one in this example).[11]

Now, what if you don't want to invest time and money in getting your own GPU card? Just use a GPU VM on the cloud!

Using a GPU-Equipped Virtual Machine

All major cloud platforms now offer GPU VMs, some preconfigured with all the drivers and libraries you need (including TensorFlow). Google Cloud Platform enforces various GPU quotas, both worldwide and per region: you cannot just create thousands of GPU VMs without prior authorization from Google.[12] By default, the worldwide GPU quota is zero, so you cannot use any GPU VMs. Therefore, the very first thing you need to do is to request a higher worldwide quota. In the GCP console, open the navigation menu and go to IAM & admin → Quotas. Click Metric, click None to uncheck all locations, then search for "GPU" and select "GPUs (all regions)" to see the corresponding quota. If this quota's value is zero (or just insufficient for your needs), then check the box next to it (it should be the only selected one) and click "Edit quotas." Fill in the requested information, then click "Submit request." It may take a few hours (or up to a few days) for your quota request to be processed and (generally) accepted. By default, there is also a quota of one GPU per region and per GPU type. You can request to increase these quotas too: click Metric, select None to uncheck all metrics, search for "GPU," and select the type of GPU you want (e.g., NVIDIA P4 GPUs). Then click the Location drop-down menu, click None to uncheck all metrics, and click the location you want; check the boxes next to the quota(s) you want to change, and click "Edit quotas" to file a request.

Once your GPU quota requests are approved, you can in no time create a VM equipped with one or more GPUs by using Google Cloud AI Platform's *Deep Learning VM Images*: go to *https://homl.info/dlvm*, click View Console, then click "Launch on Compute Engine" and fill in the VM configuration form. Note that some locations do not have all types of GPUs, and some have no GPUs at all (change the location to see the types of GPUs available, if any). Make sure to select TensorFlow 2.0 as the framework, and check "Install NVIDIA GPU driver automatically on first startup." It is also a good idea to check "Enable access to JupyterLab via URL instead of SSH": this will make it very easy to start a Jupyter notebook running on this GPU VM, powered by

11 Many code examples in this chapter use experimental APIs. They are very likely to be moved to the core API in future versions. So if an experimental function fails, try simply removing the word `experimental`, and hopefully it will work. If not, then perhaps the API has changed a bit; please check the Jupyter notebook, as I will ensure it contains the correct code.

12 Presumably, these quotas are meant to stop bad guys who might be tempted to use GCP with stolen credit cards to mine cryptocurrencies.

JupyterLab (this is an alternative web interface to run Jupyter notebooks). Once the VM is created, scroll down the navigation menu to the Artificial Intelligence section, then click AI Platform → Notebooks. Once the Notebook instance appears in the list (this may take a few minutes, so click Refresh once in a while until it appears), click its Open JupyterLab link. This will run JupyterLab on the VM and connect your browser to it. You can create notebooks and run any code you want on this VM, and benefit from its GPUs!

But if you just want to run some quick tests or easily share notebooks with your colleagues, then you should try Colaboratory.

Colaboratory

The simplest and cheapest way to access a GPU VM is to use *Colaboratory* (or *Colab*, for short). It's free! Just go to *https://colab.research.google.com/* and create a new Python 3 notebook: this will create a Jupyter notebook, stored on your Google Drive (alternatively, you can open any notebook on GitHub, or on Google Drive, or you can even upload your own notebooks). Colab's user interface is similar to Jupyter's, except you can share and use the notebooks like regular Google Docs, and there are a few other minor differences (e.g., you can create handy widgets using special comments in your code).

When you open a Colab notebook, it runs on a free Google VM dedicated to that notebook, called a *Colab Runtime* (see Figure 19-11). By default the Runtime is CPU-only, but you can change this by going to Runtime → "Change runtime type," selecting GPU in the "Hardware accelerator" drop-down menu, then clicking Save. In fact, you could even select TPU! (Yes, you can actually use a TPU for free; we will talk about TPUs later in this chapter, though, so for now just select GPU.)

Figure 19-11. Colab Runtimes and notebooks

Colab does have some restrictions: first, there is a limit to the number of Colab notebooks you can run simultaneously (currently 5 per Runtime type). Moreover, as the FAQ states, "Colaboratory is intended for interactive use. Long-running background computations, particularly on GPUs, may be stopped. Please do not use Colaboratory for cryptocurrency mining." Also, the web interface will automatically disconnect from the Colab Runtime if you leave it unattended for a while (~30 minutes). When you reconnect to the Colab Runtime, it may have been reset, so make sure you always export any data you care about (e.g., download it or save it to Google Drive). Even if you never disconnect, the Colab Runtime will automatically shut down after 12 hours, as it is not meant for long-running computations. Despite these limitations, it's a fantastic tool to run tests easily, get quick results, and collaborate with your colleagues.

Managing the GPU RAM

By default TensorFlow automatically grabs all the RAM in all available GPUs the first time you run a computation. It does this to limit GPU RAM fragmentation. This means that if you try to start a second TensorFlow program (or any program that requires the GPU), it will quickly run out of RAM. This does not happen as often as you might think, as you will most often have a single TensorFlow program running on a machine: usually a training script, a TF Serving node, or a Jupyter notebook. If you need to run multiple programs for some reason (e.g., to train two different models in parallel on the same machine), then you will need to split the GPU RAM between these processes more evenly.

If you have multiple GPU cards on your machine, a simple solution is to assign each of them to a single process. To do this, you can set the `CUDA_VISIBLE_DEVICES` environment variable so that each process only sees the appropriate GPU card(s). Also set the `CUDA_DEVICE_ORDER` environment variable to `PCI_BUS_ID` to ensure that

each ID always refers to the same GPU card. For example, if you have four GPU cards, you could start two programs, assigning two GPUs to each of them, by executing commands like the following in two separate terminal windows:

```
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py
# and in another terminal:
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

Program 1 will then only see GPU cards 0 and 1, named `/gpu:0` and `/gpu:1` respectively, and program 2 will only see GPU cards 2 and 3, named `/gpu:1` and `/gpu:0` respectively (note the order). Everything will work fine (see Figure 19-12). Of course, you can also define these environment variables in Python by setting `os.environ["CUDA_DEVICE_ORDER"]` and `os.environ["CUDA_VISIBLE_DEVICES"]`, as long as you do so before using TensorFlow.

Figure 19-12. Each program gets two GPUs

Another option is to tell TensorFlow to grab only a specific amount of GPU RAM. This must be done immediately after importing TensorFlow. For example, to make TensorFlow grab only 2 GiB of RAM on each GPU, you must create a *virtual GPU device* (also called a *logical GPU device*) for each physical GPU device and set its memory limit to 2 GiB (i.e., 2,048 MiB):

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):
    tf.config.experimental.set_virtual_device_configuration(
        gpu,
        [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048)])
```

Now (supposing you have four GPUs, each with at least 4 GiB of RAM) two programs like this one can run in parallel, each using all four GPU cards (see Figure 19-13).

Figure 19-13. Each program gets all four GPUs, but with only 2 GiB of RAM on each GPU

If you run the `nvidia-smi` command while both programs are running, you should see that each process holds 2 GiB of RAM on each card:

```
$ nvidia-smi
[...]
+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      2373      C   /usr/bin/python3                            2241MiB |
|    0      2533      C   /usr/bin/python3                            2241MiB |
|    1      2373      C   /usr/bin/python3                            2241MiB |
|    1      2533      C   /usr/bin/python3                            2241MiB |
[...]
```

Yet another option is to tell TensorFlow to grab memory only when it needs it (this also must be done immediately after importing TensorFlow):

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):
    tf.config.experimental.set_memory_growth(gpu, True)
```

Another way to do this is to set the `TF_FORCE_GPU_ALLOW_GROWTH` environment variable to `true`. With this option, TensorFlow will never release memory once it has grabbed it (again, to avoid memory fragmentation), except of course when the program ends. It can be harder to guarantee deterministic behavior using this option (e.g., one program may crash because another program's memory usage went through the roof), so in production you'll probably want to stick with one of the previous options. However, there are some cases where it is very useful: for example, when you use a machine to run multiple Jupyter notebooks, several of which use TensorFlow. This is why the `TF_FORCE_GPU_ALLOW_GROWTH` environment variable is set to `true` in Colab Runtimes.

Lastly, in some cases you may want to split a GPU into two or more *virtual GPUs*—for example, if you want to test a distribution algorithm (this is a handy way to try out the code examples in the rest of this chapter even if you have a single GPU, such

as in a Colab Runtime). The following code splits the first GPU into two virtual devices, with 2 GiB of RAM each (again, this must be done immediately after importing TensorFlow):

```
physical_gpus = tf.config.experimental.list_physical_devices("GPU")
tf.config.experimental.set_virtual_device_configuration(
    physical_gpus[0],
    [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048),
     tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048)])
```

These two virtual devices will then be called `/gpu:0` and `/gpu:1`, and you can place operations and variables on each of them as if they were really two independent GPUs. Now let's see how TensorFlow decides which devices it should place variables and execute operations on.

Placing Operations and Variables on Devices

The TensorFlow whitepaper[13] presents a friendly *dynamic placer* algorithm that automagically distributes operations across all available devices, taking into account things like the measured computation time in previous runs of the graph, estimations of the size of the input and output tensors for each operation, the amount of RAM available in each device, communication delay when transferring data into and out of devices, and hints and constraints from the user. In practice this algorithm turned out to be less efficient than a small set of placement rules specified by the user, so the TensorFlow team ended up dropping the dynamic placer.

That said, tf.keras and tf.data generally do a good job of placing operations and variables where they belong (e.g., heavy computations on the GPU, and data preprocessing on the CPU). But you can also place operations and variables manually on each device, if you want more control:

- As just mentioned, you generally want to place the data preprocessing operations on the CPU, and place the neural network operations on the GPUs.
- GPUs usually have a fairly limited communication bandwidth, so it is important to avoid unnecessary data transfers in and out of the GPUs.
- Adding more CPU RAM to a machine is simple and fairly cheap, so there's usually plenty of it, whereas the GPU RAM is baked into the GPU: it is an expensive and thus limited resource, so if a variable is not needed in the next few training steps, it should probably be placed on the CPU (e.g., datasets generally belong on the CPU).

13 Martín Abadi et al., "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems" Google Research whitepaper (2015).

By default, all variables and all operations will be placed on the first GPU (named `/gpu:0`), except for variables and operations that don't have a GPU kernel:[14] these are placed on the CPU (named `/cpu:0`). A tensor or variable's `device` attribute tells you which device it was placed on:[15]

```
>>> a = tf.Variable(42.0)
>>> a.device
'/job:localhost/replica:0/task:0/device:GPU:0'
>>> b = tf.Variable(42)
>>> b.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

You can safely ignore the prefix `/job:localhost/replica:0/task:0` for now (it allows you to place operations on other machines when using a TensorFlow cluster; we will talk about jobs, replicas, and tasks later in this chapter). As you can see, the first variable was placed on GPU 0, which is the default device. However, the second variable was placed on the CPU: this is because there are no GPU kernels for integer variables (or for operations involving integer tensors), so TensorFlow fell back to the CPU.

If you want to place an operation on a different device than the default one, use a `tf.device()` context:

```
>>> with tf.device("/cpu:0"):
...     c = tf.Variable(42.0)
...
>>> c.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

The CPU is always treated as a single device (`/cpu:0`), even if your machine has multiple CPU cores. Any operation placed on the CPU may run in parallel across multiple cores if it has a multi-threaded kernel.

If you explicitly try to place an operation or variable on a device that does not exist or for which there is no kernel, then you will get an exception. However, in some cases you may prefer to fall back to the CPU; for example, if your program may run both on CPU-only machines and on GPU machines, you may want TensorFlow to ignore your `tf.device("/gpu:*")` on CPU-only machines. To do this, you can call `tf.config.set_soft_device_placement(True)` just after importing TensorFlow: when a

14 As we saw in Chapter 12, a kernel is a variable or operation's implementation for a specific data type and device type. For example, there is a GPU kernel for the `float32` `tf.matmul()` operation, but there is no GPU kernel for `int32` `tf.matmul()` (only a CPU kernel).

15 You can also use `tf.debugging.set_log_device_placement(True)` to log all device placements.

placement request fails, TensorFlow will fall back to its default placement rules (i.e., GPU 0 by default if it exists and there is a GPU kernel, and CPU 0 otherwise).

Now how exactly will TensorFlow execute all these operations across multiple devices?

Parallel Execution Across Multiple Devices

As we saw in Chapter 12, one of the benefits of using TF Functions is parallelism. Let's look at this a bit more closely. When TensorFlow runs a TF Function, it starts by analyzing its graph to find the list of operations that need to be evaluated, and it counts how many dependencies each of them has. TensorFlow then adds each operation with zero dependencies (i.e., each source operation) to the evaluation queue of this operation's device (see Figure 19-14). Once an operation has been evaluated, the dependency counter of each operation that depends on it is decremented. Once an operation's dependency counter reaches zero, it is pushed to the evaluation queue of its device. And once all the nodes that TensorFlow needs have been evaluated, it returns their outputs.

Figure 19-14. Parallelized execution of a TensorFlow graph

Operations in the CPU's evaluation queue are dispatched to a thread pool called the *inter-op thread pool*. If the CPU has multiple cores, then these operations will effectively be evaluated in parallel. Some operations have multithreaded CPU kernels: these kernels split their tasks into multiple suboperations, which are placed in another evaluation queue and dispatched to a second thread pool called the *intra-op*

thread pool (shared by all multithreaded CPU kernels). In short, multiple operations and suboperations may be evaluated in parallel on different CPU cores.

For the GPU, things are a bit simpler. Operations in a GPU's evaluation queue are evaluated sequentially. However, most operations have multithreaded GPU kernels, typically implemented by libraries that TensorFlow depends on, such as CUDA and cuDNN. These implementations have their own thread pools, and they typically exploit as many GPU threads as they can (which is the reason why there is no need for an inter-op thread pool in GPUs: each operation already floods most GPU threads).

For example, in Figure 19-14, operations A, B, and C are source ops, so they can immediately be evaluated. Operations A and B are placed on the CPU, so they are sent to the CPU's evaluation queue, then they are dispatched to the inter-op thread pool and immediately evaluated in parallel. Operation A happens to have a multithreaded kernel; its computations are split into three parts, which are executed in parallel by the intra-op thread pool. Operation C goes to GPU 0's evaluation queue, and in this example its GPU kernel happens to use cuDNN, which manages its own intra-op thread pool and runs the operation across many GPU threads in parallel. Suppose C finishes first. The dependency counters of D and E are decremented and they reach zero, so both operations are pushed to GPU 0's evaluation queue, and they are executed sequentially. Note that C only gets evaluated once, even though both D and E depend on it. Suppose B finishes next. Then F's dependency counter is decremented from 4 to 3, and since that's not 0, it does not run yet. Once A, D, and E are finished, then F's dependency counter reaches 0, and it is pushed to the CPU's evaluation queue and evaluated. Finally, TensorFlow returns the requested outputs.

An extra bit of magic that TensorFlow performs is when the TF Function modifies a stateful resource, such as a variable: it ensures that the order of execution matches the order in the code, even if there is no explicit dependency between the statements. For example, if your TF Function contains `v.assign_add(1)` followed by `v.assign(v * 2)`, TensorFlow will ensure that these operations are executed in that order.

You can control the number of threads in the inter-op thread pool by calling `tf.config.threading.set_inter_op_parallelism_threads()`. To set the number of intra-op threads, use `tf.config.threading.set_intra_op_parallelism_threads()`. This is useful if you want do not want TensorFlow to use all the CPU cores or if you want it to be single-threaded.[16]

16 This can be useful if you want to guarantee perfect reproducibility, as I explain in this video, based on TF 1.

With that, you have all you need to run any operation on any device, and exploit the power of your GPUs! Here are some of the things you could do:

- You could train several models in parallel, each on its own GPU: just write a training script for each model and run them in parallel, setting `CUDA_DEVICE_ORDER` and `CUDA_VISIBLE_DEVICES` so that each script only sees a single GPU device. This is great for hyperparameter tuning, as you can train in parallel multiple models with different hyperparameters. If you have a single machine with two GPUs, and it takes one hour to train one model on one GPU, then training two models in parallel, each on its own dedicated GPU, will take just one hour. Simple!
- You could train a model on a single GPU and perform all the preprocessing in parallel on the CPU, using the dataset's `prefetch()` method[17] to prepare the next few batches in advance so that they are ready when the GPU needs them (see Chapter 13).
- If your model takes two images as input and processes them using two CNNs before joining their outputs, then it will probably run much faster if you place each CNN on a different GPU.
- You can create an efficient ensemble: just place a different trained model on each GPU so that you can get all the predictions much faster to produce the ensemble's final prediction.

But what if you want to *train* a single model across multiple GPUs?

Training Models Across Multiple Devices

There are two main approaches to training a single model across multiple devices: *model parallelism*, where the model is split across the devices, and *data parallelism*, where the model is replicated across every device, and each replica is trained on a subset of the data. Let's look at these two options closely before we train a model on multiple GPUs.

Model Parallelism

So far we have trained each neural network on a single device. What if we want to train a single neural network across multiple devices? This requires chopping the model into separate chunks and running each chunk on a different device.

17 At the time of this writing it only prefetches the data to the CPU RAM, but you can use `tf.data.experimental.prefetch_to_device()` to make it prefetch the data and push it to the device of your choice so that the GPU does not waste time waiting for the data to be transferred.

Unfortunately, such model parallelism turns out to be pretty tricky, and it really depends on the architecture of your neural network. For fully connected networks, there is generally not much to be gained from this approach (see Figure 19-15). Intuitively, it may seem that an easy way to split the model is to place each layer on a different device, but this does not work because each layer needs to wait for the output of the previous layer before it can do anything. So perhaps you can slice it vertically—for example, with the left half of each layer on one device, and the right part on another device? This is slightly better, since both halves of each layer can indeed work in parallel, but the problem is that each half of the next layer requires the output of both halves, so there will be a lot of cross-device communication (represented by the dashed arrows). This is likely to completely cancel out the benefit of the parallel computation, since cross-device communication is slow (especially when the devices are located on different machines).

Figure 19-15. Splitting a fully connected neural network

Some neural network architectures, such as convolutional neural networks (see Chapter 14), contain layers that are only partially connected to the lower layers, so it is much easier to distribute chunks across devices in an efficient way (Figure 19-16).

Figure 19-16. Splitting a partially connected neural network

Deep recurrent neural networks (see Chapter 15) can be split a bit more efficiently across multiple GPUs. If you split the network horizontally by placing each layer on a different device, and you feed the network with an input sequence to process, then at the first time step only one device will be active (working on the sequence's first value), at the second step two will be active (the second layer will be handling the output of the first layer for the first value, while the first layer will be handling the second value), and by the time the signal propagates to the output layer, all devices will be active simultaneously (Figure 19-17). There is still a lot of cross-device communication going on, but since each cell may be fairly complex, the benefit of running multiple cells in parallel may (in theory) outweigh the communication penalty. However, in practice a regular stack of `LSTM` layers running on a single GPU actually runs much faster.

Figure 19-17. Splitting a deep recurrent neural network

In short, model parallelism may speed up running or training some types of neural networks, but not all, and it requires special care and tuning, such as making sure that devices that need to communicate the most run on the same machine.[18] Let's look at a much simpler and generally more efficient option: data parallelism.

Data Parallelism

Another way to parallelize the training of a neural network is to replicate it on every device and run each training step simultaneously on all replicas, using a different mini-batch for each. The gradients computed by each replica are then averaged, and the result is used to update the model parameters. This is called *data parallelism*. There are many variants of this idea, so let's look at the most important ones.

Data parallelism using the mirrored strategy

Arguably the simplest approach is to completely mirror all the model parameters across all the GPUs and always apply the exact same parameter updates on every GPU. This way, all replicas always remain perfectly identical. This is called the *mirrored strategy*, and it turns out to be quite efficient, especially when using a single machine (see Figure 19-18).

18 If you are interested in going further with model parallelism, check out Mesh TensorFlow.

Figure 19-18. Data parallelism using the mirrored strategy

The tricky part when using this approach is to efficiently compute the mean of all the gradients from all the GPUs and distribute the result across all the GPUs. This can be done using an *AllReduce* algorithm, a class of algorithms where multiple nodes collaborate to efficiently perform a reduce operation (such as computing the mean, sum, and max), while ensuring that all nodes obtain the same final result. Fortunately, there are off-the-shelf implementations of such algorithms, as we will see.

Data parallelism with centralized parameters

Another approach is to store the model parameters outside of the GPU devices performing the computations (called *workers*), for example on the CPU (see Figure 19-19). In a distributed setup, you may place all the parameters on one or more CPU-only servers called *parameter servers*, whose only role is to host and update the parameters.

Figure 19-19. Data parallelism with centralized parameters

Whereas the mirrored strategy imposes synchronous weight updates across all GPUs, this centralized approach allows either synchronous or asynchronous updates. Let's see the pros and cons of both options.

Synchronous updates. With *synchronous updates*, the aggregator waits until all gradients are available before it computes the average gradients and passes them to the optimizer, which will update the model parameters. Once a replica has finished computing its gradients, it must wait for the parameters to be updated before it can proceed to the next mini-batch. The downside is that some devices may be slower than others, so all other devices will have to wait for them at every step. Moreover, the parameters will be copied to every device almost at the same time (immediately after the gradients are applied), which may saturate the parameter servers' bandwidth.

To reduce the waiting time at each step, you could ignore the gradients from the slowest few replicas (typically ~10%). For example, you could run 20 replicas, but only aggregate the gradients from the fastest 18 replicas at each step, and just ignore the gradients from the last 2. As soon as the parameters are updated, the first 18 replicas can start working again immediately, without having to wait for the 2 slowest replicas. This setup is generally described as having 18 replicas plus 2 *spare replicas*.[19]

19 This name is slightly confusing because it sounds like some replicas are special, doing nothing. In reality, all replicas are equivalent: they all work hard to be among the fastest at each training step, and the losers vary at every step (unless some devices are really slower than others). However, it does mean that if a server crashes, training will continue just fine.

Asynchronous updates. With asynchronous updates, whenever a replica has finished computing the gradients, it immediately uses them to update the model parameters. There is no aggregation (it removes the "mean" step in Figure 19-19) and no synchronization. Replicas work independently of the other replicas. Since there is no waiting for the other replicas, this approach runs more training steps per minute. Moreover, although the parameters still need to be copied to every device at every step, this happens at different times for each replica, so the risk of bandwidth saturation is reduced.

Data parallelism with asynchronous updates is an attractive choice because of its simplicity, the absence of synchronization delay, and a better use of the bandwidth. However, although it works reasonably well in practice, it is almost surprising that it works at all! Indeed, by the time a replica has finished computing the gradients based on some parameter values, these parameters will have been updated several times by other replicas (on average $N - 1$ times, if there are N replicas), and there is no guarantee that the computed gradients will still be pointing in the right direction (see Figure 19-20). When gradients are severely out-of-date, they are called *stale gradients*: they can slow down convergence, introducing noise and wobble effects (the learning curve may contain temporary oscillations), or they can even make the training algorithm diverge.

Figure 19-20. Stale gradients when using asynchronous updates

There are a few ways you can reduce the effect of stale gradients:

- Reduce the learning rate.
- Drop stale gradients or scale them down.
- Adjust the mini-batch size.

- Start the first few epochs using just one replica (this is called the *warmup phase*). Stale gradients tend to be more damaging at the beginning of training, when gradients are typically large and the parameters have not settled into a valley of the cost function yet, so different replicas may push the parameters in quite different directions.

A paper published by the Google Brain team in 2016[20] benchmarked various approaches and found that using synchronous updates with a few spare replicas was more efficient than using asynchronous updates, not only converging faster but also producing a better model. However, this is still an active area of research, so you should not rule out asynchronous updates just yet.

Bandwidth saturation

Whether you use synchronous or asynchronous updates, data parallelism with centralized parameters still requires communicating the model parameters from the parameter servers to every replica at the beginning of each training step, and the gradients in the other direction at the end of each training step. Similarly, when using the mirrored strategy, the gradients produced by each GPU will need to be shared with every other GPU. Unfortunately, there always comes a point where adding an extra GPU will not improve performance at all because the time spent moving the data into and out of GPU RAM (and across the network in a distributed setup) will outweigh the speedup obtained by splitting the computation load. At that point, adding more GPUs will just worsen the bandwidth saturation and actually slow down training.

For some models, typically relatively small and trained on a very large training set, you are often better off training the model on a single machine with a single powerful GPU with a large memory bandwidth.

Saturation is more severe for large dense models, since they have a lot of parameters and gradients to transfer. It is less severe for small models (but the parallelization gain is limited) and for large sparse models, where the gradients are typically mostly zeros and so can be communicated efficiently. Jeff Dean, initiator and lead of the Google Brain project, reported typical speedups of 25–40× when distributing computations across 50 GPUs for dense models, and a 300× speedup for sparser models trained across 500 GPUs. As you can see, sparse models really do scale better. Here are a few concrete examples:

20 Jianmin Chen et al., "Revisiting Distributed Synchronous SGD," arXiv preprint arXiv:1604.00981 (2016).

- Neural machine translation: 6× speedup on 8 GPUs
- Inception/ImageNet: 32× speedup on 50 GPUs
- RankBrain: 300× speedup on 500 GPUs

Beyond a few dozen GPUs for a dense model or few hundred GPUs for a sparse model, saturation kicks in and performance degrades. There is plenty of research going on to solve this problem (exploring peer-to-peer architectures rather than centralized parameter servers, using lossy model compression, optimizing when and what the replicas need to communicate, and so on), so there will likely be a lot of progress in parallelizing neural networks in the next few years.

In the meantime, to reduce the saturation problem, you probably want to use a few powerful GPUs rather than plenty of weak GPUs, and you should also group your GPUs on few and very well interconnected servers. You can also try dropping the float precision from 32 bits (`tf.float32`) to 16 bits (`tf.bfloat16`). This will cut in half the amount of data to transfer, often without much impact on the convergence rate or the model's performance. Lastly, if you are using centralized parameters, you can shard (split) the parameters across multiple parameter servers: adding more parameter servers will reduce the network load on each server and limit the risk of bandwidth saturation.

OK, now let's train a model across multiple GPUs!

Training at Scale Using the Distribution Strategies API

Many models can be trained quite well on a single GPU, or even on a CPU. But if training is too slow, you can try distributing it across multiple GPUs on the same machine. If that's still too slow, try using more powerful GPUs, or add more GPUs to the machine. If your model performs heavy computations (such as large matrix multiplications), then it will run much faster on powerful GPUs, and you could even try to use TPUs on Google Cloud AI Platform, which will usually run even faster for such models. But if you can't fit any more GPUs on the same machine, and if TPUs aren't for you (e.g., perhaps your model doesn't benefit much from TPUs, or perhaps you want to use your own hardware infrastructure), then you can try training it across several servers, each with multiple GPUs (if this is still not enough, as a last resort you can try adding some model parallelism, but this requires a lot more effort). In this section we will see how to train models at scale, starting with multiple GPUs on the same machine (or TPUs) and then moving on to multiple GPUs across multiple machines.

Luckily, TensorFlow comes with a very simple API that takes care of all the complexity for you: the *Distribution Strategies API*. To train a Keras model across all available GPUs (on a single machine, for now) using data parallelism with the mirrored

strategy, create a `MirroredStrategy` object, call its `scope()` method to get a distribution context, and wrap the creation and compilation of your model inside that context. Then call the model's `fit()` method normally:

```
distribution = tf.distribute.MirroredStrategy()

with distribution.scope():
    mirrored_model = keras.models.Sequential([...])
    mirrored_model.compile([...])

batch_size = 100 # must be divisible by the number of replicas
history = mirrored_model.fit(X_train, y_train, epochs=10)
```

Under the hood, tf.keras is distribution-aware, so in this `MirroredStrategy` context it knows that it must replicate all variables and operations across all available GPU devices. Note that the `fit()` method will automatically split each training batch across all the replicas, so it's important that the batch size be divisible by the number of replicas. And that's all! Training will generally be significantly faster than using a single device, and the code change was really minimal.

Once you have finished training your model, you can use it to make predictions efficiently: call the `predict()` method, and it will automatically split the batch across all replicas, making predictions in parallel (again, the batch size must be divisible by the number of replicas). If you call the model's `save()` method, it will be saved as a regular model, *not* as a mirrored model with multiple replicas. So when you load it, it will run like a regular model, on a single device (by default GPU 0, or the CPU if there are no GPUs). If you want to load a model and run it on all available devices, you must call `keras.models.load_model()` within a distribution context:

```
with distribution.scope():
    mirrored_model = keras.models.load_model("my_mnist_model.h5")
```

If you only want to use a subset of all the available GPU devices, you can pass the list to the `MirroredStrategy`'s constructor:

```
distribution = tf.distribute.MirroredStrategy(["/gpu:0", "/gpu:1"])
```

By default, the `MirroredStrategy` class uses the *NVIDIA Collective Communications Library* (NCCL) for the AllReduce mean operation, but you can change it by setting the `cross_device_ops` argument to an instance of the `tf.distribute.HierarchicalCopyAllReduce` class, or an instance of the `tf.distribute.ReductionToOneDevice` class. The default NCCL option is based on the `tf.distribute.NcclAllReduce` class, which is usually faster, but this depends on the number and types of GPUs, so you may want to give the alternatives a try.[21]

21 For more details on AllReduce algorithms, read this great post by Yuichiro Ueno, and this page on scaling with NCCL.

If you want to try using data parallelism with centralized parameters, replace the `MirroredStrategy` with the `CentralStorageStrategy`:

```
distribution = tf.distribute.experimental.CentralStorageStrategy()
```

You can optionally set the `compute_devices` argument to specify the list of devices you want to use as workers (by default it will use all available GPUs), and you can optionally set the `parameter_device` argument to specify the device you want to store the parameters on (by default it will use the CPU, or the GPU if there is just one).

Now let's see how to train a model across a cluster of TensorFlow servers!

Training a Model on a TensorFlow Cluster

A *TensorFlow cluster* is a group of TensorFlow processes running in parallel, usually on different machines, and talking to each other to complete some work—for example, training or executing a neural network. Each TF process in the cluster is called a *task*, or a *TF server*. It has an IP address, a port, and a type (also called its *role* or its *job*). The type can be either `"worker"`, `"chief"`, `"ps"` (parameter server), or `"evaluator"`:

- Each *worker* performs computations, usually on a machine with one or more GPUs.
- The *chief* performs computations as well (it is a worker), but it also handles extra work such as writing TensorBoard logs or saving checkpoints. There is a single chief in a cluster. If no chief is specified, then the first worker is the chief.
- A *parameter server* only keeps track of variable values, and it is usually on a CPU-only machine. This type of task is only used with the `ParameterServerStrategy`.
- An *evaluator* obviously takes care of evaluation.

To start a TensorFlow cluster, you must first specify it. This means defining each task's IP address, TCP port, and type. For example, the following *cluster specification* defines a cluster with three tasks (two workers and one parameter server; see Figure 19-21). The cluster spec is a dictionary with one key per job, and the values are lists of task addresses (*IP:port*):

```
cluster_spec = {
    "worker": [
        "machine-a.example.com:2222",  # /job:worker/task:0
        "machine-b.example.com:2222"   # /job:worker/task:1
    ],
    "ps": ["machine-a.example.com:2221"] # /job:ps/task:0
}
```

Figure 19-21. TensorFlow cluster

In general there will be a single task per machine, but as this example shows, you can configure multiple tasks on the same machine if you want (if they share the same GPUs, make sure the RAM is split appropriately, as discussed earlier).

By default, every task in the cluster may communicate with every other task, so make sure to configure your firewall to authorize all communications between these machines on these ports (it's usually simpler if you use the same port on every machine).

When you start a task, you must give it the cluster spec, and you must also tell it what its type and index are (e.g., worker 0). The simplest way to specify everything at once (both the cluster spec and the current task's type and index) is to set the `TF_CONFIG` environment variable before starting TensorFlow. It must be a JSON-encoded dictionary containing a cluster specification (under the `"cluster"` key) and the type and index of the current task (under the `"task"` key). For example, the following `TF_CONFIG` environment variable uses the cluster we just defined and specifies that the task to start is the first worker:

```
import os
import json

os.environ["TF_CONFIG"] = json.dumps({
    "cluster": cluster_spec,
    "task": {"type": "worker", "index": 0}
})
```

In general you want to define the `TF_CONFIG` environment variable outside of Python, so the code does not need to include the current task's type and index (this makes it possible to use the same code across all workers).

Now let's train a model on a cluster! We will start with the mirrored strategy—it's surprisingly simple! First, you need to set the `TF_CONFIG` environment variable appropriately for each task. There should be no parameter server (remove the "ps" key in the cluster spec), and in general you will want a single worker per machine. Make extra sure you set a different task index for each task. Finally, run the following training code on every worker:

```
distribution = tf.distribute.experimental.MultiWorkerMirroredStrategy()

with distribution.scope():
    mirrored_model = keras.models.Sequential([...])
    mirrored_model.compile([...])

batch_size = 100 # must be divisible by the number of replicas
history = mirrored_model.fit(X_train, y_train, epochs=10)
```

Yes, that's exactly the same code we used earlier, except this time we are using the `MultiWorkerMirroredStrategy` (in future versions, the `MirroredStrategy` will probably handle both the single machine and multimachine cases). When you start this script on the first workers, they will remain blocked at the AllReduce step, but as soon as the last worker starts up training will begin, and you will see them all advancing at exactly the same rate (since they synchronize at each step).

You can choose from two AllReduce implementations for this distribution strategy: a ring AllReduce algorithm based on gRPC for the network communications, and NCCL's implementation. The best algorithm to use depends on the number of workers, the number and types of GPUs, and the network. By default, TensorFlow will apply some heuristics to select the right algorithm for you, but if you want to force one algorithm, pass `CollectiveCommunication.RING` or `CollectiveCommunication.NCCL` (from `tf.distribute.experimental`) to the strategy's constructor.

If you prefer to implement asynchronous data parallelism with parameter servers, change the strategy to `ParameterServerStrategy`, add one or more parameter servers, and configure `TF_CONFIG` appropriately for each task. Note that although the workers will work asynchronously, the replicas on each worker will work synchronously.

Lastly, if you have access to TPUs on Google Cloud, you can create a `TPUStrategy` like this (then use it like the other strategies):

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()
tf.tpu.experimental.initialize_tpu_system(resolver)
tpu_strategy = tf.distribute.experimental.TPUStrategy(resolver)
```

If you are a researcher, you may be eligible to use TPUs for free; see *https://tensorflow.org/tfrc* for more details.

You can now train models across multiple GPUs and multiple servers: give yourself a pat on the back! If you want to train a large model, you will need many GPUs, across many servers, which will require either buying a lot of hardware or managing a lot of cloud VMs. In many cases, it's going to be less hassle and less expensive to use a cloud service that takes care of provisioning and managing all this infrastructure for you, just when you need it. Let's see how to do that on GCP.

Running Large Training Jobs on Google Cloud AI Platform

If you decide to use Google AI Platform, you can deploy a training job with the same training code as you would run on your own TF cluster, and the platform will take care of provisioning and configuring as many GPU VMs as you desire (within your quotas).

To start the job, you will need the `gcloud` command-line tool, which is part of the Google Cloud SDK. You can either install the SDK on your own machine, or just use the Google Cloud Shell on GCP. This is a terminal you can use directly in your web browser; it runs on a free Linux VM (Debian), with the SDK already installed and preconfigured for you. The Cloud Shell is available anywhere in GCP: just click the Activate Cloud Shell icon at the top right of the page (see Figure 19-22).

Figure 19-22. Activating the Google Cloud Shell

If you prefer to install the SDK on your machine, once you have installed it, you need to initialize it by running `gcloud init`: you will need to log in to GCP and grant access to your GCP resources, then select the GCP project you want to use (if you have more than one), as well as the region where you want the job to run. The `gcloud` command gives you access to every GCP feature, including the ones we used earlier. You don't have to go through the web interface every time; you can write scripts that start or stop VMs for you, deploy models, or perform any other GCP action.

Before you can run the training job, you need to write the training code, exactly like you did earlier for a distributed setup (e.g., using the `ParameterServerStrategy`). AI Platform will take care of setting `TF_CONFIG` for you on each VM. Once that's done, you can deploy it and run it on a TF cluster with a command line like this:

```
$ gcloud ai-platform jobs submit training my_job_20190531_164700 \
    --region asia-southeast1 \
    --scale-tier PREMIUM_1 \
    --runtime-version 2.0 \
    --python-version 3.5 \
    --package-path /my_project/src/trainer \
    --module-name trainer.task \
    --staging-bucket gs://my-staging-bucket \
    --job-dir gs://my-mnist-model-bucket/trained_model \
    --
    --my-extra-argument1 foo --my-extra-argument2 bar
```

Let's go through these options. The command will start a training job named `my_job_20190531_164700`, in the `asia-southeast1` region, using a `PREMIUM_1` *scale tier*: this corresponds to 20 workers (including a chief) and 11 parameter servers (check out the other available scale tiers). All these VMs will be based on AI Platform's 2.0 runtime (a VM configuration that includes TensorFlow 2.0 and many other packages)[22] and Python 3.5. The training code is located in the */my_project/src/trainer* directory, and the `gcloud` command will automatically bundle it into a pip package and upload it to GCS at *gs://my-staging-bucket*. Next, AI Platform will start several VMs, deploy the package to them, and run the `trainer.task` module. Lastly, the `--job-dir` argument and the extra arguments (i.e., all the arguments located after the `--` separator) will be passed to the training program: the chief task will usually use the `--job-dir` argument to find out where to save the final model on GCS, in this case at *gs://my-mnist-model-bucket/trained_model*. And that's it! In the GCP console, you can then open the navigation menu, scroll down to the Artificial Intelligence section, and open AI Platform → Jobs. You should see your job running, and if you click it you will see graphs showing the CPU, GPU, and RAM utilization for every task. You can click View Logs to access the detailed logs using Stackdriver.

If you place the training data on GCS, you can create a `tf.data.TextLineDataset` or `tf.data.TFRecordDataset` to access it: just use the GCS paths as the filenames (e.g., *gs://my-data-bucket/my_data_001.csv*). These datasets rely on the `tf.io.gfile` package to access files: it supports both local files and GCS files (but make sure the service account you use has access to GCS).

22 At the time of this writing, the 2.0 runtime is not yet available, but it should be ready by the time you read this. Check out the list of available runtimes.

If you want to explore a few hyperparameter values, you can simply run multiple jobs and specify the hyperparameter values using the extra arguments for your tasks. However, if you want to explore many hyperparameters efficiently, it's a good idea to use AI Platform's hyperparameter tuning service instead.

Black Box Hyperparameter Tuning on AI Platform

AI Platform provides a powerful Bayesian optimization hyperparameter tuning service called Google Vizier.[23] To use it, you need to pass a YAML configuration file when creating the job (`--config tuning.yaml`). For example, it may look like this:

```yaml
trainingInput:
  hyperparameters:
    goal: MAXIMIZE
    hyperparameterMetricTag: accuracy
    maxTrials: 10
    maxParallelTrials: 2
    params:
      - parameterName: n_layers
        type: INTEGER
        minValue: 10
        maxValue: 100
        scaleType: UNIT_LINEAR_SCALE
      - parameterName: momentum
        type: DOUBLE
        minValue: 0.1
        maxValue: 1.0
        scaleType: UNIT_LOG_SCALE
```

This tells AI Platform that we want to maximize the metric named `"accuracy"`, the job will run a maximum of 10 trials (each trial will run our training code to train the model from scratch), and it will run a maximum of 2 trials in parallel. We want it to tune two hyperparameters: the `n_layers` hyperparameter (an integer between 10 and 100) and the `momentum` hyperparameter (a float between 0.1 and 1.0). The `scaleType` argument specifies the prior for the hyperparameter value: `UNIT_LINEAR_SCALE` means a flat prior (i.e., no a priori preference), while `UNIT_LOG_SCALE` says we have a prior belief that the optimal value lies closer to the max value (the other possible prior is `UNIT_REVERSE_LOG_SCALE`, when we believe the optimal value to be close to the min value).

The `n_layers` and `momentum` arguments will be passed as command-line arguments to the training code, and of course it is expected to use them. The question is, how will the training code communicate the metric back to the AI Platform so that it can

23 Daniel Golovin et al., "Google Vizier: A Service for Black-Box Optimization," *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017): 1487–1495.

decide which hyperparameter values to use during the next trial? Well, AI Platform just monitors the output directory (specified via `--job-dir`) for any event file (introduced in Chapter 10) containing summaries for a metric named `"accuracy"` (or whatever metric name is specified as the `hyperparameterMetricTag`), and it reads those values. So your training code simply has to use the `TensorBoard()` callback (which you will want to do anyway for monitoring), and you're good to go!

Once the job is finished, all the hyperparameter values used in each trial and the resulting accuracy will be available in the job's output (available via the AI Platform → Jobs page).

AI Platform jobs can also be used to efficiently execute your model on large amounts of data: each worker can read part of the data from GCS, make predictions, and save them to GCS.

Now you have all the tools and knowledge you need to create state-of-the-art neural net architectures and train them at scale using various distribution strategies, on your own infrastructure or on the cloud—and you can even perform powerful Bayesian optimization to fine-tune the hyperparameters!

Exercises

1. What does a SavedModel contain? How do you inspect its content?
2. When should you use TF Serving? What are its main features? What are some tools you can use to deploy it?
3. How do you deploy a model across multiple TF Serving instances?
4. When should you use the gRPC API rather than the REST API to query a model served by TF Serving?
5. What are the different ways TFLite reduces a model's size to make it run on a mobile or embedded device?
6. What is quantization-aware training, and why would you need it?
7. What are model parallelism and data parallelism? Why is the latter generally recommended?
8. When training a model across multiple servers, what distribution strategies can you use? How do you choose which one to use?
9. Train a model (any model you like) and deploy it to TF Serving or Google Cloud AI Platform. Write the client code to query it using the REST API or the gRPC

API. Update the model and deploy the new version. Your client code will now query the new version. Roll back to the first version.

10. Train any model across multiple GPUs on the same machine using the `MirroredStrategy` (if you do not have access to GPUs, you can use Colaboratory with a GPU Runtime and create two virtual GPUs). Train the model again using the `CentralStorageStrategy` and compare the training time.
11. Train a small model on Google Cloud AI Platform, using black box hyperparameter tuning.

Thank You!

Before we close the last chapter of this book, I would like to thank you for reading it up to the last paragraph. I truly hope that you had as much pleasure reading this book as I had writing it, and that it will be useful for your projects, big or small.

If you find errors, please send feedback. More generally, I would love to know what you think, so please don't hesitate to contact me via O'Reilly, through the *ageron/handson-ml2* GitHub project, or on Twitter at @aureliengeron.

Going forward, my best advice to you is to practice and practice: try going through all the exercises (if you have not done so already), play with the Jupyter notebooks, join Kaggle.com or some other ML community, watch ML courses, read papers, attend conferences, and meet experts. It also helps tremendously to have a concrete project to work on, whether it is for work or for fun (ideally for both), so if there's anything you have always dreamt of building, give it a shot! Work incrementally; don't shoot for the moon right away, but stay focused on your project and build it piece by piece. It will require patience and perseverance, but when you have a walking robot, or a working chatbot, or whatever else you fancy to build, it will be immensely rewarding.

My greatest hope is that this book will inspire you to build a wonderful ML application that will benefit all of us! What will it be?

—Aurélien Géron, June 17, 2019

APPENDIX A
Exercise Solutions

Solutions to the coding exercises are available in the online Jupyter notebooks at *https://github.com/ageron/handson-ml2*.

Chapter 1: The Machine Learning Landscape

1. Machine Learning is about building systems that can learn from data. Learning means getting better at some task, given some performance measure.
2. Machine Learning is great for complex problems for which we have no algorithmic solution, to replace long lists of hand-tuned rules, to build systems that adapt to fluctuating environments, and finally to help humans learn (e.g., data mining).
3. A labeled training set is a training set that contains the desired solution (a.k.a. a label) for each instance.
4. The two most common supervised tasks are regression and classification.
5. Common unsupervised tasks include clustering, visualization, dimensionality reduction, and association rule learning.
6. Reinforcement Learning is likely to perform best if we want a robot to learn to walk in various unknown terrains, since this is typically the type of problem that Reinforcement Learning tackles. It might be possible to express the problem as a supervised or semisupervised learning problem, but it would be less natural.
7. If you don't know how to define the groups, then you can use a clustering algorithm (unsupervised learning) to segment your customers into clusters of similar customers. However, if you know what groups you would like to have, then you

can feed many examples of each group to a classification algorithm (supervised learning), and it will classify all your customers into these groups.

8. Spam detection is a typical supervised learning problem: the algorithm is fed many emails along with their labels (spam or not spam).
9. An online learning system can learn incrementally, as opposed to a batch learning system. This makes it capable of adapting rapidly to both changing data and autonomous systems, and of training on very large quantities of data.
10. Out-of-core algorithms can handle vast quantities of data that cannot fit in a computer's main memory. An out-of-core learning algorithm chops the data into mini-batches and uses online learning techniques to learn from these mini-batches.
11. An instance-based learning system learns the training data by heart; then, when given a new instance, it uses a similarity measure to find the most similar learned instances and uses them to make predictions.
12. A model has one or more model parameters that determine what it will predict given a new instance (e.g., the slope of a linear model). A learning algorithm tries to find optimal values for these parameters such that the model generalizes well to new instances. A hyperparameter is a parameter of the learning algorithm itself, not of the model (e.g., the amount of regularization to apply).
13. Model-based learning algorithms search for an optimal value for the model parameters such that the model will generalize well to new instances. We usually train such systems by minimizing a cost function that measures how bad the system is at making predictions on the training data, plus a penalty for model complexity if the model is regularized. To make predictions, we feed the new instance's features into the model's prediction function, using the parameter values found by the learning algorithm.
14. Some of the main challenges in Machine Learning are the lack of data, poor data quality, nonrepresentative data, uninformative features, excessively simple models that underfit the training data, and excessively complex models that overfit the data.
15. If a model performs great on the training data but generalizes poorly to new instances, the model is likely overfitting the training data (or we got extremely lucky on the training data). Possible solutions to overfitting are getting more data, simplifying the model (selecting a simpler algorithm, reducing the number of parameters or features used, or regularizing the model), or reducing the noise in the training data.
16. A test set is used to estimate the generalization error that a model will make on new instances, before the model is launched in production.

17. A validation set is used to compare models. It makes it possible to select the best model and tune the hyperparameters.
18. The train-dev set is used when there is a risk of mismatch between the training data and the data used in the validation and test datasets (which should always be as close as possible to the data used once the model is in production). The train-dev set is a part of the training set that's held out (the model is not trained on it). The model is trained on the rest of the training set, and evaluated on both the train-dev set and the validation set. If the model performs well on the training set but not on the train-dev set, then the model is likely overfitting the training set. If it performs well on both the training set and the train-dev set, but not on the validation set, then there is probably a significant data mismatch between the training data and the validation + test data, and you should try to improve the training data to make it look more like the validation + test data.
19. If you tune hyperparameters using the test set, you risk overfitting the test set, and the generalization error you measure will be optimistic (you may launch a model that performs worse than you expect).

Chapter 2: End-to-End Machine Learning Project

See the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 3: Classification

See the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 4: Training Models

1. If you have a training set with millions of features you can use Stochastic Gradient Descent or Mini-batch Gradient Descent, and perhaps Batch Gradient Descent if the training set fits in memory. But you cannot use the Normal Equation or the SVD approach because the computational complexity grows quickly (more than quadratically) with the number of features.
2. If the features in your training set have very different scales, the cost function will have the shape of an elongated bowl, so the Gradient Descent algorithms will take a long time to converge. To solve this you should scale the data before training the model. Note that the Normal Equation or SVD approach will work just fine without scaling. Moreover, regularized models may converge to a suboptimal solution if the features are not scaled: since regularization penalizes large weights, features with smaller values will tend to be ignored compared to features with larger values.

3. Gradient Descent cannot get stuck in a local minimum when training a Logistic Regression model because the cost function is convex.[1]
4. If the optimization problem is convex (such as Linear Regression or Logistic Regression), and assuming the learning rate is not too high, then all Gradient Descent algorithms will approach the global optimum and end up producing fairly similar models. However, unless you gradually reduce the learning rate, Stochastic GD and Mini-batch GD will never truly converge; instead, they will keep jumping back and forth around the global optimum. This means that even if you let them run for a very long time, these Gradient Descent algorithms will produce slightly different models.
5. If the validation error consistently goes up after every epoch, then one possibility is that the learning rate is too high and the algorithm is diverging. If the training error also goes up, then this is clearly the problem and you should reduce the learning rate. However, if the training error is not going up, then your model is overfitting the training set and you should stop training.
6. Due to their random nature, neither Stochastic Gradient Descent nor Mini-batch Gradient Descent is guaranteed to make progress at every single training iteration. So if you immediately stop training when the validation error goes up, you may stop much too early, before the optimum is reached. A better option is to save the model at regular intervals; then, when it has not improved for a long time (meaning it will probably never beat the record), you can revert to the best saved model.
7. Stochastic Gradient Descent has the fastest training iteration since it considers only one training instance at a time, so it is generally the first to reach the vicinity of the global optimum (or Mini-batch GD with a very small mini-batch size). However, only Batch Gradient Descent will actually converge, given enough training time. As mentioned, Stochastic GD and Mini-batch GD will bounce around the optimum, unless you gradually reduce the learning rate.
8. If the validation error is much higher than the training error, this is likely because your model is overfitting the training set. One way to try to fix this is to reduce the polynomial degree: a model with fewer degrees of freedom is less likely to overfit. Another thing you can try is to regularize the model—for example, by adding an ℓ_2 penalty (Ridge) or an ℓ_1 penalty (Lasso) to the cost function. This will also reduce the degrees of freedom of the model. Lastly, you can try to increase the size of the training set.

1 If you draw a straight line between any two points on the curve, the line never crosses the curve.

9. If both the training error and the validation error are almost equal and fairly high, the model is likely underfitting the training set, which means it has a high bias. You should try reducing the regularization hyperparameter α.
10. Let's see:
    - A model with some regularization typically performs better than a model without any regularization, so you should generally prefer Ridge Regression over plain Linear Regression.
    - Lasso Regression uses an ℓ_1 penalty, which tends to push the weights down to exactly zero. This leads to sparse models, where all weights are zero except for the most important weights. This is a way to perform feature selection automatically, which is good if you suspect that only a few features actually matter. When you are not sure, you should prefer Ridge Regression.
    - Elastic Net is generally preferred over Lasso since Lasso may behave erratically in some cases (when several features are strongly correlated or when there are more features than training instances). However, it does add an extra hyperparameter to tune. If you want Lasso without the erratic behavior, you can just use Elastic Net with an `l1_ratio` close to 1.
11. If you want to classify pictures as outdoor/indoor and daytime/nighttime, since these are not exclusive classes (i.e., all four combinations are possible) you should train two Logistic Regression classifiers.
12. See the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 5: Support Vector Machines

1. The fundamental idea behind Support Vector Machines is to fit the widest possible "street" between the classes. In other words, the goal is to have the largest possible margin between the decision boundary that separates the two classes and the training instances. When performing soft margin classification, the SVM searches for a compromise between perfectly separating the two classes and having the widest possible street (i.e., a few instances may end up on the street). Another key idea is to use kernels when training on nonlinear datasets.
2. After training an SVM, a *support vector* is any instance located on the "street" (see the previous answer), including its border. The decision boundary is entirely determined by the support vectors. Any instance that is *not* a support vector (i.e., is off the street) has no influence whatsoever; you could remove them, add more instances, or move them around, and as long as they stay off the street they won't affect the decision boundary. Computing the predictions only involves the support vectors, not the whole training set.

3. SVMs try to fit the largest possible "street" between the classes (see the first answer), so if the training set is not scaled, the SVM will tend to neglect small features (see Figure 5-2).
4. An SVM classifier can output the distance between the test instance and the decision boundary, and you can use this as a confidence score. However, this score cannot be directly converted into an estimation of the class probability. If you set `probability=True` when creating an SVM in Scikit-Learn, then after training it will calibrate the probabilities using Logistic Regression on the SVM's scores (trained by an additional five-fold cross-validation on the training data). This will add the `predict_proba()` and `predict_log_proba()` methods to the SVM.
5. This question applies only to linear SVMs since kernelized SVMs can only use the dual form. The computational complexity of the primal form of the SVM problem is proportional to the number of training instances m, while the computational complexity of the dual form is proportional to a number between m^2 and m^3. So if there are millions of instances, you should definitely use the primal form, because the dual form will be much too slow.
6. If an SVM classifier trained with an RBF kernel underfits the training set, there might be too much regularization. To decrease it, you need to increase `gamma` or `C` (or both).
7. Let's call the QP parameters for the hard margin problem $\mathbf{H}'$, $\mathbf{f}'$, $\mathbf{A}'$, and $\mathbf{b}'$ (see "Quadratic Programming" on page 167). The QP parameters for the soft margin problem have m additional parameters ($n_p = n + 1 + m$) and m additional constraints ($n_c = 2m$). They can be defined like so:
   - $\mathbf{H}$ is equal to $\mathbf{H}'$, plus m columns of 0s on the right and m rows of 0s at the bottom: $\mathbf{H} = \begin{pmatrix} \mathbf{H}' & 0 & \cdots \\ 0 & 0 & \\ \vdots & & \ddots \end{pmatrix}$
   - $\mathbf{f}$ is equal to $\mathbf{f}'$ with m additional elements, all equal to the value of the hyperparameter C.
   - $\mathbf{b}$ is equal to $\mathbf{b}'$ with m additional elements, all equal to 0.
   - $\mathbf{A}$ is equal to $\mathbf{A}'$, with an extra $m \times m$ identity matrix $\mathbf{I}_m$ appended to the right, $-*\mathrm{I}*_m$ just below it, and the rest filled with 0s: $\mathbf{A} = \begin{pmatrix} \mathbf{A}' & \mathbf{I}_m \\ \mathbf{0} & -\mathbf{I}_m \end{pmatrix}$

For the solutions to exercises 8, 9, and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 6: Decision Trees

1. The depth of a well-balanced binary tree containing m leaves is equal to $\log_2(m)$,[2] rounded up. A binary Decision Tree (one that makes only binary decisions, as is the case with all trees in Scikit-Learn) will end up more or less well balanced at the end of training, with one leaf per training instance if it is trained without restrictions. Thus, if the training set contains one million instances, the Decision Tree will have a depth of $\log_2(10^6) \approx 20$ (actually a bit more since the tree will generally not be perfectly well balanced).
2. A node's Gini impurity is generally lower than its parent's. This is due to the CART training algorithm's cost function, which splits each node in a way that minimizes the weighted sum of its children's Gini impurities. However, it is possible for a node to have a higher Gini impurity than its parent, as long as this increase is more than compensated for by a decrease in the other child's impurity. For example, consider a node containing four instances of class A and one of class B. Its Gini impurity is $1 - (1/5)^2 - (4/5)^2 = 0.32$. Now suppose the dataset is one-dimensional and the instances are lined up in the following order: A, B, A, A, A. You can verify that the algorithm will split this node after the second instance, producing one child node with instances A, B, and the other child node with instances A, A, A. The first child node's Gini impurity is $1 - (1/2)^2 - (1/2)^2 = 0.5$, which is higher than its parent's. This is compensated for by the fact that the other node is pure, so its overall weighted Gini impurity is $2/5 \times 0.5 + 3/5 \times 0 = 0.2$, which is lower than the parent's Gini impurity.
3. If a Decision Tree is overfitting the training set, it may be a good idea to decrease `max_depth`, since this will constrain the model, regularizing it.
4. Decision Trees don't care whether or not the training data is scaled or centered; that's one of the nice things about them. So if a Decision Tree underfits the training set, scaling the input features will just be a waste of time.
5. The computational complexity of training a Decision Tree is $O(n \times m \log(m))$. So if you multiply the training set size by 10, the training time will be multiplied by $K = (n \times 10m \times \log(10m)) / (n \times m \times \log(m)) = 10 \times \log(10m) / \log(m)$. If $m = 10^6$, then $K \approx 11.7$, so you can expect the training time to be roughly 11.7 hours.
6. Presorting the training set speeds up training only if the dataset is smaller than a few thousand instances. If it contains 100,000 instances, setting `presort=True` will considerably slow down training.

For the solutions to exercises 7 and 8, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

2 $\log_2$ is the binary log; $\log_2(m) = \log(m) / \log(2)$.

Chapter 7: Ensemble Learning and Random Forests

1. If you have trained five different models and they all achieve 95% precision, you can try combining them into a voting ensemble, which will often give you even better results. It works better if the models are very different (e.g., an SVM classifier, a Decision Tree classifier, a Logistic Regression classifier, and so on). It is even better if they are trained on different training instances (that's the whole point of bagging and pasting ensembles), but if not this will still be effective as long as the models are very different.
2. A hard voting classifier just counts the votes of each classifier in the ensemble and picks the class that gets the most votes. A soft voting classifier computes the average estimated class probability for each class and picks the class with the highest probability. This gives high-confidence votes more weight and often performs better, but it works only if every classifier is able to estimate class probabilities (e.g., for the SVM classifiers in Scikit-Learn you must set `probability=True`).
3. It is quite possible to speed up training of a bagging ensemble by distributing it across multiple servers, since each predictor in the ensemble is independent of the others. The same goes for pasting ensembles and Random Forests, for the same reason. However, each predictor in a boosting ensemble is built based on the previous predictor, so training is necessarily sequential, and you will not gain anything by distributing training across multiple servers. Regarding stacking ensembles, all the predictors in a given layer are independent of each other, so they can be trained in parallel on multiple servers. However, the predictors in one layer can only be trained after the predictors in the previous layer have all been trained.
4. With out-of-bag evaluation, each predictor in a bagging ensemble is evaluated using instances that it was not trained on (they were held out). This makes it possible to have a fairly unbiased evaluation of the ensemble without the need for an additional validation set. Thus, you have more instances available for training, and your ensemble can perform slightly better.
5. When you are growing a tree in a Random Forest, only a random subset of the features is considered for splitting at each node. This is true as well for Extra-Trees, but they go one step further: rather than searching for the best possible thresholds, like regular Decision Trees do, they use random thresholds for each feature. This extra randomness acts like a form of regularization: if a Random Forest overfits the training data, Extra-Trees might perform better. Moreover, since Extra-Trees don't search for the best possible thresholds, they are much faster to train than Random Forests. However, they are neither faster nor slower than Random Forests when making predictions.

6. If your AdaBoost ensemble underfits the training data, you can try increasing the number of estimators or reducing the regularization hyperparameters of the base estimator. You may also try slightly increasing the learning rate.
7. If your Gradient Boosting ensemble overfits the training set, you should try decreasing the learning rate. You could also use early stopping to find the right number of predictors (you probably have too many).

For the solutions to exercises 8 and 9, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 8: Dimensionality Reduction

1. The main motivations for dimensionality reduction are:

   - To speed up a subsequent training algorithm (in some cases it may even remove noise and redundant features, making the training algorithm perform better)
   - To visualize the data and gain insights on the most important features
   - To save space (compression)

   The main drawbacks are:

   - Some information is lost, possibly degrading the performance of subsequent training algorithms.
   - It can be computationally intensive.
   - It adds some complexity to your Machine Learning pipelines.
   - Transformed features are often hard to interpret.

2. The curse of dimensionality refers to the fact that many problems that do not exist in low-dimensional space arise in high-dimensional space. In Machine Learning, one common manifestation is the fact that randomly sampled high-dimensional vectors are generally very sparse, increasing the risk of overfitting and making it very difficult to identify patterns in the data without having plenty of training data.
3. Once a dataset's dimensionality has been reduced using one of the algorithms we discussed, it is almost always impossible to perfectly reverse the operation, because some information gets lost during dimensionality reduction. Moreover, while some algorithms (such as PCA) have a simple reverse transformation procedure that can reconstruct a dataset relatively similar to the original, other algorithms (such as T-SNE) do not.

4. PCA can be used to significantly reduce the dimensionality of most datasets, even if they are highly nonlinear, because it can at least get rid of useless dimensions. However, if there are no useless dimensions—as in a Swiss roll dataset—then reducing dimensionality with PCA will lose too much information. You want to unroll the Swiss roll, not squash it.
5. That's a trick question: it depends on the dataset. Let's look at two extreme examples. First, suppose the dataset is composed of points that are almost perfectly aligned. In this case, PCA can reduce the dataset down to just one dimension while still preserving 95% of the variance. Now imagine that the dataset is composed of perfectly random points, scattered all around the 1,000 dimensions. In this case roughly 950 dimensions are required to preserve 95% of the variance. So the answer is, it depends on the dataset, and it could be any number between 1 and 950. Plotting the explained variance as a function of the number of dimensions is one way to get a rough idea of the dataset's intrinsic dimensionality.
6. Regular PCA is the default, but it works only if the dataset fits in memory. Incremental PCA is useful for large datasets that don't fit in memory, but it is slower than regular PCA, so if the dataset fits in memory you should prefer regular PCA. Incremental PCA is also useful for online tasks, when you need to apply PCA on the fly, every time a new instance arrives. Randomized PCA is useful when you want to considerably reduce dimensionality and the dataset fits in memory; in this case, it is much faster than regular PCA. Finally, Kernel PCA is useful for nonlinear datasets.
7. Intuitively, a dimensionality reduction algorithm performs well if it eliminates a lot of dimensions from the dataset without losing too much information. One way to measure this is to apply the reverse transformation and measure the reconstruction error. However, not all dimensionality reduction algorithms provide a reverse transformation. Alternatively, if you are using dimensionality reduction as a preprocessing step before another Machine Learning algorithm (e.g., a Random Forest classifier), then you can simply measure the performance of that second algorithm; if dimensionality reduction did not lose too much information, then the algorithm should perform just as well as when using the original dataset.
8. It can absolutely make sense to chain two different dimensionality reduction algorithms. A common example is using PCA to quickly get rid of a large number of useless dimensions, then applying another much slower dimensionality reduction algorithm, such as LLE. This two-step approach will likely yield the same performance as using LLE only, but in a fraction of the time.

For the solutions to exercises 9 and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 9: Unsupervised Learning Techniques

1. In Machine Learning, clustering is the unsupervised task of grouping similar instances together. The notion of similarity depends on the task at hand: for example, in some cases two nearby instances will be considered similar, while in others similar instances may be far apart as long as they belong to the same densely packed group. Popular clustering algorithms include K-Means, DBSCAN, agglomerative clustering, BIRCH, Mean-Shift, affinity propagation, and spectral clustering.
2. The main applications of clustering algorithms include data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction, anomaly detection, and novelty detection.
3. The elbow rule is a simple technique to select the number of clusters when using K-Means: just plot the inertia (the mean squared distance from each instance to its nearest centroid) as a function of the number of clusters, and find the point in the curve where the inertia stops dropping fast (the "elbow"). This is generally close to the optimal number of clusters. Another approach is to plot the silhouette score as a function of the number of clusters. There will often be a peak, and the optimal number of clusters is generally nearby. The silhouette score is the mean silhouette coefficient over all instances. This coefficient varies from +1 for instances that are well inside their cluster and far from other clusters, to −1 for instances that are very close to another cluster. You may also plot the silhouette diagrams and perform a more thorough analysis.
4. Labeling a dataset is costly and time-consuming. Therefore, it is common to have plenty of unlabeled instances, but few labeled instances. Label propagation is a technique that consists in copying some (or all) of the labels from the labeled instances to similar unlabeled instances. This can greatly extend the number of labeled instances, and thereby allow a supervised algorithm to reach better performance (this is a form of semi-supervised learning). One approach is to use a clustering algorithm such as K-Means on all the instances, then for each cluster find the most common label or the label of the most representative instance (i.e., the one closest to the centroid) and propagate it to the unlabeled instances in the same cluster.
5. K-Means and BIRCH scale well to large datasets. DBSCAN and Mean-Shift look for regions of high density.
6. Active learning is useful whenever you have plenty of unlabeled instances but labeling is costly. In this case (which is very common), rather than randomly selecting instances to label, it is often preferable to perform active learning, where human experts interact with the learning algorithm, providing labels for

specific instances when the algorithm requests them. A common approach is uncertainty sampling (see the description in "Active Learning" on page 255).

7. Many people use the terms *anomaly detection* and *novelty detection* interchangeably, but they are not exactly the same. In anomaly detection, the algorithm is trained on a dataset that may contain outliers, and the goal is typically to identify these outliers (within the training set), as well as outliers among new instances. In novelty detection, the algorithm is trained on a dataset that is presumed to be "clean," and the objective is to detect novelties strictly among new instances. Some algorithms work best for anomaly detection (e.g., Isolation Forest), while others are better suited for novelty detection (e.g., one-class SVM).
8. A Gaussian mixture model (GMM) is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions whose parameters are unknown. In other words, the assumption is that the data is grouped into a finite number of clusters, each with an ellipsoidal shape (but the clusters may have different ellipsoidal shapes, sizes, orientations, and densities), and we don't know which cluster each instance belongs to. This model is useful for density estimation, clustering, and anomaly detection.
9. One way to find the right number of clusters when using a Gaussian mixture model is to plot the Bayesian information criterion (BIC) or the Akaike information criterion (AIC) as a function of the number of clusters, then choose the number of clusters that minimizes the BIC or AIC. Another technique is to use a Bayesian Gaussian mixture model, which automatically selects the number of clusters.

For the solutions to exercises 10 to 13, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 10: Introduction to Artificial Neural Networks with Keras

1. Visit the TensorFlow Playground and play around with it, as described in this exercise.
2. Here is a neural network based on the original artificial neurons that computes $A \oplus B$ (where $\oplus$ represents the exclusive OR), using the fact that $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$. There are other solutions—for example, using the fact that $A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$, or the fact that $A \oplus B = (A \vee B) \wedge (\neg A \vee \wedge B)$, and so on.

3. A classical Perceptron will converge only if the dataset is linearly separable, and it won't be able to estimate class probabilities. In contrast, a Logistic Regression classifier will converge to a good solution even if the dataset is not linearly separable, and it will output class probabilities. If you change the Perceptron's activation function to the logistic activation function (or the softmax activation function if there are multiple neurons), and if you train it using Gradient Descent (or some other optimization algorithm minimizing the cost function, typically cross entropy), then it becomes equivalent to a Logistic Regression classifier.
4. The logistic activation function was a key ingredient in training the first MLPs because its derivative is always nonzero, so Gradient Descent can always roll down the slope. When the activation function is a step function, Gradient Descent cannot move, as there is no slope at all.
5. Popular activation functions include the step function, the logistic (sigmoid) function, the hyperbolic tangent (tanh) function, and the Rectified Linear Unit (ReLU) function (see Figure 10-8). See Chapter 11 for other examples, such as ELU and variants of the ReLU function.
6. Considering the MLP described in the question, composed of one input layer with 10 passthrough neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons, where all artificial neurons use the ReLU activation function: ..The shape of the input matrix $\mathbf{X}$ is $m \times 10$, where m represents the training batch size.
   a. The shape of the hidden layer's weight vector $\mathbf{W}_h$ is 10×50, and the length of its bias vector $\mathbf{b}_h$ is 50.
   b. The shape of the output layer's weight vector $\mathbf{W}_o$ is 50×3, and the length of its bias vector $\mathbf{b}_o$ is 3.
   c. The shape of the network's output matrix $\mathbf{Y}$ is $m \times 3$.
   d. $Y^* = \text{ReLU}(\text{ReLU}(\mathbf{X}\,\mathbf{W}_h + \mathbf{b}_h)\,\mathbf{W}_o + \mathbf{b}_o)$. Recall that the ReLU function just sets every negative number in the matrix to zero. Also note that when you are adding a bias vector to a matrix, it is added to every single row in the matrix, which is called *broadcasting*.

7. To classify email into spam or ham, you just need one neuron in the output layer of a neural network—for example, indicating the probability that the email is spam. You would typically use the logistic activation function in the output layer when estimating a probability. If instead you want to tackle MNIST, you need 10 neurons in the output layer, and you must replace the logistic function with the softmax activation function, which can handle multiple classes, outputting one probability per class. If you want your neural network to predict housing prices like in Chapter 2, then you need one output neuron, using no activation function at all in the output layer.[3]
8. Backpropagation is a technique used to train artificial neural networks. It first computes the gradients of the cost function with regard to every model parameter (all the weights and biases), then it performs a Gradient Descent step using these gradients. This backpropagation step is typically performed thousands or millions of times, using many training batches, until the model parameters converge to values that (hopefully) minimize the cost function. To compute the gradients, backpropagation uses reverse-mode autodiff (although it wasn't called that when backpropagation was invented, and it has been reinvented several times). Reverse-mode autodiff performs a forward pass through a computation graph, computing every node's value for the current training batch, and then it performs a reverse pass, computing all the gradients at once (see Appendix D for more details). So what's the difference? Well, backpropagation refers to the whole process of training an artificial neural network using multiple backpropagation steps, each of which computes gradients and uses them to perform a Gradient Descent step. In contrast, reverse-mode autodiff is just a technique to compute gradients efficiently, and it happens to be used by backpropagation.
9. Here is a list of all the hyperparameters you can tweak in a basic MLP: the number of hidden layers, the number of neurons in each hidden layer, and the activation function used in each hidden layer and in the output layer.[4] In general, the ReLU activation function (or one of its variants; see Chapter 11) is a good default for the hidden layers. For the output layer, in general you will want the logistic activation function for binary classification, the softmax activation function for multiclass classification, or no activation function for regression.

3 When the values to predict can vary by many orders of magnitude, you may want to predict the logarithm of the target value rather than the target value directly. Simply computing the exponential of the neural network's output will give you the estimated value (since $\exp(\log v) = v$).

4 In Chapter 11 we discuss many techniques that introduce additional hyperparameters: type of weight initialization, activation function hyperparameters (e.g., the amount of leak in leaky ReLU), Gradient Clipping threshold, type of optimizer and its hyperparameters (e.g., the momentum hyperparameter when using a `MomentumOptimizer`), type of regularization for each layer and regularization hyperparameters (e.g., dropout rate when using dropout), and so on.

If the MLP overfits the training data, you can try reducing the number of hidden layers and reducing the number of neurons per hidden layer.

10. See the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 11: Training Deep Neural Networks

1. No, all weights should be sampled independently; they should not all have the same initial value. One important goal of sampling weights randomly is to break symmetry: if all the weights have the same initial value, even if that value is not zero, then symmetry is not broken (i.e., all neurons in a given layer are equivalent), and backpropagation will be unable to break it. Concretely, this means that all the neurons in any given layer will always have the same weights. It's like having just one neuron per layer, and much slower. It is virtually impossible for such a configuration to converge to a good solution.
2. It is perfectly fine to initialize the bias terms to zero. Some people like to initialize them just like weights, and that's okay too; it does not make much difference.
3. A few advantages of the SELU function over the ReLU function are:
   - It can take on negative values, so the average output of the neurons in any given layer is typically closer to zero than when using the ReLU activation function (which never outputs negative values). This helps alleviate the vanishing gradients problem.
   - It always has a nonzero derivative, which avoids the dying units issue that can affect ReLU units.
   - When the conditions are right (i.e., if the model is sequential, and the weights are initialized using LeCun initialization, and the inputs are standardized, and there's no incompatible layer or regularization, such as dropout or ℓ_1 regularization), then the SELU activation function ensures the model is self-normalized, which solves the exploding/vanishing gradients problems.
4. The SELU activation function is a good default. If you need the neural network to be as fast as possible, you can use one of the leaky ReLU variants instead (e.g., a simple leaky ReLU using the default hyperparameter value). The simplicity of the ReLU activation function makes it many people's preferred option, despite the fact that it is generally outperformed by SELU and leaky ReLU. However, the ReLU activation function's ability to output precisely zero can be useful in some cases (e.g., see Chapter 17). Moreover, it can sometimes benefit from optimized implementation as well as from hardware acceleration. The hyperbolic tangent (tanh) can be useful in the output layer if you need to output a number between –1 and 1, but nowadays it is not used much in hidden layers (except in recurrent

nets). The logistic activation function is also useful in the output layer when you need to estimate a probability (e.g., for binary classification), but is rarely used in hidden layers (there are exceptions—for example, for the coding layer of variational autoencoders; see Chapter 17). Finally, the softmax activation function is useful in the output layer to output probabilities for mutually exclusive classes, but it is rarely (if ever) used in hidden layers.

5. If you set the `momentum` hyperparameter too close to 1 (e.g., 0.99999) when using an `SGD` optimizer, then the algorithm will likely pick up a lot of speed, hopefully moving roughly toward the global minimum, but its momentum will carry it right past the minimum. Then it will slow down and come back, accelerate again, overshoot again, and so on. It may oscillate this way many times before converging, so overall it will take much longer to converge than with a smaller `momentum` value.
6. One way to produce a sparse model (i.e., with most weights equal to zero) is to train the model normally, then zero out tiny weights. For more sparsity, you can apply ℓ_1 regularization during training, which pushes the optimizer toward sparsity. A third option is to use the TensorFlow Model Optimization Toolkit.
7. Yes, dropout does slow down training, in general roughly by a factor of two. However, it has no impact on inference speed since it is only turned on during training. MC Dropout is exactly like dropout during training, but it is still active during inference, so each inference is slowed down slightly. More importantly, when using MC Dropout you generally want to run inference 10 times or more to get better predictions. This means that making predictions is slowed down by a factor of 10 or more.

For the solutions to exercises 8, 9, and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 12: Custom Models and Training with TensorFlow

1. TensorFlow is an open-source library for numerical computation, particularly well suited and fine-tuned for large-scale Machine Learning. Its core is similar to NumPy, but it also features GPU support, support for distributed computing, computation graph analysis and optimization capabilities (with a portable graph format that allows you to train a TensorFlow model in one environment and run it in another), an optimization API based on reverse-mode autodiff, and several powerful APIs such as tf.keras, tf.data, tf.image, tf.signal, and more. Other popular Deep Learning libraries include PyTorch, MXNet, Microsoft Cognitive Toolkit, Theano, Caffe2, and Chainer.
2. Although TensorFlow offers most of the functionalities provided by NumPy, it is not a drop-in replacement, for a few reasons. First, the names of the functions are

not always the same (for example, `tf.reduce_sum()` versus `np.sum()`). Second, some functions do not behave in exactly the same way (for example, `tf.transpose()` creates a transposed copy of a tensor, while NumPy's `T` attribute creates a transposed view, without actually copying any data). Lastly, NumPy arrays are mutable, while TensorFlow tensors are not (but you can use a `tf.Variable` if you need a mutable object).

3. Both `tf.range(10)` and `tf.constant(np.arange(10))` return a one-dimensional tensor containing the integers 0 to 9. However, the former uses 32-bit integers while the latter uses 64-bit integers. Indeed, TensorFlow defaults to 32 bits, while NumPy defaults to 64 bits.
4. Beyond regular tensors, TensorFlow offers several other data structures, including sparse tensors, tensor arrays, ragged tensors, queues, string tensors, and sets. The last two are actually represented as regular tensors, but TensorFlow provides special functions to manipulate them (in `tf.strings` and `tf.sets`).
5. When you want to define a custom loss function, in general you can just implement it as a regular Python function. However, if your custom loss function must support some hyperparameters (or any other state), then you should subclass the `keras.losses.Loss` class and implement the `__init__()` and `call()` methods. If you want the loss function's hyperparameters to be saved along with the model, then you must also implement the `get_config()` method.
6. Much like custom loss functions, most metrics can be defined as regular Python functions. But if you want your custom metric to support some hyperparameters (or any other state), then you should subclass the `keras.metrics.Metric` class. Moreover, if computing the metric over a whole epoch is not equivalent to computing the mean metric over all batches in that epoch (e.g., as for the precision and recall metrics), then you should subclass the `keras.metrics.Metric` class and implement the `__init__()`, `update_state()`, and `result()` methods to keep track of a running metric during each epoch. You should also implement the `reset_states()` method unless all it needs to do is reset all variables to 0.0. If you want the state to be saved along with the model, then you should implement the `get_config()` method as well.
7. You should distinguish the internal components of your model (i.e., layers or reusable blocks of layers) from the model itself (i.e., the object you will train). The former should subclass the `keras.layers.Layer` class, while the latter should subclass the `keras.models.Model` class.
8. Writing your own custom training loop is fairly advanced, so you should only do it if you really need to. Keras provides several tools to customize training without having to write a custom training loop: callbacks, custom regularizers, custom constraints, custom losses, and so on. You should use these instead of writing a custom training loop whenever possible: writing a custom training loop is more

error-prone, and it will be harder to reuse the custom code you write. However, in some cases writing a custom training loop is necessary—for example, if you want to use different optimizers for different parts of your neural network, like in the Wide & Deep paper. A custom training loop can also be useful when debugging, or when trying to understand exactly how training works.

9. Custom Keras components should be convertible to TF Functions, which means they should stick to TF operations as much as possible and respect all the rules listed in "TF Function Rules" on page 409. If you absolutely need to include arbitrary Python code in a custom component, you can either wrap it in a `tf.py_function()` operation (but this will reduce performance and limit your model's portability) or set `dynamic=True` when creating the custom layer or model (or set `run_eagerly=True` when calling the model's `compile()` method).
10. Please refer to "TF Function Rules" on page 409 for the list of rules to respect when creating a TF Function.
11. Creating a dynamic Keras model can be useful for debugging, as it will not compile any custom component to a TF Function, and you can use any Python debugger to debug your code. It can also be useful if you want to include arbitrary Python code in your model (or in your training code), including calls to external libraries. To make a model dynamic, you must set `dynamic=True` when creating it. Alternatively, you can set `run_eagerly=True` when calling the model's `compile()` method. Making a model dynamic prevents Keras from using any of TensorFlow's graph features, so it will slow down training and inference, and you will not have the possibility to export the computation graph, which will limit your model's portability.

For the solutions to exercises 12 and 13, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 13: Loading and Preprocessing Data with TensorFlow

1. Ingesting a large dataset and preprocessing it efficiently can be a complex engineering challenge. The Data API makes it fairly simple. It offers many features, including loading data from various sources (such as text or binary files), reading data in parallel from multiple sources, transforming it, interleaving the records, shuffling the data, batching it, and prefetching it.
2. Splitting a large dataset into multiple files makes it possible to shuffle it at a coarse level before shuffling it at a finer level using a shuffling buffer. It also makes it possible to handle huge datasets that do not fit on a single machine. It's also simpler to manipulate thousands of small files rather than one huge file; for

example, it's easier to split the data into multiple subsets. Lastly, if the data is split across multiple files spread across multiple servers, it is possible to download several files from different servers simultaneously, which improves the bandwidth usage.

3. You can use TensorBoard to visualize profiling data: if the GPU is not fully utilized then your input pipeline is likely to be the bottleneck. You can fix it by making sure it reads and preprocesses the data in multiple threads in parallel, and ensuring it prefetches a few batches. If this is insufficient to get your GPU to 100% usage during training, make sure your preprocessing code is optimized. You can also try saving the dataset into multiple TFRecord files, and if necessary perform some of the preprocessing ahead of time so that it does not need to be done on the fly during training (TF Transform can help with this). If necessary, use a machine with more CPU and RAM, and ensure that the GPU bandwidth is large enough.
4. A TFRecord file is composed of a sequence of arbitrary binary records: you can store absolutely any binary data you want in each record. However, in practice most TFRecord files contain sequences of serialized protocol buffers. This makes it possible to benefit from the advantages of protocol buffers, such as the fact that they can be read easily across multiple platforms and languages and their definition can be updated later in a backward-compatible way.
5. The `Example` protobuf format has the advantage that TensorFlow provides some operations to parse it (the `tf.io.parse*example()` functions) without you having to define your own format. It is sufficiently flexible to represent instances in most datasets. However, if it does not cover your use case, you can define your own protocol buffer, compile it using `protoc` (setting the `--descriptor_set_out` and `--include_imports` arguments to export the protobuf descriptor), and use the `tf.io.decode_proto()` function to parse the serialized protobufs (see the "Custom protobuf" section of the notebook for an example). It's more complicated, and it requires deploying the descriptor along with the model, but it can be done.
6. When using TFRecords, you will generally want to activate compression if the TFRecord files will need to be downloaded by the training script, as compression will make files smaller and thus reduce download time. But if the files are located on the same machine as the training script, it's usually preferable to leave compression off, to avoid wasting CPU for decompression.
7. Let's look at the pros and cons of each preprocessing option:

   - If you preprocess the data when creating the data files, the training script will run faster, since it will not have to perform preprocessing on the fly. In some cases, the preprocessed data will also be much smaller than the original data, so you can save some space and speed up downloads. It may also be helpful to

materialize the preprocessed data, for example to inspect it or archive it. However, this approach has a few cons. First, it's not easy to experiment with various preprocessing logics if you need to generate a preprocessed dataset for each variant. Second, if you want to perform data augmentation, you have to materialize many variants of your dataset, which will use a large amount of disk space and take a lot of time to generate. Lastly, the trained model will expect preprocessed data, so you will have to add preprocessing code in your application before it calls the model.

- If the data is preprocessed with the tf.data pipeline, it's much easier to tweak the preprocessing logic and apply data augmentation. Also, tf.data makes it easy to build highly efficient preprocessing pipelines (e.g., with multithreading and prefetching). However, preprocessing the data this way will slow down training. Moreover, each training instance will be preprocessed once per epoch rather than just once if the data was preprocessed when creating the data files. Lastly, the trained model will still expect preprocessed data.
- If you add preprocessing layers to your model, you will only have to write the preprocessing code once for both training and inference. If your model needs to be deployed to many different platforms, you will not need to write the preprocessing code multiple times. Plus, you will not run the risk of using the wrong preprocessing logic for your model, since it will be part of the model. On the downside, preprocessing the data will slow down training, and each training instance will be preprocessed once per epoch. Moreover, by default the preprocessing operations will run on the GPU for the current batch (you will not benefit from parallel preprocessing on the CPU, and prefetching). Fortunately, the upcoming Keras preprocessing layers should be able to lift the preprocessing operations from the preprocessing layers and run them as part of the tf.data pipeline, so you will benefit from multithreaded execution on the CPU and prefetching.
- Lastly, using TF Transform for preprocessing gives you many of the benefits from the previous options: the preprocessed data is materialized, each instance is preprocessed just once (speeding up training), and preprocessing layers get generated automatically so you only need to write the preprocessing code once. The main drawback is the fact that you need to learn how to use this tool.

8. Let's look at how to encode categorical features and text:

   - To encode a categorical feature that has a natural order, such as a movie rating (e.g., "bad," "average," "good"), the simplest option is to use ordinal encoding: sort the categories in their natural order and map each category to its rank (e.g., "bad" maps to 0, "average" maps to 1, and "good" maps to 2). However, most categorical features don't have such a natural order. For example, there's

no natural order for professions or countries. In this case, you can use one-hot encoding or, if there are many categories, embeddings.

- For text, one option is to use a bag-of-words representation: a sentence is represented by a vector counting the counts of each possible word. Since common words are usually not very important, you'll want to use TF-IDF to reduce their weight. Instead of counting words, it is also common to count *n*-grams, which are sequences of *n* consecutive words—nice and simple. Alternatively, you can encode each word using word embeddings, possibly pretrained. Rather than encoding words, it is also possible to encode each letter, or subword tokens (e.g., splitting "smartest" into "smart" and "est"). These last two options are discussed in Chapter 16.

For the solutions to exercises 9 and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 14: Deep Computer Vision Using Convolutional Neural Networks

1. These are the main advantages of a CNN over a fully connected DNN for image classification:

   - Because consecutive layers are only partially connected and because it heavily reuses its weights, a CNN has many fewer parameters than a fully connected DNN, which makes it much faster to train, reduces the risk of overfitting, and requires much less training data.
   - When a CNN has learned a kernel that can detect a particular feature, it can detect that feature anywhere in the image. In contrast, when a DNN learns a feature in one location, it can detect it only in that particular location. Since images typically have very repetitive features, CNNs are able to generalize much better than DNNs for image processing tasks such as classification, using fewer training examples.
   - Finally, a DNN has no prior knowledge of how pixels are organized; it does not know that nearby pixels are close. A CNN's architecture embeds this prior knowledge. Lower layers typically identify features in small areas of the images, while higher layers combine the lower-level features into larger features. This works well with most natural images, giving CNNs a decisive head start compared to DNNs.

2. Let's compute how many parameters the CNN has. Since its first convolutional layer has 3×3 kernels, and the input has three channels (red, green, and blue), each feature map has $3 \times 3 \times 3$ weights, plus a bias term. That's 28 parameters per

feature map. Since this first convolutional layer has 100 feature maps, it has a total of 2,800 parameters. The second convolutional layer has 3 × 3 kernels and its input is the set of 100 feature maps of the previous layer, so each feature map has 3 × 3 × 100 = 900 weights, plus a bias term. Since it has 200 feature maps, this layer has 901 × 200 = 180,200 parameters. Finally, the third and last convolutional layer also has 3 × 3 kernels, and its input is the set of 200 feature maps of the previous layers, so each feature map has 3 × 3 × 200 = 1,800 weights, plus a bias term. Since it has 400 feature maps, this layer has a total of 1,801 × 400 = 720,400 parameters. All in all, the CNN has 2,800 + 180,200 + 720,400 = 903,400 parameters.

Now let's compute how much RAM this neural network will require (at least) when making a prediction for a single instance. First let's compute the feature map size for each layer. Since we are using a stride of 2 and `"same"` padding, the horizontal and vertical dimensions of the feature maps are divided by 2 at each layer (rounding up if necessary). So, as the input channels are 200 × 300 pixels, the first layer's feature maps are 100 × 150, the second layer's feature maps are 50 × 75, and the third layer's feature maps are 25 × 38. Since 32 bits is 4 bytes and the first convolutional layer has 100 feature maps, this first layer takes up 4 × 100 × 150 × 100 = 6 million bytes (6 MB). The second layer takes up 4 × 50 × 75 × 200 = 3 million bytes (3 MB). Finally, the third layer takes up 4 × 25 × 38 × 400 = 1,520,000 bytes (about 1.5 MB). However, once a layer has been computed, the memory occupied by the previous layer can be released, so if everything is well optimized, only 6 + 3 = 9 million bytes (9 MB) of RAM will be required (when the second layer has just been computed, but the memory occupied by the first layer has not been released yet). But wait, you also need to add the memory occupied by the CNN's parameters! We computed earlier that it has 903,400 parameters, each using up 4 bytes, so this adds 3,613,600 bytes (about 3.6 MB). The total RAM required is therefore (at least) 12,613,600 bytes (about 12.6 MB).

Lastly, let's compute the minimum amount of RAM required when training the CNN on a mini-batch of 50 images. During training TensorFlow uses backpropagation, which requires keeping all values computed during the forward pass until the reverse pass begins. So we must compute the total RAM required by all layers for a single instance and multiply that by 50. At this point, let's start counting in megabytes rather than bytes. We computed before that the three layers require respectively 6, 3, and 1.5 MB for each instance. That's a total of 10.5 MB per instance, so for 50 instances the total RAM required is 525 MB. Add to that the RAM required by the input images, which is 50 × 4 × 200 × 300 × 3 = 36 million bytes (36 MB), plus the RAM required for the model parameters, which is about 3.6 MB (computed earlier), plus some RAM for the gradients (we will neglect this since it can be released gradually as backpropagation goes down the layers during the reverse pass). We are up to a total of roughly 525 + 36 + 3.6 = 564.6 MB, and that's really an optimistic bare minimum.

3. If your GPU runs out of memory while training a CNN, here are five things you could try to solve the problem (other than purchasing a GPU with more RAM):
   - Reduce the mini-batch size.
   - Reduce dimensionality using a larger stride in one or more layers.
   - Remove one or more layers.
   - Use 16-bit floats instead of 32-bit floats.
   - Distribute the CNN across multiple devices.
4. A max pooling layer has no parameters at all, whereas a convolutional layer has quite a few (see the previous questions).
5. A local response normalization layer makes the neurons that most strongly activate inhibit neurons at the same location but in neighboring feature maps, which encourages different feature maps to specialize and pushes them apart, forcing them to explore a wider range of features. It is typically used in the lower layers to have a larger pool of low-level features that the upper layers can build upon.
6. The main innovations in AlexNet compared to LeNet-5 are that it is much larger and deeper, and it stacks convolutional layers directly on top of each other, instead of stacking a pooling layer on top of each convolutional layer. The main innovation in GoogLeNet is the introduction of *inception modules*, which make it possible to have a much deeper net than previous CNN architectures, with fewer parameters. ResNet's main innovation is the introduction of skip connections, which make it possible to go well beyond 100 layers. Arguably, its simplicity and consistency are also rather innovative. SENet's main innovation was the idea of using an SE block (a two-layer dense network) after every inception module in an inception network or every residual unit in a ResNet to recalibrate the relative importance of feature maps. Finally, Xception's main innovation was the use of depthwise separable convolutional layers, which look at spatial patterns and depthwise patterns separately.
7. Fully convolutional networks are neural networks composed exclusively of convolutional and pooling layers. FCNs can efficiently process images of any width and height (at least above the minimum size). They are most useful for object detection and semantic segmentation because they only need to look at the image once (instead of having to run a CNN multiple times on different parts of the image). If you have a CNN with some dense layers on top, you can convert these dense layers to convolutional layers to create an FCN: just replace the lowest dense layer with a convolutional layer with a kernel size equal to the layer's input size, with one filter per neuron in the dense layer, and using `"valid"` padding. Generally the stride should be 1, but you can set it to a higher value if you want. The activation function should be the same as the dense layer's. The other dense layers should be converted the same way, but using 1×1 filters. It is actually pos-

sible to convert a trained CNN this way by appropriately reshaping the dense layers' weight matrices.

8. The main technical difficulty of semantic segmentation is the fact that a lot of the spatial information gets lost in a CNN as the signal flows through each layer, especially in pooling layers and layers with a stride greater than 1. This spatial information needs to be restored somehow to accurately predict the class of each pixel.

For the solutions to exercises 9 to 12, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 15: Processing Sequences Using RNNs and CNNs

1. Here are a few RNN applications:

   - For a sequence-to-sequence RNN: predicting the weather (or any other time series), machine translation (using an Encoder–Decoder architecture), video captioning, speech to text, music generation (or other sequence generation), identifying the chords of a song
   - For a sequence-to-vector RNN: classifying music samples by music genre, analyzing the sentiment of a book review, predicting what word an aphasic patient is thinking of based on readings from brain implants, predicting the probability that a user will want to watch a movie based on their watch history (this is one of many possible implementations of *collaborative filtering* for a recommender system)
   - For a vector-to-sequence RNN: image captioning, creating a music playlist based on an embedding of the current artist, generating a melody based on a set of parameters, locating pedestrians in a picture (e.g., a video frame from a self-driving car's camera)

2. An RNN layer must have three-dimensional inputs: the first dimension is the batch dimension (its size is the batch size), the second dimension represents the time (its size is the number of time steps), and the third dimension holds the inputs at each time step (its size is the number of input features per time step). For example, if you want to process a batch containing 5 time series of 10 time steps each, with 2 values per time step (e.g., the temperature and the wind speed), the shape will be [5, 10, 2]. The outputs are also three-dimensional, with the same first two dimensions, but the last dimension is equal to the number of neurons. For example, if an RNN layer with 32 neurons processes the batch we just discussed, the output will have a shape of [5, 10, 32].

3. To build a deep sequence-to-sequence RNN using Keras, you must set `return_sequences=True` for all RNN layers. To build a sequence-to-vector RNN, you must set `return_sequences=True` for all RNN layers except for the top RNN layer, which must have `return_sequences=False` (or do not set this argument at all, since `False` is the default).
4. If you have a daily univariate time series, and you want to forecast the next seven days, the simplest RNN architecture you can use is a stack of RNN layers (all with `return_sequences=True` except for the top RNN layer), using seven neurons in the output RNN layer. You can then train this model using random windows from the time series (e.g., sequences of 30 consecutive days as the inputs, and a vector containing the values of the next 7 days as the target). This is a sequence-to-vector RNN. Alternatively, you could set `return_sequences=True` for all RNN layers to create a sequence-to-sequence RNN. You can train this model using random windows from the time series, with sequences of the same length as the inputs as the targets. Each target sequence should have seven values per time step (e.g., for time step t, the target should be a vector containing the values at time steps $t + 1$ to $t + 7$).
5. The two main difficulties when training RNNs are unstable gradients (exploding or vanishing) and a very limited short-term memory. These problems both get worse when dealing with long sequences. To alleviate the unstable gradients problem, you can use a smaller learning rate, use a saturating activation function such as the hyperbolic tangent (which is the default), and possibly use gradient clipping, Layer Normalization, or dropout at each time step. To tackle the limited short-term memory problem, you can use `LSTM` or `GRU` layers (this also helps with the unstable gradients problem).
6. An LSTM cell's architecture looks complicated, but it's actually not too hard if you understand the underlying logic. The cell has a short-term state vector and a long-term state vector. At each time step, the inputs and the previous short-term state are fed to a simple RNN cell and three gates: the forget gate decides what to remove from the long-term state, the input gate decides which part of the output of the simple RNN cell should be added to the long-term state, and the output gate decides which part of the long-term state should be output at this time step (after going through the tanh activation function). The new short-term state is equal to the output of the cell. See Figure 15-9.
7. An RNN layer is fundamentally sequential: in order to compute the outputs at time step t, it has to first compute the outputs at all earlier time steps. This makes it impossible to parallelize. On the other hand, a 1D convolutional layer lends itself well to parallelization since it does not hold a state between time steps. In other words, it has no memory: the output at any time step can be computed based only on a small window of values from the inputs without having to know all the past values. Moreover, since a 1D convolutional layer is not recurrent, it

suffers less from unstable gradients. One or more 1D convolutional layers can be useful in an RNN to efficiently preprocess the inputs, for example to reduce their temporal resolution (downsampling) and thereby help the RNN layers detect long-term patterns. In fact, it is possible to use only convolutional layers, for example by building a WaveNet architecture.

8. To classify videos based on their visual content, one possible architecture could be to take (say) one frame per second, then run every frame through the same convolutional neural network (e.g., a pretrained Xception model, possibly frozen if your dataset is not large), feed the sequence of outputs from the CNN to a sequence-to-vector RNN, and finally run its output through a softmax layer, giving you all the class probabilities. For training you would use cross entropy as the cost function. If you wanted to use the audio for classification as well, you could use a stack of strided 1D convolutional layers to reduce the temporal resolution from thousands of audio frames per second to just one per second (to match the number of images per second), and concatenate the output sequence to the inputs of the sequence-to-vector RNN (along the last dimension).

For the solutions to exercises 9 and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 16: Natural Language Processing with RNNs and Attention

1. Stateless RNNs can only capture patterns whose length is less than, or equal to, the size of the windows the RNN is trained on. Conversely, stateful RNNs can capture longer-term patterns. However, implementing a stateful RNN is much harder—especially preparing the dataset properly. Moreover, stateful RNNs do not always work better, in part because consecutive batches are not independent and identically distributed (IID). Gradient Descent is not fond of non-IID datasets.
2. In general, if you translate a sentence one word at a time, the result will be terrible. For example, the French sentence "Je vous en prie" means "You are welcome," but if you translate it one word at a time, you get "I you in pray." Huh? It is much better to read the whole sentence first and then translate it. A plain sequence-to-sequence RNN would start translating a sentence immediately after reading the first word, while an Encoder–Decoder RNN will first read the whole sentence and then translate it. That said, one could imagine a plain sequence-to-sequence RNN that would output silence whenever it is unsure about what to say next (just like human translators do when they must translate a live broadcast).
3. Variable-length input sequences can be handled by padding the shorter sequences so that all sequences in a batch have the same length, and using masking to

ensure the RNN ignores the padding token. For better performance, you may also want to create batches containing sequences of similar sizes. Ragged tensors can hold sequences of variable lengths, and tf.keras will likely support them eventually, which will greatly simplify handling variable-length input sequences (at the time of this writing, it is not the case yet). Regarding variable-length output sequences, if the length of the output sequence is known in advance (e.g., if you know that it is the same as the input sequence), then you just need to configure the loss function so that it ignores tokens that come after the end of the sequence. Similarly, the code that will use the model should ignore tokens beyond the end of the sequence. But generally the length of the output sequence is not known ahead of time, so the solution is to train the model so that it outputs an end-of-sequence token at the end of each sequence.

4. Beam search is a technique used to improve the performance of a trained Encoder–Decoder model, for example in a neural machine translation system. The algorithm keeps track of a short list of the *k* most promising output sentences (say, the top three), and at each decoder step it tries to extend them by one word; then it keeps only the *k* most likely sentences. The parameter *k* is called the *beam width*: the larger it is, the more CPU and RAM will be used, but also the more accurate the system will be. Instead of greedily choosing the most likely next word at each step to extend a single sentence, this technique allows the system to explore several promising sentences simultaneously. Moreover, this technique lends itself well to parallelization. You can implement beam search fairly easily using TensorFlow Addons.
5. An attention mechanism is a technique initially used in Encoder–Decoder models to give the decoder more direct access to the input sequence, allowing it to deal with longer input sequences. At each decoder time step, the current decoder's state and the full output of the encoder are processed by an alignment model that outputs an alignment score for each input time step. This score indicates which part of the input is most relevant to the current decoder time step. The weighted sum of the encoder output (weighted by their alignment score) is then fed to the decoder, which produces the next decoder state and the output for this time step. The main benefit of using an attention mechanism is the fact that the Encoder–Decoder model can successfully process longer input sequences. Another benefit is that the alignment scores makes the model easier to debug and interpret: for example, if the model makes a mistake, you can look at which part of the input it was paying attention to, and this can help diagnose the issue. An attention mechanism is also at the core of the Transformer architecture, in the Multi-Head Attention layers. See the next answer.
6. The most important layer in the Transformer architecture is the Multi-Head Attention layer (the original Transformer architecture contains 18 of them, including 6 Masked Multi-Head Attention layers). It is at the core of language

models such as BERT and GPT-2. Its purpose is to allow the model to identify which words are most aligned with each other, and then improve each word's representation using these contextual clues.

7. Sampled softmax is used when training a classification model when there are many classes (e.g., thousands). It computes an approximation of the cross-entropy loss based on the logit predicted by the model for the correct class, and the predicted logits for a sample of incorrect words. This speeds up training considerably compared to computing the softmax over all logits and then estimating the cross-entropy loss. After training, the model can be used normally, using the regular softmax function to compute all the class probabilities based on all the logits.

For the solutions to exercises 8 to 11, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 17: Representation Learning and Generative Learning Using Autoencoders and GANs

1. Here are some of the main tasks that autoencoders are used for:

   - Feature extraction
   - Unsupervised pretraining
   - Dimensionality reduction
   - Generative models
   - Anomaly detection (an autoencoder is generally bad at reconstructing outliers)

2. If you want to train a classifier and you have plenty of unlabeled training data but only a few thousand labeled instances, then you could first train a deep autoencoder on the full dataset (labeled + unlabeled), then reuse its lower half for the classifier (i.e., reuse the layers up to the codings layer, included) and train the classifier using the labeled data. If you have little labeled data, you probably want to freeze the reused layers when training the classifier.

3. The fact that an autoencoder perfectly reconstructs its inputs does not necessarily mean that it is a good autoencoder; perhaps it is simply an overcomplete autoencoder that learned to copy its inputs to the codings layer and then to the outputs. In fact, even if the codings layer contained a single neuron, it would be possible for a very deep autoencoder to learn to map each training instance to a different coding (e.g., the first instance could be mapped to 0.001, the second to 0.002, the third to 0.003, and so on), and it could learn "by heart" to reconstruct the right training instance for each coding. It would perfectly reconstruct its inputs

without really learning any useful pattern in the data. In practice such a mapping is unlikely to happen, but it illustrates the fact that perfect reconstructions are not a guarantee that the autoencoder learned anything useful. However, if it produces very bad reconstructions, then it is almost guaranteed to be a bad autoencoder. To evaluate the performance of an autoencoder, one option is to measure the reconstruction loss (e.g., compute the MSE, or the mean square of the outputs minus the inputs). Again, a high reconstruction loss is a good sign that the autoencoder is bad, but a low reconstruction loss is not a guarantee that it is good. You should also evaluate the autoencoder according to what it will be used for. For example, if you are using it for unsupervised pretraining of a classifier, then you should also evaluate the classifier's performance.

4. An undercomplete autoencoder is one whose codings layer is smaller than the input and output layers. If it is larger, then it is an overcomplete autoencoder. The main risk of an excessively undercomplete autoencoder is that it may fail to reconstruct the inputs. The main risk of an overcomplete autoencoder is that it may just copy the inputs to the outputs, without learning any useful features.
5. To tie the weights of an encoder layer and its corresponding decoder layer, you simply make the decoder weights equal to the transpose of the encoder weights. This reduces the number of parameters in the model by half, often making training converge faster with less training data and reducing the risk of overfitting the training set.
6. A generative model is a model capable of randomly generating outputs that resemble the training instances. For example, once trained successfully on the MNIST dataset, a generative model can be used to randomly generate realistic images of digits. The output distribution is typically similar to the training data. For example, since MNIST contains many images of each digit, the generative model would output roughly the same number of images of each digit. Some generative models can be parametrized—for example, to generate only some kinds of outputs. An example of a generative autoencoder is the variational autoencoder.
7. A generative adversarial network is a neural network architecture composed of two parts, the generator and the discriminator, which have opposing objectives. The generator's goal is to generate instances similar to those in the training set, to fool the discriminator. The discriminator must distinguish the real instances from the generated ones. At each training iteration, the discriminator is trained like a normal binary classifier, then the generator is trained to maximize the discriminator's error. GANs are used for advanced image processing tasks such as super resolution, colorization, image editing (replacing objects with realistic background), turning a simple sketch into a photorealistic image, or predicting the next frames in a video. They are also used to augment a dataset (to train other

models), to generate other types of data (such as text, audio, and time series), and to identify the weaknesses in other models and strengthen them.

8. Training GANs is notoriously difficult, because of the complex dynamics between the generator and the discriminator. The biggest difficulty is mode collapse, where the generator produces outputs with very little diversity. Moreover, training can be terribly unstable: it may start out fine and then suddenly start oscillating or diverging, without any apparent reason. GANs are also very sensitive to the choice of hyperparameters.

For the solutions to exercises 9, 10, and 11, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 18: Reinforcement Learning

1. Reinforcement Learning is an area of Machine Learning aimed at creating agents capable of taking actions in an environment in a way that maximizes rewards over time. There are many differences between RL and regular supervised and unsupervised learning. Here are a few:

   - In supervised and unsupervised learning, the goal is generally to find patterns in the data and use them to make predictions. In Reinforcement Learning, the goal is to find a good policy.
   - Unlike in supervised learning, the agent is not explicitly given the "right" answer. It must learn by trial and error.
   - Unlike in unsupervised learning, there is a form of supervision, through rewards. We do not tell the agent how to perform the task, but we do tell it when it is making progress or when it is failing.
   - A Reinforcement Learning agent needs to find the right balance between exploring the environment, looking for new ways of getting rewards, and exploiting sources of rewards that it already knows. In contrast, supervised and unsupervised learning systems generally don't need to worry about exploration; they just feed on the training data they are given.
   - In supervised and unsupervised learning, training instances are typically independent (in fact, they are generally shuffled). In Reinforcement Learning, consecutive observations are generally *not* independent. An agent may remain in the same region of the environment for a while before it moves on, so consecutive observations will be very correlated. In some cases a replay memory (buffer) is used to ensure that the training algorithm gets fairly independent observations.

2. Here are a few possible applications of Reinforcement Learning, other than those mentioned in Chapter 18:

   Music personalization
   The environment is a user's personalized web radio. The agent is the software deciding what song to play next for that user. Its possible actions are to play any song in the catalog (it must try to choose a song the user will enjoy) or to play an advertisement (it must try to choose an ad that the user will be interested in). It gets a small reward every time the user listens to a song, a larger reward every time the user listens to an ad, a negative reward when the user skips a song or an ad, and a very negative reward if the user leaves.

   Marketing
   The environment is your company's marketing department. The agent is the software that defines which customers a mailing campaign should be sent to, given their profile and purchase history (for each customer it has two possible actions: send or don't send). It gets a negative reward for the cost of the mailing campaign, and a positive reward for estimated revenue generated from this campaign.

   Product delivery
   Let the agent control a fleet of delivery trucks, deciding what they should pick up at the depots, where they should go, what they should drop off, and so on. It will get positive rewards for each product delivered on time, and negative rewards for late deliveries.

3. When estimating the value of an action, Reinforcement Learning algorithms typically sum all the rewards that this action led to, giving more weight to immediate rewards and less weight to later rewards (considering that an action has more influence on the near future than on the distant future). To model this, a discount factor is typically applied at each time step. For example, with a discount factor of 0.9, a reward of 100 that is received two time steps later is counted as only $0.9^2 \times 100 = 81$ when you are estimating the value of the action. You can think of the discount factor as a measure of how much the future is valued relative to the present: if it is very close to 1, then the future is valued almost as much as the present; if it is close to 0, then only immediate rewards matter. Of course, this impacts the optimal policy tremendously: if you value the future, you may be willing to put up with a lot of immediate pain for the prospect of eventual rewards, while if you don't value the future, you will just grab any immediate reward you can find, never investing in the future.

4. To measure the performance of a Reinforcement Learning agent, you can simply sum up the rewards it gets. In a simulated environment, you can run many episodes and look at the total rewards it gets on average (and possibly look at the min, max, standard deviation, and so on).

5. The credit assignment problem is the fact that when a Reinforcement Learning agent receives a reward, it has no direct way of knowing which of its previous actions contributed to this reward. It typically occurs when there is a large delay between an action and the resulting reward (e.g., during a game of Atari's *Pong*, there may be a few dozen time steps between the moment the agent hits the ball and the moment it wins the point). One way to alleviate it is to provide the agent with shorter-term rewards, when possible. This usually requires prior knowledge about the task. For example, if we want to build an agent that will learn to play chess, instead of giving it a reward only when it wins the game, we could give it a reward every time it captures one of the opponent's pieces.
6. An agent can often remain in the same region of its environment for a while, so all of its experiences will be very similar for that period of time. This can introduce some bias in the learning algorithm. It may tune its policy for this region of the environment, but it will not perform well as soon as it moves out of this region. To solve this problem, you can use a replay memory; instead of using only the most immediate experiences for learning, the agent will learn based on a buffer of its past experiences, recent and not so recent (perhaps this is why we dream at night: to replay our experiences of the day and better learn from them?).
7. An off-policy RL algorithm learns the value of the optimal policy (i.e., the sum of discounted rewards that can be expected for each state if the agent acts optimally) while the agent follows a different policy. Q-Learning is a good example of such an algorithm. In contrast, an on-policy algorithm learns the value of the policy that the agent actually executes, including both exploration and exploitation.

For the solutions to exercises 8, 9, and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 19: Training and Deploying TensorFlow Models at Scale

1. A SavedModel contains a TensorFlow model, including its architecture (a computation graph) and its weights. It is stored as a directory containing a *saved_model.pb* file, which defines the computation graph (represented as a serialized protocol buffer), and a *variables* subdirectory containing the variable values. For models containing a large number of weights, these variable values may be split across multiple files. A SavedModel also includes an *assets* subdirectory that may contain additional data, such as vocabulary files, class names, or some example instances for this model. To be more accurate, a SavedModel can contain one or more *metagraphs*. A metagraph is a computation graph plus some function signature definitions (including their input and output names, types, and shapes). Each metagraph is identified by a set of tags. To inspect a SavedMo-

del, you can use the command-line tool `saved_model_cli` or just load it using `tf.saved_model.load()` and inspect it in Python.

2. TF Serving allows you to deploy multiple TensorFlow models (or multiple versions of the same model) and make them accessible to all your applications easily via a REST API or a gRPC API. Using your models directly in your applications would make it harder to deploy a new version of a model across all applications. Implementing your own microservice to wrap a TF model would require extra work, and it would be hard to match TF Serving's features. TF Serving has many features: it can monitor a directory and autodeploy the models that are placed there, and you won't have to change or even restart any of your applications to benefit from the new model versions; it's fast, well tested, and scales very well; and it supports A/B testing of experimental models and deploying a new model version to just a subset of your users (in this case the model is called a *canary*). TF Serving is also capable of grouping individual requests into batches to run them jointly on the GPU. To deploy TF Serving, you can install it from source, but it is much simpler to install it using a Docker image. To deploy a cluster of TF Serving Docker images, you can use an orchestration tool such as Kubernetes, or use a fully hosted solution such as Google Cloud AI Platform.
3. To deploy a model across multiple TF Serving instances, all you need to do is configure these TF Serving instances to monitor the same *models* directory, and then export your new model as a SavedModel into a subdirectory.
4. The gRPC API is more efficient than the REST API. However, its client libraries are not as widely available, and if you activate compression when using the REST API, you can get almost the same performance. So, the gRPC API is most useful when you need the highest possible performance and the clients are not limited to the REST API.
5. To reduce a model's size so it can run on a mobile or embedded device, TFLite uses several techniques:
   - It provides a converter which can optimize a SavedModel: it shrinks the model and reduces its latency. To do this, it prunes all the operations that are not needed to make predictions (such as training operations), and it optimizes and fuses operations whenever possible.
   - The converter can also perform post-training quantization: this technique dramatically reduces the model's size, so it's much faster to download and store.
   - It saves the optimized model using the FlatBuffer format, which can be loaded to RAM directly, without parsing. This reduces the loading time and memory footprint.

6. Quantization-aware training consists in adding fake quantization operations to the model during training. This allows the model to learn to ignore the quantization noise; the final weights will be more robust to quantization.
7. Model parallelism means chopping your model into multiple parts and running them in parallel across multiple devices, hopefully speeding up the model during training or inference. Data parallelism means creating multiple exact replicas of your model and deploying them across multiple devices. At each iteration during training, each replica is given a different batch of data, and it computes the gradients of the loss with regard to the model parameters. In synchronous data parallelism, the gradients from all replicas are then aggregated and the optimizer performs a Gradient Descent step. The parameters may be centralized (e.g., on parameter servers) or replicated across all replicas and kept in sync using AllReduce. In asynchronous data parallelism, the parameters are centralized and the replicas run independently from each other, each updating the central parameters directly at the end of each training iteration, without having to wait for the other replicas. To speed up training, data parallelism turns out to work better than model parallelism, in general. This is mostly because it requires less communication across devices. Moreover, it is much easier to implement, and it works the same way for any model, whereas model parallelism requires analyzing the model to determine the best way to chop it into pieces.
8. When training a model across multiple servers, you can use the following distribution strategies:
    - The `MultiWorkerMirroredStrategy` performs mirrored data parallelism. The model is replicated across all available servers and devices, and each replica gets a different batch of data at each training iteration and computes its own gradients. The mean of the gradients is computed and shared across all replicas using a distributed AllReduce implementation (NCCL by default), and all replicas perform the same Gradient Descent step. This strategy is the simplest to use since all servers and devices are treated in exactly the same way, and it performs fairly well. In general, you should use this strategy. Its main limitation is that it requires the model to fit in RAM on every replica.
    - The `ParameterServerStrategy` performs asynchronous data parallelism. The model is replicated across all devices on all workers, and the parameters are sharded across all parameter servers. Each worker has its own training loop, running asynchronously with the other workers; at each training iteration, each worker gets its own batch of data and fetches the latest version of the model parameters from the parameter servers, then it computes the gradients of the loss with regard to these parameters, and it sends them to the parameter servers. Lastly, the parameter servers perform a Gradient Descent step using these gradients. This strategy is generally slower than the previous strategy,

and a bit harder to deploy, since it requires managing parameter servers. However, it is useful to train huge models that don't fit in GPU RAM.

For the solutions to exercises 9, 10, and 11, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

APPENDIX B
Machine Learning Project Checklist

This checklist can guide you through your Machine Learning projects. There are eight main steps:

1. Frame the problem and look at the big picture.
2. Get the data.
3. Explore the data to gain insights.
4. Prepare the data to better expose the underlying data patterns to Machine Learning algorithms.
5. Explore many different models and shortlist the best ones.
6. Fine-tune your models and combine them into a great solution.
7. Present your solution.
8. Launch, monitor, and maintain your system.

Obviously, you should feel free to adapt this checklist to your needs.

Frame the Problem and Look at the Big Picture

1. Define the objective in business terms.
2. How will your solution be used?
3. What are the current solutions/workarounds (if any)?
4. How should you frame this problem (supervised/unsupervised, online/offline, etc.)?
5. How should performance be measured?
6. Is the performance measure aligned with the business objective?

7. What would be the minimum performance needed to reach the business objective?
8. What are comparable problems? Can you reuse experience or tools?
9. Is human expertise available?
10. How would you solve the problem manually?
11. List the assumptions you (or others) have made so far.
12. Verify assumptions if possible.

Get the Data

Note: automate as much as possible so you can easily get fresh data.

1. List the data you need and how much you need.
2. Find and document where you can get that data.
3. Check how much space it will take.
4. Check legal obligations, and get authorization if necessary.
5. Get access authorizations.
6. Create a workspace (with enough storage space).
7. Get the data.
8. Convert the data to a format you can easily manipulate (without changing the data itself).
9. Ensure sensitive information is deleted or protected (e.g., anonymized).
10. Check the size and type of data (time series, sample, geographical, etc.).
11. Sample a test set, put it aside, and never look at it (no data snooping!).

Explore the Data

Note: try to get insights from a field expert for these steps.

1. Create a copy of the data for exploration (sampling it down to a manageable size if necessary).
2. Create a Jupyter notebook to keep a record of your data exploration.
3. Study each attribute and its characteristics:
   - Name
   - Type (categorical, int/float, bounded/unbounded, text, structured, etc.)

- % of missing values
- Noisiness and type of noise (stochastic, outliers, rounding errors, etc.)
- Usefulness for the task
- Type of distribution (Gaussian, uniform, logarithmic, etc.)

4. For supervised learning tasks, identify the target attribute(s).
5. Visualize the data.
6. Study the correlations between attributes.
7. Study how you would solve the problem manually.
8. Identify the promising transformations you may want to apply.
9. Identify extra data that would be useful (go back to "Get the Data" on page 756).
10. Document what you have learned.

Prepare the Data

Notes:

- Work on copies of the data (keep the original dataset intact).
- Write functions for all data transformations you apply, for five reasons:
  — So you can easily prepare the data the next time you get a fresh dataset
  — So you can apply these transformations in future projects
  — To clean and prepare the test set
  — To clean and prepare new data instances once your solution is live
  — To make it easy to treat your preparation choices as hyperparameters

1. Data cleaning:
   - Fix or remove outliers (optional).
   - Fill in missing values (e.g., with zero, mean, median…) or drop their rows (or columns).
2. Feature selection (optional):
   - Drop the attributes that provide no useful information for the task.
3. Feature engineering, where appropriate:
   - Discretize continuous features.

- Decompose features (e.g., categorical, date/time, etc.).
- Add promising transformations of features (e.g., log(x), sqrt(x), x^2, etc.).
- Aggregate features into promising new features.

4. Feature scaling:
   - Standardize or normalize features.

Shortlist Promising Models

Notes:

- If the data is huge, you may want to sample smaller training sets so you can train many different models in a reasonable time (be aware that this penalizes complex models such as large neural nets or Random Forests).
- Once again, try to automate these steps as much as possible.

1. Train many quick-and-dirty models from different categories (e.g., linear, naive Bayes, SVM, Random Forest, neural net, etc.) using standard parameters.
2. Measure and compare their performance.
   - For each model, use N-fold cross-validation and compute the mean and standard deviation of the performance measure on the N folds.
3. Analyze the most significant variables for each algorithm.
4. Analyze the types of errors the models make.
   - What data would a human have used to avoid these errors?
5. Perform a quick round of feature selection and engineering.
6. Perform one or two more quick iterations of the five previous steps.
7. Shortlist the top three to five most promising models, preferring models that make different types of errors.

Fine-Tune the System

Notes:

- You will want to use as much data as possible for this step, especially as you move toward the end of fine-tuning.

- As always, automate what you can.

1. Fine-tune the hyperparameters using cross-validation:
   - Treat your data transformation choices as hyperparameters, especially when you are not sure about them (e.g., if you're not sure whether to replace missing values with zeros or with the median value, or to just drop the rows).
   - Unless there are very few hyperparameter values to explore, prefer random search over grid search. If training is very long, you may prefer a Bayesian optimization approach (e.g., using Gaussian process priors, as described by Jasper Snoek et al.).[1]
2. Try Ensemble methods. Combining your best models will often produce better performance than running them individually.
3. Once you are confident about your final model, measure its performance on the test set to estimate the generalization error.

Don't tweak your model after measuring the generalization error: you would just start overfitting the test set.

Present Your Solution

1. Document what you have done.
2. Create a nice presentation.
   - Make sure you highlight the big picture first.
3. Explain why your solution achieves the business objective.
4. Don't forget to present interesting points you noticed along the way.
   - Describe what worked and what did not.
   - List your assumptions and your system's limitations.

1 Jasper Snoek et al., "Practical Bayesian Optimization of Machine Learning Algorithms," *Proceedings of the 25th International Conference on Neural Information Processing Systems* 2 (2012): 2951–2959.

5. Ensure your key findings are communicated through beautiful visualizations or easy-to-remember statements (e.g., "the median income is the number-one predictor of housing prices").

Launch!

1. Get your solution ready for production (plug into production data inputs, write unit tests, etc.).
2. Write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops.
   - Beware of slow degradation: models tend to "rot" as data evolves.
   - Measuring performance may require a human pipeline (e.g., via a crowdsourcing service).
   - Also monitor your inputs' quality (e.g., a malfunctioning sensor sending random values, or another team's output becoming stale). This is particularly important for online learning systems.
3. Retrain your models on a regular basis on fresh data (automate as much as possible).

APPENDIX C

SVM Dual Problem

To understand *duality*, you first need to understand the *Lagrange multipliers* method. The general idea is to transform a constrained optimization objective into an unconstrained one, by moving the constraints into the objective function. Let's look at a simple example. Suppose you want to find the values of x and y that minimize the function $f(x, y) = x^2 + 2y$, subject to an *equality constraint*: $3x + 2y + 1 = 0$. Using the Lagrange multipliers method, we start by defining a new function called the *Lagrangian* (or *Lagrange function*): $g(x, y, \alpha) = f(x, y) - \alpha(3x + 2y + 1)$. Each constraint (in this case just one) is subtracted from the original objective, multiplied by a new variable called a Lagrange multiplier.

Joseph-Louis Lagrange showed that if $(\hat{x}, \hat{y})$ is a solution to the constrained optimization problem, then there must exist an $\hat{\alpha}$ such that $(\hat{x}, \hat{y}, \hat{\alpha})$ is a *stationary point* of the Lagrangian (a stationary point is a point where all partial derivatives are equal to zero). In other words, we can compute the partial derivatives of $g(x, y, \alpha)$ with regard to x, y, and α; we can find the points where these derivatives are all equal to zero; and the solutions to the constrained optimization problem (if they exist) must be among these stationary points.

In this example the partial derivatives are: $\begin{cases} \frac{\partial}{\partial x} g(x, y, \alpha) = 2x - 3\alpha \\ \frac{\partial}{\partial y} g(x, y, \alpha) = 2 - 2\alpha \\ \frac{\partial}{\partial \alpha} g(x, y, \alpha) = -3x - 2y - 1 \end{cases}$

When all these partial derivatives are equal to 0, we find that $2\hat{x} - 3\hat{\alpha} = 2 - 2\hat{\alpha} = -3\hat{x} - 2\hat{y} - 1 = 0$, from which we can easily find that $\hat{x} = \frac{3}{2}$, $\hat{y} = -\frac{11}{4}$, and $\hat{\alpha} = 1$. This is the only stationary point, and as it respects the constraint, it must be the solution to the constrained optimization problem.

However, this method applies only to equality constraints. Fortunately, under some regularity conditions (which are respected by the SVM objectives), this method can be generalized to *inequality constraints* as well (e.g., $3x + 2y + 1 \geq 0$). The *generalized Lagrangian* for the hard margin problem is given by Equation C-1, where the $\alpha^{(i)}$ variables are called the *Karush–Kuhn–Tucker* (KKT) multipliers, and they must be greater or equal to zero.

Equation C-1. Generalized Lagrangian for the hard margin problem

$$\mathscr{L}(\mathbf{w}, b, \alpha) = \frac{1}{2}\mathbf{w}^\top\mathbf{w} - \sum_{i=1}^{m} \alpha^{(i)}\left(t^{(i)}\left(\mathbf{w}^\top\mathbf{x}^{(i)} + b\right) - 1\right)$$

$$\text{with} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

Just like with the Lagrange multipliers method, you can compute the partial derivatives and locate the stationary points. If there is a solution, it will necessarily be among the stationary points $\left(\widehat{\mathbf{w}}, \hat{b}, \hat{\alpha}\right)$ that respect the *KKT conditions*:

- Respect the problem's constraints: $t^{(i)}\left(\widehat{\mathbf{w}}^\top\mathbf{x}^{(i)} + \hat{b}\right) \geq 1 \quad \text{for } i = 1, 2, \ldots, m$.
- Verify $\hat{\alpha}^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$.
- Either $\hat{\alpha}^{(i)} = 0$ or the ith constraint must be an *active constraint*, meaning it must hold by equality: $t^{(i)}\left(\widehat{\mathbf{w}}^\top\mathbf{x}^{(i)} + \hat{b}\right) = 1$. This condition is called the *complementary slackness* condition. It implies that either $\hat{\alpha}^{(i)} = 0$ or the ith instance lies on the boundary (it is a support vector).

Note that the KKT conditions are necessary conditions for a stationary point to be a solution of the constrained optimization problem. Under some conditions, they are also sufficient conditions. Luckily, the SVM optimization problem happens to meet these conditions, so any stationary point that meets the KKT conditions is guaranteed to be a solution to the constrained optimization problem.

We can compute the partial derivatives of the generalized Lagrangian with regard to $\mathbf{w}$ and b with Equation C-2.

Equation C-2. Partial derivatives of the generalized Lagrangian

$$\nabla_{\mathbf{w}}\mathscr{L}(\mathbf{w}, b, \alpha) = \mathbf{w} - \sum_{i=1}^{m} \alpha^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\frac{\partial}{\partial b}\mathscr{L}(\mathbf{w}, b, \alpha) = -\sum_{i=1}^{m} \alpha^{(i)} t^{(i)}$$

When these partial derivatives are equal to zero, we have Equation C-3.

Equation C-3. Properties of the stationary points

$$\widehat{\mathbf{w}} = \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} = 0$$

If we plug these results into the definition of the generalized Lagrangian, some terms disappear and we find Equation C-4.

Equation C-4. Dual form of the SVM problem

$$\mathscr{L}\left(\widehat{\mathbf{w}}, \hat{b}, \alpha\right) = \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)\top} \mathbf{x}^{(j)} \quad - \quad \sum_{i=1}^{m} \alpha^{(i)}$$

$$\text{with} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

The goal is now to find the vector $\widehat{\boldsymbol{\alpha}}$ that minimizes this function, with $\hat{\alpha}^{(i)} \geq 0$ for all instances. This constrained optimization problem is the dual problem we were looking for.

Once you find the optimal $\widehat{\boldsymbol{\alpha}}$, you can compute $\widehat{\mathbf{w}}$ using the first line of Equation C-3. To compute $\hat{b}$, you can use the fact that a support vector must verify $t^{(i)}(\widehat{\mathbf{w}}^\top \mathbf{x}^{(i)} + \hat{b}) = 1$, so if the kth instance is a support vector (i.e., $\hat{\alpha}^{(k)} > 0$), you can use it to compute $\hat{b} = t^{(k)} - \widehat{\mathbf{w}}^\top \mathbf{x}^{(k)}$. However, it is often preferred to compute the average over all support vectors to get a more stable and precise value, as in Equation C-5.

Equation C-5. Bias term estimation using the dual form

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left[t^{(i)} - \widehat{\mathbf{w}}^\top \mathbf{x}^{(i)} \right]$$

APPENDIX D
Autodiff

This appendix explains how TensorFlow's autodifferentiation (autodiff) feature works, and how it compares to other solutions.

Suppose you define a function $f(x, y) = x^2y + y + 2$, and you need its partial derivatives $\partial f/\partial x$ and $\partial f/\partial y$, typically to perform Gradient Descent (or some other optimization algorithm). Your main options are manual differentiation, finite difference approximation, forward-mode autodiff, and reverse-mode autodiff. TensorFlow implements reverse-mode autodiff, but to understand it, it's useful to look at the other options first. So let's go through each of them, starting with manual differentiation.

Manual Differentiation

The first approach to compute derivatives is to pick up a pencil and a piece of paper and use your calculus knowledge to derive the appropriate equation. For the function $f(x, y)$ just defined, it is not too hard; you just need to use five rules:

- The derivative of a constant is 0.
- The derivative of λx is λ (where λ is a constant).
- The derivative of x^λ is $\lambda x^{\lambda - 1}$, so the derivative of x^2 is $2x$.
- The derivative of a sum of functions is the sum of these functions' derivatives.
- The derivative of λ times a function is λ times its derivative.

From these rules, you can derive Equation D-1.

Equation D-1. Partial derivatives of f(x, y)

$$\frac{\partial f}{\partial x} = \frac{\partial\left(x^2 y\right)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y\frac{\partial\left(x^2\right)}{\partial x} + 0 + 0 = 2xy$$

$$\frac{\partial f}{\partial y} = \frac{\partial\left(x^2 y\right)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1$$

This approach can become very tedious for more complex functions, and you run the risk of making mistakes. Fortunately, there are other options. Let's look at finite difference approximation now.

Finite Difference Approximation

Recall that the derivative $h'(x_0)$ of a function $h(x)$ at a point x_0 is the slope of the function at that point. More precisely, the derivative is defined as the limit of the slope of a straight line going through this point x_0 and another point x on the function, as x gets infinitely close to x_0 (see Equation D-2).

Equation D-2. Definition of the derivative of a function h(x) at point x_0

$$\begin{aligned} h'\left(x_0\right) &= \lim_{x \to x_0} \frac{h(x) - h\left(x_0\right)}{x - x_0} \\ &= \lim_{\varepsilon \to 0} \frac{h\left(x_0 + \varepsilon\right) - h\left(x_0\right)}{\varepsilon} \end{aligned}$$

So, if we wanted to calculate the partial derivative of $f(x, y)$ with regard to x at $x = 3$ and $y = 4$, we could compute $f(3 + \varepsilon, 4) - f(3, 4)$ and divide the result by ε, using a very small value for ε. This type of numerical approximation of the derivative is called a *finite difference approximation*, and this specific equation is called *Newton's difference quotient*. That's exactly what the following code does:

```
def f(x, y):
    return x**2*y + y + 2

def derivative(f, x, y, x_eps, y_eps):
    return (f(x + x_eps, y + y_eps) - f(x, y)) / (x_eps + y_eps)

df_dx = derivative(f, 3, 4, 0.00001, 0)
df_dy = derivative(f, 3, 4, 0, 0.00001)
```

Unfortunately, the result is imprecise (and it gets worse for more complicated functions). The correct results are respectively 24 and 10, but instead we get:

```
>>> print(df_dx)
24.000039999805264
>>> print(df_dy)
10.000000000331966
```

Notice that to compute both partial derivatives, we have to call `f()` at least three times (we called it four times in the preceding code, but it could be optimized). If there were 1,000 parameters, we would need to call `f()` at least 1,001 times. When you are dealing with large neural networks, this makes finite difference approximation way too inefficient.

However, this method is so simple to implement that it is a great tool to check that the other methods are implemented correctly. For example, if it disagrees with your manually derived function, then your function probably contains a mistake.

So far, we have considered two ways to compute gradients: using manual differentiation and using finite difference approximation. Unfortunately, both were fatally flawed to train a large-scale neural network. So let's turn to autodiff, starting with forward mode.

Forward-Mode Autodiff

Figure D-1 shows how forward-mode autodiff works on an even simpler function, $g(x, y) = 5 + xy$. The graph for that function is represented on the left. After forward-mode autodiff, we get the graph on the right, which represents the partial derivative $\partial g/\partial x = 0 + (0 \times x + y \times 1) = y$ (we could similarly obtain the partial derivative with regard to y).

Figure D-1. Forward-mode autodiff

The algorithm will go through the computation graph from the inputs to the outputs (hence the name "forward mode"). It starts by getting the partial derivatives of the leaf nodes. The constant node (5) returns the constant 0, since the derivative of a constant is always 0. The variable x returns the constant 1 since $\partial x/\partial x = 1$, and the variable y returns the constant 0 since $\partial y/\partial x = 0$ (if we were looking for the partial derivative with regard to y, it would be the reverse).

Now we have all we need to move up the graph to the multiplication node in function g. Calculus tells us that the derivative of the product of two functions u and v is $\partial(u \times v)/\partial x = \partial v/\partial x \times u + v \times \partial u/\partial x$. We can therefore construct a large part of the graph on the right, representing $0 \times x + y \times 1$.

Finally, we can go up to the addition node in function g. As mentioned, the derivative of a sum of functions is the sum of these functions' derivatives. So we just need to create an addition node and connect it to the parts of the graph we have already computed. We get the correct partial derivative: $\partial g/\partial x = 0 + (0 \times x + y \times 1)$.

However, this equation can be simplified (a lot). A few pruning steps can be applied to the computation graph to get rid of all unnecessary operations, and we get a much smaller graph with just one node: $\partial g/\partial x = y$. In this case simplification is fairly easy, but for a more complex function forward-mode autodiff can produce a huge graph that may be tough to simplify and lead to suboptimal performance.

Note that we started with a computation graph, and forward-mode autodiff produced another computation graph. This is called *symbolic differentiation*, and it has two nice features: first, once the computation graph of the derivative has been produced, we can use it as many times as we want to compute the derivatives of the given function for any value of x and y; second, we can run forward-mode autodiff again on the resulting graph to get second-order derivatives if we ever need to (i.e., derivatives of derivatives). We could even compute third-order derivatives, and so on.

But it is also possible to run forward-mode autodiff without constructing a graph (i.e., numerically, not symbolically), just by computing intermediate results on the fly. One way to do this is to use *dual numbers*, which are weird but fascinating numbers of the form $a + b\varepsilon$, where a and b are real numbers and ε is an infinitesimal number such that $\varepsilon^2 = 0$ (but $\varepsilon \neq 0$). You can think of the dual number $42 + 24\varepsilon$ as something akin to $42.0000\cdots000024$ with an infinite number of 0s (but of course this is simplified just to give you some idea of what dual numbers are). A dual number is represented in memory as a pair of floats. For example, $42 + 24\varepsilon$ is represented by the pair (42.0, 24.0).

Dual numbers can be added, multiplied, and so on, as shown in Equation D-3.

Equation D-3. A few operations with dual numbers

$$\lambda(a + b\varepsilon) = \lambda a + \lambda b\varepsilon$$
$$(a + b\varepsilon) + (c + d\varepsilon) = (a + c) + (b + d)\varepsilon$$
$$(a + b\varepsilon) \times (c + d\varepsilon) = ac + (ad + bc)\varepsilon + (bd)\varepsilon^2 = ac + (ad + bc)\varepsilon$$

Most importantly, it can be shown that $h(a + b\varepsilon) = h(a) + b \times h'(a)\varepsilon$, so computing $h(a + \varepsilon)$ gives you both $h(a)$ and the derivative $h'(a)$ in just one shot. Figure D-2 shows that the partial derivative of $f(x, y)$ with regard to x at $x = 3$ and $y = 4$ (which we will write $\partial f/\partial x$ (3, 4)) can be computed using dual numbers. All we need to do is compute $f(3 + \varepsilon, 4)$; this will output a dual number whose first component is equal to $f(3, 4)$ and whose second component is equal to $\partial f/\partial x$ (3, 4).

Figure D-2. Forward-mode autodiff using dual numbers

To compute $\partial f/\partial x$ (3, 4) we would have to go through the graph again, but this time with $x = 3$ and $y = 4 + \varepsilon$.

So forward-mode autodiff is much more accurate than finite difference approximation, but it suffers from the same major flaw, at least when there are many inputs and few outputs (as is the case when dealing with neural networks): if there were 1,000 parameters, it would require 1,000 passes through the graph to compute all the partial

derivatives. This is where reverse-mode autodiff shines: it can compute all of them in just two passes through the graph. Let's see how.

Reverse-Mode Autodiff

Reverse-mode autodiff is the solution implemented by TensorFlow. It first goes through the graph in the forward direction (i.e., from the inputs to the output) to compute the value of each node. Then it does a second pass, this time in the reverse direction (i.e., from the output to the inputs), to compute all the partial derivatives. The name "reverse mode" comes from this second pass through the graph, where gradients flow in the reverse direction. Figure D-3 represents the second pass. During the first pass, all the node values were computed, starting from $x = 3$ and $y = 4$. You can see those values at the bottom right of each node (e.g., $x \times x = 9$). The nodes are labeled n_1 to n_7 for clarity. The output node is n_7: $f(3, 4) = n_7 = 42$.

Figure D-3. Reverse-mode autodiff

The idea is to gradually go down the graph, computing the partial derivative of $f(x, y)$ with regard to each consecutive node, until we reach the variable nodes. For this, reverse-mode autodiff relies heavily on the *chain rule*, shown in Equation D-4.

Equation D-4. Chain rule

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

Since n_7 is the output node, $f = n_7$ so $\partial f/\partial n_7 = 1$.

Let's continue down the graph to n_5: how much does f vary when n_5 varies? The answer is $\partial f/\partial n_5 = \partial f/\partial n_7 \times \partial n_7/\partial n_5$. We already know that $\partial f/\partial n_7 = 1$, so all we need is $\partial n_7/\partial n_5$. Since n_7 simply performs the sum $n_5 + n_6$, we find that $\partial n_7/\partial n_5 = 1$, so $\partial f/\partial n_5 = 1 \times 1 = 1$.

Now we can proceed to node n_4: how much does f vary when n_4 varies? The answer is $\partial f/\partial n_4 = \partial f/\partial n_5 \times \partial n_5/\partial n_4$. Since $n_5 = n_4 \times n_2$, we find that $\partial n_5/\partial n_4 = n_2$, so $\partial f/\partial n_4 = 1 \times n_2 = 4$.

The process continues until we reach the bottom of the graph. At that point we will have calculated all the partial derivatives of $f(x, y)$ at the point $x = 3$ and $y = 4$. In this example, we find $\partial f/\partial x = 24$ and $\partial f/\partial y = 10$. Sounds about right!

Reverse-mode autodiff is a very powerful and accurate technique, especially when there are many inputs and few outputs, since it requires only one forward pass plus one reverse pass per output to compute all the partial derivatives for all outputs with regard to all the inputs. When training neural networks, we generally want to minimize the loss, so there is a single output (the loss), and hence only two passes through the graph are needed to compute the gradients. Reverse-mode autodiff can also handle functions that are not entirely differentiable, as long as you ask it to compute the partial derivatives at points that are differentiable.

In Figure D-3, the numerical results are computed on the fly, at each node. However, that's not exactly what TensorFlow does: instead, it creates a new computation graph. In other words, it implements *symbolic* reverse-mode autodiff. This way, the computation graph to compute the gradients of the loss with regard to all the parameters in the neural network only needs to be generated once, and then it can be executed over and over again, whenever the optimizer needs to compute the gradients. Moreover, this makes it possible to compute higher-order derivatives if needed.

If you ever want to implement a new type of low-level TensorFlow operation in C++, and you want to make it compatible with autodiff, then you will need to provide a function that returns the partial derivatives of the function's outputs with regard to its inputs. For example, suppose you implement a function that computes the square of its input: $f(x) = x^2$. In that case you would need to provide the corresponding derivative function: $f'(x) = 2x$.

APPENDIX E

Other Popular ANN Architectures

In this appendix I will give a quick overview of a few historically important neural network architectures that are much less used today than deep Multilayer Perceptrons (Chapter 10), convolutional neural networks (Chapter 14), recurrent neural networks (Chapter 15), or autoencoders (Chapter 17). They are often mentioned in the literature, and some are still used in a range of applications, so it is worth knowing about them. Additionally, we will discuss *deep belief nets*, which were the state of the art in Deep Learning until the early 2010s. They are still the subject of very active research, so they may well come back with a vengeance in the future.

Hopfield Networks

Hopfield networks were first introduced by W. A. Little in 1974, then popularized by J. Hopfield in 1982. They are *associative memory* networks: you first teach them some patterns, and then when they see a new pattern they (hopefully) output the closest learned pattern. This made them useful for character recognition, in particular, before they were outperformed by other approaches: you first train the network by showing it examples of character images (each binary pixel maps to one neuron), and then when you show it a new character image, after a few iterations it outputs the closest learned character.

Hopfield networks are fully connected graphs (see Figure E-1); that is, every neuron is connected to every other neuron. Note that in the diagram the images are 6×6 pixels, so the neural network on the left should contain 36 neurons (and 630 connections), but for visual clarity a much smaller network is represented.

Figure E-1. Hopfield network

The training algorithm works by using Hebb's rule (see "The Perceptron" on page 284): for each training image, the weight between two neurons is increased if the corresponding pixels are both on or both off, but decreased if one pixel is on and the other is off.

To show a new image to the network, you just activate the neurons that correspond to active pixels. The network then computes the output of every neuron, and this gives you a new image. You can then take this new image and repeat the whole process. After a while, the network reaches a stable state. Generally, this corresponds to the training image that most resembles the input image.

A so-called *energy function* is associated with Hopfield nets. At each iteration, the energy decreases, so the network is guaranteed to eventually stabilize to a low-energy state. The training algorithm tweaks the weights in a way that decreases the energy level of the training patterns, so the network is likely to stabilize in one of these low-energy configurations. Unfortunately, some patterns that were not in the training set also end up with low energy, so the network sometimes stabilizes in a configuration that was not learned. These are called *spurious patterns*.

Another major flaw with Hopfield nets is that they don't scale very well—their memory capacity is roughly equal to 14% of the number of neurons. For example, to classify 28 × 28–pixel images, you would need a Hopfield net with 784 fully connected neurons and 306,936 weights. Such a network would only be able to learn about 110 different characters (14% of 784). That's a lot of parameters for such a small memory.

Boltzmann Machines

Boltzmann machines were invented in 1985 by Geoffrey Hinton and Terrence Sejnowski. Just like Hopfield nets, they are fully connected ANNs, but they are based on *stochastic neurons*: instead of using a deterministic step function to decide what value to output, these neurons output 1 with some probability, and 0 otherwise. The probability function that these ANNs use is based on the Boltzmann distribution (used in statistical mechanics), hence their name. Equation E-1 gives the probability that a particular neuron will output 1.

Equation E-1. Probability that the ith neuron will output 1

$$p\left(s_i^{(\text{next step})} = 1\right) = \sigma\left(\frac{\Sigma_{j=1}^{N} w_{i,j} s_j + b_i}{T}\right)$$

- s_j is the jth neuron's state (0 or 1).
- $w_{i,j}$ is the connection weight between the ith and j^{th} neurons. Note that $w_{i,i} = 0$.
- b_i is the ith neuron's bias term. We can implement this term by adding a bias neuron to the network.
- N is the number of neurons in the network.
- T is a number called the network's *temperature*; the higher the temperature, the more random the output is (i.e., the more the probability approaches 50%).
- σ is the logistic function.

Neurons in Boltzmann machines are separated into two groups: *visible units* and *hidden units* (see Figure E-2). All neurons work in the same stochastic way, but the visible units are the ones that receive the inputs and from which outputs are read.

Because of its stochastic nature, a Boltzmann machine will never stabilize into a fixed configuration; instead, it will keep switching between many configurations. If it is left running for a sufficiently long time, the probability of observing a particular configuration will only be a function of the connection weights and bias terms, not of the original configuration (similarly, after you shuffle a deck of cards for long enough, the configuration of the deck does not depend on the initial state). When the network reaches this state where the original configuration is "forgotten," it is said to be in *thermal equilibrium* (although its configuration keeps changing all the time). By setting the network parameters appropriately, letting the network reach thermal equilibrium, and then observing its state, we can simulate a wide range of probability distributions. This is called a *generative model*.

Figure E-2. Boltzmann machine

Training a Boltzmann machine means finding the parameters that will make the network approximate the training set's probability distribution. For example, if there are three visible neurons and the training set contains 75% (0, 1, 1) triplets, 10% (0, 0, 1) triplets, and 15% (1, 1, 1) triplets, then after training a Boltzmann machine, you could use it to generate random binary triplets with about the same probability distribution. For example, about 75% of the time it would output the (0, 1, 1) triplet.

Such a generative model can be used in a variety of ways. For example, if it is trained on images, and you provide an incomplete or noisy image to the network, it will automatically "repair" the image in a reasonable way. You can also use a generative model for classification. Just add a few visible neurons to encode the training image's class (e.g., add 10 visible neurons and turn on only the fifth neuron when the training image represents a 5). Then, when given a new image, the network will automatically turn on the appropriate visible neurons, indicating the image's class (e.g., it will turn on the fifth visible neuron if the image represents a 5).

Unfortunately, there is no efficient technique to train Boltzmann machines. However, fairly efficient algorithms have been developed to train *restricted Boltzmann machines* (RBMs).

Restricted Boltzmann Machines

An RBM is simply a Boltzmann machine in which there are no connections between visible units or between hidden units, only between visible and hidden units. For example, Figure E-3 represents an RBM with three visible units and four hidden units.

Figure E-3. Restricted Boltzmann machine

A very efficient training algorithm called *Contrastive Divergence* was introduced in 2005 by Miguel Á. Carreira-Perpiñán and Geoffrey Hinton.[1] Here is how it works: for each training instance **x**, the algorithm starts by feeding it to the network by setting the state of the visible units to $x_1, x_2, \cdots, x_n$. Then you compute the state of the hidden units by applying the stochastic equation described before (Equation E-1). This gives you a hidden vector **h** (where h_i is equal to the state of the ith unit). Next you compute the state of the visible units, by applying the same stochastic equation. This gives you a vector $\mathbf{x}'$. Then once again you compute the state of the hidden units, which gives you a vector $\mathbf{h}'$. Now you can update each connection weight by applying the rule in Equation E-2, where η is the learning rate.

Equation E-2. Contrastive divergence weight update

$$w_{i,j} \leftarrow w_{i,j} + \eta\left(\mathbf{x}\mathbf{h}^{\mathsf{T}} - \mathbf{x}'\mathbf{h}'^{\mathsf{T}}\right)$$

The great benefit of this algorithm is that it does not require waiting for the network to reach thermal equilibrium: it just goes forward, backward, and forward again, and that's it. This makes it incomparably more efficient than previous algorithms, and it was a key ingredient to the first success of Deep Learning based on multiple stacked RBMs.

Deep Belief Nets

Several layers of RBMs can be stacked; the hidden units of the first-level RBM serve as the visible units for the second-layer RBM, and so on. Such an RBM stack is called a *deep belief net* (DBN).

1 Miguel Á. Carreira-Perpiñán and Geoffrey E. Hinton, "On Contrastive Divergence Learning," *Proceedings of the 10th International Workshop on Artificial Intelligence and Statistics* (2005): 59–66.

Yee-Whye Teh, one of Geoffrey Hinton's students, observed that it was possible to train DBNs one layer at a time using Contrastive Divergence, starting with the lower layers and then gradually moving up to the top layers. This led to the groundbreaking article that kickstarted the Deep Learning tsunami in 2006.[2]

Just like RBMs, DBNs learn to reproduce the probability distribution of their inputs, without any supervision. However, they are much better at it, for the same reason that deep neural networks are more powerful than shallow ones: real-world data is often organized in hierarchical patterns, and DBNs take advantage of that. Their lower layers learn low-level features in the input data, while higher layers learn high-level features.

Just like RBMs, DBNs are fundamentally unsupervised, but you can also train them in a supervised manner by adding some visible units to represent the labels. Moreover, one great feature of DBNs is that they can be trained in a semisupervised fashion. Figure E-4 represents such a DBN configured for semisupervised learning.

Figure E-4. A deep belief network configured for semisupervised learning

First, RBM 1 is trained without supervision. It learns low-level features in the training data. Then RBM 2 is trained with RBM 1's hidden units as inputs, again without

2 Geoffrey E. Hinton et al., "A Fast Learning Algorithm for Deep Belief Nets," *Neural Computation* 18 (2006): 1527–1554.

supervision: it learns higher-level features (note that RBM 2's hidden units include only the three rightmost units, not the label units). Several more RBMs could be stacked this way, but you get the idea. So far, training was 100% unsupervised. Lastly, RBM 3 is trained using RBM 2's hidden units as inputs, as well as extra visible units used to represent the target labels (e.g., a one-hot vector representing the instance class). It learns to associate high-level features with training labels. This is the supervised step.

At the end of training, if you feed RBM 1 a new instance, the signal will propagate up to RBM 2, then up to the top of RBM 3, and then back down to the label units; hopefully, the appropriate label will light up. This is how a DBN can be used for classification.

One great benefit of this semisupervised approach is that you don't need much labeled training data. If the unsupervised RBMs do a good enough job, then only a small amount of labeled training instances per class will be necessary. Similarly, a baby learns to recognize objects without supervision, so when you point to a chair and say "chair," the baby can associate the word "chair" with the class of objects it has already learned to recognize on its own. You don't need to point to every single chair and say "chair"; only a few examples will suffice (just enough so the baby can be sure that you are indeed referring to the chair, not to its color or one of the chair's parts).

Quite amazingly, DBNs can also work in reverse. If you activate one of the label units, the signal will propagate up to the hidden units of RBM 3, then down to RBM 2, and then RBM 1, and a new instance will be output by the visible units of RBM 1. This new instance will usually look like a regular instance of the class whose label unit you activated. This generative capability of DBNs is quite powerful. For example, it has been used to automatically generate captions for images, and vice versa: first a DBN is trained (without supervision) to learn features in images, and another DBN is trained (again without supervision) to learn features in sets of captions (e.g., "car" often comes with "automobile"). Then an RBM is stacked on top of both DBNs and trained with a set of images along with their captions; it learns to associate high-level features in images with high-level features in captions. Next, if you feed the image DBN an image of a car, the signal will propagate through the network, up to the top-level RBM, and back down to the bottom of the caption DBN, producing a caption. Due to the stochastic nature of RBMs and DBNs, the caption will keep changing randomly, but it will generally be appropriate for the image. If you generate a few hundred captions, the most frequently generated ones will likely be a good description of the image.[3]

3 See this video by Geoffrey Hinton for more details and a demo: *https://homl.info/137*.

Self-Organizing Maps

Self-organizing maps (SOMs) are quite different from all the other types of neural networks we have discussed so far. They are used to produce a low-dimensional representation of a high-dimensional dataset, generally for visualization, clustering, or classification. The neurons are spread across a map (typically 2D for visualization, but it can be any number of dimensions you want), as shown in Figure E-5, and each neuron has a weighted connection to every input (note that the diagram shows just two inputs, but there are typically a very large number, since the whole point of SOMs is to reduce dimensionality).

Figure E-5. Self-organizing map

Once the network is trained, you can feed it a new instance and this will activate only one neuron (i.e., one point on the map): the neuron whose weight vector is closest to the input vector. In general, instances that are nearby in the original input space will activate neurons that are nearby on the map. This makes SOMs useful not only for visualization (in particular, you can easily identify clusters on the map), but also for applications like speech recognition. For example, if each instance represents an audio recording of a person pronouncing a vowel, then different pronunciations of the vowel "a" will activate neurons in the same area of the map, while instances of the vowel "e" will activate neurons in another area, and intermediate sounds will generally activate intermediate neurons on the map.

One important difference from the other dimensionality reduction techniques discussed in Chapter 8 is that all instances get mapped to a discrete number of points in the low-dimensional space (one point per neuron). When there are very few neurons, this technique is better described as clustering rather than dimensionality reduction.

The training algorithm is unsupervised. It works by having all the neurons compete against each other. First, all the weights are initialized randomly. Then a training instance is picked randomly and fed to the network. All neurons compute the distance between their weight vector and the input vector (this is very different from the artificial neurons we have seen so far). The neuron that measures the smallest distance wins and tweaks its weight vector to be slightly closer to the input vector, making it more likely to win future competitions for other inputs similar to this one. It also recruits its neighboring neurons, and they too update their weight vectors to be slightly closer to the input vector (but they don't update their weights as much as the winning neuron). Then the algorithm picks another training instance and repeats the process, again and again. This algorithm tends to make nearby neurons gradually specialize in similar inputs.[4]

4 You can imagine a class of young children with roughly similar skills. One child happens to be slightly better at basketball. This motivates them to practice more, especially with their friends. After a while, this group of friends gets so good at basketball that other kids cannot compete. But that's okay, because the other kids specialize in other areas. After a while, the class is full of little specialized groups.

APPENDIX F
Special Data Structures

In this appendix we will take a very quick look at the data structures supported by TensorFlow, beyond regular float or integer tensors. This includes strings, ragged tensors, sparse tensors, tensor arrays, sets, and queues.

Strings

Tensors can hold byte strings, which is useful in particular for natural language processing (see Chapter 16):

```
>>> tf.constant(b"hello world")
<tf.Tensor: id=149, shape=(), dtype=string, numpy=b'hello world'>
```

If you try to build a tensor with a Unicode string, TensorFlow automatically encodes it to UTF-8:

```
>>> tf.constant("café")
<tf.Tensor: id=138, shape=(), dtype=string, numpy=b'caf\xc3\xa9'>
```

It is also possible to create tensors representing Unicode strings. Just create an array of 32-bit integers, each representing a single Unicode code point:[1]

```
>>> tf.constant([ord(c) for c in "café"])
<tf.Tensor: id=211, shape=(4,), dtype=int32,
            numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

1 If you are not familiar with Unicode code points, please check out *https://homl.info/unicode*.

In tensors of type tf.string, the string length is not part of the tensor's shape. In other words, strings are considered as atomic values. However, in a Unicode string tensor (i.e., an int32 tensor), the length of the string *is* part of the tensor's shape.

The `tf.strings` package contains several functions to manipulate string tensors, such as `length()` to count the number of bytes in a byte string (or the number of code points if you set `unit="UTF8_CHAR"`), `unicode_encode()` to convert a Unicode string tensor (i.e., int32 tensor) to a byte string tensor, and `unicode_decode()` to do the reverse:

```
>>> b = tf.strings.unicode_encode(u, "UTF-8")
>>> tf.strings.length(b, unit="UTF8_CHAR")
<tf.Tensor: id=386, shape=(), dtype=int32, numpy=4>
>>> tf.strings.unicode_decode(b, "UTF-8")
<tf.Tensor: id=393, shape=(4,), dtype=int32,
            numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

You can also manipulate tensors containing multiple strings:

```
>>> p = tf.constant(["Café", "Coffee", "caffè", "咖啡"])
>>> tf.strings.length(p, unit="UTF8_CHAR")
<tf.Tensor: id=299, shape=(4,), dtype=int32,
            numpy=array([4, 6, 5, 2], dtype=int32)>
>>> r = tf.strings.unicode_decode(p, "UTF8")
>>> r
tf.RaggedTensor(values=tf.Tensor(
[   67    97   102   233    67   111   102   102   101   101    99    97
   102   102   232 21654 21857], shape=(17,), dtype=int32),
   row_splits=tf.Tensor([ 0  4 10 15 17], shape=(5,), dtype=int64))
>>> print(r)
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101],
                  [99, 97, 102, 102, 232], [21654, 21857]]>
```

Notice that the decoded strings are stored in a `RaggedTensor`. What is that?

Ragged Tensors

A ragged tensor is a special kind of tensor that represents a list of arrays of different sizes. More generally, it is a tensor with one or more *ragged dimensions*, meaning dimensions whose slices may have different lengths. In the ragged tensor `r`, the second dimension is a ragged dimension. In all ragged tensors, the first dimension is always a regular dimension (also called a *uniform dimension*).

All the elements of the ragged tensor `r` are regular tensors. For example, let's look at the second element of the ragged tensor:

```
>>> print(r[1])
tf.Tensor([ 67 111 102 102 101 101], shape=(6,), dtype=int32)
```

The `tf.ragged` package contains several functions to create and manipulate ragged tensors. Let's create a second ragged tensor using `tf.ragged.constant()` and concatenate it with the first ragged tensor, along axis 0:

```
>>> r2 = tf.ragged.constant([[65, 66], [], [67]])
>>> print(tf.concat([r, r2], axis=0))
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101], [99, 97,
102, 102, 232], [21654, 21857], [65, 66], [], [67]]>
```

The result is not too surprising: the tensors in `r2` were appended after the tensors in `r` along axis 0. But what if we concatenate `r` and another ragged tensor along axis 1?

```
>>> r3 = tf.ragged.constant([[68, 69, 70], [71], [], [72, 73]])
>>> print(tf.concat([r, r3], axis=1))
<tf.RaggedTensor [[67, 97, 102, 233, 68, 69, 70], [67, 111, 102, 102, 101, 101,
71], [99, 97, 102, 102, 232], [21654, 21857, 72, 73]]>
```

This time, notice that the ith tensor in `r` and the ith tensor in `r3` were concatenated. Now that's more unusual, since all of these tensors can have different lengths.

If you call the `to_tensor()` method, it gets converted to a regular tensor, padding shorter tensors with zeros to get tensors of equal lengths (you can change the default value by setting the `default_value` argument):

```
>>> r.to_tensor()
<tf.Tensor: id=1056, shape=(4, 6), dtype=int32, numpy=
array([[   67,    97,   102,   233,     0,     0],
       [   67,   111,   102,   102,   101,   101],
       [   99,    97,   102,   102,   232,     0],
       [21654, 21857,     0,     0,     0,     0]], dtype=int32)>
```

Many TF operations support ragged tensors. For the full list, see the documentation of the `tf.RaggedTensor` class.

Sparse Tensors

TensorFlow can also efficiently represent *sparse tensors* (i.e., tensors containing mostly zeros). Just create a `tf.SparseTensor`, specifying the indices and values of the nonzero elements and the tensor's shape. The indices must be listed in "reading order" (from left to right, and top to bottom). If you are unsure, just use `tf.sparse.reorder()`. You can convert a sparse tensor to a dense tensor (i.e., a regular tensor) using `tf.sparse.to_dense()`:

```
>>> s = tf.SparseTensor(indices=[[0, 1], [1, 0], [2, 3]],
                        values=[1., 2., 3.],
                        dense_shape=[3, 4])
>>> tf.sparse.to_dense(s)
<tf.Tensor: id=1074, shape=(3, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [2., 0., 0., 0.],
       [0., 0., 0., 3.]], dtype=float32)>
```

Note that sparse tensors do not support as many operations as dense tensors. For example, you can multiply a sparse tensor by any scalar value, and you get a new sparse tensor, but you cannot add a scalar value to a sparse tensor, as this would not return a sparse tensor:

```
>>> s * 3.14
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x13205d470>
>>> s + 42.0
[...] TypeError: unsupported operand type(s) for +: 'SparseTensor' and 'float'
```

Tensor Arrays

A `tf.TensorArray` represents a list of tensors. This can be handy in dynamic models containing loops, to accumulate results and later compute some statistics. You can read or write tensors at any location in the array:

```
array = tf.TensorArray(dtype=tf.float32, size=3)
array = array.write(0, tf.constant([1., 2.]))
array = array.write(1, tf.constant([3., 10.]))
array = array.write(2, tf.constant([5., 7.]))
tensor1 = array.read(1) # => returns (and pops!) tf.constant([3., 10.])
```

Notice that reading an item pops it from the array, replacing it with a tensor of the same shape, full of zeros.

When you write to the array, you must assign the output back to the array, as shown in this code example. If you don't, although your code will work fine in eager mode, it will break in graph mode (these modes were presented in Chapter 12).

When creating a `TensorArray`, you must provide its `size`, except in graph mode. Alternatively, you can leave the `size` unset and instead set `dynamic_size=True`, but this will hinder performance, so if you know the `size` in advance, you should set it. You must also specify the `dtype`, and all elements must have the same shape as the first one written to the array.

You can stack all the items into a regular tensor by calling the `stack()` method:

```
>>> array.stack()
<tf.Tensor: id=2110875, shape=(3, 2), dtype=float32, numpy=
array([[1., 2.],
       [0., 0.],
       [5., 7.]], dtype=float32)>
```

Sets

TensorFlow supports sets of integers or strings (but not floats). It represents them using regular tensors. For example, the set `{1, 5, 9}` is just represented as the tensor `[[1, 5, 9]]`. Note that the tensor must have at least two dimensions, and the sets must be in the last dimension. For example, `[[1, 5, 9], [2, 5, 11]]` is a tensor holding two independent sets: `{1, 5, 9}` and `{2, 5, 11}`. If some sets are shorter than others, you must pad them with a padding value (0 by default, but you can use any other value you prefer).

The `tf.sets` package contains several functions to manipulate sets. For example, let's create two sets and compute their union (the result is a sparse tensor, so we call `to_dense()` to display it):

```
>>> a = tf.constant([[1, 5, 9]])
>>> b = tf.constant([[5, 6, 9, 11]])
>>> u = tf.sets.union(a, b)
>>> u
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x132b60d30>
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11]], dtype=int32)>
```

You can also compute the union of multiple pairs of sets simultaneously:

```
>>> a = tf.constant([[1, 5, 9], [10, 0, 0]])
>>> b = tf.constant([[5, 6, 9, 11], [13, 0, 0, 0, 0]])
>>> u = tf.sets.union(a, b)
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],
                               [ 0, 10, 13,  0,  0]], dtype=int32)>
```

If you prefer to use a different padding value, you must set `default_value` when calling `to_dense()`:

```
>>> tf.sparse.to_dense(u, default_value=-1)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],
                               [ 0, 10, 13, -1, -1]], dtype=int32)>
```

The default `default_value` is 0, so when dealing with string sets, you must set the `default_value` (e.g., to an empty string).

Other functions available in `tf.sets` include `difference()`, `intersection()`, and `size()`, which are self-explanatory. If you want to check whether or not a set contains some given values, you can compute the intersection of that set and the values. If you want to add some values to a set, you can compute the union of the set and the values.

Queues

A queue is a data structure to which you can push data records, and later pull them out. TensorFlow implements several types of queues in the `tf.queue` package. They used to be very important when implementing efficient data loading and preprocessing pipelines, but the tf.data API has essentially rendered them useless (except perhaps in some rare cases) because it is much simpler to use and provides all the tools you need to build efficient pipelines. For the sake of completeness, though, let's take a quick look at them.

The simplest kind of queue is the first-in, first-out (FIFO) queue. To build it, you need to specify the maximum number of records it can contain. Moreover, each record is a tuple of tensors, so you must specify the type of each tensor, and optionally their shapes. For example, the following code example creates a FIFO queue with maximum three records, each containing a tuple with a 32-bit integer and a string. Then it pushes two records to it, looks at the size (which is 2 at this point), and pulls a record out:

```
>>> q = tf.queue.FIFOQueue(3, [tf.int32, tf.string], shapes=[(), ()])
>>> q.enqueue([10, b"windy"])
>>> q.enqueue([15, b"sunny"])
>>> q.size()
<tf.Tensor: id=62, shape=(), dtype=int32, numpy=2>
>>> q.dequeue()
[<tf.Tensor: id=6, shape=(), dtype=int32, numpy=10>,
 <tf.Tensor: id=7, shape=(), dtype=string, numpy=b'windy'>]
```

It is also possible to enqueue and dequeue multiple records at once (the latter requires specifying the shapes when creating the queue):

```
>>> q.enqueue_many([[13, 16], [b'cloudy', b'rainy']])
>>> q.dequeue_many(3)
[<tf.Tensor: [...] numpy=array([15, 13, 16], dtype=int32)>,
 <tf.Tensor: [...] numpy=array([b'sunny', b'cloudy', b'rainy'], dtype=object)>]
```

Other queue types include:

`PaddingFIFOQueue`
: Same as `FIFOQueue`, but its `dequeue_many()` method supports dequeueing multiple records of different shapes. It automatically pads the shortest records to ensure all the records in the batch have the same shape.

`PriorityQueue`
: A queue that dequeues records in a prioritized order. The priority must be a 64-bit integer included as the first element of each record. Surprisingly, records with a lower priority will be dequeued first. Records with the same priority will be dequeued in FIFO order.

`RandomShuffleQueue`
: A queue whose records are dequeued in random order. This was useful to implement a shuffle buffer before tf.data existed.

If a queue is already full and you try to enqueue another record, the `enqueue*()` method will freeze until a record is dequeued by another thread. Similarly, if a queue is empty and you try to dequeue a record, the `dequeue*()` method will freeze until records are pushed to the queue by another thread.

APPENDIX G
TensorFlow Graphs

In this appendix, we will explore the graphs generated by TF Functions (see Chapter 12).

TF Functions and Concrete Functions

TF Functions are polymorphic, meaning they support inputs of different types (and shapes). For example, consider the following `tf_cube()` function:

```
@tf.function
def tf_cube(x):
    return x ** 3
```

Every time you call a TF Function with a new combination of input types or shapes, it generates a new *concrete function*, with its own graph specialized for this particular combination. Such a combination of argument types and shapes is called an *input signature*. If you call the TF Function with an input signature it has already seen before, it will reuse the concrete function it generated earlier. For example, if you call `tf_cube(tf.constant(3.0))`, the TF Function will reuse the same concrete function it used for `tf_cube(tf.constant(2.0))` (for float32 scalar tensors). But it will generate a new concrete function if you call `tf_cube(tf.constant([2.0]))` or `tf_cube(tf.constant([3.0]))` (for float32 tensors of shape [1]), and yet another for `tf_cube(tf.constant([[1.0, 2.0], [3.0, 4.0]]))` (for float32 tensors of shape [2, 2]). You can get the concrete function for a particular combination of inputs by calling the TF Function's `get_concrete_function()` method. It can then be called like a regular function, but it will only support one input signature (in this example, float32 scalar tensors):

```
>>> concrete_function = tf_cube.get_concrete_function(tf.constant(2.0))
>>> concrete_function
<tensorflow.python.eager.function.ConcreteFunction at 0x155c29240>
>>> concrete_function(tf.constant(2.0))
<tf.Tensor: id=19068249, shape=(), dtype=float32, numpy=8.0>
```

Figure G-1 shows the `tf_cube()` TF Function, after we called `tf_cube(2)` and `tf_cube(tf.constant(2.0))`: two concrete functions were generated, one for each signature, each with its own optimized *function graph* (`FuncGraph`), and its own *function definition* (`FunctionDef`). A function definition points to the parts of the graph that correspond to the function's inputs and outputs. In each `FuncGraph`, the nodes (ovals) represent operations (e.g., power, constants, or placeholders for arguments like `x`), while the edges (the solid arrows between the operations) represent the tensors that will flow through the graph. The concrete function on the left is specialized for `x = 2`, so TensorFlow managed to simplify it to just output 8 all the time (note that the function definition does not even have an input). The concrete function on the right is specialized for float32 scalar tensors, and it could not be simplified. If we call `tf_cube(tf.constant(5.0))`, the second concrete function will be called, the placeholder operation for `x` will output 5.0, then the power operation will compute `5.0 ** 3`, so the output will be 125.0.

Figure G-1. The tf_cube() TF Function, with its ConcreteFunctions and their FunctionGraphs

The tensors in these graphs are *symbolic tensors*, meaning they don't have an actual value, just a data type, a shape, and a name. They represent the future tensors that will flow through the graph once an actual value is fed to the placeholder `x` and the graph is executed. Symbolic tensors make it possible to specify ahead of time how to

connect operations, and they also allow TensorFlow to recursively infer the data types and shapes of all tensors, given the data types and shapes of their inputs.

Now let's continue to peek under the hood, and see how to access function definitions and function graphs and how to explore a graph's operations and tensors.

Exploring Function Definitions and Graphs

You can access a concrete function's computation graph using the `graph` attribute, and get the list of its operations by calling the graph's `get_operations()` method:

```
>>> concrete_function.graph
<tensorflow.python.framework.func_graph.FuncGraph at 0x14db5ef98>
>>> ops = concrete_function.graph.get_operations()
>>> ops
[<tf.Operation 'x' type=Placeholder>,
 <tf.Operation 'pow/y' type=Const>,
 <tf.Operation 'pow' type=Pow>,
 <tf.Operation 'Identity' type=Identity>]
```

In this example, the first operation represents the input argument `x` (it is called a *placeholder*), the second "operation" represents the constant 3, the third operation represents the power operation (`**`), and the final operation represents the output of this function (it is an identity operation, meaning it will do nothing more than copy the output of the addition operation[1]). Each operation has a list of input and output tensors that you can easily access using the operation's `inputs` and `outputs` attributes. For example, let's get the list of inputs and outputs of the power operation:

```
>>> pow_op = ops[2]
>>> list(pow_op.inputs)
[<tf.Tensor 'x:0' shape=() dtype=float32>,
 <tf.Tensor 'pow/y:0' shape=() dtype=float32>]
>>> pow_op.outputs
[<tf.Tensor 'pow:0' shape=() dtype=float32>]
```

This computation graph is represented in Figure G-2.

1 You can safely ignore it—it is only here for technical reasons, to ensure that TF Functions don't leak internal structures.

Figure G-2. Example of a computation graph

Note that each operation has a name. It defaults to the name of the operation (e.g., `"pow"`), but you can define it manually when calling the operation (e.g., `tf.pow(x, 3, name="other_name")`). If a name already exists, TensorFlow automatically adds a unique index (e.g., `"pow_1"`, `"pow_2"`, etc.). Each tensor also has a unique name: it is always the name of the operation that outputs this tensor, plus `:0` if it is the operation's first output, or `:1` if it is the second output, and so on. You can fetch an operation or a tensor by name using the graph's `get_operation_by_name()` or `get_tensor_by_name()` methods:

```
>>> concrete_function.graph.get_operation_by_name('x')
<tf.Operation 'x' type=Placeholder>
>>> concrete_function.graph.get_tensor_by_name('Identity:0')
<tf.Tensor 'Identity:0' shape=() dtype=float32>
```

The concrete function also contains the function definition (represented as a protocol buffer[2]), which includes the function's signature. This signature allows the concrete function to know which placeholders to feed with the input values, and which tensors to return:

```
>>> concrete_function.function_def.signature
name: "__inference_cube_19068241"
input_arg {
  name: "x"
  type: DT_FLOAT
}
output_arg {
  name: "identity"
  type: DT_FLOAT
}
```

2 A popular binary format discussed in Chapter 13.

Now let's look more closely at tracing.

A Closer Look at Tracing

Let's tweak the `tf_cube()` function to print its input:

```
@tf.function
def tf_cube(x):
    print("x =", x)
    return x ** 3
```

Now let's call it:

```
>>> result = tf_cube(tf.constant(2.0))
x = Tensor("x:0", shape=(), dtype=float32)
>>> result
<tf.Tensor: id=19068290, shape=(), dtype=float32, numpy=8.0>
```

The `result` looks good, but look at what was printed: `x` is a symbolic tensor! It has a shape and a data type, but no value. Plus it has a name (`"x:0"`). This is because the `print()` function is not a TensorFlow operation, so it will only run when the Python function is traced, which happens in graph mode, with arguments replaced with symbolic tensors (same type and shape, but no value). Since the `print()` function was not captured into the graph, the next times we call `tf_cube()` with float32 scalar tensors, nothing is printed:

```
>>> result = tf_cube(tf.constant(3.0))
>>> result = tf_cube(tf.constant(4.0))
```

But if we call `tf_cube()` with a tensor of a different type or shape, or with a new Python value, the function will be traced again, so the `print()` function will be called:

```
>>> result = tf_cube(2) # new Python value: trace!
x = 2
>>> result = tf_cube(3) # new Python value: trace!
x = 3
>>> result = tf_cube(tf.constant([[1., 2.]])) # New shape: trace!
x = Tensor("x:0", shape=(1, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[3., 4.], [5., 6.]])) # New shape: trace!
x = Tensor("x:0", shape=(None, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[7., 8.], [9., 10.]])) # Same shape: no trace
```

If your function has Python side effects (e.g., it saves some logs to disk), be aware that this code will only run when the function is traced (i.e., every time the TF Function is called with a new input signature). It best to assume that the function may be traced (or not) any time the TF Function is called.

In some cases, you may want to restrict a TF Function to a specific input signature. For example, suppose you know that you will only ever call a TF Function with batches of 28 × 28–pixel images, but the batches will have very different sizes. You may not want TensorFlow to generate a different concrete function for each batch size, or count on it to figure out on its own when to use `None`. In this case, you can specify the input signature like this:

```
@tf.function(input_signature=[tf.TensorSpec([None, 28, 28], tf.float32)])
def shrink(images):
    return images[:, ::2, ::2] # drop half the rows and columns
```

This TF Function will accept any float32 tensor of shape [*, 28, 28], and it will reuse the same concrete function every time:

```
img_batch_1 = tf.random.uniform(shape=[100, 28, 28])
img_batch_2 = tf.random.uniform(shape=[50, 28, 28])
preprocessed_images = shrink(img_batch_1) # Works fine. Traces the function.
preprocessed_images = shrink(img_batch_2) # Works fine. Same concrete function.
```

However, if you try to call this TF Function with a Python value, or a tensor of an unexpected data type or shape, you will get an exception:

```
img_batch_3 = tf.random.uniform(shape=[2, 2, 2])
preprocessed_images = shrink(img_batch_3)  # ValueError! Unexpected signature.
```

Using AutoGraph to Capture Control Flow

If your function contains a simple `for` loop, what do you expect will happen? For example, let's write a function that will add 10 to its input, by just adding 1 10 times:

```
@tf.function
def add_10(x):
    for i in range(10):
        x += 1
    return x
```

It works fine, but when we look at its graph, we find that it does not contain a loop: it just contains 10 addition operations!

```
>>> add_10(tf.constant(0))
<tf.Tensor: id=19280066, shape=(), dtype=int32, numpy=10>
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'add' type=Add>, [...],
 <tf.Operation 'add_1' type=Add>, [...],
 <tf.Operation 'add_2' type=Add>, [...],
 [...]
 <tf.Operation 'add_9' type=Add>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

This actually makes sense: when the function got traced, the loop ran 10 times, so the `x += 1` operation was run 10 times, and since it was in graph mode, it recorded this operation 10 times in the graph. You can think of this `for` loop as a "static" loop that gets unrolled when the graph is created.

If you want the graph to contain a "dynamic" loop instead (i.e., one that runs when the graph is executed), you can create one manually using the `tf.while_loop()` operation, but it is not very intuitive (see the "Using AutoGraph to Capture Control Flow" section of the Chapter 12 notebook for an example). Instead, it is much simpler to use TensorFlow's *AutoGraph* feature, discussed in Chapter 12. AutoGraph is actually activated by default (if you ever need to turn it off, you can pass `autograph=False` to `tf.function()`). So if it is on, why didn't it capture the `for` loop in the `add_10()` function? Well, it only captures `for` loops that iterate over `tf.range()`, not `range()`. This is to give you the choice:

- If you use `range()`, the `for` loop will be static, meaning it will only be executed when the function is traced. The loop will be "unrolled" into a set of operations for each iteration, as we saw.
- If you use `tf.range()`, the loop will be dynamic, meaning that it will be included in the graph itself (but it will not run during tracing).

Let's look at the graph that gets generated if you just replace `range()` with `tf.range()` in the `add_10()` function:

```
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'range' type=Range>, [...],
 <tf.Operation 'while' type=While>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

As you can see, the graph now contains a `While` loop operation, as if you had called the `tf.while_loop()` function.

Handling Variables and Other Resources in TF Functions

In TensorFlow, variables and other stateful objects, such as queues or datasets, are called *resources*. TF Functions treat them with special care: any operation that reads or updates a resource is considered stateful, and TF Functions ensure that stateful operations are executed in the order they appear (as opposed to stateless operations, which may be run in parallel, so their order of execution is not guaranteed). Moreover, when you pass a resource as an argument to a TF Function, it gets passed by reference, so the function may modify it. For example:

```
counter = tf.Variable(0)

@tf.function
def increment(counter, c=1):
    return counter.assign_add(c)

increment(counter) # counter is now equal to 1
increment(counter) # counter is now equal to 2
```

If you peek at the function definition, the first argument is marked as a resource:

```
>>> function_def = increment.get_concrete_function(counter).function_def
>>> function_def.signature.input_arg[0]
name: "counter"
type: DT_RESOURCE
```

It is also possible to use a `tf.Variable` defined outside of the function, without explicitly passing it as an argument:

```
counter = tf.Variable(0)

@tf.function
def increment(c=1):
    return counter.assign_add(c)
```

The TF Function will treat this as an implicit first argument, so it will actually end up with the same signature (except for the name of the argument). However, using global variables can quickly become messy, so you should generally wrap variables (and other resources) inside classes. The good news is `@tf.function` works fine with methods too:

```
class Counter:
    def __init__(self):
        self.counter = tf.Variable(0)

    @tf.function
    def increment(self, c=1):
        return self.counter.assign_add(c)
```

Do not use =, +=, -=, or any other Python assignment operator with TF variables. Instead, you must use the `assign()`, `assign_add()`, or `assign_sub()` methods. If you try to use a Python assignment operator, you will get an exception when you call the method.

A good example of this object-oriented approach is, of course, tf.keras. Let's see how to use TF Functions with tf.keras.

Using TF Functions with tf.keras (or Not)

By default, any custom function, layer, or model you use with tf.keras will automatically be converted to a TF Function; you do not need to do anything at all! However, in some cases you may want to deactivate this automatic conversion—for example, if your custom code cannot be turned into a TF Function, or if you just want to debug your code, which is much easier in eager mode. To do this, you can simply pass `dynamic=True` when creating the model or any of its layers:

```
model = MyModel(dynamic=True)
```

If your custom model or layer will always be dynamic, you can instead call the base class's constructor with `dynamic=True`:

```
class MyLayer(keras.layers.Layer):
    def __init__(self, units, **kwargs):
        super().__init__(dynamic=True, **kwargs)
        [...]
```

Alternatively, you can pass `run_eagerly=True` when calling the `compile()` method:

```
model.compile(loss=my_mse, optimizer="nadam", metrics=[my_mae],
              run_eagerly=True)
```

Now you know how TF Functions handle polymorphism (with multiple concrete functions), how graphs are automatically generated using AutoGraph and tracing, what graphs look like, how to explore their symbolic operations and tensors, how to handle variables and resources, and how to use TF Functions with tf.keras.

Index

Symbols

1cycle scheduling, 361
1D convolutional layers, 520

A

A/B experiments, 667
accelerated K-Means, 244
accuracy
  defined, 89
  example of, 2
  measuring using cross-validation, 89
action advantage, 620
action step, 656
actions
  evaluating, 619
  exploiting versus exploring, 618
activation functions
  exponential linear unit (ELU), 336-338
  hyperbolic tangent (tanh), 291
  Logistic (sigmoid), 143, 293, 302, 332
  nonsaturating, 335
  Rectified Linear Unit function (ReLU), 292-293
  Scaled Exponential Linear Unit (SELU), 334, 337-338, 368
  softmax, 294, 299, 470, 482, 488, 543
  softplus, 293
active constraint, 762
active learning, 255
Actor-Critic algorithms, 625, 662
AdaBoost, 200
AdaGrad, 354
Adam and Nadam optimization, 356
Adaptive Boosting, 200
adaptive instance normalization (AdaIN), 604
adaptive learning rate, 355
adaptive moment estimation, 356
additive attention, 550
Advantage Actor-Critic (A2C), 663
adversarial learning, 495, 568
affine transformations, 604
affinity, 237
affinity propagation, 259
agents, 14
agglomerative clustering, 258
AI Platform, 680
Akaike information criterion (AIC), 267
AlexNet, 464
algorithms
  Actor-Critic algorithms, 625, 662
  Advantage Actor-Critic (A2C), 663
  AllReduce algorithm, 705
  Asynchronous Advantage Actor-Critic (A3C), 662
  BIRCH algorithm, 259
  CART training algorithm, 177, 179
  clustering algorithms, 10
  Dueling DQN algorithm, 641
  dynamic placer algorithm, 697
  Expectation-Maximization (EM) algorithm, 262
  for anomaly detection, 274
  genetic algorithms, 612
  greedy algorithms, 180
  hierarchical clustering algorithms, 10
  importance of data over, 24
  Isolation Forest algorithm, 274
  isomap algorithm, 233

K-Means algorithm, 238
Lloyd–Forgy algorithm, 238
Mean-Shift algorithm, 259
off-policy algorithms, 632
on-policy algorithms, 632
one-class SVM algorithm, 275
Proximal Policy Optimization (PPO), 663
Randomized PCA algorithm, 225
REINFORCE algorithms, 620
Soft Actor-Critic algorithm, 663
supervised learning, 8
unsupervised learning, 9
Value Iteration algorithm, 627
visualization algorithms, 11
AllReduce algorithm, 705
alpha channels, 250
anchor boxes, 490
anomaly detection
additional algorithms for, 274
examples of, 12
goal of, 236
using clustering, 237
using Gaussian Mixtures, 266
Approximate Q-Learning, 633
area under the curve (AUC), 98
argmax operator, 149
artificial neural networks (ANNs)
Boltzmann machines, 775
fine-tuning hyperparameters for, 320-327
from biological to artificial neurons, 280-295
Hopfield networks, 773
implementing MLPs with Keras, 295-320
overview of, 279
restricted Boltzmann machines (RBMs), 776
self-organizing maps (SOMs), 780
artificial neurons, 283
association rule learning, 12
associative memory networks, 773
Asynchronous Advantage Actor-Critic (A3C), 662
asynchronous updates, 707
Atari preprocessing, 645
attention mechanisms
defined, 526
explainability and, 553
overview of, 549
Transformer architecture, 554
visual attention, 552
attributes, 8
autoencoders
convolutional, 579
denoising, 581
efficient data representations, 569
generative, 586
versus Generative Adversarial Networks (GANs), 568
overview of, 567
parts of, 569
PCA with undercomplete linear autoencoders, 570
probabilistic, 586
recurrent, 580
sparse, 582
stacked, 572-575
undercomplete, 570
unsupervised pretraining using stacked, 576-579
variational, 586-591
AutoGraphs, 407
automatic differentiation (autodiff), 290, 399, 765-772
AutoML, 323
autonomous driving systems, 497
autoregressive integrated moving average (ARIMA) models, 506
average absolute deviation, 41
average pooling layer, 459
Average Precision (AP), 491

B

backpropagation, 289-292
backpropagation through time (BPTT), 502
bag of words, 438
bagging and pasting
out-of-bag evaluation, 195
overview of, 192
in Scikit-Learn, 194
Bahdanau attention, 550
bandwidth saturation, 708
basic cells, 500
Batch Gradient Descent, 121
batch learning, 15
Batch Normalization (BN), 339
batch size, 325
batched action step, 657
batched time step, 657
batched trajectory, 657

Bayesian Gaussian Mixture models, 270
Bayesian inference, 586
Bayesian information criterion (BIC), 267
beam search, 547
beam width, 547
Bellman Optimality Equation, 627
Better Life Index, 19
bias neurons, 285
bias terms, 112
bias/variance trade-off, 134
bidirectional recurrent layers, 546
bidirectional RNNs, 546
binary classifiers, 88
binary trees, 177
biological neural networks (BNN), 282
biological neurons, 280
BIRCH algorithm, 259
black box models, 178
black box stochastic variational inference (BBSVI), 273
blenders, 208
Boltzmann machines, 775
boosting
  AdaBoost, 200
  Gradient Boosting, 203
  overview of, 199
bottleneck layers, 467
boundary transitions, 660
bounding box priors, 490
break the symmetry, 291
Byte-Pair Encoding, 536

C

calculus, 112
California Housing Prices dataset, 36
callbacks, 315
canary testing, 684
CART training algorithm, 177, 179
catastrophic forgetting, 637
categorical distribution, 261
categorical features
  encoding using embeddings, 433
  encoding using one-hot vectors, 431
causal models, 510
centroids, 238
chain rule, 290
chaining transformations, 415
character RNNs (Char-RNNs)
  building and training, 530
  chopping sequential datasets, 528
  generating Shakespearean text, 531
  overview of, 526
  splitting sequential datasets, 527
  stateful RNNs and, 532
  training dataset creation, 527
  using, 531
chatbots, 525
chi-squared test, 182
Classification and Regression Tree (CART), 177, 179
classification problems
  AdaBoost classifiers, 200
  binary classifiers, 88
  classification and localization, 483
  classification MLPs, 294
  error analysis, 102
  example of, 8
  Extra-Trees classifier, 198
  hard margin classification, 154
  image classifiers using Sequential APIs, 297-307
  large margin classification, 153
  linear SVM classification, 153
  MNIST dataset, 85
  multiclass classification, 100
  multilabel classification, 106
  multioutput classification, 107
  multitask classification, 311
  nonlinear SVM classification, 157-162
  performance measures, 88-100
  soft margin classification, 154
  voting classifiers, 189
closed-form solution, 114
cluster specification, 711
clustering algorithms
  additional algorithms, 258
  applications for, 10, 237
  DBSCAN, 255
  goal of, 236
  for image segmentation, 238, 249
  K-Means, 238-249
  overview of, 236
  for preprocessing, 251
  for semi-supervised learning, 253
code examples, obtaining and using, xxi
codings, 567
Colab Runtime, 693
Colaboratory (Colab), 693

collect policy, 649
color channels, 451
color segmentation, 249
column vectors, 113
comments and questions, xxiii, 718
complementary slackness, 762
components, 38
compression, 224
computation graphs, 376
Compute Unified Device Architecture library (CUDA), 690
concatenative attention, 550
concrete functions, 791
conditional probability, 547
confusion matrix, 90
connectionism, 280
constrained optimization, 166
Contrastive Divergence, 777
convergence, 118
convex function, 120
convolution kernels, 450
convolutional autoencoders, 579
convolutional layer
  filters, 450
  memory requirements, 456
  overview of, 448
  stacking multiple feature maps, 451
  TensorFlow implementation, 453
Convolutional Neural Networks (CNNs)
  architecture of visual cortex, 446
  classification and localization, 483
  CNN architectures, 460-478
  convolutional layer, 448-456
  object detection, 485-492
  overview of, 445
  pooling layer, 456
  pretrained models for transfer learning, 481
  pretrained models from Keras, 479
  ResNet-34 using Keras, 478
  semantic segmentation, 492
core instances, 255
corpus development, 24
correlation coefficient, 58
cost functions
  cross-entropy loss (log loss), 149
  hinge loss, 155, 173
  mean absolute error (MAE), 41, 293
  mean squared error, 120, 293, 308, 384, 570, 573, 583, 636
  role of, 20
credit assignment problem, 619
cross-entropy loss (log loss), 149, 295
cross-validation, 31, 73, 89
CUDA Deep Neural Network library (cuDNN), 690
curiosity-based exploration, 664
curse of dimensionality, 214
custom models
  about, 375
  activation functions, initializers, regularizers, and constraints, 387
  computing gradients using Autodiff, 399, 765-772
  layers, 391
  loss functions, 384
  losses and metrics, 397
  metrics, 388
  models, 394
  saving and loading, 385
  training loops, 402
customer segmentation, 237

D

data (see also data preparation; data visualization; training data)
  analyzing through clustering, 237
  California Housing Prices dataset, 36
  chopping sequential datasets, 528
  compressing, 224
  data mismatch, 32
  decompressing, 224
  downloading, 46
  efficient data representations, 569
  Fashion MNIST dataset, 297, 574, 590
  flat datasets, 529
  geographical data, 56
  Google News 7B corpus, 541
  helper function creation, 420
  importance of over algorithms, 24
  Internet Movie Database, 534
  iris dataset, 145
  loading and preprocessing with TensorFlow, 413-442
  MNIST dataset, 85
  nested datasets, 529
  noisy data, 19
  prefetching, 421
  preprocessing, 251, 419, 430-439

reconstruction error, 224
reducing dimensionality of, 222
shuffling, 416
skewed datasets, 89
sources for, 35
splitting sequential datasets, 527
training dataset creation, 527
training sparse models, 359
using datasets with tf.Keras, 423
Data API (TensorFlow)
chaining transformations, 415
helper function creation, 420
overview of, 414
prefetching data, 421
preprocessing data, 419
shuffling data, 416
using datasets with tf.keras, 423
data augmentation, 464
data parallelism, 701, 704
data preparation
benefits of functions for, 62
custom transformers, 68
data cleaning, 63
feature scaling, 69
handling text and categorical attributes, 65
transformation pipelines, 70
data snooping bias, 51
data visualization
attribute combinations, 61
computing correlations, 58
dimensionality reduction, 213
geographical data, 56
test, training, and exploration sets, 56
using TensorBoard for, 317
visualizing Fashion MNIST Dataset, 574
visualizing reconstructions, 574
datasets, defined, 414
DataViz (see data visualization)
DBSCAN (density-based spatial clustering of applications with noise), 255
decision boundaries, 145
decision function, 93
Decision Stumps, 203
Decision Trees
benefits of, 175
CART training algorithm, 179
computational complexity, 180
estimating class probabilities, 178
evaluating, 73
Gini impurity versus entropy, 180
instability drawbacks, 185
making predictions, 176
regression tasks, 183
regularization hyperparameters, 181
training and visualizing, 175
decoders, 501, 569
decompression, 224
deep autoencoders, 572
deep belief networks (DBNs), 13, 777
deep computer vision (see Convolutional Neural Networks (CNNs))
deep convolutional GANs, 598
Deep Learning VM Images, 692
deep neural networks (DNNs)
avoiding overfitting, 364-371
default configuration, 371
defined, xv, 289
faster optimizers, 351-364
overview of, 331
reusing pretrained layers, 345-351
vanishing/exploding gradients problems, 332-345
Deep Neuroevolution, 323
Deep Q-Learning
Double DQN, 640
Dueling DQN, 641
fixed Q-Value targets, 639
implementing, 634
overview of, 633
prioritized experience replay, 640
variants of, 639
deep Q-networks (DQNs), 633, 650, 650
denoising autoencoders, 581
dense layer, 285
dense vectors, 556
density estimation, 236, 264
depth concat layer, 467
depth radius, 466
depthwise separable convolution, 474
deques, 635
development sets (dev sets), 31
differencing, 506
dimensionality reduction
additional techniques, 232
approaches for, 215-218
using clustering, 237
curse of dimensionality, 214
goal of, 12

LLE (Locally Linear Embedding), 230
overview of, 213
PCA (Principal Component Analysis), 219-230
discount factors, 619
discriminators, 568
Distribution Strategies API, 668, 709
dot product, 551
Double DQN, 640
Double Dueling DQN, 642
DQN agents, 652
dropout, 365
dual numbers, 768
dual problem, 168, 761
duck typing, 68
Dueling DQN algorithm, 641
dummy attributes, 67
dying ReLUs problem, 335
dynamic models, 313
dynamic placer algorithm, 697
Dynamic Programming, 628

E

eager execution/eager mode, 408
early stopping, 141
Elastic Net, 140
ELU (exponential linear unit), 336-338
embedded devices, 685
Embedded Reber grammars, 566
embedding, 68, 413, 433
embedding matrix, 435
encoders, 501, 569
Encoder–Decoder model, 501, 542-548
end-of-sequence (EoS) token, 542, 556
energy function, 774
Ensemble Learning
bagging and pasting, 192-196
benefits of, 74
best uses of, 191
boosting, 199-208
defined, 189
examples of, 189
Random Forests, 189, 197
random patches and random subspaces, 196
stacking, 208
voting classifiers, 189
Ensemble methods, 189
ensembles, 189
entailment, 564
entropy impurity measure, 180
epochs, 125, 290
equalized learning rates, 603
equivariance, 458
error analysis, 102
estimators, 64
Euclidean norm, 41
event files, 317
evidence lower bound (ELBO), 272
example project
data downloading, 42-55, 756
data preparation, 62-72, 757
data visualization, 56-62, 756
framing the problem, 37, 755
launching, monitoring, and maintaining, 80, 760
Machine Learning project checklist, 37, 755
model fine-tuning, 75-80, 759
model selection and training, 72, 758
overview of, 35
project goals, 37
real-world data for, 35
selecting performance measure, 39
verifying assumptions, 42
Exclusive OR (XOR) classification problem, 288
exercise solutions, 719-753
expectation step, 262
Expectation-Maximization (EM) algorithm, 262
experience replay, 597
explainability, 553
explained variance ratio, 222
exploding gradients problem, 332
exploration policy, 630, 632
exploration sets, 56
exponential linear unit (ELU), 336-338
exponential scheduling, 360
Extra-Trees classifier, 198
Extremely Randomized Trees ensemble, 198

F

F1 score, 92
fake quantization, 687
false positive rate (FPR), 97
fan-in/fan-out numbers, 333
Fashion MNIST dataset, 297, 574, 590
Fast-MCD (minimum covariance determinant), 274

feature engineering, 27
feature extraction, 12, 27
feature maps, 228, 450
feature scaling, 69
feature selection, 27
feature space, 226
feature vector, 113
features, 8
feedforward neural networks (FNNs), 289
filters, 450
final trained models, 20
finite difference approximation, 766
First In, First Out (FIFO) queues, 383
first-order partial derivatives (Jacobians), 358
fitness functions, 20
fixed Q-Value targets, 639
flat datasets, 529
folds, 73, 89
forecasting, 503
forget gate, 516
forward pass, 290
forward-mode autodiff, 767
fraud detection, 237
Full Gradient Descent, 122
fully connected layer, 285
fully convolutional networks (FCNs), 487
fully-specified model architecture, 20
function definitions, 792
function graphs, 792
Functional API, 308-313

G

gate controllers, 516
Gated Recurrent Unit (GRU) cell, 518
Gaussian mixture model (GMM)
  additional algorithms for anomaly and novelty detection, 274
  anomaly detection using, 266
  Bayesian Gaussian Mixture models, 270
  graphical model of, 260
  overview of, 260
  selecting cluster number, 267
  variants, 260
Gaussian Radial Basis Function (RBF), 159
generalization error, 30
generalized Lagrangian, 762
Generative Adversarial Networks (GANs)
  versus autoencoders, 568
  deep convolutional GANs (DCGANs), 598
  difficulties of training, 596
  overview of, 592
  progressive growing of, 601
  StyleGANs, 604
  uses for, 567
generative autoencoders, 586
generative models, 263, 567, 775 (see also autencoders; Generative Adversarial Networks (GANs))
generative network, 569
generators, 568
genetic algorithms, 612
Gini impurity measure, 180
global average pooling layer, 460
global minimum, 119
Glorot and He initialization, 333
Google Cloud Platform (GCP)
  prediction service creation, 677-681
  prediction service use, 682-685
Google Cloud Storage (GCS), 679
Google News 7B corpus, 541
GoogLeNet, 466
GPUs (graphics processing units)
  adding to single machines, 689
  Colaboratory (Colab), 693
  GPU-equipped virtual machines, 692
  managing GPU RAM, 694
  parallel execution across multiple devices, 699
  placing operations and variables on devices, 697
  selecting, 690
  speeding computations with, 689
Gradient Boosted Regression Trees (GBRT), 203
Gradient Boosting, 203
gradient clipping, 345
Gradient Descent (GD)
  Batch Gradient Descent, 121
  Mini-batch Gradient Descent, 127
  overview of, 111, 118
  Stochastic Gradient Descent, 124
Gradient Tree Boosting, 203
graph mode, 408
greedy algorithms, 180
greedy layer-wise pretraining, 349
greedy layer-wise training, 578

H

hard clustering, 240
hard margin classification, 154
hard voting classifiers, 190
harmonic mean, 92
HDF5 format, 314
He initialization, 333
Heaviside step function, 285
Hebb's rule, 286
Hebbian learning, 286
helper functions, 420
hidden layers
  in MLPs, 289
  neurons per hidden layer, 324
  number of, 323
hidden units, 775
hierarchical clustering algorithms, 10
Hierarchical DBSCAN (HDBSCAN), 258
high-dimensional training sets, 213
hinge loss function, 155, 173
Hinton, Geoffrey, xv
histograms, 50
hold outs, 31
holdout validation, 31
Hopfield networks, 773
Huber loss, 293, 384
Hyperas, 322
Hyperband, 323
hyperbolic tangent function (tanh), 291
Hyperopt, 322
hyperparameters
  defined, 29
  fine-tuning for neural networks, 320-327
  hyperparameter tuning, 31, 75
  learning rate, 118
  Python libraries for optimization, 322
  regularization hyperparameters, 181
hyperplanes, 165
hypothesis boosting, 199

I

identity matrix, 137
image classification
  multitask classification, 311
  using Sequential API, 297-307
image generation, 495
image segmentation, 238, 249
importance sampling (IS), 640
impurity, 177, 180
imputation, 503
incremental learning, 16
Incremental PCA (IPCA), 225
independent and identically distributed (IID), 126
inequality constraints, 762
inertia, 243
inference, 23
information theory, 180
initialization
  centroid initialization methods, 243
  Glorot and He initialization, 333
  LeCun initialization, 334
  random initialization, 118
  Xavier initialization, 333
inliers, 266
input and output sequences, 501
input gate, 516
input layers, 289
input neurons, 285
input signatures, 791
instability, 185
instance segmentation, 249, 495
instance-based learning, 17, 22
inter-op thread pool, 699
intercept terms, 112
Internet Movie Database, 534
intra-op thread pool, 699
invariance, 457
inverse transformation, 225
iris dataset, 145
isolated environments, 43
Isolation Forest algorithm, 274
isomap algorithm, 233

J

JupyterLab, 692
just-in-time (JIT) compiler, 376

K

K-fold cross-validation, 73, 89
K-Means
  accelerated and mini-batch, 244
  centroid initialization methods, 243
  hard and soft clustering, 240
  image segmentation, 249
  K-Means algorithm, 241
  limits of, 248
  optimal cluster number, 245

overview of, 238
preprocessing with, 251
proposed improvement to, 243
scaling input features, 249
for semi-supervised learning, 253
k-Nearest Neighbors regression, 22
Karush–Kuhn–Tucker (KKT) multipliers, 762
keep probability, 367
Keras
benefits of, xvi
complex architectures, 314
gradient clipping in, 345
implementing Batch Normalization with, 341
implementing dropout using, 367
implementing MLPs with, 295-320
implementing ResNet-34 with, 478
keras.callbacks package, 316
loading datasets with, 297
low-level API, 381
multibackend Keras, 295
preprocessing layers, 437
saving and restoring models in, 314
stacked autoencoders using, 572
transfer learning with, 347
using code examples from keras.io, 300
using pretrained models from, 479
Keras Tuner, 322
Kernel PCA (kPCA), 226-230
kernel trick, 158, 228
kernelized SVM, 169
kernels, 170, 226, 377
kopt library, 322
Kullback–Leibler divergence, 150

L

label propagation, 254
labels, 8, 39, 239
Lagrange multipliers, 761
landmarks, 159
language models, 563 (see also natural language processing (NLP))
large margin classification, 153
Lasso Regression, 137
latent loss, 587
latent representations, 567
latent variables, 262
law of large numbers, 191
Layer Normalization, 512
layers
1D convolutional layer, 520
adaptive instance normalization (AdaIN), 604
bidirectional recurrent layer, 546
convolutional layer, 448-456
dense (fully connected) layer, 285
hidden layer, 289
input layer, 289
Masked Multi-Head Attention layer, 556
minibatch standard deviation layer, 603
Multi-Head Attention layer, 556, 559
output layer, 289
pooling layer, 456
recurrent, 498-502
reusing pretrained, 345-351
Scaled Dot-Product Attention layer, 559
leaf nodes, 176
leaky ReLU function, 335
learning curves, 130-134
learning rate, 16, 118, 325, 603
learning rate scheduling, 359
learning schedules, 125, 360
LeCun initialization, 334
LeNet-5, 463
Levenshtein distance, 161
liblinear library, 162
libsvm library, 162
likelihood function, 267
linear algebra, 112
linear autoencoders, 570
Linear Discriminant Analysis (LDA), 233
linear models, 19
Linear Regression model
approaches to training, 111, 113
computational complexity, 117
Normal Equation, 114
overview of, 112
linear SVM classification, 153
lists of lists, using SequenceExample Protobuf, 429
LLE (Locally Linear Embedding), 230
Lloyd-Forgy algorithm, 238
local minimum, 119
Local Outlier Factor (LOF), 274
local response normalization, 465
localization, 483
log loss, 144
log-odds, 144

logical computations, 283
logical GPU devices, 695
Logistic (sigmoid) function, 143, 293-294, 302, 332
Logistic Regression
  classification with, 8
  decision boundaries, 145
  estimating probabilities, 143
  overview of, 142
  Softmax Regression, 148
  training and cost function, 144
logit, 144
Logit Regression (see Logistic Regression)
long sequences
  overview of, 511
  short-term memory problems, 514-523
  unstable gradients problem, 512
Long Short-Term Memory (LSTM) cell, 514
loss functions (see cost functions)
Luong attention, 551

M

Machine Learning (ML)
  additional resources, xix
  applications for, xv, 5
  approach to learning, xvi
  benefits of, 2
  challenges of, 23-30
  defined, 1
  history of, xv
  locating papers on, 378
  notations for, 40, 164
  overview of, 30
  prerequisites to learning, xvii
  testing and validating, 30-33
  topics covered, xvii
  types of, 7-23
Machine Learning project checklist, 37, 755
majority-vote classifiers, 190
majority-vote predictions, 187
Manhattan norm, 41
manifold assumption, 218
manifold hypothesis, 218
Manifold Learning, 218
manual differentiation, 765
margin violations, 155
Markov chains, 625
Markov Decision Processes (MDP), 625-629
Mask R-CNN, 495
mask tensors, 539
masked language model (MLM), 564
Masked Multi-Head Attention layer, 556
masking, 538
max pooling layer, 457
max-norm regularization, 370
maximization step, 262
maximum a-posteriori (MAP) estimation, 269
maximum likelihood estimate (MLE), 269
mean absolute error (MAE), 41
mean Average Precision (mAP), 491
mean coding, 586
mean field variational inference, 273
Mean-Shift algorithm, 259
measure of similarity, 18
memory bandwidth, 422
memory cells, 500
Mercer's conditions, 171
Mercer's theorem, 171
meta learners, 208
metagraphs, 671
metrics
  accuracy, 388
  area under the curve (AUC), 98
  confusion matrix, 90, 90
  F1 score, 92
  mean absolute error (MAE), 41, 293
  mean average precision, 491
  mean squared error, 183, 505
  precision, 91-97
  recall, 91-97
  RMSE, 39
  ROC curve, 97
Microsoft Cognitive Toolkit (CNTK), 295
min-max scaling, 69
Mini-batch Gradient Descent, 127
mini-batch K-Means, 244
mini-batches, 15, 127
minibatch discrimination, 597
minibatch standard deviation layer, 603
mirrored strategy, 704
mixing regularization, 606
ML Engine, 680
MNIST dataset, 85
mobile devices, 685
mode collapse, 597
model parallelism, 701
model parameters, 20
model selection, 19, 31, 72

model-based learning, 18
models (see also custom models)
causal models, 510
complex using Functional API, 308-313
custom with TensorFlow, 384-405
defined, 20
dynamic using Subclassing API, 313
fine-tuning, 75-80
parametric versus nonparametric, 181
pretrained models for transfer learning, 481
pretrained models from Keras, 479
saving and restoring, 314
sequence-to-sequence models, 510
training, 20, 72 (see also training models)
training across multiple devices, 701-717
training sparse models, 359
using callbacks, 315
using TensorBoard for visualization, 317
white versus black box, 178
modules, 540
momentum optimization, 351
momentum vector, 352
Monte Carlo (MC) dropout, 368
Multi-Head Attention layer, 556, 559
multibackend Keras, 295
multiclass classification, 100
Multidimensional Scaling (MDS), 232
multilabel classification, 106
Multilayer Perceptrons (MLPs)
backpropagation and, 289-292
classification MLPs, 294
regression MLPs, 292
multinomial classifiers, 100
Multinomial Logistic Regression, 148
multioutput classification, 107
multiple outputs, 311
multiple regression problems, 39
multiplicative attention, 551
multitask classification, 311
multivariate regression problems, 39
multivariate time series, 503

N

naive forecasting, 505
Nash equilibrium, 596
natural language processing (NLP)
attention mechanisms, 549-563
CNNs for, 445
Encoder–Decoder network for, 542-548
generating text using character RNNs, 526-534
overview of, 525
recent innovations in, 563
RNNS for, 497
sentiment analysis, 534-542
uses for, 351
nested datasets, 529
Nesterov Accelerated Gradient (NAG), 353
Nesterov momentum optimization, 353
neural machine translation (NMT), 542-563
(see also natural language processing (NLP))
neurons
bias neurons, 285
fan-in/fan-out numbers, 333
from biological to artificial, 280-295
input neurons, 285
logical computations with, 283
per hidden layer, 324
recurrent neurons, 498-502
stochastic neurons, 775
Newton's difference quotient, 766
next sentence prediction (NSP), 565
No Free Lunch (NFL) theorem, 33
noisy data, 19
non-max suppression, 486
nonlinear dimensionality reduction (NLDR), 230
nonlinear SVM classification, 157-162
nonparametric models, 181
nonsaturating activation functions, 335
nonsequential neural networks, 308
Normal Equation, 114
normalization, 69, 339, 603
normalized exponential, 148
novelty detection, 12, 267, 274
NP-Complete problem, 180
null hypothesis, 182
NumPy
array_split() function, 226
dense arrays, 67
installing, 42
inv() function, 115
memmap class, 226
randint() function, 107
serializing large arrays, 75
svd() function, 221
using TensorFlow like, 379-384

NVIDIA Collective Communications Library (NCCL), 710
Nvidia GPU cards, 690

O

object detection
  fully convolutional networks (FCNs), 487
  overview of, 485
  You Only Look Once (YOLO), 489
objectness output, 486
observed variables, 262
observers, 654
off-policy algorithms, 632
offline learning, 15
on-policy algorithms, 632
one-class SVM algorithm, 275
one-hot encoding, 67
one-hot vectors, 431
one-versus-all (OvA) strategy, 100
one-versus-one (OvO) strategy, 100
one-versus-the-rest (OvR) strategy, 100
online learning, 15, 88
online model, 639
online SVMs, 172
OpenAI Gym, 613-617
Optical Character Recognition (OCR), 1
optimal state value, 627
optimizers
  AdaGrad, 354
  Adam and Nadam optimization, 356
  creating faster, 351
  first- and second-order partial derivatives, 358
  learning rate scheduling, 359
  momentum optimization, 351
  Nesterov Accelerated Gradient (NAG), 353
  RMSProp, 355
  Stochastic Gradient Descent (SGD), 88, 124
original space, 226
out-of-core learning, 16
out-of-sample error, 30
out-of-vocabulary (oov) buckets, 432
outlier detection, 237, 266
output gate, 516
output layers, 289
overcomplete autoencoders, 580, 580
overfitting
  avoiding through regularization, 364-371
  defined, 27
  limiting risk of, 457

P

p (posterior) distribution, 272
p (prior) distribution, 271
p-value, 182
parameter efficiency, 323
parameter matrix, 148
parameter servers, 705
parameter space, 121
parameter vector, 113
parametric leaky ReLU (PReLU), 335
parametric models, 181
partial derivatives, 121
pasting (see bagging and pasting)
pattern matching, 569
PCA (Principal Component Analysis)
  anomaly and novelty detection using, 274
  choosing dimension number, 223
  for compression, 224
  explained variance ratio, 222
  incremental, 225
  Kernel PCA (kPCA), 226-230
  overview of, 219
  preserving variance, 219
  principal component axis, 220
  projecting down to d dimensions, 221
  randomized, 225
  using Scikit-Learn, 222
  undercomplete linear autoencoders for, 570
Pearson's r, 58
peephole connections, 518
penalties, 14
Perceptron, 284-288
Perceptron convergence theorem, 287
performance measures (see metrics)
performance scheduling, 361
piecewise constant scheduling, 361
pipelines, 38, 424
pixelwise normalization layers, 603
policies, 14, 612
policy gradients (PG), 613, 620-625
policy parameters, 612
policy search, 612
policy space, 612
polynomial features, 158
polynomial kernels, 170
Polynomial Regression, 112, 128
pooling kernel, 457

pooling layer, 456
positional embeddings, 556
post-training quantization, 686
power scheduling, 360
pre-images, 228
precision, 91-97
prediction problems, 8, 17, 189
prediction service
  creating on GCP AI, 677-681
  using, 682-685
predictors, 65
preprocessing, 251, 430-439
pretraining
  for transfer learning, 481
  greedy layer-wise pretraining, 349
  models from Keras, 479
  on auxiliary tasks, 350
  reusing pretrained embeddings, 540
  reusing pretrained layers, 345-351
  unsupervised pretraining, 349
  using stacked autoencoders, 576-579
primal problem, 168
prioritized experience replay (PER), 640
probabilistic autoencoders, 586
probability density function (PDF), 236, 264
projection, 215
propositional logic, 280
protocol buffers (protobufs), 425
Proximal Policy Optimization (PPO), 663
pruning, 182
PyTorch library, 296

Q

Q-Learning
  Approximate Q-Learning and Deep Q-Learning, 633
  exploration policy, 632
  implementing, 631
  overview of, 630
Q-Value Iteration, 628
Q-Values, 628
Quadratic Programming (QP) problems, 167
quantization-aware training, 687
queries per second (QPS), 667
questions and comments, xxiii, 718
queues, 383, 788

R

Radial Basis Function (RBF), 159
ragged tensors, 383, 784
Rainbow agent, 642
Random Forests
  benefits of, 189
  Extra-Trees, 198
  feature importance, 198
  overview of, 197
random initialization, 118
random patches and random subspaces, 196
random projections, 232
randomized leaky ReLU (RReLU), 335
Randomized PCA, 225
recall, 91-97
receiver operating characteristic (ROC) curve, 97
recognition network, 569
recommender systems, 237
reconstruction error, 224
reconstruction loss, 397, 570
reconstruction pre-images, 228
reconstructions, 570
Rectified Linear Unit function (ReLU), 292-293
recurrent autoencoders, 580
recurrent neural networks (RNNs)
  bidirectional RNNs, 546
  forecasting time series, 503-511
  generating text using character RNNS, 526-534
  handling long sequences, 511-523
  overview of, 497
  recurrent neurons and layers, 498-502
  stateless and stateful, 525, 532
  training, 502
recurrent neurons, 498
Region Proposal Network (RPN), 492
regression problems
  Decision Trees, 183
  defined, 8
  k-Nearest Neighbors regression, 22
  Lasso Regression, 137
  Linear Regression, 112-117
  Logistic Regression, 142-151
  multiple regression problems, 39
  multivariate regression problems, 39
  Polynomial Regression, 128
  regression MLPs, 292
  regression MLPs using Sequential API, 307
  Ridge Regression, 135
  Softmax Regression, 148-151

SVM regression, 162
univariate regression problems, 39
regular expressions, 536
regularization
avoiding overfitting through, 364-371
defined, 28
hyperparameters for Decision Trees, 181
multiple outputs for, 311
shrinkage technique, 205
regularization terms, 135
regularized linear models
Elastic Net, 140
Lasso Regression, 137
overview of, 134
Ridge Regression, 135
REINFORCE algorithms, 620
Reinforcement Learning (RL)
algorithms for, 662
Deep Q-Learning, 633-638
evaluating actions, 619
Markov Decision Processes (MDP), 625-629
neural network policies, 617
OpenAI Gym, 613-617
optimizing rewards, 610
overview of, 14, 609
policy gradients, 620-625
policy search, 612
Q-Learning, 630-634
Temporal Difference Learning, 629
TF-Agents library, 642-662
ReLU (Rectified Linear Unit function), 292-293
replay buffers, 635, 649, 654
replay memory, 635
representation learning, 68, 434 (see also autoencoders)
residual blocks, 395
residual errors, 203
residual learning, 471
residual units, 471
ResNet (Residual Network), 471
ResNet-34 CNN, 478
responsibilities (clustering), 262
restoring models, 314
restricted Boltzmann machines (RBMs), 13, 349, 776
reverse-mode autodiff, 290, 770
rewards, 14
Ridge Regression, 135
RMSProp, 355
Root Mean Square Error (RMSE), 39, 120
root nodes, 176

S

SAMME (Stagewise Additive Modeling using a Multiclass Exponential loss function), 203
sample inefficiency, 625
sampled softmax technique, 544
sampling bias, 25
sampling noise, 25
SavedModel format, 669
saving and restoring models, 314
Scaled Dot-Product Attention layer, 559
Scaled Exponential Linear Unit (SELU) function, 334, 337-338, 368
Scikit-Learn
AdaBoost version used in, 203
anomaly and novelty detection, 274
automatic reconstruction with, 229
bagging and pasting in, 194
benefits of, xvi
CART training algorithm, 177, 179
clustering algorithms in, 258
computing classifier metrics, 92-107
converting text to numbers, 66
cross_val_score() function, 89
data centering in, 221
dataset dictionary structure, 85
DecisionTreeRegressor class, 183
design principles, 64
dimensionality reduction in, 232
ExtraTreesClassifier class, 198
feature importance scoring, 198
feature scaling, 154
full SVD approach, 225
GBRT ensemble training in, 204
GridSearchCV, 76
incremental training in, 207
IncrementalPCA class, 226
installing, 42
K-fold cross-validation feature, 73
KernelPCA class, 227
launching, monitoring, and maintaining your system, 80
linear model using, 21
linear regression using, 116
LLE (Locally Linear Embedding), 230, 232
max_depth hyperparameter, 181
mean_squared_error function, 72

missing value handling, 63
one-hot vectors, 67
out-of-bag evaluation, 195
PCA using, 222
Perceptron class, 287
presorting data with, 180
Randomized PCA algorithm, 225
random_state hyperparameter, 185
saving models, 75
SGDClassifier class, 88
splitting datasets into subsets, 53
stratified sampling using, 54
SVM classification classes, 162
SVM models, 155
tolerance hyperparameter, 162
transformation sequences, 70
transformers and, 68
voting classifiers in, 191
Scikit-Optimize, 322
SE block, 476
SE-Inception, 476
SE-ResNet, 476
search engines, 238
second-order partial derivatives (Hessians), 358
self-attention mechanism, 556
self-normalization, 337
self-organizing maps (SOMs), 780
self-supervised learning, 351
SELU (Scaled Exponential Linear Unit) function (see Scaled Exponential Linear Unit (SELU) function)
semantic interpolation, 590
semantic segmentation, 249, 458, 492
semi-supervised learning
clustering algorithms for, 237, 253
defined, 13
examples of, 13
SENet (Squeeze-and-Excitation Network), 476
sensitivity, 91
sentence encoders, 541
sentiment analysis
defined, 526
masking, 538
overview of, 534
reusing pretrained embeddings, 540
separable convolution, 474
sequence-to-sequence models, 510
sequence-to-vector networks, 501
SequenceExample protobuf (TensorFlow), 429
sequences
forecasting time series, 503-511
handling long, 511-523
input and output, 501
RNNS for, 497
Sequential API
image classifiers using, 297-307
regression MLP using, 307
service account, 682
sets, 383, 787
Shannon's information theory, 180
short-term memory problems, 514-523
shortcut connections, 471
shrinkage, 205
shuffling-buffer approach, 417
sigmoid (Logistic) activation function, 143, 293-294, 302, 332
sigmoid kernel, 171
silhouette coefficient, 246
silhouette diagram, 247
silhouette score, 246
similarity functions, 159
simulated annealing, 125
simulated environments, 614
single-shot learning, 495
Singular Value Decomposition (SVD), 117, 221
skewed datasets, 89
skip connections, 337, 471
Sklearn-Deap, 323
slack variables, 167
smoothing term, 340
Soft Actor-Critic algorithm, 663
soft clustering, 240
soft margin classification, 154
soft voting, 192
softmax function, 148, 294, 299, 470, 482, 488, 543
Softmax Regression, 148
softplus activation function, 293
spam filters, 1, 2
spare replicas, 706
sparse autoencoders, 582
sparse matrix, 67
sparse models, 359
sparse tensors, 383, 785
sparsity, 582
sparsity loss, 583
Spearmint library, 322
spectral clustering, 259

spurious patterns, 774
stacked autoencoders
overview of, 572
stacked denoising autoencoders, 581
unsupervised pretraining using, 576-579
using Keras, 572
visualizing Fashion MNIST Dataset, 574
visualizing reconstructions, 574
stacked denoising autoencoders, 581
stacked generalization, 208
stacking, 208
stale gradients, 707
standard correlation coefficient, 58
standardization, 69
start of sequence (SoS) token, 535
state-action values, 628
stateful metrics, 389
stationary point, 761
statistical mode, 193
statistical significance, 182
step function, 284
Stochastic Gradient Boosting, 207
Stochastic Gradient Descent (SGD), 88, 124
stochastic neurons, 775
stochastic policy, 612
stratified sampling, 53
streaming metrics, 389
stride, 449
string kernels, 161
string subsequence kernel, 161
string tensors, 383, 783
strong learners, 190
style mixing, 606
style transfer, 604
StyleGANs, 567, 604
Subclassing API, 313
subderivatives, 173
subgradient vector, 140
subsampling, 456
subspace, 215
summaries (TensorFlow), 317
supervised learning
algorithms covered, 9
common tasks, 8
defined, 8
Support Vector Machines (SVMs)
benefits of, 153
decision function and prediction, 165
dual problem, 168, 761
kernelized SVM, 169
linear SVM classification, 153
nonlinear SVM classification, 157-162
online SVMs, 172
SVM regression, 162
training objective, 166
support vectors, 154
symbolic differentiation, 768
symbolic tensors, 408, 792
symmetry, breaking in backpropagation, 291
synchronous updates, 706

T

t-Distributed Stochastic Neighbor Embedding (t-SNE), 233
tail-heavy histograms, 51
Talos library, 322
target model, 639
TD error, 630
TD target, 630
temperature
in Boltzmann machines, 775
in text generation, 531
Temporal Difference Learning (TD Learning), 629
tensor arrays, 383, 786
TensorBoard, 317
TensorFlow Addons, 545
TensorFlow cluster, 711
TensorFlow Extended (TFX), 440
TensorFlow Hub, 378, 540
TensorFlow Lite, 378
TensorFlow Model Optimization Toolkit (TF-MOT), 359
TensorFlow Playground, 295
TensorFlow, basics of
architecture, 377
benefits, xvi, 376
community support, 379
features, 376
getting help, 379
installing, 296
library ecosystem, 378
operating system compatibility, 378
PyTorch library and, 296
versions covered, 375
TensorFlow, CNNs
convolution operations, 494
convolutional layers, 453

pooling layer, 458
TensorFlow, custom models and training
about, 375
activation functions, initializers, regularizers, and constraints, 387
computing gradients using Autodiff, 399, 765-772
implementing learning rate scheduling, 363
layers, 391
loss functions, 384
losses and metrics, 397
metrics, 388
models, 394
saving and loading, 385
special data structures, 783-789
training loops, 402
TensorFlow, data loading and preprocessing
Data API, 414-424
overview of, 413
preprocessing input features, 430-439
TensorFlow Datasets (TFDS) Project, 441, 441
TF Transform, 439
TFRecord format, 424-430
TensorFlow, functions and graphs
AutoGraph and tracing, 407, 791-799
overview of, 405
TF Function rules, 409
TensorFlow, model deployment at scale
deploying on AI platforms, 81
deploying to mobile and embedded devices, 685-688
overview of, 667
serving TensorFlow models, 668-685
training models across multiple devices, 701-717
using GPUs to speed computations, 689-701
TensorFlow, NumPy-like operations
other data structures, 383
tensors and NumPy, 381
tensors and operations, 379
type conversions, 381
variables, 382
TensorFlow.js, 378
tensors, 379
Term-Frequency × Inverse-Document-Frequency (TF-IDF), 439
terminal state, 626
test sets, 30, 51
testing and validation
data mismatch, 32
hyperparameter tuning, 31
model selection, 31
text generation
building and training models for, 530
chopping sequential datasets, 528
generating Shakespearean text, 531
overview of, 526
splitting sequential datasets, 527
stateful RNNs and, 532
training dataset creation, 527
using models for, 531
TF Datasets (TFDS), 414, 441
TF Functions
graphs generated by, 791-799
rules, 409
TF Transform (tf.Transform), 414, 439
TF-Agents library
collect driver, 656
datasets, 658
deep Q-networks (DQNs), 650
DQN agents, 652
environment specifications, 644
environment wrappers, 645
environments, 643
installing, 643
overview of, 642
replay buffer and observer, 654
training architecture, 649
training loops, 661
training metrics, 655
tf.keras, 295, 363, 363, 423
tf.summary package, 319
TF.Text library, 536
TFRecord format
compressed TFRecord files, 425
lists of lists using SequenceExample Protobuf, 429
loading and parsing examples, 428
overview of, 424
protocol buffers (protobufs), 425
TensorFlow protobufs, 427
Theano, 295
theoretical information criterion, 267
thermal equilibrium, 775
threshold logic unit (TLU), 284
Tikhonov regularization, 135
time series data

additional models for, 506
baseline metrics, 505
deep RNNS, 506
forecasting several steps ahead, 508
overview of, 503
RNNS for, 497
simple RNNs, 505
time step, 498
tokenization, 536
tolerance, 123
TPUs (tensor processing units), 377
train-dev sets, 32
training data
defined, 2
hold outs, 31
insufficient quantity of, 23
irrelevant features, 27
nonrepresentative, 25
overfitting, 27
poor quality, 26
training dataset creation, 527
underfitting, 29
training instances, 2, 215
training models
defined, 20
example project, 72
Gradient Descent, 118-128
learning curves, 130-134
Linear Regression, 112-117
Logistic Regression, 142-151
overview of, 111
Polynomial Regression, 128-130
regularized linear models, 134-142
training samples, 2
training set rotation, 185
training sets, 2, 30, 213
training/serving skew, 440
trajectories, 649
trajectory, 650
transfer learning, 324, 345, 481
transformations
affine transformations, 604
chaining, 415
custom, 68
inverse transformation, 225
purpose of, 64
transformation pipelines, 70
Transformer architecture, 554
transposed convolutional layer, 493
true negative rate (TNR), 97
true positive rate (TPR), 91
truncated backpropagation through time, 529
Turing test, 525
tying weights, 577
type conversions, 381

U

uncertainty sampling, 255
undercomplete autoencoders, 570
underfitting, 29
undiscounted rewards, 656
univariate regression problems, 39
univariate time series, 503
unrolling the network through time, 498
unstable gradients problem, 512
unsupervised learning
algorithms covered, 10
clustering, 236-260
common tasks, 10
defined, 9
Gaussian mixtures model (GMM), 260-275
overview of, 235
pretraining using stacked autoencoders, 576-579
unsupervised pretraining, 349
upsampling layer, 493
utility functions, 20

V

validation sets, 31
Value Iteration algorithm, 627
vanishing/exploding gradients problems, 332-345
variables, 382
variance
explained variance ratio, 222
preserving, 219
variational autoencoders, 586-591
variational inference, 272
variational parameters, 272
vector-to-sequence networks, 501
vectors
column vectors, 113
feature vectors, 113
momentum vector, 352
parameter vectors, 113
subgradient vectors, 140
VGGNet, 470

virtual GPU devices, 695
visible units, 775
visual attention, 552
visualization algorithms, 11
vocabulary, 432
voice recognition, 445

W

wall time, 341
warmup phase, 708
WaveNet, 498, 521
weak learners, 190
weighted moving average model, 506
white box models, 178
Wide & Deep neural networks, 308
wisdom of the crowd, 189
word embeddings, 434
word tokenization, 536
WordTrees, 490
workspace creation, 42

X

Xavier initialization, 333
Xception (Extreme Inception), 474
XGBoost, 208

Y

You Only Look Once (YOLO), 489

Z

zero padding, 449
zero-shot learning (ZSL), 564
ZF Net, 466

About the Author

Aurélien Géron is a Machine Learning consultant and lecturer. A former Googler, he led YouTube's video classification team from 2013 to 2016. He's been a founder of and CTO at a few different companies: Wifirst, a leading wireless ISP in France; Polyconseil, a consulting firm focused on telecoms, media, and strategy; and Kiwisoft, a consulting firm focused on Machine Learning and data privacy.

Before all that he worked as an engineer in a variety of domains: finance (JP Morgan and Société Générale), defense (Canada's DOD), and healthcare (blood transfusion). He also published a few technical books (on C++, WiFi, and internet architectures) and lectured about computer science at a French engineering school.

A few fun facts: he taught his three children to count in binary with their fingers (up to 1,023), he studied microbiology and evolutionary genetics before going into software engineering, and his parachute didn't open on the second jump.

Colophon

The animal on the cover of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* is the fire salamander (*Salamandra salamandra*), an amphibian found across most of Europe. Its black, glossy skin features large yellow spots on the head and back, signaling the presence of alkaloid toxins. This is a possible source of this amphibian's common name: contact with these toxins (which they can also spray short distances) causes convulsions and hyperventilation. Either the painful poisons or the moistness of the salamander's skin (or both) led to a misguided belief that these creatures not only could survive being placed in fire but could extinguish it as well.

Fire salamanders live in shaded forests, hiding in moist crevices and under logs near the pools or other freshwater bodies that facilitate their breeding. Though they spend most of their lives on land, they give birth to their young in water. They subsist mostly on a diet of insects, spiders, slugs, and worms. Fire salamanders can grow up to a foot in length, and in captivity may live as long as 50 years.

The fire salamander's numbers have been reduced by destruction of their forest habitat and capture for the pet trade, but the greatest threat they face is the susceptibility of their moisture-permeable skin to pollutants and microbes. Since 2014, they have become extinct in parts of the Netherlands and Belgium due to an introduced fungus.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. The cover illustration is by Karen Montgomery, based on an engraving from *Wood's Illustrated Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning

©2019 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly Media, Inc. | 175

AUTOMATE THE BORING STUFF WITH PYTHON

PRACTICAL PROGRAMMING FOR TOTAL BEGINNERS

AL SWEIGART

no starch press

www.allitebooks.com

AUTOMATE THE BORING STUFF WITH PYTHON

www.allitebooks.com

AUTOMATE THE BORING STUFF WITH PYTHON

Practical Programming for Total Beginners

by Al Sweigart

no starch press

San Francisco

www.allitebooks.com

AUTOMATE THE BORING STUFF WITH PYTHON. Copyright © 2015 by Al Sweigart.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Printed in USA

Second printing

19 18 17 16 15 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-599-4
ISBN-13: 978-1-59327-599-0

Publisher: William Pollock
Production Editor: Laurel Chun
Cover Illustration: Josh Ellingson
Interior Design: Octopod Studios
Developmental Editors: Jennifer Griffith-Delgado, Greg Poulos, and Leslie Shen
Technical Reviewer: Ari Lacenski
Copyeditor: Kim Wimpsett
Compositor: Susan Glinert Stevens
Proofreader: Lisa Devoto Farrell
Indexer: BIM Indexing and Proofreading Services

For information on distribution, translations, or bulk sales,
please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 415.863.9900; info@nostarch.com
www.nostarch.com

Library of Congress Control Number: 2014953114

SUSTAINABLE FORESTRY INITIATIVE Certified Sourcing www.sfiprogram.org SFI-00000

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/us/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

www.allitebooks.com

For my nephew Jack

www.allitebooks.com

About the Author

Al Sweigart is a software developer and tech book author living in San Francisco. Python is his favorite programming language, and he is the developer of several open source modules for it. His other books are freely available under a Creative Commons license on his website *http://www.inventwithpython.com/*. His cat weighs 14 pounds.

About the Tech Reviewer

Ari Lacenski is a developer of Android applications and Python software. She lives in San Francisco, where she writes about Android programming at *http://gradlewhy.ghost.io/* and mentors with Women Who Code. She's also a folk guitarist.

BRIEF CONTENTS

Acknowledgments xxiii

Introduction 1

PART I: PYTHON PROGRAMMING BASICS 11

Chapter 1: Python Basics 13

Chapter 2: Flow Control 31

Chapter 3: Functions 61

Chapter 4: Lists 79

Chapter 5: Dictionaries and Structuring Data 105

Chapter 6: Manipulating Strings 123

PART II: AUTOMATING TASKS 145

Chapter 7: Pattern Matching with Regular Expressions 147

Chapter 8: Reading and Writing Files 173

Chapter 9: Organizing Files 197

Chapter 10: Debugging 215

Chapter 11: Web Scraping 233

Chapter 12: Working with Excel Spreadsheets 265

Chapter 13: Working with PDF and Word Documents 295

Chapter 14: Working with CSV Files and JSON Data 319

Chapter 15: Keeping Time, Scheduling Tasks, and Launching Programs 335

Chapter 16: Sending Email and Text Messages 361

Chapter 17: Manipulating Images 387

Chapter 18: Controlling the Keyboard and Mouse with GUI Automation 413

Appendix A: Installing Third-Party Modules 441

Appendix B: Running Programs 443

Appendix C: Answers to the Practice Questions 447

Index 461

CONTENTS IN DETAIL

ACKNOWLEDGMENTS **xxiii**

INTRODUCTION **1**

Whom Is This Book For? . . . 2
Conventions . . . 2
What Is Programming? . . . 3
    What Is Python? . . . 4
    Programmers Don't Need to Know Much Math . . . 4
    Programming Is a Creative Activity . . . 5
About This Book . . . 5
Downloading and Installing Python . . . 6
Starting IDLE . . . 7
    The Interactive Shell . . . 8
How to Find Help . . . 8
Asking Smart Programming Questions . . . 9
Summary . . . 10

PART I: PYTHON PROGRAMMING BASICS **11**

1
PYTHON BASICS **13**

Entering Expressions into the Interactive Shell . . . 14
The Integer, Floating-Point, and String Data Types . . . 16
String Concatenation and Replication . . . 17
Storing Values in Variables . . . 18
    Assignment Statements . . . 18
    Variable Names . . . 20
Your First Program . . . 21
Dissecting Your Program . . . 22
    Comments . . . 23
    The print() Function . . . 23
    The input() Function . . . 23
    Printing the User's Name . . . 24
    The len() Function . . . 24
    The str(), int(), and float() Functions . . . 25
Summary . . . 28
Practice Questions . . . 28

2
FLOW CONTROL **31**

Boolean Values . . . 32
Comparison Operators . . . 33
Boolean Operators . . . 35

Binary Boolean Operators . . . 35
The not Operator . . . 36
Mixing Boolean and Comparison Operators . . . 36
Elements of Flow Control . . . 37
Conditions . . . 37
Blocks of Code . . . 37
Program Execution . . . 38
Flow Control Statements . . . 38
if Statements . . . 38
else Statements . . . 39
elif Statements . . . 40
while Loop Statements . . . 45
break Statements . . . 49
continue Statements . . . 50
for Loops and the range() Function . . . 53
Importing Modules . . . 57
from import Statements . . . 58
Ending a Program Early with sys.exit() . . . 58
Summary . . . 58
Practice Questions . . . 59

3
FUNCTIONS **61**

def Statements with Parameters . . . 63
Return Values and return Statements . . . 63
The None Value . . . 65
Keyword Arguments and print() . . . 65
Local and Global Scope . . . 67
Local Variables Cannot Be Used in the Global Scope . . . 67
Local Scopes Cannot Use Variables in Other Local Scopes . . . 68
Global Variables Can Be Read from a Local Scope . . . 69
Local and Global Variables with the Same Name . . . 69
The global Statement . . . 70
Exception Handling . . . 72
A Short Program: Guess the Number . . . 74
Summary . . . 76
Practice Questions . . . 76
Practice Projects . . . 77
The Collatz Sequence . . . 77
Input Validation . . . 77

4
LISTS **79**

The List Data Type . . . 80
Getting Individual Values in a List with Indexes . . . 80
Negative Indexes . . . 82
Getting Sublists with Slices . . . 82
Getting a List's Length with len() . . . 83
Changing Values in a List with Indexes . . . 83

List Concatenation and List Replication . . . 83
Removing Values from Lists with del Statements . . . 84
Working with Lists . . . 84
Using for Loops with Lists . . . 86
The in and not in Operators . . . 87
The Multiple Assignment Trick . . . 87
Augmented Assignment Operators . . . 88
Methods . . . 89
Finding a Value in a List with the index() Method . . . 89
Adding Values to Lists with the append() and insert() Methods . . . 89
Removing Values from Lists with remove() . . . 90
Sorting the Values in a List with the sort() Method . . . 91
Example Program: Magic 8 Ball with a List . . . 92
List-like Types: Strings and Tuples . . . 93
Mutable and Immutable Data Types . . . 94
The Tuple Data Type . . . 96
Converting Types with the list() and tuple() Functions . . . 97
References . . . 97
Passing References . . . 100
The copy Module's copy() and deepcopy() Functions . . . 100
Summary . . . 101
Practice Questions . . . 102
Practice Projects . . . 102
Comma Code . . . 102
Character Picture Grid . . . 103

5
DICTIONARIES AND STRUCTURING DATA 105

The Dictionary Data Type . . . 105
Dictionaries vs. Lists . . . 106
The keys(), values(), and items() Methods . . . 107
Checking Whether a Key or Value Exists in a Dictionary . . . 109
The get() Method . . . 109
The setdefault() Method . . . 110
Pretty Printing . . . 111
Using Data Structures to Model Real-World Things . . . 112
A Tic-Tac-Toe Board . . . 113
Nested Dictionaries and Lists . . . 117
Summary . . . 119
Practice Questions . . . 119
Practice Projects . . . 120
Fantasy Game Inventory . . . 120
List to Dictionary Function for Fantasy Game Inventory . . . 120

6
MANIPULATING STRINGS 123

Working with Strings . . . 123
String Literals . . . 124
Indexing and Slicing Strings . . . 126
The in and not in Operators with Strings . . . 127

Useful String Methods . . . 127
The upper(), lower(), isupper(), and islower() String Methods . . . 128
The isX String Methods . . . 129
The startswith() and endswith() String Methods . . . 131
The join() and split() String Methods . . . 131
Justifying Text with rjust(), ljust(), and center() . . . 133
Removing Whitespace with strip(), rstrip(), and lstrip() . . . 134
Copying and Pasting Strings with the pyperclip Module . . . 135
Project: Password Locker . . . 136
Step 1: Program Design and Data Structures . . . 136
Step 2: Handle Command Line Arguments . . . 137
Step 3: Copy the Right Password . . . 137
Project: Adding Bullets to Wiki Markup . . . 139
Step 1: Copy and Paste from the Clipboard . . . 139
Step 2: Separate the Lines of Text and Add the Star . . . 140
Step 3: Join the Modified Lines . . . 141
Summary . . . 141
Practice Questions . . . 142
Practice Project . . . 142
Table Printer . . . 142

PART II: AUTOMATING TASKS 145

7
PATTERN MATCHING WITH REGULAR EXPRESSIONS 147

Finding Patterns of Text Without Regular Expressions . . . 148
Finding Patterns of Text with Regular Expressions . . . 150
Creating Regex Objects . . . 150
Matching Regex Objects . . . 151
Review of Regular Expression Matching . . . 152
More Pattern Matching with Regular Expressions . . . 152
Grouping with Parentheses . . . 152
Matching Multiple Groups with the Pipe . . . 153
Optional Matching with the Question Mark . . . 154
Matching Zero or More with the Star . . . 155
Matching One or More with the Plus . . . 155
Matching Specific Repetitions with Curly Brackets . . . 156
Greedy and Nongreedy Matching . . . 156
The findall() Method . . . 157
Character Classes . . . 158
Making Your Own Character Classes . . . 159
The Caret and Dollar Sign Characters . . . 159
The Wildcard Character . . . 160
Matching Everything with Dot-Star . . . 161
Matching Newlines with the Dot Character . . . 162
Review of Regex Symbols . . . 162
Case-Insensitive Matching . . . 163

Substituting Strings with the sub() Method 163
Managing Complex Regexes. 164
Combining re.IGNORECASE, re.DOTALL, and re.VERBOSE 164
Project: Phone Number and Email Address Extractor 165
Step 1: Create a Regex for Phone Numbers 166
Step 2: Create a Regex for Email Addresses. 166
Step 3: Find All Matches in the Clipboard Text 167
Step 4: Join the Matches into a String for the Clipboard. 168
Running the Program 169
Ideas for Similar Programs 169
Summary 169
Practice Questions 170
Practice Projects. 171
Strong Password Detection 171
Regex Version of strip() 171

8
READING AND WRITING FILES 173

Files and File Paths. 173
Backslash on Windows and Forward Slash on OS X and Linux. 174
The Current Working Directory 175
Absolute vs. Relative Paths 175
Creating New Folders with os.makedirs() 176
The os.path Module 177
Handling Absolute and Relative Paths 177
Finding File Sizes and Folder Contents. 179
Checking Path Validity. 180
The File Reading/Writing Process 180
Opening Files with the open() Function 181
Reading the Contents of Files 182
Writing to Files 183
Saving Variables with the shelve Module 184
Saving Variables with the pprint.pformat() Function 185
Project: Generating Random Quiz Files. 186
Step 1: Store the Quiz Data in a Dictionary 187
Step 2: Create the Quiz File and Shuffle the Question Order 188
Step 3: Create the Answer Options 189
Step 4: Write Content to the Quiz and Answer Key Files 189
Project: Multiclipboard 191
Step 1: Comments and Shelf Setup 192
Step 2: Save Clipboard Content with a Keyword 192
Step 3: List Keywords and Load a Keyword's Content 193
Summary 194
Practice Questions 194
Practice Projects. 194
Extending the Multiclipboard 194
Mad Libs 195
Regex Search 195

9
ORGANIZING FILES 197

The shutil Module . . . 198
Copying Files and Folders . . . 198
Moving and Renaming Files and Folders . . . 199
Permanently Deleting Files and Folders . . . 200
Safe Deletes with the send2trash Module . . . 201
Walking a Directory Tree . . . 202
Compressing Files with the zipfile Module . . . 203
Reading ZIP Files . . . 204
Extracting from ZIP Files . . . 205
Creating and Adding to ZIP Files . . . 205
Project: Renaming Files with American-Style Dates to European-Style Dates . . . 206
Step 1: Create a Regex for American-Style Dates . . . 206
Step 2: Identify the Date Parts from the Filenames . . . 207
Step 3: Form the New Filename and Rename the Files . . . 209
Ideas for Similar Programs . . . 209
Project: Backing Up a Folder into a ZIP File . . . 209
Step 1: Figure Out the ZIP File's Name . . . 210
Step 2: Create the New ZIP File . . . 211
Step 3: Walk the Directory Tree and Add to the ZIP File . . . 211
Ideas for Similar Programs . . . 212
Summary . . . 212
Practice Questions . . . 213
Practice Projects . . . 213
Selective Copy . . . 213
Deleting Unneeded Files . . . 213
Filling in the Gaps . . . 214

10
DEBUGGING 215

Raising Exceptions . . . 216
Getting the Traceback as a String . . . 217
Assertions . . . 219
Using an Assertion in a Traffic Light Simulation . . . 219
Disabling Assertions . . . 221
Logging . . . 221
Using the logging Module . . . 221
Don't Debug with print() . . . 223
Logging Levels . . . 223
Disabling Logging . . . 224
Logging to a File . . . 225
IDLE's Debugger . . . 225
Go . . . 226
Step . . . 226
Over . . . 226
Out . . . 227

Quit . . . 227
Debugging a Number Adding Program . . . 227
Breakpoints . . . 229
Summary . . . 231
Practice Questions . . . 231
Practice Project . . . 232
Debugging Coin Toss . . . 232

11
WEB SCRAPING 233

Project: mapIt.py with the webbrowser Module . . . 234
Step 1: Figure Out the URL . . . 234
Step 2: Handle the Command Line Arguments . . . 235
Step 3: Handle the Clipboard Content and Launch the Browser . . . 236
Ideas for Similar Programs . . . 236
Downloading Files from the Web with the requests Module . . . 237
Downloading a Web Page with the requests.get() Function . . . 237
Checking for Errors . . . 238
Saving Downloaded Files to the Hard Drive . . . 239
HTML . . . 240
Resources for Learning HTML . . . 240
A Quick Refresher . . . 240
Viewing the Source HTML of a Web Page . . . 241
Opening Your Browser's Developer Tools . . . 242
Using the Developer Tools to Find HTML Elements . . . 244
Parsing HTML with the BeautifulSoup Module . . . 245
Creating a BeautifulSoup Object from HTML . . . 245
Finding an Element with the select() Method . . . 246
Getting Data from an Element's Attributes . . . 248
Project: "I'm Feeling Lucky" Google Search . . . 248
Step 1: Get the Command Line Arguments and Request the Search Page . . . 249
Step 2: Find All the Results . . . 249
Step 3: Open Web Browsers for Each Result . . . 250
Ideas for Similar Programs . . . 251
Project: Downloading All XKCD Comics . . . 251
Step 1: Design the Program . . . 252
Step 2: Download the Web Page . . . 253
Step 3: Find and Download the Comic Image . . . 254
Step 4: Save the Image and Find the Previous Comic . . . 255
Ideas for Similar Programs . . . 256
Controlling the Browser with the selenium Module . . . 256
Starting a Selenium-Controlled Browser . . . 256
Finding Elements on the Page . . . 257
Clicking the Page . . . 259
Filling Out and Submitting Forms . . . 259
Sending Special Keys . . . 260
Clicking Browser Buttons . . . 261
More Information on Selenium . . . 261

Summary 261
Practice Questions 261
Practice Projects 262
Command Line Emailer 262
Image Site Downloader 263
2048 263
Link Verification 263

12
WORKING WITH EXCEL SPREADSHEETS 265

Excel Documents 266
Installing the openpyxl Module 266
Reading Excel Documents 266
Opening Excel Documents with OpenPyXL 267
Getting Sheets from the Workbook 268
Getting Cells from the Sheets 268
Converting Between Column Letters and Numbers 270
Getting Rows and Columns from the Sheets 270
Workbooks, Sheets, Cells 272
Project: Reading Data from a Spreadsheet 272
Step 1: Read the Spreadsheet Data 273
Step 2: Populate the Data Structure 274
Step 3: Write the Results to a File 275
Ideas for Similar Programs 276
Writing Excel Documents 277
Creating and Saving Excel Documents 277
Creating and Removing Sheets 278
Writing Values to Cells 278
Project: Updating a Spreadsheet 279
Step 1: Set Up a Data Structure with the Update Information 280
Step 2: Check All Rows and Update Incorrect Prices 281
Ideas for Similar Programs 281
Setting the Font Style of Cells 282
Font Objects 282
Formulas 284
Adjusting Rows and Columns 285
Setting Row Height and Column Width 285
Merging and Unmerging Cells 286
Freeze Panes 287
Charts 288
Summary 290
Practice Questions 291
Practice Projects 291
Multiplication Table Maker 291
Blank Row Inserter 292
Spreadsheet Cell Inverter 292
Text Files to Spreadsheet 293
Spreadsheet to Text Files 293

13
WORKING WITH PDF AND WORD DOCUMENTS 295

PDF Documents . . . 295
Extracting Text from PDFs . . . 296
Decrypting PDFs . . . 297
Creating PDFs . . . 298
Project: Combining Select Pages from Many PDFs . . . 303
Step 1: Find All PDF Files . . . 304
Step 2: Open Each PDF . . . 304
Step 3: Add Each Page . . . 305
Step 4: Save the Results . . . 305
Ideas for Similar Programs . . . 306
Word Documents . . . 306
Reading Word Documents . . . 307
Getting the Full Text from a .docx File . . . 308
Styling Paragraph and Run Objects . . . 309
Creating Word Documents with Nondefault Styles . . . 310
Run Attributes . . . 311
Writing Word Documents . . . 312
Adding Headings . . . 314
Adding Line and Page Breaks . . . 315
Adding Pictures . . . 315
Summary . . . 316
Practice Questions . . . 316
Practice Projects . . . 317
PDF Paranoia . . . 317
Custom Invitations as Word Documents . . . 317
Brute-Force PDF Password Breaker . . . 318

14
WORKING WITH CSV FILES AND JSON DATA 319

The csv Module . . . 320
Reader Objects . . . 321
Reading Data from Reader Objects in a for Loop . . . 322
Writer Objects . . . 322
The delimiter and lineterminator Keyword Arguments . . . 323
Project: Removing the Header from CSV Files . . . 324
Step 1: Loop Through Each CSV File . . . 325
Step 2: Read in the CSV File . . . 325
Step 3: Write Out the CSV File Without the First Row . . . 326
Ideas for Similar Programs . . . 327
JSON and APIs . . . 327
The json Module . . . 328
Reading JSON with the loads() Function . . . 328
Writing JSON with the dumps() Function . . . 329
Project: Fetching Current Weather Data . . . 329
Step 1: Get Location from the Command Line Argument . . . 330
Step 2: Download the JSON Data . . . 330
Step 3: Load JSON Data and Print Weather . . . 331
Ideas for Similar Programs . . . 332
Summary . . . 333

Practice Questions 333
Practice Project 333
Excel-to-CSV Converter 333

15
KEEPING TIME, SCHEDULING TASKS, AND LAUNCHING PROGRAMS 335

The time Module 336
The time.time() Function 336
The time.sleep() Function 337
Rounding Numbers 338
Project: Super Stopwatch 338
Step 1: Set Up the Program to Track Times 339
Step 2: Track and Print Lap Times 339
Ideas for Similar Programs 340
The datetime Module 341
The timedelta Data Type 342
Pausing Until a Specific Date 344
Converting datetime Objects into Strings 344
Converting Strings into datetime Objects 345
Review of Python's Time Functions 346
Multithreading 347
Passing Arguments to the Thread's Target Function 348
Concurrency Issues 349
Project: Multithreaded XKCD Downloader 350
Step 1: Modify the Program to Use a Function 350
Step 2: Create and Start Threads 351
Step 3: Wait for All Threads to End 352
Launching Other Programs from Python 352
Passing Command Line Arguments to Popen() 354
Task Scheduler, launchd, and cron 354
Opening Websites with Python 355
Running Other Python Scripts 355
Opening Files with Default Applications 355
Project: Simple Countdown Program 357
Step 1: Count Down 357
Step 2: Play the Sound File 357
Ideas for Similar Programs 358
Summary 358
Practice Questions 359
Practice Projects 359
Prettified Stopwatch 360
Scheduled Web Comic Downloader 360

16
SENDING EMAIL AND TEXT MESSAGES 361

SMTP 362
Sending Email 362
Connecting to an SMTP Server 363
Sending the SMTP "Hello" Message 364

Starting TLS Encryption . . . 364
Logging in to the SMTP Server . . . 364
Sending an Email . . . 365
Disconnecting from the SMTP Server . . . 366
IMAP . . . 366
Retrieving and Deleting Emails with IMAP . . . 366
Connecting to an IMAP Server . . . 367
Logging in to the IMAP Server . . . 368
Searching for Email . . . 368
Fetching an Email and Marking It As Read . . . 372
Getting Email Addresses from a Raw Message . . . 373
Getting the Body from a Raw Message . . . 374
Deleting Emails . . . 375
Disconnecting from the IMAP Server . . . 375
Project: Sending Member Dues Reminder Emails . . . 376
Step 1: Open the Excel File . . . 376
Step 2: Find All Unpaid Members . . . 378
Step 3: Send Customized Email Reminders . . . 378
Sending Text Messages with Twilio . . . 380
Signing Up for a Twilio Account . . . 380
Sending Text Messages . . . 381
Project: "Just Text Me" Module . . . 383
Summary . . . 384
Practice Questions . . . 384
Practice Projects . . . 385
Random Chore Assignment Emailer . . . 385
Umbrella Reminder . . . 385
Auto Unsubscriber . . . 385
Controlling Your Computer Through Email . . . 386

17
MANIPULATING IMAGES 387

Computer Image Fundamentals . . . 388
Colors and RGBA Values . . . 388
Coordinates and Box Tuples . . . 389
Manipulating Images with Pillow . . . 390
Working with the Image Data Type . . . 392
Cropping Images . . . 393
Copying and Pasting Images onto Other Images . . . 394
Resizing an Image . . . 397
Rotating and Flipping Images . . . 398
Changing Individual Pixels . . . 400
Project: Adding a Logo . . . 401
Step 1: Open the Logo Image . . . 401
Step 2: Loop Over All Files and Open Images . . . 402
Step 3: Resize the Images . . . 403
Step 4: Add the Logo and Save the Changes . . . 404
Ideas for Similar Programs . . . 406
Drawing on Images . . . 406
Drawing Shapes . . . 406
Drawing Text . . . 408

Summary 410
Practice Questions 410
Practice Projects 411
Extending and Fixing the Chapter Project Programs 411
Identifying Photo Folders on the Hard Drive 411
Custom Seating Cards 412

18
CONTROLLING THE KEYBOARD AND MOUSE WITH GUI AUTOMATION 413

Installing the pyautogui Module 414
Staying on Track 414
Shutting Down Everything by Logging Out 414
Pauses and Fail-Safes 415
Controlling Mouse Movement 415
Moving the Mouse 416
Getting the Mouse Position 417
Project: "Where Is the Mouse Right Now?" 417
Step 1: Import the Module 418
Step 2: Set Up the Quit Code and Infinite Loop 418
Step 3: Get and Print the Mouse Coordinates 418
Controlling Mouse Interaction 419
Clicking the Mouse 420
Dragging the Mouse 420
Scrolling the Mouse 422
Working with the Screen 423
Getting a Screenshot 423
Analyzing the Screenshot 424
Project: Extending the mouseNow Program 424
Image Recognition 425
Controlling the Keyboard 426
Sending a String from the Keyboard 426
Key Names 427
Pressing and Releasing the Keyboard 428
Hotkey Combinations 429
Review of the PyAutoGUI Functions 430
Project: Automatic Form Filler 430
Step 1: Figure Out the Steps 432
Step 2: Set Up Coordinates 432
Step 3: Start Typing Data 434
Step 4: Handle Select Lists and Radio Buttons 435
Step 5: Submit the Form and Wait 436
Summary 437
Practice Questions 438
Practice Projects 438
Looking Busy 438
Instant Messenger Bot 438
Game-Playing Bot Tutorial 439

A
INSTALLING THIRD-PARTY MODULES **441**
The pip Tool 441
Installing Third-Party Modules 442

B
RUNNING PROGRAMS **443**
Shebang Line. 443
Running Python Programs on Windows. 444
Running Python Programs on OS X and Linux 445
Running Python Programs with Assertions Disabled 445

C
ANSWERS TO THE PRACTICE QUESTIONS **447**
Chapter 1 448
Chapter 2 448
Chapter 3 450
Chapter 4 450
Chapter 5 451
Chapter 6 451
Chapter 7 452
Chapter 8 453
Chapter 9 453
Chapter 10 454
Chapter 11 455
Chapter 12 456
Chapter 13 456
Chapter 14 457
Chapter 15 457
Chapter 16 458
Chapter 17 458
Chapter 18 458

INDEX **461**

ACKNOWLEDGMENTS

I couldn't have written a book like this without the help of a lot of people. I'd like to thank Bill Pollock; my editors, Laurel Chun, Leslie Shen, Greg Poulos, and Jennifer Griffith-Delgado; and the rest of the staff at No Starch Press for their invaluable help. Thanks to my tech reviewer, Ari Lacenski, for great suggestions, edits, and support.

Many thanks to our Benevolent Dictator For Life, Guido van Rossum, and everyone at the Python Software Foundation for their great work. The Python community is the best one I've found in the tech industry.

Finally, I would like to thank my family, friends, and the gang at Shotwell's for not minding the busy life I've had while writing this book. Cheers!

INTRODUCTION

"You've just done in two hours what it takes the three of us two days to do." My college roommate was working at a retail electronics store in the early 2000s. Occasionally, the store would receive a spreadsheet of thousands of product prices from its competitor. A team of three employees would print the spreadsheet onto a thick stack of paper and split it among themselves. For each product price, they would look up their store's price and note all the products that their competitors sold for less. It usually took a couple of days.

"You know, I could write a program to do that if you have the original file for the printouts," my roommate told them, when he saw them sitting on the floor with papers scattered and stacked around them.

After a couple of hours, he had a short program that read a competitor's price from a file, found the product in the store's database, and noted whether the competitor was cheaper. He was still new to programming, and

he spent most of his time looking up documentation in a programming book. The actual program took only a few seconds to run. My roommate and his co-workers took an extra-long lunch that day.

This is the power of computer programming. A computer is like a Swiss Army knife that you can configure for countless tasks. Many people spend hours clicking and typing to perform repetitive tasks, unaware that the machine they're using could do their job in seconds if they gave it the right instructions.

Whom Is This Book For?

Software is at the core of so many of the tools we use today: Nearly everyone uses social networks to communicate, many people have Internet-connected computers in their phones, and most office jobs involve interacting with a computer to get work done. As a result, the demand for people who can code has skyrocketed. Countless books, interactive web tutorials, and developer boot camps promise to turn ambitious beginners into software engineers with six-figure salaries.

This book is not for those people. It's for everyone else.

On its own, this book won't turn you into a professional software developer any more than a few guitar lessons will turn you into a rock star. But if you're an office worker, administrator, academic, or anyone else who uses a computer for work or fun, you will learn the basics of programming so that you can automate simple tasks such as the following:

- Moving and renaming thousands of files and sorting them into folders
- Filling out online forms, no typing required
- Downloading files or copy text from a website whenever it updates
- Having your computer text you custom notifications
- Updating or formatting Excel spreadsheets
- Checking your email and sending out prewritten responses

These tasks are simple but time-consuming for humans, and they're often so trivial or specific that there's no ready-made software to perform them. Armed with a little bit of programming knowledge, you can have your computer do these tasks for you.

Conventions

This book is not designed as a reference manual; it's a guide for beginners. The coding style sometimes goes against best practices (for example, some programs use global variables), but that's a trade-off to make the code simpler to learn. This book is made for people to write throwaway code, so there's not much time spent on style and elegance. Sophisticated programming concepts—like object-oriented programming, list comprehensions,

and generators—aren't covered because of the complexity they add. Veteran programmers may point out ways the code in this book could be changed to improve efficiency, but this book is mostly concerned with getting programs to work with the least amount of effort.

What Is Programming?

Television shows and films often show programmers furiously typing cryptic streams of 1s and 0s on glowing screens, but modern programming isn't that mysterious. *Programming* is simply the act of entering instructions for the computer to perform. These instructions might crunch some numbers, modify text, look up information in files, or communicate with other computers over the Internet.

All programs use basic instructions as building blocks. Here are a few of the most common ones, in English:

"Do this; then do that."

"If this condition is true, perform this action; otherwise, do that action."

"Do this action that number of times."

"Keep doing that until this condition is true."

You can combine these building blocks to implement more intricate decisions, too. For example, here are the programming instructions, called the *source code*, for a simple program written in the Python programming language. Starting at the top, the Python software runs each line of code (some lines are run only *if* a certain condition is true or *else* Python runs some other line) until it reaches the bottom.

```
❶ passwordFile = open('SecretPasswordFile.txt')
❷ secretPassword = passwordFile.read()
❸ print('Enter your password.')
  typedPassword = input()
❹ if typedPassword == secretPassword:
❺    print('Access granted')
❻    if typedPassword == '12345':
❼       print('That password is one that an idiot puts on their luggage.')
  else:
❽    print('Access denied')
```

You might not know anything about programming, but you could probably make a reasonable guess at what the previous code does just by reading it. First, the file *SecretPasswordFile.txt* is opened ❶, and the secret password in it is read ❷. Then, the user is prompted to input a password (from the keyboard) ❸. These two passwords are compared ❹, and if they're the same, the program prints *Access granted* to the screen ❺. Next, the program checks to see whether the password is *12345* ❻ and hints that this choice might not be the best for a password ❼. If the passwords are not the same, the program prints *Access denied* to the screen ❽.

What Is Python?

Python refers to the Python programming language (with syntax rules for writing what is considered valid Python code) and the Python interpreter software that reads source code (written in the Python language) and performs its instructions. The Python interpreter is free to download from *http://python.org/*, and there are versions for Linux, OS X, and Windows.

The name Python comes from the surreal British comedy group Monty Python, not from the snake. Python programmers are affectionately called Pythonistas, and both Monty Python and serpentine references usually pepper Python tutorials and documentation.

Programmers Don't Need to Know Much Math

The most common anxiety I hear about learning to program is that people think it requires a lot of math. Actually, most programming doesn't require math beyond basic arithmetic. In fact, being good at programming isn't that different from being good at solving Sudoku puzzles.

To solve a Sudoku puzzle, the numbers 1 through 9 must be filled in for each row, each column, and each 3×3 interior square of the full 9×9 board. You find a solution by applying deduction and logic from the starting numbers. For example, since 5 appears in the top left of the Sudoku puzzle shown in Figure 0-1, it cannot appear elsewhere in the top row, in the leftmost column, or in the top-left 3×3 square. Solving one row, column, or square at a time will provide more number clues for the rest of the puzzle.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 0-1: A new Sudoku puzzle (left) and its solution (right). Despite using numbers, Sudoku doesn't involve much math. (Images © Wikimedia Commons)

Just because Sudoku involves numbers doesn't mean you have to be good at math to figure out the solution. The same is true of programming. Like solving a Sudoku puzzle, writing programs involves breaking down a problem into individual, detailed steps. Similarly, when *debugging* programs (that is, finding and fixing errors), you'll patiently observe what the program is doing and find the cause of the bugs. And like all skills, the more you program, the better you'll become.

Programming Is a Creative Activity

Programming is a creative task, somewhat like constructing a castle out of LEGO bricks. You start with a basic idea of what you want your castle to look like and inventory your available blocks. Then you start building. Once you've finished building your program, you can pretty up your code just like you would your castle.

The difference between programming and other creative activities is that when programming, you have all the raw materials you need in your computer; you don't need to buy any additional canvas, paint, film, yarn, LEGO bricks, or electronic components. When your program is written, it can easily be shared online with the entire world. And though you'll make mistakes when programming, the activity is still a lot of fun.

About This Book

The first part of this book covers basic Python programming concepts, and the second part covers various tasks you can have your computer automate. Each chapter in the second part has project programs for you to study. Here's a brief rundown of what you'll find in each chapter:

Part I: Python Programming Basics

Chapter 1: Python Basics Covers expressions, the most basic type of Python instruction, and how to use the Python interactive shell software to experiment with code.

Chapter 2: Flow Control Explains how to make programs decide which instructions to execute so your code can intelligently respond to different conditions.

Chapter 3: Functions Instructs you on how to define your own functions so that you can organize your code into more manageable chunks.

Chapter 4: Lists Introduces the list data type and explains how to organize data.

Chapter 5: Dictionaries and Structuring Data Introduces the dictionary data type and shows you more powerful ways to organize data.

Chapter 6: Manipulating Strings Covers working with text data (called *strings* in Python).

Part II: Automating Tasks

Chapter 7: Pattern Matching with Regular Expressions Covers how Python can manipulate strings and search for text patterns with regular expressions.

Chapter 8: Reading and Writing Files Explains how your programs can read the contents of text files and save information to files on your hard drive.

Chapter 9: Organizing Files Shows how Python can copy, move, rename, and delete large numbers of files much faster than a human user can. It also explains compressing and decompressing files.

Chapter 10: Debugging Shows how to use Python's various bug-finding and bug-fixing tools.

Chapter 11: Web Scraping Shows how to write programs that can automatically download web pages and parse them for information. This is called *web scraping*.

Chapter 12: Working with Excel Spreadsheets Covers programmatically manipulating Excel spreadsheets so that you don't have to read them. This is helpful when the number of documents you have to analyze is in the hundreds or thousands.

Chapter 13: Working with PDF and Word Documents Covers programmatically reading Word and PDF documents.

Chapter 14: Working with CSV Files and JSON Data Continues to explain how to programmatically manipulate documents with CSV and JSON files.

Chapter 15: Keeping Time, Scheduling Tasks, and Launching Programs Explains how time and dates are handled by Python programs and how to schedule your computer to perform tasks at certain times. This chapter also shows how your Python programs can launch non-Python programs.

Chapter 16: Sending Email and Text Messages Explains how to write programs that can send emails and text messages on your behalf.

Chapter 17: Manipulating Images Explains how to programmatically manipulate images such as JPEG or PNG files.

Chapter 18: Controlling the Keyboard and Mouse with GUI Automation Explains how to programmatically control the mouse and keyboard to automate clicks and keypresses.

Downloading and Installing Python

You can download Python for Windows, OS X, and Ubuntu for free from *http://python.org/downloads/*. If you download the latest version from the website's download page, all of the programs in this book should work.

WARNING *Be sure to download a version of Python 3 (such as 3.4.0). The programs in this book are written to run on Python 3 and may not run correctly, if at all, on Python 2.*

You'll find Python installers for 64-bit and 32-bit computers for each operating system on the download page, so first figure out which installer you need. If you bought your computer in 2007 or later, it is most likely a 64-bit system. Otherwise, you have a 32-bit version, but here's how to find out for sure:

- On Windows, select **Start ▸ Control Panel ▸ System** and check whether System Type says 64-bit or 32-bit.

- On OS X, go the Apple menu, select **About This Mac ▸ More Info ▸ System Report ▸ Hardware**, and then look at the Processor Name field. If it says Intel Core Solo or Intel Core Duo, you have a 32-bit machine. If it says anything else (including Intel Core 2 Duo), you have a 64-bit machine.
- On Ubuntu Linux, open a Terminal and run the command `uname -m`. A response of `i686` means 32-bit, and `x86_64` means 64-bit.

On Windows, download the Python installer (the filename will end with *.msi*) and double-click it. Follow the instructions the installer displays on the screen to install Python, as listed here:

1. Select **Install for All Users** and then click **Next**.
2. Install to the *C:\Python34* folder by clicking **Next**.
3. Click **Next** again to skip the Customize Python section.

On Mac OS X, download the *.dmg* file that's right for your version of OS X and double-click it. Follow the instructions the installer displays on the screen to install Python, as listed here:

1. When the DMG package opens in a new window, double-click the *Python.mpkg* file. You may have to enter the administrator password.
2. Click **Continue** through the Welcome section and click **Agree** to accept the license.
3. Select **HD Macintosh** (or whatever name your hard drive has) and click **Install**.

If you're running Ubuntu, you can install Python from the Terminal by following these steps:

1. Open the Terminal window.
2. Enter `sudo apt-get install python3`.
3. Enter `sudo apt-get install idle3`.
4. Enter `sudo apt-get install python3-pip`.

Starting IDLE

While the *Python interpreter* is the software that runs your Python programs, the *interactive development environment (IDLE)* software is where you'll enter your programs, much like a word processor. Let's start IDLE now.

- On Windows 7 or newer, click the Start icon in the lower-left corner of your screen, enter `IDLE` in the search box, and select **IDLE (Python GUI)**.
- On Windows XP, click the **Start** button and then select **Programs ▸ Python 3.4 ▸ IDLE (Python GUI)**.

- On Mac OS X, open the Finder window, click **Applications**, click **Python 3.4**, and then click the IDLE icon.
- On Ubuntu, select **Applications ▸ Accessories ▸ Terminal** and then enter `idle3`. (You may also be able to click **Applications** at the top of the screen, select **Programming**, and then click **IDLE 3**.)

The Interactive Shell

No matter which operating system you're running, the IDLE window that first appears should be mostly blank except for text that looks something like this:

```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.1600 64
bit (AMD64)] on win32Type "copyright", "credits" or "license()" for more
information.
>>>
```

This window is called the *interactive shell*. A shell is a program that lets you type instructions into the computer, much like the Terminal or Command Prompt on OS X and Windows, respectively. Python's interactive shell lets you enter instructions for the Python interpreter software to run. The computer reads the instructions you enter and runs them immediately.

For example, enter the following into the interactive shell next to the >>> prompt:

```
>>> print('Hello world!')
```

After you type that line and press ENTER, the interactive shell should display this in response:

```
>>> print('Hello world!')
Hello world!
```

How to Find Help

Solving programming problems on your own is easier than you might think. If you're not convinced, then let's cause an error on purpose: Enter `'42' + 3` into the interactive shell. You don't need to know what this instruction means right now, but the result should look like this:

```
  >>> '42' + 3
❶ Traceback (most recent call last):
    File "<pyshell#0>", line 1, in <module>
      '42' + 3
❷ TypeError: Can't convert 'int' object to str implicitly
  >>>
```

The error message ❷ appeared here because Python couldn't understand your instruction. The traceback part ❶ of the error message shows the specific instruction and line number that Python had trouble with. If you're not sure what to make of a particular error message, search online for the exact error message. Enter **"TypeError: Can't convert 'int' object to str implicitly"** (including the quotes) into your favorite search engine, and you should see tons of links explaining what the error message means and what causes it, as shown in Figure 0-2.

Figure 0-2: The Google results for an error message can be very helpful.

You'll often find that someone else had the same question as you and that some other helpful person has already answered it. No one person can know everything about programming, so an everyday part of any software developer's job is looking up answers to technical questions.

Asking Smart Programming Questions

If you can't find the answer by searching online, try asking people in a web forum such as Stack Overlow (*http://stackoverflow.com/*) or the "learn programming" subreddit at *http://reddit.com/r/learnprogramming/*. But keep in mind there are smart ways to ask programming questions that help others help you. Be sure to read the Frequently Asked Questions sections these websites have about the proper way to post questions.

When asking programming questions, remember to do the following:

- Explain what you are trying to do, not just what you did. This lets your helper know if you are on the wrong track.
- Specify the point at which the error happens. Does it occur at the very start of the program or only after you do a certain action?
- Copy and paste the *entire* error message and your code to *http://pastebin.com/* or *http://gist.github.com/*.

  These websites make it easy to share large amounts of code with people over the Web, without the risk of losing any text formatting. You can then put the URL of the posted code in your email or forum post. For example, here some pieces of code I've posted: *http://pastebin.com/SzP2DbFx/* and *https://gist.github.com/asweigart/6912168/*.
- Explain what you've already tried to do to solve your problem. This tells people you've already put in some work to figure things out on your own.
- List the version of Python you're using. (There are some key differences between version 2 Python interpreters and version 3 Python interpreters.) Also, say which operating system and version you're running.
- If the error came up after you made a change to your code, explain exactly what you changed.
- Say whether you're able to reproduce the error every time you run the program or whether it happens only after you perform certain actions. Explain what those actions are, if so.

Always follow good online etiquette as well. For example, don't post your questions in all caps or make unreasonable demands of the people trying to help you.

Summary

For most people, their computer is just an appliance instead of a tool. But by learning how to program, you'll gain access to one of the most powerful tools of the modern world, and you'll have fun along the way. Programming isn't brain surgery—it's fine for amateurs to experiment and make mistakes.

I love helping people discover Python. I write programming tutorials on my blog at *http://inventwithpython.com/blog/*, and you can contact me with questions at *al@inventwithpython.com*.

This book will start you off from zero programming knowledge, but you may have questions beyond its scope. Remember that asking effective questions and knowing how to find answers are invaluable tools on your programming journey.

Let's begin!

PART I

PYTHON PROGRAMMING BASICS

1

PYTHON BASICS

The Python programming language has a wide range of syntactical constructions, standard library functions, and interactive development environment features. Fortunately, you can ignore most of that; you just need to learn enough to write some handy little programs.

You will, however, have to learn some basic programming concepts before you can do anything. Like a wizard-in-training, you might think these concepts seem arcane and tedious, but with some knowledge and practice, you'll be able to command your computer like a magic wand to perform incredible feats.

This chapter has a few examples that encourage you to type into the interactive shell, which lets you execute Python instructions one at a time and shows you the results instantly. Using the interactive shell is great for learning what basic Python instructions do, so give it a try as you follow along. You'll remember the things you do much better than the things you only read.

Entering Expressions into the Interactive Shell

You run the interactive shell by launching IDLE, which you installed with Python in the introduction. On Windows, open the Start menu, select **All Programs ▸ Python 3.3**, and then select **IDLE (Python GUI)**. On OS X, select **Applications ▸ MacPython 3.3 ▸ IDLE**. On Ubuntu, open a new Terminal window and enter `idle3`.

A window with the >>> prompt should appear; that's the interactive shell. Enter `2 + 2` at the prompt to have Python do some simple math.

```
>>> 2 + 2
4
```

The IDLE window should now show some text like this:

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2 + 2
4
>>>
```

In Python, `2 + 2` is called an *expression*, which is the most basic kind of programming instruction in the language. Expressions consist of *values* (such as `2`) and *operators* (such as `+`), and they can always *evaluate* (that is, reduce) down to a single value. That means you can use expressions anywhere in Python code that you could also use a value.

In the previous example, `2 + 2` is evaluated down to a single value, `4`. A single value with no operators is also considered an expression, though it evaluates only to itself, as shown here:

```
>>> 2
2
```

ERRORS ARE OKAY!

Programs will crash if they contain code the computer can't understand, which will cause Python to show an error message. An error message won't break your computer, though, so don't be afraid to make mistakes. A *crash* just means the program stopped running unexpectedly.

If you want to know more about an error message, you can search for the exact message text online to find out more about that specific error. You can also check out the resources at *http://nostarch.com/automatestuff/* to see a list of common Python error messages and their meanings.

There are plenty of other operators you can use in Python expressions, too. For example, Table 1-1 lists all the math operators in Python.

Table 1-1: Math Operators from Highest to Lowest Precedence

Operator	Operation	Example	Evaluates to...
**	Exponent	`2 ** 3`	8
%	Modulus/remainder	`22 % 8`	6
//	Integer division/floored quotient	`22 // 8`	2
/	Division	`22 / 8`	2.75
*	Multiplication	`3 * 5`	15
-	Subtraction	`5 - 2`	3
+	Addition	`2 + 2`	4

The order of operations (also called *precedence*) of Python math operators is similar to that of mathematics. The ** operator is evaluated first; the *, /, //, and % operators are evaluated next, from left to right; and the + and - operators are evaluated last (also from left to right). You can use parentheses to override the usual precedence if you need to. Enter the following expressions into the interactive shell:

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2      +           2
4
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```

In each case, you as the programmer must enter the expression, but Python does the hard part of evaluating it down to a single value. Python will keep evaluating parts of the expression until it becomes a single value, as shown in Figure 1-1.

```
(5 - 1) * ((7 + 1) / (3 - 1))
        ↓
      4 * ((7 + 1) / (3 - 1))
        ↓
      4 * (   8   ) / (3 - 1))
        ↓
      4 * (   8   ) / (   2   )
        ↓
      4 * 4.0
        ↓
       16.0
```

Figure 1-1: Evaluating an expression reduces it to a single value.

These rules for putting operators and values together to form expressions are a fundamental part of Python as a programming language, just like the grammar rules that help us communicate. Here's an example:

This is a grammatically correct English sentence.

This grammatically is sentence not English correct a.

The second line is difficult to parse because it doesn't follow the rules of English. Similarly, if you type in a bad Python instruction, Python won't be able to understand it and will display a `SyntaxError` error message, as shown here:

```
>>> 5 +
  File "<stdin>", line 1
    5 +
      ^
SyntaxError: invalid syntax
>>> 42 + 5 + * 2
  File "<stdin>", line 1
    42 + 5 + * 2
             ^
SyntaxError: invalid syntax
```

You can always test to see whether an instruction works by typing it into the interactive shell. Don't worry about breaking the computer: The worst thing that could happen is that Python responds with an error message. Professional software developers get error messages while writing code all the time.

The Integer, Floating-Point, and String Data Types

Remember that expressions are just values combined with operators, and they always evaluate down to a single value. A *data type* is a category for values, and every value belongs to exactly one data type. The most

common data types in Python are listed in Table 1-2. The values -2 and 30, for example, are said to be *integer* values. The integer (or *int*) data type indicates values that are whole numbers. Numbers with a decimal point, such as 3.14, are called *floating-point numbers* (or *floats*). Note that even though the value 42 is an integer, the value 42.0 would be a floating-point number.

Table 1-2: Common Data Types

Data type	Examples
Integers	-2, -1, 0, 1, 2, 3, 4, 5
Floating-point numbers	-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25
Strings	'a', 'aa', 'aaa', 'Hello!', '11 cats'

Python programs can also have text values called *strings*, or *strs* (pronounced "stirs"). Always surround your string in single quote (') characters (as in 'Hello' or 'Goodbye cruel world!') so Python knows where the string begins and ends. You can even have a string with no characters in it, '', called a *blank string*. Strings are explained in greater detail in Chapter 4.

If you ever see the error message SyntaxError: EOL while scanning string literal, you probably forgot the final single quote character at the end of the string, such as in this example:

```
>>> 'Hello world!
SyntaxError: EOL while scanning string literal
```

String Concatenation and Replication

The meaning of an operator may change based on the data types of the values next to it. For example, + is the addition operator when it operates on two integers or floating-point values. However, when + is used on two string values, it joins the strings as the *string concatenation* operator. Enter the following into the interactive shell:

```
>>> 'Alice' + 'Bob'
'AliceBob'
```

The expression evaluates down to a single, new string value that combines the text of the two strings. However, if you try to use the + operator on a string and an integer value, Python will not know how to handle this, and it will display an error message.

```
>>> 'Alice' + 42
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    'Alice' + 42
TypeError: Can't convert 'int' object to str implicitly
```

The error message `Can't convert 'int' object to str implicitly` means that Python thought you were trying to concatenate an integer to the string `'Alice'`. Your code will have to explicitly convert the integer to a string, because Python cannot do this automatically. (Converting data types will be explained in "Dissecting Your Program" on page 22 when talking about the `str()`, `int()`, and `float()` functions.)

The * operator is used for multiplication when it operates on two integer or floating-point values. But when the * operator is used on one string value and one integer value, it becomes the *string replication* operator. Enter a string multiplied by a number into the interactive shell to see this in action.

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

The expression evaluates down to a single string value that repeats the original a number of times equal to the integer value. String replication is a useful trick, but it's not used as often as string concatenation.

The * operator can be used with only two numeric values (for multiplication) or one string value and one integer value (for string replication). Otherwise, Python will just display an error message.

```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int of type 'str'
>>> 'Alice' * 5.0
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    'Alice' * 5.0
TypeError: can't multiply sequence by non-int of type 'float'
```

It makes sense that Python wouldn't understand these expressions: You can't multiply two words, and it's hard to replicate an arbitrary string a fractional number of times.

Storing Values in Variables

A *variable* is like a box in the computer's memory where you can store a single value. If you want to use the result of an evaluated expression later in your program, you can save it inside a variable.

Assignment Statements

You'll store values in variables with an *assignment statement*. An assignment statement consists of a variable name, an equal sign (called the *assignment operator*), and the value to be stored. If you enter the assignment statement `spam = 42`, then a variable named `spam` will have the integer value `42` stored in it.

Think of a variable as a labeled box that a value is placed in, as in Figure 1-2.

Figure 1-2: `spam = 42` is like telling the program, "The variable `spam` now has the integer value `42` in it."

For example, enter the following into the interactive shell:

```
❶ >>> spam = 40
  >>> spam
  40
  >>> eggs = 2
❷ >>> spam + eggs
  42
  >>> spam + eggs + spam
  82
❸ >>> spam = spam + 2
  >>> spam
  42
```

A variable is *initialized* (or created) the first time a value is stored in it ❶. After that, you can use it in expressions with other variables and values ❷. When a variable is assigned a new value ❸, the old value is forgotten, which is why `spam` evaluated to `42` instead of `40` at the end of the example. This is called *overwriting* the variable. Enter the following code into the interactive shell to try overwriting a string:

```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```

Just like the box in Figure 1-3, the `spam` variable in this example stores `'Hello'` until you replace it with `'Goodbye'`.

Figure 1-3: When a new value is assigned to a variable, the old one is forgotten.

Variable Names

Table 1-3 has examples of legal variable names. You can name a variable anything as long as it obeys the following three rules:

1. It can be only one word.
2. It can use only letters, numbers, and the underscore (_) character.
3. It can't begin with a number.

Table 1-3: Valid and Invalid Variable Names

Valid variable names	Invalid variable names
`balance`	`current-balance` (hyphens are not allowed)
`currentBalance`	`current balance` (spaces are not allowed)
`current_balance`	`4account` (can't begin with a number)
`_spam`	`42` (can't begin with a number)
`SPAM`	`total_$um` (special characters like `$` are not allowed)
`account4`	`'hello'` (special characters like `'` are not allowed)

Variable names are case-sensitive, meaning that `spam`, `SPAM`, `Spam`, and `sPaM` are four different variables. It is a Python convention to start your variables with a lowercase letter.

This book uses camelcase for variable names instead of underscores; that is, variables `lookLikeThis` instead of `looking_like_this`. Some experienced programmers may point out that the official Python code style, PEP 8, says that underscores should be used. I unapologetically prefer camelcase and point to "A Foolish Consistency Is the Hobgoblin of Little Minds" in PEP 8 itself:

> "Consistency with the style guide is important. But most importantly: know when to be inconsistent—sometimes the style guide just doesn't apply. When in doubt, use your best judgment."

A good variable name describes the data it contains. Imagine that you moved to a new house and labeled all of your moving boxes as *Stuff*. You'd never find anything! The variable names `spam`, `eggs`, and `bacon` are used as generic names for the examples in this book and in much of Python's documentation (inspired by the Monty Python "Spam" sketch), but in your programs, a descriptive name will help make your code more readable.

Your First Program

While the interactive shell is good for running Python instructions one at a time, to write entire Python programs, you'll type the instructions into the file editor. The *file editor* is similar to text editors such as Notepad or TextMate, but it has some specific features for typing in source code. To open the file editor in IDLE, select **File ▸ New Window**.

The window that appears should contain a cursor awaiting your input, but it's different from the interactive shell, which runs Python instructions as soon as you press ENTER. The file editor lets you type in many instructions, save the file, and run the program. Here's how you can tell the difference between the two:

- The interactive shell window will always be the one with the >>> prompt.
- The file editor window will not have the >>> prompt.

Now it's time to create your first program! When the file editor window opens, type the following into it:

```
❶ # This program says hello and asks for my name.

❷ print('Hello world!')
  print('What is your name?')    # ask for their name
❸ myName = input()
❹ print('It is good to meet you, ' + myName)
❺ print('The length of your name is:')
  print(len(myName))
```

```
❻ print('What is your age?')    # ask for their age
myAge = input()
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

Once you've entered your source code, save it so that you won't have to retype it each time you start IDLE. From the menu at the top of the file editor window, select **File ▸ Save As**. In the Save As window, enter **`hello.py`** in the File Name field and then click **Save**.

You should save your programs every once in a while as you type them. That way, if the computer crashes or you accidentally exit from IDLE, you won't lose the code. As a shortcut, you can press CTRL-S on Windows and Linux or ⌘-S on OS X to save your file.

Once you've saved, let's run our program. Select **Run ▸ Run Module** or just press the **F5** key. Your program should run in the interactive shell window that appeared when you first started IDLE. Remember, you have to press F5 from the file editor window, not the interactive shell window. Enter your name when your program asks for it. The program's output in the interactive shell should look something like this:

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Hello world!
What is your name?
Al
It is good to meet you, Al
The length of your name is:
2
What is your age?
4
You will be 5 in a year.
>>>
```

When there are no more lines of code to execute, the Python program *terminates*; that is, it stops running. (You can also say that the Python program *exits*.)

You can close the file editor by clicking the X at the top of the window. To reload a saved program, select **File ▸ Open** from the menu. Do that now, and in the window that appears, choose ***hello.py***, and click the **Open** button. Your previously saved *hello.py* program should open in the file editor window.

Dissecting Your Program

With your new program open in the file editor, let's take a quick tour of the Python instructions it uses by looking at what each line of code does.

Comments

The following line is called a *comment*.

```
❶ # This program says hello and asks for my name.
```

Python ignores comments, and you can use them to write notes or remind yourself what the code is trying to do. Any text for the rest of the line following a hash mark (#) is part of a comment.

Sometimes, programmers will put a # in front of a line of code to temporarily remove it while testing a program. This is called *commenting out* code, and it can be useful when you're trying to figure out why a program doesn't work. You can remove the # later when you are ready to put the line back in.

Python also ignores the blank line after the comment. You can add as many blank lines to your program as you want. This can make your code easier to read, like paragraphs in a book.

The print() Function

The `print()` function displays the string value inside the parentheses on the screen.

```
❷ print('Hello world!')
  print('What is your name?') # ask for their name
```

The line `print('Hello world!')` means "Print out the text in the string `'Hello world!'`." When Python executes this line, you say that Python is *calling* the `print()` function and the string value is being *passed* to the function. A value that is passed to a function call is an *argument*. Notice that the quotes are not printed to the screen. They just mark where the string begins and ends; they are not part of the string value.

NOTE *You can also use this function to put a blank line on the screen; just call `print()` with nothing in between the parentheses.*

When writing a function name, the opening and closing parentheses at the end identify it as the name of a function. This is why in this book you'll see `print()` rather than `print`. Chapter 2 describes functions in more detail.

The input() Function

The `input()` function waits for the user to type some text on the keyboard and press ENTER.

```
❸ myName = input()
```

This function call evaluates to a string equal to the user's text, and the previous line of code assigns the `myName` variable to this string value.

You can think of the `input()` function call as an expression that evaluates to whatever string the user typed in. If the user entered `'Al'`, then the expression would evaluate to `myName = 'Al'`.

Printing the User's Name

The following call to `print()` actually contains the expression `'It is good to meet you, ' + myName` between the parentheses.

```
❹ print('It is good to meet you, ' + myName)
```

Remember that expressions can always evaluate to a single value. If `'Al'` is the value stored in `myName` on the previous line, then this expression evaluates to `'It is good to meet you, Al'`. This single string value is then passed to `print()`, which prints it on the screen.

The len() Function

You can pass the `len()` function a string value (or a variable containing a string), and the function evaluates to the integer value of the number of characters in that string.

```
❺ print('The length of your name is:')
  print(len(myName))
```

Enter the following into the interactive shell to try this:

```
>>> len('hello')
5
>>> len('My very energetic monster just scarfed nachos.')
46
>>> len('')
0
```

Just like those examples, `len(myName)` evaluates to an integer. It is then passed to `print()` to be displayed on the screen. Notice that `print()` allows you to pass it either integer values or string values. But notice the error that shows up when you type the following into the interactive shell:

```
 >>> print('I am ' + 29 + ' years old.')
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    print('I am ' + 29 + ' years old.')
TypeError: Can't convert 'int' object to str implicitly
```

The `print()` function isn't causing that error, but rather it's the expression you tried to pass to `print()`. You get the same error message if you type the expression into the interactive shell on its own.

```
>>> 'I am ' + 29 + ' years old.'
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    'I am ' + 29 + ' years old.'
TypeError: Can't convert 'int' object to str implicitly
```

Python gives an error because you can use the + operator only to add two integers together or concatenate two strings. You can't add an integer to a string because this is ungrammatical in Python. You can fix this by using a string version of the integer instead, as explained in the next section.

The str(), int(), and float() Functions

If you want to concatenate an integer such as `29` with a string to pass to `print()`, you'll need to get the value `'29'`, which is the string form of `29`. The `str()` function can be passed an integer value and will evaluate to a string value version of it, as follows:

```
>>> str(29)
'29'
>>> print('I am ' + str(29) + ' years old.')
I am 29 years old.
```

Because `str(29)` evaluates to `'29'`, the expression `'I am ' + str(29) + ' years old.'` evaluates to `'I am ' + '29' + ' years old.'`, which in turn evaluates to `'I am 29 years old.'`. This is the value that is passed to the `print()` function.

The `str()`, `int()`, and `float()` functions will evaluate to the string, integer, and floating-point forms of the value you pass, respectively. Try converting some values in the interactive shell with these functions, and watch what happens.

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
-99
>>> int(1.25)
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

The previous examples call the str(), int(), and float() functions and pass them values of the other data types to obtain a string, integer, or floating-point form of those values.

The str() function is handy when you have an integer or float that you want to concatenate to a string. The int() function is also helpful if you have a number as a string value that you want to use in some mathematics. For example, the input() function always returns a string, even if the user enters a number. Enter **spam = input()** into the interactive shell and enter **101** when it waits for your text.

```
>>> spam = input()
101
>>> spam
'101'
```

The value stored inside spam isn't the integer 101 but the string '101'. If you want to do math using the value in spam, use the int() function to get the integer form of spam and then store this as the new value in spam.

```
>>> spam = int(spam)
>>> spam
101
```

Now you should be able to treat the spam variable as an integer instead of a string.

```
>>> spam * 10 / 5
202.0
```

Note that if you pass a value to int() that it cannot evaluate as an integer, Python will display an error message.

```
>>> int('99.99')
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    int('99.99')
ValueError: invalid literal for int() with base 10: '99.99'
>>> int('twelve')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    int('twelve')
ValueError: invalid literal for int() with base 10: 'twelve'
```

The int() function is also useful if you need to round a floating-point number down.

```
>>> int(7.7)
7
>>> int(7.7) + 1
8
```

In your program, you used the `int()` and `str()` functions in the last three lines to get a value of the appropriate data type for the code.

```
❻ print('What is your age?') # ask for their age
myAge = input()
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

The `myAge` variable contains the value returned from `input()`. Because the `input()` function always returns a string (even if the user typed in a number), you can use the `int(myAge)` code to return an integer value of the string in `myAge`. This integer value is then added to `1` in the expression `int(myAge) + 1`.

The result of this addition is passed to the `str()` function: `str(int(myAge) + 1)`. The string value returned is then concatenated with the strings `'You will be '` and `' in a year.'` to evaluate to one large string value. This large string is finally passed to `print()` to be displayed on the screen.

Let's say the user enters the string `'4'` for `myAge`. The string `'4'` is converted to an integer, so you can add one to it. The result is `5`. The `str()` function converts the result back to a string, so you can concatenate it with the second string, `'in a year.'`, to create the final message. These evaluation steps would look something like Figure 1-4.

TEXT AND NUMBER EQUIVALENCE

Although the string value of a number is considered a completely different value from the integer or floating-point version, an integer can be equal to a floating point.

```
>>> 42 == '42'
False
>>> 42 == 42.0
True
>>> 42.0 == 0042.000
True
```

Python makes this distinction because strings are text, while integers and floats are both numbers.

```
print('You will be ' + str(int(myAge) + 1) + ' in a year.')

print('You will be ' + str(int( '4' ) + 1) + ' in a year.')

print('You will be ' + str(    4 + 1     ) + ' in a year.')

print('You will be ' + str(      5       ) + ' in a year.')

print('You will be ' +         '5'        + ' in a year.')

print('You will be 5'                     + ' in a year.')

print('You will be 5 in a year.')
```

Figure 1-4: The evaluation steps, if `4` was stored in `myAge`

Summary

You can compute expressions with a calculator or type string concatenations with a word processor. You can even do string replication easily by copying and pasting text. But expressions, and their component values—operators, variables, and function calls—are the basic building blocks that make programs. Once you know how to handle these elements, you will be able to instruct Python to operate on large amounts of data for you.

It is good to remember the different types of operators (`+`, `-`, `*`, `/`, `//`, `%`, and `**` for math operations, and `+` and `*` for string operations) and the three data types (integers, floating-point numbers, and strings) introduced in this chapter.

A few different functions were introduced as well. The `print()` and `input()` functions handle simple text output (to the screen) and input (from the keyboard). The `len()` function takes a string and evaluates to an int of the number of characters in the string. The `str()`, `int()`, and `float()` functions will evaluate to the string, integer, or floating-point number form of the value they are passed.

In the next chapter, you will learn how to tell Python to make intelligent decisions about what code to run, what code to skip, and what code to repeat based on the values it has. This is known as *flow control*, and it allows you to write programs that make intelligent decisions.

Practice Questions

1. Which of the following are operators, and which are values?

```
*
'hello'
-88.8
-
/
+
5
```

2. Which of the following is a variable, and which is a string?

```
spam
'spam'
```

3. Name three data types.
4. What is an expression made up of? What do all expressions do?
5. This chapter introduced assignment statements, like `spam = 10`. What is the difference between an expression and a statement?
6. What does the variable `bacon` contain after the following code runs?

```
bacon = 20
bacon + 1
```

7. What should the following two expressions evaluate to?

```
'spam' + 'spamspam'
'spam' * 3
```

8. Why is `eggs` a valid variable name while `100` is invalid?
9. What three functions can be used to get the integer, floating-point number, or string version of a value?
10. Why does this expression cause an error? How can you fix it?

```
'I have eaten ' + 99 + ' burritos.'
```

Extra credit: Search online for the Python documentation for the `len()` function. It will be on a web page titled "Built-in Functions." Skim the list of other functions Python has, look up what the `round()` function does, and experiment with it in the interactive shell.

2

FLOW CONTROL

So you know the basics of individual instructions and that a program is just a series of instructions. But the real strength of programming isn't just running (or *executing*) one instruction after another like a weekend errand list. Based on how the expressions evaluate, the program can decide to skip instructions, repeat them, or choose one of several instructions to run. In fact, you almost never want your programs to start from the first line of code and simply execute every line, straight to the end. *Flow control statements* can decide which Python instructions to execute under which conditions.

These flow control statements directly correspond to the symbols in a flowchart, so I'll provide flowchart versions of the code discussed in this chapter. Figure 2-1 shows a flowchart for what to do if it's raining. Follow the path made by the arrows from Start to End.

Figure 2-1: A flowchart to tell you what to do if it is raining

In a flowchart, there is usually more than one way to go from the start to the end. The same is true for lines of code in a computer program. Flowcharts represent these branching points with diamonds, while the other steps are represented with rectangles. The starting and ending steps are represented with rounded rectangles.

But before you learn about flow control statements, you first need to learn how to represent those *yes* and *no* options, and you need to understand how to write those branching points as Python code. To that end, let's explore Boolean values, comparison operators, and Boolean operators.

Boolean Values

While the integer, floating-point, and string data types have an unlimited number of possible values, the *Boolean* data type has only two values: `True` and `False`. (Boolean is capitalized because the data type is named after mathematician George Boole.) When typed as Python code, the Boolean values `True` and `False` lack the quotes you place around strings, and they always start with a capital *T* or *F*, with the rest of the word in lowercase. Enter the following into the interactive shell. (Some of these instructions are intentionally incorrect, and they'll cause error messages to appear.)

```
❶ >>> spam = True
  >>> spam
  True
❷ >>> true
  Traceback (most recent call last):
    File "<pyshell#2>", line 1, in <module>
      true
  NameError: name 'true' is not defined
❸ >>> True = 2 + 2
  SyntaxError: assignment to keyword
```

Like any other value, Boolean values are used in expressions and can be stored in variables ❶. If you don't use the proper case ❷ or you try to use `True` and `False` for variable names ❸, Python will give you an error message.

Comparison Operators

Comparison operators compare two values and evaluate down to a single Boolean value. Table 2-1 lists the comparison operators.

Table 2-1: Comparison Operators

Operator	Meaning
`==`	Equal to
`!=`	Not equal to
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to

These operators evaluate to `True` or `False` depending on the values you give them. Let's try some operators now, starting with `==` and `!=`.

```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
```

As you might expect, `==` (equal to) evaluates to `True` when the values on both sides are the same, and `!=` (not equal to) evaluates to `True` when the two values are different. The `==` and `!=` operators can actually work with values of any data type.

```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
True
❶ >>> 42 == '42'
False
```

Note that an integer or floating-point value will always be unequal to a string value. The expression `42 == '42'` ❶ evaluates to `False` because Python considers the integer `42` to be different from the string `'42'`.

The `<`, `>`, `<=`, and `>=` operators, on the other hand, work properly only with integer and floating-point values.

```
>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
False
>>> eggCount = 42
❶ >>> eggCount <= 42
True
>>> myAge = 29
❷ >>> myAge >= 10
True
```

THE DIFFERENCE BETWEEN THE == AND = OPERATORS

You might have noticed that the `==` operator (equal to) has two equal signs, while the `=` operator (assignment) has just one equal sign. It's easy to confuse these two operators with each other. Just remember these points:

- The `==` operator (equal to) asks whether two values are the same as each other.
- The `=` operator (assignment) puts the value on the right into the variable on the left.

To help remember which is which, notice that the `==` operator (equal to) consists of two characters, just like the `!=` operator (not equal to) consists of two characters.

You'll often use comparison operators to compare a variable's value to some other value, like in the `eggCount <= 42` ❶ and `myAge >= 10` ❷ examples. (After all, instead of typing **`'dog' != 'cat'`** in your code, you could have just typed **`True`**.) You'll see more examples of this later when you learn about flow control statements.

Boolean Operators

The three Boolean operators (`and`, `or`, and `not`) are used to compare Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value. Let's explore these operators in detail, starting with the `and` operator.

Binary Boolean Operators

The `and` and `or` operators always take two Boolean values (or expressions), so they're considered *binary* operators. The `and` operator evaluates an expression to `True` if *both* Boolean values are `True`; otherwise, it evaluates to `False`. Enter some expressions using `and` into the interactive shell to see it in action.

```
>>> True and True
True
>>> True and False
False
```

A *truth table* shows every possible result of a Boolean operator. Table 2-2 is the truth table for the `and` operator.

Table 2-2: The `and` Operator's Truth Table

Expression	Evaluates to…
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

On the other hand, the `or` operator evaluates an expression to `True` if *either* of the two Boolean values is `True`. If both are `False`, it evaluates to `False`.

```
>>> False or True
True
>>> False or False
False
```

You can see every possible outcome of the `or` operator in its truth table, shown in Table 2-3.

Table 2-3: The `or` Operator's Truth Table

Expression	Evaluates to...
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

The not Operator

Unlike `and` and `or`, the `not` operator operates on only one Boolean value (or expression). The `not` operator simply evaluates to the opposite Boolean value.

```
>>> not True
False
❶ >>> not not not not True
True
```

Much like using double negatives in speech and writing, you can nest `not` operators ❶, though there's never not no reason to do this in real programs. Table 2-4 shows the truth table for `not`.

Table 2-4: The `not` Operator's Truth Table

Expression	Evaluates to...
`not True`	`False`
`not False`	`True`

Mixing Boolean and Comparison Operators

Since the comparison operators evaluate to Boolean values, you can use them in expressions with the Boolean operators.

Recall that the `and`, `or`, and `not` operators are called Boolean operators because they always operate on the Boolean values `True` and `False`. While expressions like `4 < 5` aren't Boolean values, they are expressions that evaluate down to Boolean values. Try entering some Boolean expressions that use comparison operators into the interactive shell.

```
>>> (4 < 5) and (5 < 6)
True
>>> (4 < 5) and (9 < 6)
False
>>> (1 == 2) or (2 == 2)
True
```

The computer will evaluate the left expression first, and then it will evaluate the right expression. When it knows the Boolean value for each, it will then evaluate the whole expression down to one Boolean value. You can think of the computer's evaluation process for `(4 < 5) and (5 < 6)` as shown in Figure 2-2.

Figure 2-2: The process of evaluating (4 < 5) and *(5 < 6) to True.*

You can also use multiple Boolean operators in an expression, along with the comparison operators.

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

The Boolean operators have an order of operations just like the math operators do. After any math and comparison operators evaluate, Python evaluates the `not` operators first, then the `and` operators, and then the `or` operators.

Elements of Flow Control

Flow control statements often start with a part called the *condition*, and all are followed by a block of code called the *clause*. Before you learn about Python's specific flow control statements, I'll cover what a condition and a block are.

Conditions

The Boolean expressions you've seen so far could all be considered conditions, which are the same thing as expressions; *condition* is just a more specific name in the context of flow control statements. Conditions always evaluate down to a Boolean value, `True` or `False`. A flow control statement decides what to do based on whether its condition is `True` or `False`, and almost every flow control statement uses a condition.

Blocks of Code

Lines of Python code can be grouped together in *blocks*. You can tell when a block begins and ends from the indentation of the lines of code. There are three rules for blocks.

1. Blocks begin when the indentation increases.
2. Blocks can contain other blocks.
3. Blocks end when the indentation decreases to zero or to a containing block's indentation.

Blocks are easier to understand by looking at some indented code, so let's find the blocks in part of a small game program, shown here:

```
if name == 'Mary':
❶    print('Hello Mary')
if password == 'swordfish':
❷    print('Access granted.')
else:
❸    print('Wrong password.')
```

The first block of code ❶ starts at the line `print('Hello Mary')` and contains all the lines after it. Inside this block is another block ❷, which has only a single line in it: `print('Access Granted.')`. The third block ❸ is also one line long: `print('Wrong password.')`.

Program Execution

In the previous chapter's *hello.py* program, Python started executing instructions at the top of the program going down, one after another. The *program execution* (or simply, *execution*) is a term for the current instruction being executed. If you print the source code on paper and put your finger on each line as it is executed, you can think of your finger as the program execution.

Not all programs execute by simply going straight down, however. If you use your finger to trace through a program with flow control statements, you'll likely find yourself jumping around the source code based on conditions, and you'll probably skip entire clauses.

Flow Control Statements

Now, let's explore the most important piece of flow control: the statements themselves. The statements represent the diamonds you saw in the flowchart in Figure 2-1, and they are the actual decisions your programs will make.

if Statements

The most common type of flow control statement is the `if` statement. An `if` statement's clause (that is, the block following the `if` statement) will execute if the statement's condition is `True`. The clause is skipped if the condition is `False`.

In plain English, an `if` statement could be read as, "If this condition is true, execute the code in the clause." In Python, an `if` statement consists of the following:

- The `if` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `if` clause)

For example, let's say you have some code that checks to see whether someone's name is Alice. (Pretend `name` was assigned some value earlier.)

```
if name == 'Alice':
    print('Hi, Alice.')
```

All flow control statements end with a colon and are followed by a new block of code (the clause). This `if` statement's clause is the block with `print('Hi, Alice.')`. Figure 2-3 shows what a flowchart of this code would look like.

Figure 2-3: The flowchart for an `if` statement

else Statements

An `if` clause can optionally be followed by an `else` statement. The `else` clause is executed only when the `if` statement's condition is `False`. In plain English, an `else` statement could be read as, "If this condition is true, execute this code. Or else, execute that code." An `else` statement doesn't have a condition, and in code, an `else` statement always consists of the following:

- The `else` keyword
- A colon
- Starting on the next line, an indented block of code (called the `else` clause)

Returning to the Alice example, let's look at some code that uses an `else` statement to offer a different greeting if the person's name isn't Alice.

```
if name == 'Alice':
    print('Hi, Alice.')
```

```
else:
    print('Hello, stranger.')
```

Figure 2-4 shows what a flowchart of this code would look like.

Figure 2-4: The flowchart for an `else` statement

elif Statements

While only one of the `if` or `else` clauses will execute, you may have a case where you want one of *many* possible clauses to execute. The `elif` statement is an "else if" statement that always follows an `if` or another `elif` statement. It provides another condition that is checked only if any of the previous conditions were `False`. In code, an `elif` statement always consists of the following:

- The `elif` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `elif` clause)

Let's add an `elif` to the name checker to see this statement in action.

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
```

This time, you check the person's age, and the program will tell them something different if they're younger than 12. You can see the flowchart for this in Figure 2-5.

Figure 2-5: The flowchart for an `elif` statement

The `elif` clause executes if `age < 12` is `True` and `name == 'Alice'` is `False`. However, if both of the conditions are `False`, then both of the clauses are skipped. It is *not* guaranteed that at least one of the clauses will be executed. When there is a chain of `elif` statements, only one or none of the clauses will be executed. Once one of the statements' conditions is found to be `True`, the rest of the `elif` clauses are automatically skipped. For example, open a new file editor window and enter the following code, saving it as *vampire.py*:

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
elif age > 100:
    print('You are not Alice, grannie.')
```

Here I've added two more `elif` statements to make the name checker greet a person with different answers based on age. Figure 2-6 shows the flowchart for this.

Figure 2-6: The flowchart for multiple `elif` statements in the vampire.py *program*

The order of the elif statements does matter, however. Let's rearrange them to introduce a bug. Remember that the rest of the elif clauses are automatically skipped once a True condition has been found, so if you swap around some of the clauses in *vampire.py*, you run into a problem. Change the code to look like the following, and save it as *vampire2.py*:

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
❶ elif age > 100:
    print('You are not Alice, grannie.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
```

Say the age variable contains the value 3000 before this code is executed. You might expect the code to print the string 'Unlike you, Alice is not an undead, immortal vampire.'. However, because the age > 100 condition is True (after all, 3000 *is* greater than 100) ❶, the string 'You are not Alice, grannie.' is printed, and the rest of the elif statements are automatically skipped. Remember, at most only one of the clauses will be executed, and for elif statements, the order matters!

Figure 2-7 shows the flowchart for the previous code. Notice how the diamonds for age > 100 and age > 2000 are swapped.

Optionally, you can have an else statement after the last elif statement. In that case, it *is* guaranteed that at least one (and only one) of the clauses will be executed. If the conditions in every if and elif statement are False, then the else clause is executed. For example, let's re-create the Alice program to use if, elif, and else clauses.

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

Figure 2-8 shows the flowchart for this new code, which we'll save as *littleKid.py*.

In plain English, this type of flow control structure would be, "If the first condition is true, do this. Else, if the second condition is true, do that. Otherwise, do something else." When you use all three of these statements together, remember these rules about how to order them to avoid bugs like the one in Figure 2-7. First, there is always exactly one if statement. Any elif statements you need should follow the if statement. Second, if you want to be sure that at least one clause is executed, close the structure with an else statement.

Figure 2-7: The flowchart for the vampire2.py *program. The crossed-out path will logically never happen, because if* age *were greater than* 2000*, it would have already been greater than* 100*.*

Figure 2-8: Flowchart for the previous littleKid.py *program*

while Loop Statements

You can make a block of code execute over and over again with a `while` statement. The code in a `while` clause will be executed as long as the `while` statement's condition is `True`. In code, a `while` statement always consists of the following:

- The `while` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `while` clause)

You can see that a `while` statement looks similar to an `if` statement. The difference is in how they behave. At the end of an `if` clause, the program execution continues after the `if` statement. But at the end of a `while` clause, the program execution jumps back to the start of the `while` statement. The `while` clause is often called the *while loop* or just the *loop*.

Let's look at an `if` statement and a `while` loop that use the same condition and take the same actions based on that condition. Here is the code with an `if` statement:

```
spam = 0
if spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

Here is the code with a `while` statement:

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

These statements are similar—both `if` and `while` check the value of `spam`, and if it's less than five, they print a message. But when you run these two code snippets, something very different happens for each one. For the `if` statement, the output is simply `"Hello, world."`. But for the `while` statement, it's `"Hello, world."` repeated five times! Take a look at the flowcharts for these two pieces of code, Figures 2-9 and 2-10, to see why this happens.

Figure 2-9: The flowchart for the `if` statement code

Figure 2-10: The flowchart for the `while` statement code

The code with the `if` statement checks the condition, and it prints `Hello, world.` only once if that condition is true. The code with the `while` loop, on the other hand, will print it five times. It stops after five prints because the integer in `spam` is incremented by one at the end of each loop iteration, which means that the loop will execute five times before `spam < 5` is `False`.

In the `while` loop, the condition is always checked at the start of each *iteration* (that is, each time the loop is executed). If the condition is `True`, then the clause is executed, and afterward, the condition is checked again. The first time the condition is found to be `False`, the `while` clause is skipped.

An Annoying while Loop

Here's a small example program that will keep asking you to type, literally, **`your name`**. Select **File ▸ New Window** to open a new file editor window, enter the following code, and save the file as *yourName.py*:

```
❶ name = ''
❷ while name != 'your name':
      print('Please type your name.')
❸     name = input()
❹ print('Thank you!')
```

First, the program sets the `name` variable ❶ to an empty string. This is so that the `name != 'your name'` condition will evaluate to `True` and the program execution will enter the `while` loop's clause ❷.

The code inside this clause asks the user to type their name, which is assigned to the `name` variable ❸. Since this is the last line of the block, the execution moves back to the start of the `while` loop and reevaluates the condition. If the value in `name` is *not equal* to the string `'your name'`, then the condition is `True`, and the execution enters the `while` clause again.

But once the user types **`your name`**, the condition of the `while` loop will be `'your name' != 'your name'`, which evaluates to `False`. The condition is now `False`, and instead of the program execution reentering the `while` loop's clause, it skips past it and continues running the rest of the program ❹. Figure 2-11 shows a flowchart for the *yourName.py* program.

Figure 2-11: A flowchart of the yourName.py *program*

Now, let's see *yourName.py* in action. Press **F5** to run it, and enter something other than **`your name`** a few times before you give the program what it wants.

```
Please type your name.
Al
Please type your name.
Albert
```

```
Please type your name.
%#@#%*(^&!!!
Please type your name.
your name
Thank you!
```

If you never enter **`your name`**, then the `while` loop's condition will never be `False`, and the program will just keep asking forever. Here, the `input()` call lets the user enter the right string to make the program move on. In other programs, the condition might never actually change, and that can be a problem. Let's look at how you can break out of a `while` loop.

break Statements

There is a shortcut to getting the program execution to break out of a `while` loop's clause early. If the execution reaches a `break` statement, it immediately exits the `while` loop's clause. In code, a `break` statement simply contains the `break` keyword.

Pretty simple, right? Here's a program that does the same thing as the previous program, but it uses a `break` statement to escape the loop. Enter the following code, and save the file as *yourName2.py*:

```
❶ while True:
      print('Please type your name.')
❷     name = input()
❸     if name == 'your name':
❹         break
❺ print('Thank you!')
```

The first line ❶ creates an *infinite loop*; it is a `while` loop whose condition is always `True`. (The expression `True`, after all, always evaluates down to the value `True`.) The program execution will always enter the loop and will exit it only when a `break` statement is executed. (An infinite loop that *never* exits is a common programming bug.)

Just like before, this program asks the user to type **`your name`** ❷. Now, however, while the execution is still inside the `while` loop, an `if` statement gets executed ❸ to check whether `name` is equal to `your name`. If this condition is `True`, the `break` statement is run ❹, and the execution moves out of the loop to `print('Thank you!')` ❺. Otherwise, the `if` statement's clause with the `break` statement is skipped, which puts the execution at the end of the `while` loop. At this point, the program execution jumps back to the start of the `while` statement ❶ to recheck the condition. Since this condition is merely the `True` Boolean value, the execution enters the loop to ask the user to type **`your name`** again. See Figure 2-12 for the flowchart of this program.

Run *yourName2.py*, and enter the same text you entered for *yourName.py*. The rewritten program should respond in the same way as the original.

Figure 2-12: The flowchart for the yourName2.py *program with an infinite loop. Note that the X path will logically never happen because the loop condition is always* `True`*.*

continue Statements

Like `break` statements, `continue` statements are used inside loops. When the program execution reaches a `continue` statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition. (This is also what happens when the execution reaches the end of the loop.)

TRAPPED IN AN INFINITE LOOP?

If you ever run a program that has a bug causing it to get stuck in an infinite loop, press CTRL-C. This will send a `KeyboardInterrupt` error to your program and cause it to stop immediately. To try it, create a simple infinite loop in the file editor, and save it as *infiniteloop.py*.

```
while True:
    print('Hello world!')
```

When you run this program, it will print `Hello world!` to the screen forever, because the `while` statement's condition is always `True`. In IDLE's interactive shell window, there are only two ways to stop this program: press CTRL-C or select **Shell ▸ Restart Shell** from the menu. CTRL-C is handy if you ever want to terminate your program immediately, even if it's not stuck in an infinite loop.

Let's use `continue` to write a program that asks for a name and password. Enter the following code into a new file editor window and save the program as *swordfish.py*.

```
  while True:
      print('Who are you?')
      name = input()
❶     if name != 'Joe':
❷         continue
      print('Hello, Joe. What is the password? (It is a fish.)')
❸     password = input()
      if password == 'swordfish':
❹         break
❺ print('Access granted.')
```

If the user enters any name besides `Joe` ❶, the `continue` statement ❷ causes the program execution to jump back to the start of the loop. When it reevaluates the condition, the execution will always enter the loop, since the condition is simply the value `True`. Once they make it past that `if` statement, the user is asked for a password ❸. If the password entered is `swordfish`, then the `break` statement ❹ is run, and the execution jumps out of the `while` loop to print `Access granted` ❺. Otherwise, the execution continues to the end of the `while` loop, where it then jumps back to the start of the loop. See Figure 2-13 for this program's flowchart.

Figure 2-13: A flowchart for swordfish.py. *The X path will logically never happen because the loop condition is always* `True`.

"TRUTHY" AND "FALSEY" VALUES

There are some values in other data types that conditions will consider equivalent to `True` and `False`. When used in conditions, `0`, `0.0`, and `''` (the empty string) are considered `False`, while all other values are considered `True`. For example, look at the following program:

```
name = ''
while not name:❶
    print('Enter your name:')
    name = input()
print('How many guests will you have?')
numOfGuests = int(input())
if numOfGuests:❷
    print('Be sure to have enough room for all your guests.')❸
print('Done')
```

If the user enters a blank string for `name`, then the `while` statement's condition will be `True` ❶, and the program continues to ask for a name. If the value for `numOfGuests` is not `0` ❷, then the condition is considered to be `True`, and the program will print a reminder for the user ❸.

You could have typed `not name != ''` instead of `not name`, and `numOfGuests != 0` instead of `numOfGuests`, but using the truthy and falsey values can make your code easier to read.

Run this program and give it some input. Until you claim to be Joe, it shouldn't ask for a password, and once you enter the correct password, it should exit.

```
Who are you?
I'm fine, thanks. Who are you?
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
Mary
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
swordfish
Access granted.
```

for Loops and the range() Function

The `while` loop keeps looping while its condition is `True` (which is the reason for its name), but what if you want to execute a block of code only a certain number of times? You can do this with a `for` loop statement and the `range()` function.

In code, a `for` statement looks something like `for i in range(5):` and always includes the following:

- The `for` keyword
- A variable name
- The `in` keyword
- A call to the `range()` method with up to three integers passed to it
- A colon
- Starting on the next line, an indented block of code (called the `for` clause)

Let's create a new program called *fiveTimes.py* to help you see a `for` loop in action.

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

The code in the `for` loop's clause is run five times. The first time it is run, the variable `i` is set to `0`. The `print()` call in the clause will print `Jimmy Five Times (0)`. After Python finishes an iteration through all the code inside the `for` loop's clause, the execution goes back to the top of the loop, and the `for` statement increments `i` by one. This is why `range(5)` results in five iterations through the clause, with `i` being set to `0`, then `1`, then `2`, then `3`, and then `4`. The variable `i` will go up to, but will not include, the integer passed to `range()`. Figure 2-14 shows a flowchart for the *fiveTimes.py* program.

Figure 2-14: The flowchart for fiveTimes.py

When you run this program, it should print `Jimmy Five Times` followed by the value of `i` five times before leaving the `for` loop.

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

NOTE *You can use `break` and `continue` statements inside `for` loops as well. The `continue` statement will continue to the next value of the `for` loop's counter, as if the program execution had reached the end of the loop and returned to the start. In fact, you can use `continue` and `break` statements only inside `while` and `for` loops. If you try to use these statements elsewhere, Python will give you an error.*

As another `for` loop example, consider this story about the mathematician Karl Friedrich Gauss. When Gauss was a boy, a teacher wanted to give the class some busywork. The teacher told them to add up all the numbers from 0 to 100. Young Gauss came up with a clever trick to figure out the answer in a few seconds, but you can write a Python program with a `for` loop to do this calculation for you.

```
❶ total = 0
❷ for num in range(101):
❸     total = total + num
❹ print(total)
```

The result should be 5,050. When the program first starts, the `total` variable is set to `0` ❶. The `for` loop ❷ then executes `total = total + num` ❸ 100 times. By the time the loop has finished all of its 100 iterations, every integer from `0` to `100` will have been added to `total`. At this point, `total` is printed to the screen ❹. Even on the slowest computers, this program takes less than a second to complete.

(Young Gauss figured out that there were 50 pairs of numbers that added up to 100: 1 + 99, 2 + 98, 3 + 97, and so on, until 49 + 51. Since 50 × 100 is 5,000, when you add that middle 50, the sum of all the numbers from 0 to 100 is 5,050. Clever kid!)

An Equivalent while Loop

You can actually use a `while` loop to do the same thing as a `for` loop; `for` loops are just more concise. Let's rewrite *fiveTimes.py* to use a `while` loop equivalent of a `for` loop.

```
print('My name is')
i = 0
while i < 5:
    print('Jimmy Five Times (' + str(i) + ')')
    i = i + 1
```

If you run this program, the output should look the same as the *fiveTimes.py* program, which uses a `for` loop.

The Starting, Stopping, and Stepping Arguments to range()

Some functions can be called with multiple arguments separated by a comma, and `range()` is one of them. This lets you change the integer passed to `range()` to follow any sequence of integers, including starting at a number other than zero.

```
for i in range(12, 16):
    print(i)
```

The first argument will be where the `for` loop's variable starts, and the second argument will be up to, but not including, the number to stop at.

```
12
13
14
15
```

The `range()` function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the *step argument*. The step is the amount that the variable is increased by after each iteration.

```
for i in range(0, 10, 2):
    print(i)
```

So calling `range(0, 10, 2)` will count from zero to eight by intervals of two.

```
0
2
4
6
8
```

The `range()` function is flexible in the sequence of numbers it produces for `for` loops. *For* example (I never apologize for my puns), you can even use a negative number for the step argument to make the `for` loop count down instead of up.

```
for i in range(5, -1, -1):
    print(i)
```

Running a `for` loop to print `i` with `range(5, -1, -1)` should print from five down to zero.

```
5
4
```

```
3
2
1
0
```

Importing Modules

All Python programs can call a basic set of functions called *built-in functions*, including the `print()`, `input()`, and `len()` functions you've seen before. Python also comes with a set of modules called the *standard library*. Each module is a Python program that contains a related group of functions that can be embedded in your programs. For example, the `math` module has mathematics-related functions, the `random` module has random number–related functions, and so on.

Before you can use the functions in a module, you must import the module with an `import` statement. In code, an `import` statement consists of the following:

- The `import` keyword
- The name of the module
- Optionally, more module names, as long as they are separated by commas

Once you import a module, you can use all the cool functions of that module. Let's give it a try with the `random` module, which will give us access to the `random.ranint()` function.

Enter this code into the file editor, and save it as *printRandom.py*:

```
import random
for i in range(5):
    print(random.randint(1, 10))
```

When you run this program, the output will look something like this:

```
4
1
8
4
1
```

The `random.randint()` function call evaluates to a random integer value between the two integers that you pass it. Since `randint()` is in the `random` module, you must first type **`random.`** in front of the function name to tell Python to look for this function inside the `random` module.

Here's an example of an `import` statement that imports four different modules:

```
import random, sys, os, math
```

Now we can use any of the functions in these four modules. We'll learn more about them later in the book.

from import Statements

An alternative form of the `import` statement is composed of the `from` keyword, followed by the module name, the `import` keyword, and a star; for example, `from random import *`.

With this form of `import` statement, calls to functions in `random` will not need the `random.` prefix. However, using the full name makes for more readable code, so it is better to use the normal form of the `import` statement.

Ending a Program Early with sys.exit()

The last flow control concept to cover is how to terminate the program. This always happens if the program execution reaches the bottom of the instructions. However, you can cause the program to terminate, or exit, by calling the `sys.exit()` function. Since this function is in the `sys` module, you have to import `sys` before your program can use it.

Open a new file editor window and enter the following code, saving it as *exitExample.py*:

```
import sys

while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

Run this program in IDLE. This program has an infinite loop with no `break` statement inside. The only way this program will end is if the user enters `exit`, causing `sys.exit()` to be called. When `response` is equal to `exit`, the program ends. Since the `response` variable is set by the `input()` function, the user must enter `exit` in order to stop the program.

Summary

By using expressions that evaluate to `True` or `False` (also called *conditions*), you can write programs that make decisions on what code to execute and what code to skip. You can also execute code over and over again in a loop while a certain condition evaluates to `True`. The `break` and `continue` statements are useful if you need to exit a loop or jump back to the start.

These flow control statements will let you write much more intelligent programs. There's another type of flow control that you can achieve by writing your own functions, which is the topic of the next chapter.

Practice Questions

1. What are the two values of the Boolean data type? How do you write them?
2. What are the three Boolean operators?
3. Write out the truth tables of each Boolean operator (that is, every possible combination of Boolean values for the operator and what they evaluate to).
4. What do the following expressions evaluate to?

```
(5 > 4) and (3 == 5)
not (5 > 4)
(5 > 4) or (3 == 5)
not ((5 > 4) or (3 == 5))
(True and True) and (True == False)
(not False) or (not True)
```

5. What are the six comparison operators?
6. What is the difference between the equal to operator and the assignment operator?
7. Explain what a condition is and where you would use one.
8. Identify the three blocks in this code:

```
spam = 0
if spam == 10:
    print('eggs')
    if spam > 5:
        print('bacon')
    else:
        print('ham')
    print('spam')
print('spam')
```

9. Write code that prints `Hello` if `1` is stored in `spam`, prints `Howdy` if `2` is stored in `spam`, and prints `Greetings!` if anything else is stored in `spam`.
10. What can you press if your program is stuck in an infinite loop?
11. What is the difference between `break` and `continue`?
12. What is the difference between `range(10)`, `range(0, 10)`, and `range(0, 10, 1)` in a `for` loop?
13. Write a short program that prints the numbers `1` to `10` using a `for` loop. Then write an equivalent program that prints the numbers `1` to `10` using a `while` loop.
14. If you had a function named `bacon()` inside a module named `spam`, how would you call it after importing `spam`?

    Extra credit: Look up the `round()` and `abs()` functions on the Internet, and find out what they do. Experiment with them in the interactive shell.

3

FUNCTIONS

You're already familiar with the `print()`, `input()`, and `len()` functions from the previous chapters. Python provides several built-in functions like these, but you can also write your own functions. A *function* is like a mini-program within a program.

To better understand how functions work, let's create one. Type this program into the file editor and save it as *helloFunc.py*:

```
❶ def hello():
❷     print('Howdy!')
      print('Howdy!!!')
      print('Hello there.')

❸ hello()
  hello()
  hello()
```

The first line is a `def` statement ❶, which defines a function named `hello()`. The code in the block that follows the `def` statement ❷ is the body of the function. This code is executed when the function is called, not when the function is first defined.

The `hello()` lines after the function ❸ are function calls. In code, a function call is just the function's name followed by parentheses, possibly with some number of arguments in between the parentheses. When the program execution reaches these calls, it will jump to the top line in the function and begin executing the code there. When it reaches the end of the function, the execution returns to the line that called the function and continues moving through the code as before.

Since this program calls `hello()` three times, the code in the `hello()` function is executed three times. When you run this program, the output looks like this:

```
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
```

A major purpose of functions is to group code that gets executed multiple times. Without a function defined, you would have to copy and paste this code each time, and the program would look like this:

```
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
```

In general, you always want to avoid duplicating code, because if you ever decide to update the code—if, for example, you find a bug you need to fix—you'll have to remember to change the code everywhere you copied it.

As you get more programming experience, you'll often find yourself *deduplicating* code, which means getting rid of duplicated or copy-and-pasted code. Deduplication makes your programs shorter, easier to read, and easier to update.

def Statements with Parameters

When you call the `print()` or `len()` function, you pass in values, called *arguments* in this context, by typing them between the parentheses. You can also define your own functions that accept arguments. Type this example into the file editor and save it as *helloFunc2.py*:

```
❶ def hello(name):
❷     print('Hello ' + name)

❸ hello('Alice')
  hello('Bob')
```

When you run this program, the output looks like this:

```
Hello Alice
Hello Bob
```

The definition of the `hello()` function in this program has a parameter called `name` ❶. A *parameter* is a variable that an argument is stored in when a function is called. The first time the `hello()` function is called, it's with the argument `'Alice'` ❸. The program execution enters the function, and the variable `name` is automatically set to `'Alice'`, which is what gets printed by the `print()` statement ❷.

One special thing to note about parameters is that the value stored in a parameter is forgotten when the function returns. For example, if you added `print(name)` after `hello('Bob')` in the previous program, the program would give you a `NameError` because there is no variable named `name`. This variable was destroyed after the function call `hello('Bob')` had returned, so `print(name)` would refer to a `name` variable that does not exist.

This is similar to how a program's variables are forgotten when the program terminates. I'll talk more about why that happens later in the chapter, when I discuss what a function's local scope is.

Return Values and return Statements

When you call the `len()` function and pass it an argument such as `'Hello'`, the function call evaluates to the integer value `5`, which is the length of the string you passed it. In general, the value that a function call evaluates to is called the *return value* of the function.

When creating a function using the `def` statement, you can specify what the return value should be with a `return` statement. A `return` statement consists of the following:

- The `return` keyword
- The value or expression that the function should return

When an expression is used with a `return` statement, the return value is what this expression evaluates to. For example, the following program defines a function that returns a different string depending on what number it is passed as an argument. Type this code into the file editor and save it as *magic8Ball.py*:

```
❶ import random

❷ def getAnswer(answerNumber):
❸     if answerNumber == 1:
          return 'It is certain'
      elif answerNumber == 2:
          return 'It is decidedly so'
      elif answerNumber == 3:
          return 'Yes'
      elif answerNumber == 4:
          return 'Reply hazy try again'
      elif answerNumber == 5:
          return 'Ask again later'
      elif answerNumber == 6:
          return 'Concentrate and ask again'
      elif answerNumber == 7:
          return 'My reply is no'
      elif answerNumber == 8:
          return 'Outlook not so good'
      elif answerNumber == 9:
          return 'Very doubtful'

❹ r = random.randint(1, 9)
❺ fortune = getAnswer(r)
❻ print(fortune)
```

When this program starts, Python first imports the `random` module ❶. Then the `getAnswer()` function is defined ❷. Because the function is being defined (and not called), the execution skips over the code in it. Next, the `random.randint()` function is called with two arguments, `1` and `9` ❹. It evaluates to a random integer between `1` and `9` (including `1` and `9` themselves), and this value is stored in a variable named `r`.

The `getAnswer()` function is called with `r` as the argument ❺. The program execution moves to the top of the `getAnswer()` function ❸, and the value `r` is stored in a parameter named `answerNumber`. Then, depending on this value in `answerNumber`, the function returns one of many possible string values. The program execution returns to the line at the bottom of the program that originally called `getAnswer()` ❺. The returned string is assigned to a variable named `fortune`, which then gets passed to a `print()` call ❻ and is printed to the screen.

Note that since you can pass return values as an argument to another function call, you could shorten these three lines:

```
r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

to this single equivalent line:

```
print(getAnswer(random.randint(1, 9)))
```

Remember, expressions are composed of values and operators. A function call can be used in an expression because it evaluates to its return value.

The None Value

In Python there is a value called `None`, which represents the absence of a value. `None` is the only value of the `NoneType` data type. (Other programming languages might call this value `null`, `nil`, or `undefined`.) Just like the Boolean `True` and `False` values, `None` must be typed with a capital *N*.

This value-without-a-value can be helpful when you need to store something that won't be confused for a real value in a variable. One place where `None` is used is as the return value of `print()`. The `print()` function displays text on the screen, but it doesn't need to return anything in the same way `len()` or `input()` does. But since all function calls need to evaluate to a return value, `print()` returns `None`. To see this in action, enter the following into the interactive shell:

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

Behind the scenes, Python adds `return None` to the end of any function definition with no `return` statement. This is similar to how a `while` or `for` loop implicitly ends with a `continue` statement. Also, if you use a `return` statement without a value (that is, just the `return` keyword by itself), then `None` is returned.

Keyword Arguments and print()

Most arguments are identified by their position in the function call. For example, `random.randint(1, 10)` is different from `random.randint(10, 1)`. The function call `random.randint(1, 10)` will return a random integer between `1` and `10`, because the first argument is the low end of the range and the second argument is the high end (while `random.randint(10, 1)` causes an error).

However, *keyword arguments* are identified by the keyword put before them in the function call. Keyword arguments are often used for optional parameters. For example, the `print()` function has the optional parameters `end` and `sep` to specify what should be printed at the end of its arguments and between its arguments (separating them), respectively.

If you ran the following program:

```
print('Hello')
print('World')
```

the output would look like this:

```
Hello
World
```

The two strings appear on separate lines because the `print()` function automatically adds a newline character to the end of the string it is passed. However, you can set the `end` keyword argument to change this to a different string. For example, if the program were this:

```
print('Hello', end='')
print('World')
```

the output would look like this:

```
HelloWorld
```

The output is printed on a single line because there is no longer a newline printed after `'Hello'`. Instead, the blank string is printed. This is useful if you need to disable the newline that gets added to the end of every `print()` function call.

Similarly, when you pass multiple string values to `print()`, the function will automatically separate them with a single space. Enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

But you could replace the default separating string by passing the `sep` keyword argument. Enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

You can add keyword arguments to the functions you write as well, but first you'll have to learn about the list and dictionary data types in the next two chapters. For now, just know that some functions have optional keyword arguments that can be specified when the function is called.

Local and Global Scope

Parameters and variables that are assigned in a called function are said to exist in that function's *local scope*. Variables that are assigned outside all functions are said to exist in the *global scope*. A variable that exists in a local scope is called a *local variable*, while a variable that exists in the global scope is called a *global variable*. A variable must be one or the other; it cannot be both local and global.

Think of a *scope* as a container for variables. When a scope is destroyed, all the values stored in the scope's variables are forgotten. There is only one global scope, and it is created when your program begins. When your program terminates, the global scope is destroyed, and all its variables are forgotten. Otherwise, the next time you ran your program, the variables would remember their values from the last time you ran it.

A local scope is created whenever a function is called. Any variables assigned in this function exist within the local scope. When the function returns, the local scope is destroyed, and these variables are forgotten. The next time you call this function, the local variables will not remember the values stored in them from the last time the function was called.

Scopes matter for several reasons:

- Code in the global scope cannot use any local variables.
- However, a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named `spam` and a global variable also named `spam`.

The reason Python has different scopes instead of just making everything a global variable is so that when variables are modified by the code in a particular call to a function, the function interacts with the rest of the program only through its parameters and the return value. This narrows down the list code lines that may be causing a bug. If your program contained nothing but global variables and had a bug because of a variable being set to a bad value, then it would be hard to track down where this bad value was set. It could have been set from anywhere in the program—and your program could be hundreds or thousands of lines long! But if the bug is because of a local variable with a bad value, you know that only the code in that one function could have set it incorrectly.

While using global variables in small programs is fine, it is a bad habit to rely on global variables as your programs get larger and larger.

Local Variables Cannot Be Used in the Global Scope

Consider this program, which will cause an error when you run it:

```
def spam():
    eggs = 31337
```

```
spam()
print(eggs)
```

If you run this program, the output will look like this:

```
Traceback (most recent call last):
  File "C:/test3784.py", line 4, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

The error happens because the eggs variable exists only in the local scope created when spam() is called. Once the program execution returns from spam, that local scope is destroyed, and there is no longer a variable named eggs. So when your program tries to run print(eggs), Python gives you an error saying that eggs is not defined. This makes sense if you think about it; when the program execution is in the global scope, no local scopes exist, so there can't be any local variables. This is why only global variables can be used in the global scope.

Local Scopes Cannot Use Variables in Other Local Scopes

A new local scope is created whenever a function is called, including when a function is called from another function. Consider this program:

```
  def spam():
❶     eggs = 99
❷     bacon()
❸     print(eggs)

  def bacon():
      ham = 101
❹     eggs = 0

❺ spam()
```

When the program starts, the spam() function is called ❺, and a local scope is created. The local variable eggs ❶ is set to 99. Then the bacon() function is called ❷, and a second local scope is created. Multiple local scopes can exist at the same time. In this new local scope, the local variable ham is set to 101, and a local variable eggs—which is different from the one in spam()'s local scope—is also created ❹ and set to 0.

When bacon() returns, the local scope for that call is destroyed. The program execution continues in the spam() function to print the value of eggs ❸, and since the local scope for the call to spam() still exists here, the eggs variable is set to 99. This is what the program prints.

The upshot is that local variables in one function are completely separate from the local variables in another function.

Global Variables Can Be Read from a Local Scope

Consider the following program:

```
def spam():
    print(eggs)
eggs = 42
spam()
print(eggs)
```

Since there is no parameter named eggs or any code that assigns eggs a value in the spam() function, when eggs is used in spam(), Python considers it a reference to the global variable eggs. This is why 42 is printed when the previous program is run.

Local and Global Variables with the Same Name

To simplify your life, avoid using local variables that have the same name as a global variable or another local variable. But technically, it's perfectly legal to do so in Python. To see what happens, type the following code into the file editor and save it as *sameName.py*:

```
  def spam():
❶     eggs = 'spam local'
      print(eggs)    # prints 'spam local'

  def bacon():
❷     eggs = 'bacon local'
      print(eggs)    # prints 'bacon local'
      spam()
      print(eggs)    # prints 'bacon local'

❸ eggs = 'global'
  bacon()
  print(eggs)        # prints 'global'
```

When you run this program, it outputs the following:

```
bacon local
spam local
bacon local
global
```

There are actually three different variables in this program, but confusingly they are all named eggs. The variables are as follows:

❶ A variable named eggs that exists in a local scope when spam() is called.

❷ A variable named eggs that exists in a local scope when bacon() is called.

❸ A variable named eggs that exists in the global scope.

Since these three separate variables all have the same name, it can be confusing to keep track of which one is being used at any given time. This is why you should avoid using the same variable name in different scopes.

The global Statement

If you need to modify a global variable from within a function, use the `global` statement. If you have a line such as `global eggs` at the top of a function, it tells Python, "In this function, `eggs` refers to the global variable, so don't create a local variable with this name." For example, type the following code into the file editor and save it as *sameName2.py*:

```
def spam():
❶    global eggs
❷    eggs = 'spam'

eggs = 'global'
spam()
print(eggs)
```

When you run this program, the final `print()` call will output this:

```
spam
```

Because `eggs` is declared `global` at the top of `spam()` ❶, when `eggs` is set to `'spam'` ❷, this assignment is done to the globally scoped `spam`. No local `spam` variable is created.

There are four rules to tell whether a variable is in a local scope or global scope:

1. If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
2. If there is a `global` statement for that variable in a function, it is a global variable.
3. Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
4. But if the variable is not used in an assignment statement, it is a global variable.

To get a better feel for these rules, here's an example program. Type the following code into the file editor and save it as *sameName3.py*:

```
  def spam():
❶   global eggs
    eggs = 'spam' # this is the global

  def bacon():
❷   eggs = 'bacon' # this is a local
```

```
  def ham():
❸   print(eggs) # this is the global

  eggs = 42 # this is the global
  spam()
  print(eggs)
```

In the spam() function, eggs is the global eggs variable, because there's a global statement for eggs at the beginning of the function ❶. In bacon(), eggs is a local variable, because there's an assignment statement for it in that function ❷. In ham() ❸, eggs is the global variable, because there is no assignment statement or global statement for it in that function. If you run *sameName3.py*, the output will look like this:

```
spam
```

In a function, a variable will either always be global or always be local. There's no way that the code in a function can use a local variable named eggs and then later in that same function use the global eggs variable.

NOTE *If you ever want to modify the value stored in a global variable from in a function, you must use a* `global` *statement on that variable.*

If you try to use a local variable in a function before you assign a value to it, as in the following program, Python will give you an error. To see this, type the following into the file editor and save it as *sameName4.py*:

```
def spam():
    print(eggs) # ERROR!
❶   eggs = 'spam local'

❷ eggs = 'global'
spam()
```

If you run the previous program, it produces an error message.

```
Traceback (most recent call last):
  File "C:/test3784.py", line 6, in <module>
    spam()
  File "C:/test3784.py", line 2, in spam
    print(eggs) # ERROR!
UnboundLocalError: local variable 'eggs' referenced before assignment
```

This error happens because Python sees that there is an assignment statement for eggs in the spam() function ❶ and therefore considers eggs to be local. But because print(eggs) is executed before eggs is assigned anything, the local variable eggs doesn't exist. Python will *not* fall back to using the global eggs variable ❷.

FUNCTIONS AS "BLACK BOXES"

Often, all you need to know about a function are its inputs (the parameters) and output value; you don't always have to burden yourself with how the function's code actually works. When you think about functions in this high-level way, it's common to say that you're treating the function as a "black box."

This idea is fundamental to modern programming. Later chapters in this book will show you several modules with functions that were written by other people. While you can take a peek at the source code if you're curious, you don't need to know how these functions work in order to use them. And because writing functions without global variables is encouraged, you usually don't have to worry about the function's code interacting with the rest of your program.

Exception Handling

Right now, getting an error, or *exception*, in your Python program means the entire program will crash. You don't want this to happen in real-world programs. Instead, you want the program to detect errors, handle them, and then continue to run.

For example, consider the following program, which has a "divide-by-zero" error. Open a new file editor window and enter the following code, saving it as *zeroDivide.py*:

```
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

We've defined a function called `spam`, given it a parameter, and then printed the value of that function with various parameters to see what happens. This is the output you get when you run the previous code:

```
21.0
3.5
Traceback (most recent call last):
  File "C:/zeroDivide.py", line 6, in <module>
    print(spam(0))
  File "C:/zeroDivide.py", line 2, in spam
    return 42 / divideBy
ZeroDivisionError: division by zero
```

A `ZeroDivisionError` happens whenever you try to divide a number by zero. From the line number given in the error message, you know that the `return` statement in `spam()` is causing an error.

Errors can be handled with try and except statements. The code that could potentially have an error is put in a try clause. The program execution moves to the start of a following except clause if an error happens.

You can put the previous divide-by-zero code in a try clause and have an except clause contain code to handle what happens when this error occurs.

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Invalid argument.')

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

When code in a try clause causes an error, the program execution immediately moves to the code in the except clause. After running that code, the execution continues as normal. The output of the previous program is as follows:

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

Note that any errors that occur in function calls in a try block will also be caught. Consider the following program, which instead has the spam() calls in the try block:

```
def spam(divideBy):
    return 42 / divideBy

try:
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
    print('Error: Invalid argument.')
```

When this program is run, the output looks like this:

```
21.0
3.5
Error: Invalid argument.
```

The reason `print(spam(1))` is never executed is because once the execution jumps to the code in the `except` clause, it does not return to the `try` clause. Instead, it just continues moving down as normal.

A Short Program: Guess the Number

The toy examples I've show you so far are useful for introducing basic concepts, but now let's see how everything you've learned comes together in a more complete program. In this section, I'll show you a simple "guess the number" game. When you run this program, the output will look something like this:

```
I am thinking of a number between 1 and 20.
Take a guess.
10
Your guess is too low.
Take a guess.
15
Your guess is too low.
Take a guess.
17
Your guess is too high.
Take a guess.
16
Good job! You guessed my number in 4 guesses!
```

Type the following source code into the file editor, and save the file as *guessTheNumber.py*:

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())

    if guess < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
    else:
        break    # This condition is the correct guess!

if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

Let's look at this code line by line, starting at the top.

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
```

First, a comment at the top of the code explains what the program does. Then, the program imports the `random` module so that it can use the `random.randint()` function to generate a number for the user to guess. The return value, a random integer between 1 and 20, is stored in the variable `secretNumber`.

```
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())
```

The program tells the player that it has come up with a secret number and will give the player six chances to guess it. The code that lets the player enter a guess and checks that guess is in a `for` loop that will loop at most six times. The first thing that happens in the loop is that the player types in a guess. Since `input()` returns a string, its return value is passed straight into `int()`, which translates the string into an integer value. This gets stored in a variable named `guess`.

```
    if guess < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
```

These few lines of code check to see whether the guess is less than or greater than the secret number. In either case, a hint is printed to the screen.

```
    else:
        break    # This condition is the correct guess!
```

If the guess is neither higher nor lower than the secret number, then it must be equal to the secret number, in which case you want the program execution to break out of the `for` loop.

```
if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

After the `for` loop, the previous `if...else` statement checks whether the player has correctly guessed the number and prints an appropriate message to the screen. In both cases, the program displays a variable that contains

an integer value (`guessesTaken` and `secretNumber`). Since it must concatenate these integer values to strings, it passes these variables to the `str()` function, which returns the string value form of these integers. Now these strings can be concatenated with the `+` operators before finally being passed to the `print()` function call.

Summary

Functions are the primary way to compartmentalize your code into logical groups. Since the variables in functions exist in their own local scopes, the code in one function cannot directly affect the values of variables in other functions. This limits what code could be changing the values of your variables, which can be helpful when it comes to debugging your code.

Functions are a great tool to help you organize your code. You can think of them as black boxes: They have inputs in the form of parameters and outputs in the form of return values, and the code in them doesn't affect variables in other functions.

In previous chapters, a single error could cause your programs to crash. In this chapter, you learned about `try` and `except` statements, which can run code when an error has been detected. This can make your programs more resilient to common error cases.

Practice Questions

1. Why are functions advantageous to have in your programs?
2. When does the code in a function execute: when the function is defined or when the function is called?
3. What statement creates a function?
4. What is the difference between a function and a function call?
5. How many global scopes are there in a Python program? How many local scopes?
6. What happens to variables in a local scope when the function call returns?
7. What is a return value? Can a return value be part of an expression?
8. If a function does not have a return statement, what is the return value of a call to that function?
9. How can you force a variable in a function to refer to the global variable?
10. What is the data type of `None`?
11. What does the `import areallyourpetsnamederic` statement do?
12. If you had a function named `bacon()` in a module named `spam`, how would you call it after importing `spam`?
13. How can you prevent a program from crashing when it gets an error?
14. What goes in the `try` clause? What goes in the `except` clause?

Practice Projects

For practice, write programs to do the following tasks.

The Collatz Sequence

Write a function named `collatz()` that has one parameter named `number`. If `number` is even, then `collatz()` should print `number // 2` and return this value. If `number` is odd, then `collatz()` should print and return `3 * number + 1`.

Then write a program that lets the user type in an integer and that keeps calling `collatz()` on that number until the function returns the value `1`. (Amazingly enough, this sequence actually works for any integer—sooner or later, using this sequence, you'll arrive at 1! Even mathematicians aren't sure why. Your program is exploring what's called the *Collatz sequence*, sometimes called "the simplest impossible math problem.")

Remember to convert the return value from `input()` to an integer with the `int()` function; otherwise, it will be a string value.

Hint: An integer `number` is even if `number % 2 == 0`, and it's odd if `number % 2 == 1`.

The output of this program could look something like this:

```
Enter number:
3
10
5
16
8
4
2
1
```

Input Validation

Add `try` and `except` statements to the previous project to detect whether the user types in a noninteger string. Normally, the `int()` function will raise a `ValueError` error if it is passed a noninteger string, as in `int('puppy')`. In the `except` clause, print a message to the user saying they must enter an integer.

4

LISTS

One more topic you'll need to understand before you can begin writing programs in earnest is the list data type and its cousin, the tuple. Lists and tuples can contain multiple values, which makes it easier to write programs that handle large amounts of data. And since lists themselves can contain other lists, you can use them to arrange data into hierarchical structures.

In this chapter, I'll discuss the basics of lists. I'll also teach you about methods, which are functions that are tied to values of a certain data type. Then I'll briefly cover the list-like tuple and string data types and how they compare to list values. In the next chapter, I'll introduce you to the dictionary data type.

The List Data Type

A *list* is a value that contains multiple values in an ordered sequence. The term *list value* refers to the list itself (which is a value that can be stored in a variable or passed to a function like any other value), not the values inside the list value. A list value looks like this: `['cat', 'bat', 'rat', 'elephant']`. Just as string values are typed with quote characters to mark where the string begins and ends, a list begins with an opening square bracket and ends with a closing square bracket, `[]`. Values inside the list are also called *items*. Items are separated with commas (that is, they are *comma-delimited*). For example, enter the following into the interactive shell:

```
>>> [1, 2, 3]
[1, 2, 3]
>>> ['cat', 'bat', 'rat', 'elephant']
['cat', 'bat', 'rat', 'elephant']
>>> ['hello', 3.1415, True, None, 42]
['hello', 3.1415, True, None, 42]
❶ >>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam
['cat', 'bat', 'rat', 'elephant']
```

The `spam` variable ❶ is still assigned only one value: the list value. But the list value itself contains other values. The value `[]` is an empty list that contains no values, similar to `''`, the empty string.

Getting Individual Values in a List with Indexes

Say you have the list `['cat', 'bat', 'rat', 'elephant']` stored in a variable named `spam`. The Python code `spam[0]` would evaluate to `'cat'`, and `spam[1]` would evaluate to `'bat'`, and so on. The integer inside the square brackets that follows the list is called an *index*. The first value in the list is at index `0`, the second value is at index `1`, the third value is at index `2`, and so on. Figure 4-1 shows a list value assigned to `spam`, along with what the index expressions would evaluate to.

Figure 4-1: A list value stored in the variable spam, showing which value each index refers to

For example, type the following expressions into the interactive shell. Start by assigning a list to the variable `spam`.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0]
'cat'
>>> spam[1]
'bat'
>>> spam[2]
'rat'
>>> spam[3]
'elephant'
```

```
>>> ['cat', 'bat', 'rat', 'elephant'][3]
'elephant'
❶ >>> 'Hello ' + spam[0]
❷ 'Hello cat'
>>> 'The ' + spam[1] + ' ate the ' + spam[0] + '.'
'The bat ate the cat.'
```

Notice that the expression `'Hello ' + spam[0]` ❶ evaluates to `'Hello ' + 'cat'` because `spam[0]` evaluates to the string `'cat'`. This expression in turn evaluates to the string value `'Hello cat'` ❷.

Python will give you an `IndexError` error message if you use an index that exceeds the number of values in your list value.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[10000]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    spam[10000]
IndexError: list index out of range
```

Indexes can be only integer values, not floats. The following example will cause a `TypeError` error:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1]
'bat'
>>> spam[1.0]
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    spam[1.0]
TypeError: list indices must be integers, not float
>>> spam[int(1.0)]
'bat'
```

Lists can also contain other list values. The values in these lists of lists can be accessed using multiple indexes, like so:

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
'bat'
>>> spam[1][4]
50
```

The first index dictates which list value to use, and the second indicates the value within the list value. For example, `spam[0][1]` prints `'bat'`, the second value in the first list. If you only use one index, the program will print the full list value at that index.

Negative Indexes

While indexes start at `0` and go up, you can also use negative integers for the index. The integer value `-1` refers to the last index in a list, the value `-2` refers to the second-to-last index in a list, and so on. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[-1]
'elephant'
>>> spam[-3]
'bat'
>>> 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'
'The elephant is afraid of the bat.'
```

Getting Sublists with Slices

Just as an index can get a single value from a list, a *slice* can get several values from a list, in the form of a new list. A slice is typed between square brackets, like an index, but it has two integers separated by a colon. Notice the difference between indexes and slices.

- `spam[2]` is a list with an index (one integer).
- `spam[1:4]` is a list with a slice (two integers).

In a slice, the first integer is the index where the slice starts. The second integer is the index where the slice ends. A slice goes up to, but will not include, the value at the second index. A slice evaluates to a new list value. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']
>>> spam[1:3]
['bat', 'rat']
>>> spam[0:-1]
['cat', 'bat', 'rat']
```

As a shortcut, you can leave out one or both of the indexes on either side of the colon in the slice. Leaving out the first index is the same as using `0`, or the beginning of the list. Leaving out the second index is the same as using the length of the list, which will slice to the end of the list. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
['cat', 'bat']
>>> spam[1:]
['bat', 'rat', 'elephant']
```

```
>>> spam[:]
['cat', 'bat', 'rat', 'elephant']
```

Getting a List's Length with len()

The `len()` function will return the number of values that are in a list value passed to it, just like it can count the number of characters in a string value. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'moose']
>>> len(spam)
3
```

Changing Values in a List with Indexes

Normally a variable name goes on the left side of an assignment statement, like `spam = 42`. However, you can also use an index of a list to change the value at that index. For example, `spam[1] = 'aardvark'` means "Assign the value at index `1` in the list `spam` to the string `'aardvark'`." Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1] = 'aardvark'
>>> spam
['cat', 'aardvark', 'rat', 'elephant']
>>> spam[2] = spam[1]
>>> spam
['cat', 'aardvark', 'aardvark', 'elephant']
>>> spam[-1] = 12345
>>> spam
['cat', 'aardvark', 'aardvark', 12345]
```

List Concatenation and List Replication

The + operator can combine two lists to create a new list value in the same way it combines two strings into a new string value. The * operator can also be used with a list and an integer value to replicate the list. Enter the following into the interactive shell:

```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']
>>> ['X', 'Y', 'Z'] * 3
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
>>> spam = [1, 2, 3]
>>> spam = spam + ['A', 'B', 'C']
>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

Removing Values from Lists with del Statements

The `del` statement will delete values at an index in a list. All of the values in the list after the deleted value will be moved up one index. For example, enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat']
```

The `del` statement can also be used on a simple variable to delete it, as if it were an "unassignment" statement. If you try to use the variable after deleting it, you will get a `NameError` error because the variable no longer exists.

In practice, you almost never need to delete simple variables. The `del` statement is mostly used to delete values from lists.

Working with Lists

When you first begin writing programs, it's tempting to create many individual variables to store a group of similar values. For example, if I wanted to store the names of my cats, I might be tempted to write code like this:

```
catName1 = 'Zophie'
catName2 = 'Pooka'
catName3 = 'Simon'
catName4 = 'Lady Macbeth'
catName5 = 'Fat-tail'
catName6 = 'Miss Cleo'
```

(I don't actually own this many cats, I swear.) It turns out that this is a bad way to write code. For one thing, if the number of cats changes, your program will never be able to store more cats than you have variables. These types of programs also have a lot of duplicate or nearly identical code in them. Consider how much duplicate code is in the following program, which you should enter into the file editor and save as *allMyCats1.py*:

```
print('Enter the name of cat 1:')
catName1 = input()
print('Enter the name of cat 2:')
catName2 = input()
print('Enter the name of cat 3:')
catName3 = input()
print('Enter the name of cat 4:')
catName4 = input()
print('Enter the name of cat 5:')
catName5 = input()
```

```
print('Enter the name of cat 6:')
catName6 = input()
print('The cat names are:')
print(catName1 + ' ' + catName2 + ' ' + catName3 + ' ' + catName4 + ' ' +
catName5 + ' ' + catName6)
```

Instead of using multiple, repetitive variables, you can use a single variable that contains a list value. For example, here's a new and improved version of the *allMyCats1.py* program. This new version uses a single list and can store any number of cats that the user types in. In a new file editor window, type the following source code and save it as *allMyCats2.py*:

```
catNames = []
while True:
    print('Enter the name of cat ' + str(len(catNames) + 1) +
      ' (Or enter nothing to stop.):')
    name = input()
    if name == '':
        break
    catNames = catNames + [name]  # list concatenation
print('The cat names are:')
for name in catNames:
    print('  ' + name)
```

When you run this program, the output will look something like this:

```
Enter the name of cat 1 (Or enter nothing to stop.):
Zophie
Enter the name of cat 2 (Or enter nothing to stop.):
Pooka
Enter the name of cat 3 (Or enter nothing to stop.):
Simon
Enter the name of cat 4 (Or enter nothing to stop.):
Lady Macbeth
Enter the name of cat 5 (Or enter nothing to stop.):
Fat-tail
Enter the name of cat 6 (Or enter nothing to stop.):
Miss Cleo
Enter the name of cat 7 (Or enter nothing to stop.):

The cat names are:
  Zophie
  Pooka
  Simon
  Lady Macbeth
  Fat-tail
  Miss Cleo
```

The benefit of using a list is that your data is now in a structure, so your program is much more flexible in processing the data than it would be with several repetitive variables.

Using for Loops with Lists

In Chapter 2, you learned about using `for` loops to execute a block of code a certain number of times. Technically, a `for` loop repeats the code block once for each value in a list or list-like value. For example, if you ran this code:

```
for i in range(4):
    print(i)
```

the output of this program would be as follows:

```
0
1
2
3
```

This is because the return value from `range(4)` is a list-like value that Python considers similar to `[0, 1, 2, 3]`. The following program has the same output as the previous one:

```
for i in [0, 1, 2, 3]:
    print(i)
```

What the previous `for` loop actually does is loop through its clause with the variable `i` set to a successive value in the `[0, 1, 2, 3]` list in each iteration.

NOTE *In this book, I use the term* list-like *to refer to data types that are technically named* sequences. *You don't need to know the technical definitions of this term, though.*

A common Python technique is to use `range(len(someList))` with a `for` loop to iterate over the indexes of a list. For example, enter the following into the interactive shell:

```
>>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
>>> for i in range(len(supplies)):
    print('Index ' + str(i) + ' in supplies is: ' + supplies[i])

Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flame-throwers
Index 3 in supplies is: binders
```

Using `range(len(supplies))` in the previously shown `for` loop is handy because the code in the loop can access the index (as the variable `i`) and the value at that index (as `supplies[i]`). Best of all, `range(len(supplies))` will iterate through all the indexes of `supplies`, no matter how many items it contains.

The in and not in Operators

You can determine whether a value is or isn't in a list with the `in` and `not in` operators. Like other operators, `in` and `not in` are used in expressions and connect two values: a value to look for in a list and the list where it may be found. These expressions will evaluate to a Boolean value. Enter the following into the interactive shell:

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
True
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> 'cat' in spam
False
>>> 'howdy' not in spam
False
>>> 'cat' not in spam
True
```

For example, the following program lets the user type in a pet name and then checks to see whether the name is in a list of pets. Open a new file editor window, enter the following code, and save it as *myPets.py*:

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')
name = input()
if name not in myPets:
    print('I do not have a pet named ' + name)
else:
    print(name + ' is my pet.')
```

The output may look something like this:

```
Enter a pet name:
Footfoot
I do not have a pet named Footfoot
```

The Multiple Assignment Trick

The *multiple assignment trick* is a shortcut that lets you assign multiple variables with the values in a list in one line of code. So instead of doing this:

```
>>> cat = ['fat', 'black', 'loud']
>>> size = cat[0]
>>> color = cat[1]
>>> disposition = cat[2]
```

you could type this line of code:

```
>>> cat = ['fat', 'black', 'loud']
>>> size, color, disposition = cat
```

The number of variables and the length of the list must be exactly equal, or Python will give you a ValueError:

```
>>> cat = ['fat', 'black', 'loud']
>>> size, color, disposition, name = cat
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    size, color, disposition, name = cat
ValueError: need more than 3 values to unpack
```

Augmented Assignment Operators

When assigning a value to a variable, you will frequently use the variable itself. For example, after assigning 42 to the variable spam, you would increase the value in spam by 1 with the following code:

```
>>> spam = 42
>>> spam = spam + 1
>>> spam
43
```

As a shortcut, you can use the augmented assignment operator += to do the same thing:

```
>>> spam = 42
>>> spam += 1
>>> spam
43
```

There are augmented assignment operators for the +, -, *, /, and % operators, described in Table 4-1.

Table 4-1: The Augmented Assignment Operators

Augmented assignment statement	Equivalent assignment statement
spam = spam + 1	spam += 1
spam = spam - 1	spam -= 1
spam = spam * 1	spam *= 1
spam = spam / 1	spam /= 1
spam = spam % 1	spam %= 1

The += operator can also do string and list concatenation, and the *= operator can do string and list replication. Enter the following into the interactive shell:

```
>>> spam = 'Hello'
>>> spam += ' world!'
>>> spam
'Hello world!'
```

```
>>> bacon = ['Zophie']
>>> bacon *= 3
>>> bacon
['Zophie', 'Zophie', 'Zophie']
```

Methods

A *method* is the same thing as a function, except it is "called on" a value. For example, if a list value were stored in `spam`, you would call the `index()` list method (which I'll explain next) on that list like so: `spam.index('hello')`. The method part comes after the value, separated by a period.

Each data type has its own set of methods. The list data type, for example, has several useful methods for finding, adding, removing, and otherwise manipulating values in a list.

Finding a Value in a List with the index() Method

List values have an `index()` method that can be passed a value, and if that value exists in the list, the index of the value is returned. If the value isn't in the list, then Python produces a `ValueError` error. Enter the following into the interactive shell:

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
>>> spam.index('heyas')
3
>>> spam.index('howdy howdy howdy')
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    spam.index('howdy howdy howdy')
ValueError: 'howdy howdy howdy' is not in list
```

When there are duplicates of the value in the list, the index of its first appearance is returned. Enter the following into the interactive shell, and notice that `index()` returns `1`, not `3`:

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
>>> spam.index('Pooka')
1
```

Adding Values to Lists with the append() and insert() Methods

To add new values to a list, use the `append()` and `insert()` methods. Enter the following into the interactive shell to call the `append()` method on a list value stored in the variable `spam`:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.append('moose')
```

```
>>> spam
['cat', 'dog', 'bat', 'moose']
```

The previous `append()` method call adds the argument to the end of the list. The `insert()` method can insert a value at any index in the list. The first argument to `insert()` is the index for the new value, and the second argument is the new value to be inserted. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.insert(1, 'chicken')
>>> spam
['cat', 'chicken', 'dog', 'bat']
```

Notice that the code is `spam.append('moose')` and `spam.insert(1, 'chicken')`, not `spam = spam.append('moose')` and `spam = spam.insert(1, 'chicken')`. Neither `append()` nor `insert()` gives the new value of `spam` as its return value. (In fact, the return value of `append()` and `insert()` is `None`, so you definitely wouldn't want to store this as the new variable value.) Rather, the list is modified *in place*. Modifying a list in place is covered in more detail later in "Mutable and Immutable Data Types" on page 94.

Methods belong to a single data type. The `append()` and `insert()` methods are list methods and can be called only on list values, not on other values such as strings or integers. Enter the following into the interactive shell, and note the `AttributeError` error messages that show up:

```
>>> eggs = 'hello'
>>> eggs.append('world')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    eggs.append('world')
AttributeError: 'str' object has no attribute 'append'
>>> bacon = 42
>>> bacon.insert(1, 'world')
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    bacon.insert(1, 'world')
AttributeError: 'int' object has no attribute 'insert'
```

Removing Values from Lists with remove()

The `remove()` method is passed the value to be removed from the list it is called on. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('bat')
>>> spam
['cat', 'rat', 'elephant']
```

Attempting to delete a value that does not exist in the list will result in a `ValueError` error. For example, enter the following into the interactive shell and notice the error that is displayed:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('chicken')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    spam.remove('chicken')
ValueError: list.remove(x): x not in list
```

If the value appears multiple times in the list, only the first instance of the value will be removed. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'cat', 'hat', 'cat']
>>> spam.remove('cat')
>>> spam
['bat', 'rat', 'cat', 'hat', 'cat']
```

The `del` statement is good to use when you know the index of the value you want to remove from the list. The `remove()` method is good when you know the value you want to remove from the list.

Sorting the Values in a List with the sort() Method

Lists of number values or lists of strings can be sorted with the `sort()` method. For example, enter the following into the interactive shell:

```
>>> spam = [2, 5, 3.14, 1, -7]
>>> spam.sort()
>>> spam
[-7, 1, 2, 3.14, 5]
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
>>> spam.sort()
>>> spam
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

You can also pass `True` for the `reverse` keyword argument to have `sort()` sort the values in reverse order. Enter the following into the interactive shell:

```
>>> spam.sort(reverse=True)
>>> spam
['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

There are three things you should note about the `sort()` method. First, the `sort()` method sorts the list in place; don't try to capture the return value by writing code like `spam = spam.sort()`.

Second, you cannot sort lists that have both number values *and* string values in them, since Python doesn't know how to compare these values. Type the following into the interactive shell and notice the `TypeError` error:

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
>>> spam.sort()
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    spam.sort()
TypeError: unorderable types: str() < int()
```

Third, `sort()` uses "ASCIIbetical order" rather than actual alphabetical order for sorting strings. This means uppercase letters come before lowercase letters. Therefore, the lowercase *a* is sorted so that it comes *after* the uppercase *Z*. For an example, enter the following into the interactive shell:

```
>>> spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']
>>> spam.sort()
>>> spam
['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

If you need to sort the values in regular alphabetical order, pass `str.lower` for the `key` keyword argument in the `sort()` method call.

```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam
['a', 'A', 'z', 'Z']
```

This causes the `sort()` function to treat all the items in the list as if they were lowercase without actually changing the values in the list.

Example Program: Magic 8 Ball with a List

Using lists, you can write a much more elegant version of the previous chapter's Magic 8 Ball program. Instead of several lines of nearly identical `elif` statements, you can create a single list that the code works with. Open a new file editor window and enter the following code. Save it as *magic8Ball2.py*.

```
import random

messages = ['It is certain',
    'It is decidedly so',
    'Yes definitely',
    'Reply hazy try again',
    'Ask again later',
    'Concentrate and ask again',
    'My reply is no',
    'Outlook not so good',
    'Very doubtful']

print(messages[random.randint(0, len(messages) - 1)])
```

EXCEPTIONS TO INDENTATION RULES IN PYTHON

In most cases, the amount of indentation for a line of code tells Python what block it is in. There are some exceptions to this rule, however. For example, lists can actually span several lines in the source code file. The indentation of these lines do not matter; Python knows that until it sees the ending square bracket, the list is not finished. For example, you can have code that looks like this:

```
spam = ['apples',
    'oranges',
                    'bananas',
'cats']
print(spam)
```

Of course, practically speaking, most people use Python's behavior to make their lists look pretty and readable, like the messages list in the Magic 8 Ball program.

You can also split up a single instruction across multiple lines using the \ line continuation character at the end. Think of \ as saying, "This instruction continues on the next line." The indentation on the line after a \ line continuation is not significant. For example, the following is valid Python code:

```
print('Four score and seven ' + \
      'years ago...')
```

These tricks are useful when you want to rearrange long lines of Python code to be a bit more readable.

When you run this program, you'll see that it works the same as the previous *magic8Ball.py* program.

Notice the expression you use as the index into `messages`: `random.randint(0, len(messages) - 1)`. This produces a random number to use for the index, regardless of the size of `messages`. That is, you'll get a random number between `0` and the value of `len(messages) - 1`. The benefit of this approach is that you can easily add and remove strings to the `messages` list without changing other lines of code. If you later update your code, there will be fewer lines you have to change and fewer chances for you to introduce bugs.

List-like Types: Strings and Tuples

Lists aren't the only data types that represent ordered sequences of values. For example, strings and lists are actually similar, if you consider a string to be a "list" of single text characters. Many of the things you can do with lists

can also be done with strings: indexing; slicing; and using them with `for` loops, with `len()`, and with the `in` and `not in` operators. To see this, enter the following into the interactive shell:

```
>>> name = 'Zophie'
>>> name[0]
'Z'
>>> name[-2]
'i'
>>> name[0:4]
'Zoph'
>>> 'Zo' in name
True
>>> 'z' in name
False
>>> 'p' not in name
False
>>> for i in name:
        print('* * * ' + i + ' * * *')


* * * Z * * *
* * * o * * *
* * * p * * *
* * * h * * *
* * * i * * *
* * * e * * *
```

Mutable and Immutable Data Types

But lists and strings are different in an important way. A list value is a *mutable* data type: It can have values added, removed, or changed. However, a string is *immutable*: It cannot be changed. Trying to reassign a single character in a string results in a `TypeError` error, as you can see by entering the following into the interactive shell:

```
>>> name = 'Zophie a cat'
>>> name[7] = 'the'
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    name[7] = 'the'
TypeError: 'str' object does not support item assignment
```

The proper way to "mutate" a string is to use slicing and concatenation to build a *new* string by copying from parts of the old string. Enter the following into the interactive shell:

```
>>> name = 'Zophie a cat'
>>> newName = name[0:7] + 'the' + name[8:12]
>>> name
'Zophie a cat'
```

```
>>> newName
'Zophie the cat'
```

We used [0:7] and [8:12] to refer to the characters that we don't wish to replace. Notice that the original 'Zophie a cat' string is not modified because strings are immutable.

Although a list value *is* mutable, the second line in the following code does not modify the list eggs:

```
>>> eggs = [1, 2, 3]
>>> eggs = [4, 5, 6]
>>> eggs
[4, 5, 6]
```

The list value in eggs isn't being changed here; rather, an entirely new and different list value ([4, 5, 6]) is overwriting the old list value ([1, 2, 3]). This is depicted in Figure 4-2.

If you wanted to actually modify the original list in eggs to contain [4, 5, 6], you would have to do something like this:

```
>>> eggs = [1, 2, 3]
>>> del eggs[2]
>>> del eggs[1]
>>> del eggs[0]
>>> eggs.append(4)
>>> eggs.append(5)
>>> eggs.append(6)
>>> eggs
[4, 5, 6]
```

Figure 4-2: When eggs = [4, 5, 6] is executed, the contents of eggs are replaced with a new list value.

In the first example, the list value that eggs ends up with is the same list value it started with. It's just that this list has been changed, rather than overwritten. Figure 4-3 depicts the seven changes made by the first seven lines in the previous interactive shell example.

Figure 4-3: The `del` statement and the `append()` method modify the same list value in place.

Changing a value of a mutable data type (like what the `del` statement and `append()` method do in the previous example) changes the value in place, since the variable's value is not replaced with a new list value.

Mutable versus immutable types may seem like a meaningless distinction, but "Passing References" on page 100 will explain the different behavior when calling functions with mutable arguments versus immutable arguments. But first, let's find out about the tuple data type, which is an immutable form of the list data type.

The Tuple Data Type

The *tuple* data type is almost identical to the list data type, except in two ways. First, tuples are typed with parentheses, (and), instead of square brackets, [and]. For example, enter the following into the interactive shell:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0]
'hello'
>>> eggs[1:3]
(42, 0.5)
>>> len(eggs)
3
```

But the main way that tuples are different from lists is that tuples, like strings, are immutable. Tuples cannot have their values modified, appended, or removed. Enter the following into the interactive shell, and look at the `TypeError` error message:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[1] = 99
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    eggs[1] = 99
TypeError: 'tuple' object does not support item assignment
```

If you have only one value in your tuple, you can indicate this by placing a trailing comma after the value inside the parentheses. Otherwise, Python will think you've just typed a value inside regular parentheses. The comma is what lets Python know this is a tuple value. (Unlike some other programming languages, in Python it's fine to have a trailing comma after the last item in a list or tuple.) Enter the following `type()` function calls into the interactive shell to see the distinction:

```
>>> type(('hello',))
<class 'tuple'>
>>> type(('hello'))
<class 'str'>
```

You can use tuples to convey to anyone reading your code that you don't intend for that sequence of values to change. If you need an ordered sequence of values that never changes, use a tuple. A second benefit of using tuples instead of lists is that, because they are immutable and their contents don't change, Python can implement some optimizations that make code using tuples slightly faster than code using lists.

Converting Types with the list() and tuple() Functions

Just like how `str(42)` will return `'42'`, the string representation of the integer `42`, the functions `list()` and `tuple()` will return list and tuple versions of the values passed to them. Enter the following into the interactive shell, and notice that the return value is of a different data type than the value passed:

```
>>> tuple(['cat', 'dog', 5])
('cat', 'dog', 5)
>>> list(('cat', 'dog', 5))
['cat', 'dog', 5]
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

Converting a tuple to a list is handy if you need a mutable version of a tuple value.

References

As you've seen, variables store strings and integer values. Enter the following into the interactive shell:

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

You assign 42 to the spam variable, and then you copy the value in spam and assign it to the variable cheese. When you later change the value in spam to 100, this doesn't affect the value in cheese. This is because spam and cheese are different variables that store different values.

But lists don't work this way. When you assign a list to a variable, you are actually assigning a list *reference* to the variable. A reference is a value that points to some bit of data, and a list reference is a value that points to a list. Here is some code that will make this distinction easier to understand. Enter this into the interactive shell:

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> cheese = spam
❸ >>> cheese[1] = 'Hello!'
  >>> spam
  [0, 'Hello!', 2, 3, 4, 5]
  >>> cheese
  [0, 'Hello!', 2, 3, 4, 5]
```

This might look odd to you. The code changed only the cheese list, but it seems that both the cheese and spam lists have changed.

When you create the list ❶, you assign a reference to it in the spam variable. But the next line ❷ copies only the list reference in spam to cheese, not the list value itself. This means the values stored in spam and cheese now both refer to the same list. There is only one underlying list because the list itself was never actually copied. So when you modify the first element of cheese ❸, you are modifying the same list that spam refers to.

Remember that variables are like boxes that contain values. The previous figures in this chapter show that lists in boxes aren't exactly accurate because list variables don't actually contain lists—they contain *references* to lists. (These references will have ID numbers that Python uses internally, but you can ignore them.) Using boxes as a metaphor for variables, Figure 4-4 shows what happens when a list is assigned to the spam variable.

Figure 4-4: spam = [0, 1, 2, 3, 4, 5] stores a reference to a list, not the actual list.

Then, in Figure 4-5, the reference in spam is copied to cheese. Only a new reference was created and stored in cheese, not a new list. Note how both references refer to the same list.

Figure 4-5: spam = cheese copies the reference, not the list.

When you alter the list that cheese refers to, the list that spam refers to is also changed, because both cheese and spam refer to the same list. You can see this in Figure 4-6.

Figure 4-6: cheese[1] = 'Hello!' modifies the list that both variables refer to.

Variables will contain references to list values rather than list values themselves. But for strings and integer values, variables simply contain the string or integer value. Python uses references whenever variables must store values of mutable data types, such as lists or dictionaries. For values of immutable data types such as strings, integers, or tuples, Python variables will store the value itself.

Although Python variables technically contain references to list or dictionary values, people often casually say that the variable contains the list or dictionary.

Passing References

References are particularly important for understanding how arguments get passed to functions. When a function is called, the values of the arguments are copied to the parameter variables. For lists (and dictionaries, which I'll describe in the next chapter), this means a copy of the reference is used for the parameter. To see the consequences of this, open a new file editor window, enter the following code, and save it as *passingReference.py*:

```
def eggs(someParameter):
    someParameter.append('Hello')

spam = [1, 2, 3]
eggs(spam)
print(spam)
```

Notice that when `eggs()` is called, a return value is not used to assign a new value to `spam`. Instead, it modifies the list in place, directly. When run, this program produces the following output:

```
[1, 2, 3, 'Hello']
```

Even though `spam` and `someParameter` contain separate references, they both refer to the same list. This is why the `append('Hello')` method call inside the function affects the list even after the function call has returned.

Keep this behavior in mind: Forgetting that Python handles list and dictionary variables this way can lead to confusing bugs.

The copy Module's copy() and deepcopy() Functions

Although passing around references is often the handiest way to deal with lists and dictionaries, if the function modifies the list or dictionary that is passed, you may not want these changes in the original list or dictionary value. For this, Python provides a module named `copy` that provides both the `copy()` and `deepcopy()` functions. The first of these, `copy.copy()`, can be used to make a duplicate copy of a mutable value like a list or dictionary, not just a copy of a reference. Enter the following into the interactive shell:

```
>>> import copy
>>> spam = ['A', 'B', 'C', 'D']
>>> cheese = copy.copy(spam)
>>> cheese[1] = 42
>>> spam
['A', 'B', 'C', 'D']
>>> cheese
['A', 42, 'C', 'D']
```

Now the spam and cheese variables refer to separate lists, which is why only the list in cheese is modified when you assign 42 at index 7. As you can see in Figure 4-7, the reference ID numbers are no longer the same for both variables because the variables refer to independent lists.

Figure 4-7: cheese = copy.copy(spam) creates a second list that can be modified independently of the first.

If the list you need to copy contains lists, then use the copy.deepcopy() function instead of copy.copy(). The deepcopy() function will copy these inner lists as well.

Summary

Lists are useful data types since they allow you to write code that works on a modifiable number of values in a single variable. Later in this book, you will see programs using lists to do things that would be difficult or impossible to do without them.

Lists are mutable, meaning that their contents can change. Tuples and strings, although list-like in some respects, are immutable and cannot be changed. A variable that contains a tuple or string value can be overwritten with a new tuple or string value, but this is not the same thing as modifying the existing value in place—like, say, the append() or remove() methods do on lists.

Variables do not store list values directly; they store *references* to lists. This is an important distinction when copying variables or passing lists as arguments in function calls. Because the value that is being copied is the list reference, be aware that any changes you make to the list might impact another variable in your program. You can use copy() or deepcopy() if you want to make changes to a list in one variable without modifying the original list.

Practice Questions

1. What is `[]`?
2. How would you assign the value `'hello'` as the third value in a list stored in a variable named `spam`? (Assume `spam` contains `[2, 4, 6, 8, 10]`.)

For the following three questions, let's say `spam` contains the list `['a', 'b', 'c', 'd']`.

3. What does `spam[int('3' * 2) / 11]` evaluate to?
4. What does `spam[-1]` evaluate to?
5. What does `spam[:2]` evaluate to?

For the following three questions, let's say `bacon` contains the list `[3.14, 'cat', 11, 'cat', True]`.

6. What does `bacon.index('cat')` evaluate to?
7. What does `bacon.append(99)` make the list value in `bacon` look like?
8. What does `bacon.remove('cat')` make the list value in `bacon` look like?
9. What are the operators for list concatenation and list replication?
10. What is the difference between the `append()` and `insert()` list methods?
11. What are two ways to remove values from a list?
12. Name a few ways that list values are similar to string values.
13. What is the difference between lists and tuples?
14. How do you type the tuple value that has just the integer value `42` in it?
15. How can you get the tuple form of a list value? How can you get the list form of a tuple value?
16. Variables that "contain" list values don't actually contain lists directly. What do they contain instead?
17. What is the difference between `copy.copy()` and `copy.deepcopy()`?

Practice Projects

For practice, write programs to do the following tasks.

Comma Code

Say you have a list value like this:

```
spam = ['apples', 'bananas', 'tofu', 'cats']
```

Write a function that takes a list value as an argument and returns a string with all the items separated by a comma and a space, with *and* inserted before the last item. For example, passing the previous `spam` list to the function would return `'apples, bananas, tofu, and cats'`. But your function should be able to work with any list value passed to it.

Character Picture Grid

Say you have a list of lists where each value in the inner lists is a one-character string, like this:

```
grid = [['.', '.', '.', '.', '.', '.'],
        ['.', 'O', 'O', '.', '.', '.'],
        ['O', 'O', 'O', 'O', '.', '.'],
        ['O', 'O', 'O', 'O', 'O', '.'],
        ['.', 'O', 'O', 'O', 'O', 'O'],
        ['O', 'O', 'O', 'O', 'O', '.'],
        ['O', 'O', 'O', 'O', '.', '.'],
        ['.', 'O', 'O', '.', '.', '.'],
        ['.', '.', '.', '.', '.', '.']]
```

You can think of `grid[x][y]` as being the character at the x- and y-coordinates of a "picture" drawn with text characters. The `(0, 0)` origin will be in the upper-left corner, the x-coordinates increase going right, and w the y-coordinates increase going down.

Copy the previous grid value, and write code that uses it to print the image.

```
..OO.OO..
.OOOOOOO.
.OOOOOOO.
..OOOOO..
...OOO...
....O....
```

Hint: You will need to use a loop in a loop in order to print `grid[0][0]`, then `grid[1][0]`, then `grid[2][0]`, and so on, up to `grid[8][0]`. This will finish the first row, so then print a newline. Then your program should print `grid[0][1]`, then `grid[1][1]`, then `grid[2][1]`, and so on. The last thing your program will print is `grid[8][5]`.

Also, remember to pass the `end` keyword argument to `print()` if you don't want a newline printed automatically after each `print()` call.

5

DICTIONARIES AND STRUCTURING DATA

In this chapter, I will cover the dictionary data type, which provides a flexible way to access and organize data. Then, combining dictionaries with your knowledge of lists from the previous chapter, you'll learn how to create a data structure to model a tic-tac-toe board.

The Dictionary Data Type

Like a list, a *dictionary* is a collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called *keys*, and a key with its associated value is called a *key-value pair*.

In code, a dictionary is typed with braces, {}. Enter the following into the interactive shell:

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

This assigns a dictionary to the `myCat` variable. This dictionary's keys are `'size'`, `'color'`, and `'disposition'`. The values for these keys are `'fat'`, `'gray'`, and `'loud'`, respectively. You can access these values through their keys:

```
>>> myCat['size']
'fat'
>>> 'My cat has ' + myCat['color'] + ' fur.'
'My cat has gray fur.'
```

Dictionaries can still use integer values as keys, just like lists use integers for indexes, but they do not have to start at `0` and can be any number.

```
>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}
```

Dictionaries vs. Lists

Unlike lists, items in dictionaries are unordered. The first item in a list named `spam` would be `spam[0]`. But there is no "first" item in a dictionary. While the order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary. Enter the following into the interactive shell:

```
>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon
False
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True
```

Because dictionaries are not ordered, they can't be sliced like lists.

Trying to access a key that does not exist in a dictionary will result in a `KeyError` error message, much like a list's "out-of-range" `IndexError` error message. Enter the following into the interactive shell, and notice the error message that shows up because there is no `'color'` key:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    spam['color']
KeyError: 'color'
```

Though dictionaries are not ordered, the fact that you can have arbitrary values for the keys allows you to organize your data in powerful ways. Say you wanted your program to store data about your friends' birthdays. You can use a dictionary with the names as keys and the birthdays as values. Open a new file editor window and enter the following code. Save it as *birthdays.py*.

```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

  while True:
      print('Enter a name: (blank to quit)')
      name = input()
      if name == '':
          break

❷     if name in birthdays:
❸         print(birthdays[name] + ' is the birthday of ' + name)
      else:
          print('I do not have birthday information for ' + name)
          print('What is their birthday?')
          bday = input()
❹         birthdays[name] = bday
          print('Birthday database updated.')
```

You create an initial dictionary and store it in `birthdays` ❶. You can see if the entered name exists as a key in the dictionary with the `in` keyword ❷, just as you did for lists. If the name is in the dictionary, you access the associated value using square brackets ❸; if not, you can add it using the same square bracket syntax combined with the assignment operator ❹.

When you run this program, it will look like this:

```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve
I do not have birthday information for Eve
What is their birthday?
Dec 5
Birthday database updated.
Enter a name: (blank to quit)
Eve
Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

Of course, all the data you enter in this program is forgotten when the program terminates. You'll learn how to save data to files on the hard drive in Chapter 8.

The keys(), values(), and items() Methods

There are three dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values: `keys()`, `values()`, and `items()`. The values returned by these methods are not true lists: They cannot be modified and do not have an `append()` method. But these data types (`dict_keys`,

dict_values, and dict_items, respectively) *can* be used in for loops. To see how these methods work, enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
        print(v)

red
42
```

Here, a for loop iterates over each of the values in the spam dictionary. A for loop can also iterate over the keys or both keys and values:

```
>>> for k in spam.keys():
        print(k)

color
age
>>> for i in spam.items():
        print(i)

('color', 'red')
('age', 42)
```

Using the keys(), values(), and items() methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively. Notice that the values in the dict_items value returned by the items() method are tuples of the key and value.

If you want a true list from one of these methods, pass its list-like return value to the list() function. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()
dict_keys(['color', 'age'])
>>> list(spam.keys())
['color', 'age']
```

The list(spam.keys()) line takes the dict_keys value returned from keys() and passes it to list(), which then returns a list value of ['color', 'age'].

You can also use the multiple assignment trick in a for loop to assign the key and value to separate variables. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
        print('Key: ' + k + ' Value: ' + str(v))

Key: age Value: 42
Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

Recall from the previous chapter that the `in` and `not in` operators can check whether a value exists in a list. You can also use these operators to see whether a certain key or value exists in a dictionary. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
False
```

In the previous example, notice that `'color' in spam` is essentially a shorter version of writing `'color' in spam.keys()`. This is always the case: If you ever want to check whether a value is (or isn't) a key in the dictionary, you can simply use the `in` (or `not in`) keyword with the dictionary value itself.

The get() Method

It's tedious to check whether a key exists in a dictionary before accessing that key's value. Fortunately, dictionaries have a `get()` method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.

Enter the following into the interactive shell:

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

Because there is no `'eggs'` key in the `picnicItems` dictionary, the default value `0` is returned by the `get()` method. Without using `get()`, the code would have caused an error message, such as in the following example:

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
KeyError: 'eggs'
```

The setdefault() Method

You'll often have to set a value in a dictionary for a certain key only if that key does not already have a value. The code looks something like this:

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

The `setdefault()` method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist. If the key does exist, the `setdefault()` method returns the key's value. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

The first time `setdefault()` is called, the dictionary in `spam` changes to `{'color': 'black', 'age': 5, 'name': 'Pooka'}`. The method returns the value `'black'` because this is now the value set for the key `'color'`. When `spam.setdefault('color', 'white')` is called next, the value for that key is *not* changed to `'white'` because `spam` already has a key named `'color'`.

The `setdefault()` method is a nice shortcut to ensure that a key exists. Here is a short program that counts the number of occurrences of each letter in a string. Open the file editor window and enter the following code, saving it as *characterCount.py*:

```
message = 'It was a bright cold day in April, and the clocks were striking thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

print(count)
```

The program loops over each character in the `message` variable's string, counting how often each character appears. The `setdefault()` method call ensures that the key is in the `count` dictionary (with a default value of `0`)

so the program doesn't throw a `KeyError` error when `count[character] = count[character] + 1` is executed. When you run this program, the output will look like this:

```
{' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, 'd': 3, 'g': 2, 'i':
6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6, 'w': 2, 'y': 1}
```

From the output, you can see that the lowercase letter *c* appears 3 times, the space character appears 13 times, and the uppercase letter *A* appears 1 time. This program will work no matter what string is inside the `message` variable, even if the string is millions of characters long!

Pretty Printing

If you import the `pprint` module into your programs, you'll have access to the `pprint()` and `pformat()` functions that will "pretty print" a dictionary's values. This is helpful when you want a cleaner display of the items in a dictionary than what `print()` provides. Modify the previous *characterCount.py* program and save it as *prettyCharacterCount.py*.

```
import pprint
message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

pprint.pprint(count)
```

This time, when the program is run, the output looks much cleaner, with the keys sorted.

```
{' ': 13,
 ',': 1,
 '.': 1,
 'A': 1,
 'I': 1,
 'a': 4,
 'b': 1,
 'c': 3,
 'd': 3,
 'e': 5,
 'g': 2,
 'h': 3,
 'i': 6,
```

```
 'k': 2,
 'l': 3,
 'n': 4,
 'o': 2,
 'p': 1,
 'r': 5,
 's': 3,
 't': 6,
 'w': 2,
 'y': 1}
```

The `pprint.pprint()` function is especially helpful when the dictionary itself contains nested lists or dictionaries.

If you want to obtain the prettified text as a string value instead of displaying it on the screen, call `pprint.pformat()` instead. These two lines are equivalent to each other:

```
pprint.pprint(someDictionaryValue)
print(pprint.pformat(someDictionaryValue))
```

Using Data Structures to Model Real-World Things

Even before the Internet, it was possible to play a game of chess with someone on the other side of the world. Each player would set up a chessboard at their home and then take turns mailing a postcard to each other describing each move. To do this, the players needed a way to unambiguously describe the state of the board and their moves.

In *algebraic chess notation*, the spaces on the chessboard are identified by a number and letter coordinate, as in Figure 5-1.

Figure 5-1: The coordinates of a chessboard in algebraic chess notation

The chess pieces are identified by letters: *K* for king, *Q* for queen, *R* for rook, *B* for bishop, and *N* for knight. Describing a move uses the letter of the piece and the coordinates of its destination. A pair of these moves describes

what happens in a single turn (with white going first); for instance, the notation *2. Nf3 Nc6* indicates that white moved a knight to f3 and black moved a knight to c6 on the second turn of the game.

There's a bit more to algebraic notation than this, but the point is that you can use it to unambiguously describe a game of chess without needing to be in front of a chessboard. Your opponent can even be on the other side of the world! In fact, you don't even need a physical chess set if you have a good memory: You can just read the mailed chess moves and update boards you have in your imagination.

Computers have good memories. A program on a modern computer can easily store billions of strings like `'2. Nf3 Nc6'`. This is how computers can play chess without having a physical chessboard. They model data to represent a chessboard, and you can write code to work with this model.

This is where lists and dictionaries can come in. You can use them to model real-world things, like chessboards. For the first example, you'll use a game that's a little simpler than chess: tic-tac-toe.

A Tic-Tac-Toe Board

A tic-tac-toe board looks like a large hash symbol (#) with nine slots that can each contain an *X*, an *O*, or a blank. To represent the board with a dictionary, you can assign each slot a string-value key, as shown in Figure 5-2.

Figure 5-2: The slots of a tic-tac-toe board with their corresponding keys

You can use string values to represent what's in each slot on the board: `'X'`, `'O'`, or `' '` (a space character). Thus, you'll need to store nine strings. You can use a dictionary of values for this. The string value with the key `'top-R'` can represent the top-right corner, the string value with the key `'low-L'` can represent the bottom-left corner, the string value with the key `'mid-M'` can represent the middle, and so on.

This dictionary is a data structure that represents a tic-tac-toe board. Store this board-as-a-dictionary in a variable named `theBoard`. Open a new file editor window, and enter the following source code, saving it as *ticTacToe.py*:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

The data structure stored in the `theBoard` variable represents the tic-tac-toe board in Figure 5-3.

Figure 5-3: An empty tic-tac-toe board

Since the value for every key in `theBoard` is a single-space string, this dictionary represents a completely clear board. If player X went first and chose the middle space, you could represent that board with this dictionary:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

The data structure in `theBoard` now represents the tic-tac-toe board in Figure 5-4.

Figure 5-4: The first move

A board where player O has won by placing *O*s across the top might look like this:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',
            'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}
```

The data structure in `theBoard` now represents the tic-tac-toe board in Figure 5-5.

Figure 5-5: Player O wins.

Of course, the player sees only what is printed to the screen, not the contents of variables. Let's create a function to print the board dictionary onto the screen. Make the following addition to *ticTacToe.py* (new code is in bold):

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

When you run this program, `printBoard()` will print out a blank tic-tac-toe board.

```
 | |
-+-+-
 | |
-+-+-
 | |
```

The `printBoard()` function can handle any tic-tac-toe data structure you pass it. Try changing the code to the following:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O', 'mid-L': 'X', 'mid-M':
'X', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

Now when you run this program, the new board will be printed to the screen.

```
O|O|O
-+-+-
X|X|
-+-+-
 | |X
```

Because you created a data structure to represent a tic-tac-toe board and wrote code in `printBoard()` to interpret that data structure, you now have a program that "models" the tic-tac-toe board. You could have organized your data structure differently (for example, using keys like `'TOP-LEFT'` instead of `'top-L'`), but as long as the code works with your data structures, you will have a correctly working program.

For example, the `printBoard()` function expects the tic-tac-toe data structure to be a dictionary with keys for all nine slots. If the dictionary you passed was missing, say, the `'mid-L'` key, your program would no longer work.

```
O|O|O
-+-+-
Traceback (most recent call last):
  File "ticTacToe.py", line 10, in <module>
    printBoard(theBoard)
  File "ticTacToe.py", line 6, in printBoard
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
KeyError: 'mid-L'
```

Now let's add code that allows the players to enter their moves. Modify the *ticTacToe.py* program to look like this:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ', 'mid-L': ' ', 'mid-M': '
', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
turn = 'X'
for i in range(9):
❶   printBoard(theBoard)
    print('Turn for ' + turn + '. Move on which space?')
❷   move = input()
❸   theBoard[move] = turn
❹   if turn == 'X':
        turn = 'O'
    else:
        turn = 'X'
printBoard(theBoard)
```

The new code prints out the board at the start of each new turn ❶, gets the active player's move ❷, updates the game board accordingly ❸, and then swaps the active player ❹ before moving on to the next turn.

When you run this program, it will look something like this:

```
 | |
-+-+-
 | |
-+-+-
 | |
Turn for X. Move on which space?
mid-M
 | |
-+-+-
 |X|
-+-+-
 | |
Turn for O. Move on which space?
low-L
 | |
-+-+-
 |X|
-+-+-
O| |

--snip--

O|O|X
-+-+-
X|X|O
-+-+-
O| |X
Turn for X. Move on which space?
low-M
O|O|X
-+-+-
X|X|O
-+-+-
O|X|X
```

This isn't a complete tic-tac-toe game—for instance, it doesn't ever check whether a player has won—but it's enough to see how data structures can be used in programs.

NOTE *If you are curious, the source code for a complete tic-tac-toe program is described in the resources available from* http://nostarch.com/automatestuff/.

Nested Dictionaries and Lists

Modeling a tic-tac-toe board was fairly simple: The board needed only a single dictionary value with nine key-value pairs. As you model more complicated things, you may find you need dictionaries and lists that contain

other dictionaries and lists. Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values. For example, here's a program that uses a dictionary that contains other dictionaries in order to see who is bringing what to a picnic. The `totalBrought()` function can read this data structure and calculate the total number of an item being brought by all the guests.

```
allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
             'Bob': {'ham sandwiches': 3, 'apples': 2},
             'Carol': {'cups': 3, 'apple pies': 1}}

def totalBrought(guests, item):
    numBrought = 0
❶   for k, v in guests.items():
❷       numBrought = numBrought + v.get(item, 0)
    return numBrought

print('Number of things being brought:')
print(' - Apples         ' + str(totalBrought(allGuests, 'apples')))
print(' - Cups           ' + str(totalBrought(allGuests, 'cups')))
print(' - Cakes          ' + str(totalBrought(allGuests, 'cakes')))
print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches')))
print(' - Apple Pies     ' + str(totalBrought(allGuests, 'apple pies')))
```

Inside the `totalBrought()` function, the `for` loop iterates over the key-value pairs in `guests` ❶. Inside the loop, the string of the guest's name is assigned to `k`, and the dictionary of picnic items they're bringing is assigned to `v`. If the item parameter exists as a key in this dictionary, it's value (the quantity) is added to `numBrought` ❷. If it does not exist as a key, the `get()` method returns `0` to be added to `numBrought`.

The output of this program looks like this:

```
Number of things being brought:
 - Apples 7
 - Cups 3
 - Cakes 0
 - Ham Sandwiches 3
 - Apple Pies     1
```

This may seem like such a simple thing to model that you wouldn't need to bother with writing a program to do it. But realize that this same `totalBrought()` function could easily handle a dictionary that contains thousands of guests, each bringing *thousands* of different picnic items. Then having this information in a data structure along with the `totalBrought()` function would save you a lot of time!

You can model things with data structures in whatever way you like, as long as the rest of the code in your program can work with the data model correctly. When you first begin programming, don't worry so much about

the "right" way to model data. As you gain more experience, you may come up with more efficient models, but the important thing is that the data model works for your program's needs.

Summary

You learned all about dictionaries in this chapter. Lists and dictionaries are values that can contain multiple values, including other lists and dictionaries. Dictionaries are useful because you can map one item (the key) to another (the value), as opposed to lists, which simply contain a series of values in order. Values inside a dictionary are accessed using square brackets just as with lists. Instead of an integer index, dictionaries can have keys of a variety of data types: integers, floats, strings, or tuples. By organizing a program's values into data structures, you can create representations of real-world objects. You saw an example of this with a tic-tac-toe board.

That just about covers all the basic concepts of Python programming! You'll continue to learn new concepts throughout the rest of this book, but you now know enough to start writing some useful programs that can automate tasks. You might not think you have enough Python knowledge to do things such as download web pages, update spreadsheets, or send text messages, but that's where Python modules come in! These modules, written by other programmers, provide functions that make it easy for you to do all these things. So let's learn how to write real programs to do useful automated tasks.

Practice Questions

1. What does the code for an empty dictionary look like?
2. What does a dictionary value with a key `'foo'` and a value `42` look like?
3. What is the main difference between a dictionary and a list?
4. What happens if you try to access `spam['foo']` if `spam` is `{'bar': 100}`?
5. If a dictionary is stored in `spam`, what is the difference between the expressions `'cat' in spam` and `'cat' in spam.keys()`?
6. If a dictionary is stored in `spam`, what is the difference between the expressions `'cat' in spam` and `'cat' in spam.values()`?
7. What is a shortcut for the following code?

```
if 'color' not in spam:
    spam['color'] = 'black'
```

8. What module and function can be used to "pretty print" dictionary values?

Practice Projects

For practice, write programs to do the following tasks.

Fantasy Game Inventory

You are creating a fantasy video game. The data structure to model the player's inventory will be a dictionary where the keys are string values describing the item in the inventory and the value is an integer value detailing how many of that item the player has. For example, the dictionary value `{'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12}` means the player has 1 rope, 6 torches, 42 gold coins, and so on.

Write a function named `displayInventory()` that would take any possible "inventory" and display it like the following:

```
Inventory:
12 arrow
42 gold coin
1 rope
6 torch
1 dagger
Total number of items: 62
```

Hint: You can use a `for` loop to loop through all the keys in a dictionary.

```
# inventory.py
stuff = {'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12}

def displayInventory(inventory):
    print("Inventory:")
    item_total = 0
    for k, v in inventory.items():
        print(str(v) + ' ' + k)
        item_total += v
    print("Total number of items: " + str(item_total))

displayInventory(stuff)
```

List to Dictionary Function for Fantasy Game Inventory

Imagine that a vanquished dragon's loot is represented as a list of strings like this:

```
dragonLoot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
```

Write a function named `addToInventory(inventory, addedItems)`, where the `inventory` parameter is a dictionary representing the player's inventory (like in the previous project) and the `addedItems` parameter is a list like `dragonLoot`.

The addToInventory() function should return a dictionary that represents the updated inventory. Note that the addedItems list can contain multiples of the same item. Your code could look something like this:

```
def addToInventory(inventory, addedItems):
    # your code goes here

inv = {'gold coin': 42, 'rope': 1}
dragonLoot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
inv = addToInventory(inv, dragonLoot)
displayInventory(inv)
```

The previous program (with your displayInventory() function from the previous project) would output the following:

```
Inventory:
45 gold coin
1 rope
1 ruby
1 dagger

Total number of items: 48
```

6

MANIPULATING STRINGS

Text is one of the most common forms of data your programs will handle. You already know how to concatenate two string values together with the + operator, but you can do much more than that. You can extract partial strings from string values, add or remove spacing, convert letters to lowercase or uppercase, and check that strings are formatted correctly. You can even write Python code to access the clipboard for copying and pasting text.

In this chapter, you'll learn all this and more. Then you'll work through two different programming projects: a simple password manager and a program to automate the boring chore of formatting pieces of text.

Working with Strings

Let's look at some of the ways Python lets you write, print, and access strings in your code.

String Literals

Typing string values in Python code is fairly straightforward: They begin and end with a single quote. But then how can you use a quote inside a string? Typing **`'That is Alice's cat.'`** won't work, because Python thinks the string ends after `Alice`, and the rest (`s cat.'`) is invalid Python code. Fortunately, there are multiple ways to type strings.

Double Quotes

Strings can begin and end with double quotes, just as they do with single quotes. One benefit of using double quotes is that the string can have a single quote character in it. Enter the following into the interactive shell:

```
>>> spam = "That is Alice's cat."
```

Since the string begins with a double quote, Python knows that the single quote is part of the string and not marking the end of the string. However, if you need to use both single quotes and double quotes in the string, you'll need to use escape characters.

Escape Characters

An *escape character* lets you use characters that are otherwise impossible to put into a string. An escape character consists of a backslash (`\`) followed by the character you want to add to the string. (Despite consisting of two characters, it is commonly referred to as a singular escape character.) For example, the escape character for a single quote is `\'`. You can use this inside a string that begins and ends with single quotes. To see how escape characters work, enter the following into the interactive shell:

```
>>> spam = 'Say hi to Bob\'s mother.'
```

Python knows that since the single quote in `Bob\'s` has a backslash, it is not a single quote meant to end the string value. The escape characters `\'` and `\"` let you put single quotes and double quotes inside your strings, respectively.

Table 6-1 lists the escape characters you can use.

Table 6-1: Escape Characters

Escape character	Prints as
`\'`	Single quote
`\"`	Double quote
`\t`	Tab
`\n`	Newline (line break)
`\\`	Backslash

Enter the following into the interactive shell:

```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")
Hello there!
How are you?
I'm doing fine.
```

Raw Strings

You can place an `r` before the beginning quotation mark of a string to make it a raw string. A *raw string* completely ignores all escape characters and prints any backslash that appears in the string. For example, type the following into the interactive shell:

```
>>> print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

Because this is a raw string, Python considers the backslash as part of the string and not as the start of an escape character. Raw strings are helpful if you are typing string values that contain many backslashes, such as the strings used for regular expressions described in the next chapter.

Multiline Strings with Triple Quotes

While you can use the `\n` escape character to put a newline into a string, it is often easier to use multiline strings. A multiline string in Python begins and ends with either three single quotes or three double quotes. Any quotes, tabs, or newlines in between the "triple quotes" are considered part of the string. Python's indentation rules for blocks do not apply to lines inside a multiline string.

Open the file editor and write the following:

```
print('''Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob''')
```

Save this program as *catnapping.py* and run it. The output will look like this:

```
Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob
```

Notice that the single quote character in `Eve's` does not need to be escaped. Escaping single and double quotes is optional in raw strings. The following `print()` call would print identical text but doesn't use a multiline string:

```
print('Dear Alice,\n\nEve\'s cat has been arrested for catnapping, cat
burglary, and extortion.\n\nSincerely,\nBob')
```

Multiline Comments

While the hash character (`#`) marks the beginning of a comment for the rest of the line, a multiline string is often used for comments that span multiple lines. The following is perfectly valid Python code:

```
"""This is a test Python program.
Written by Al Sweigart al@inventwithpython.com

This program was designed for Python 3, not Python 2.
"""

def spam():
    """This is a multiline comment to help
    explain what the spam() function does."""
    print('Hello!')
```

Indexing and Slicing Strings

Strings use indexes and slices the same way lists do. You can think of the string `'Hello world!'` as a list and each character in the string as an item with a corresponding index.

'	H	e	l	l	o		w	o	r	l	d	!	'
	0	1	2	3	4	5	6	7	8	9	10	11	

The space and exclamation point are included in the character count, so `'Hello world!'` is 12 characters long, from `H` at index 0 to `!` at index 11.

Enter the following into the interactive shell:

```
>>> spam = 'Hello world!'
>>> spam[0]
'H'
>>> spam[4]
'o'
>>> spam[-1]
'!'
>>> spam[0:5]
'Hello'
```

```
>>> spam[:5]
'Hello'
>>> spam[6:]
'world!'
```

If you specify an index, you'll get the character at that position in the string. If you specify a range from one index to another, the starting index is included and the ending index is not. That's why, if spam is `'Hello world!'`, `spam[0:5]` is `'Hello'`. The substring you get from `spam[0:5]` will include everything from `spam[0]` to `spam[4]`, leaving out the space at index 5.

Note that slicing a string does not modify the original string. You can capture a slice from one variable in a separate variable. Try typing the following into the interactive shell:

```
>>> spam = 'Hello world!'
>>> fizz = spam[0:5]
>>> fizz
'Hello'
```

By slicing and storing the resulting substring in another variable, you can have both the whole string and the substring handy for quick, easy access.

The in and not in Operators with Strings

The `in` and `not in` operators can be used with strings just like with list values. An expression with two strings joined using `in` or `not in` will evaluate to a Boolean `True` or `False`. Enter the following into the interactive shell:

```
>>> 'Hello' in 'Hello World'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello World'
False
>>> '' in 'spam'
True
>>> 'cats' not in 'cats and dogs'
False
```

These expressions test whether the first string (the exact string, case sensitive) can be found within the second string.

Useful String Methods

Several string methods analyze strings or create transformed string values. This section describes the methods you'll be using most often.

The upper(), lower(), isupper(), and islower() String Methods

The `upper()` and `lower()` string methods return a new string where all the letters in the original string have been converted to uppercase or lowercase, respectively. Nonletter characters in the string remain unchanged. Enter the following into the interactive shell:

```
>>> spam = 'Hello world!'
>>> spam = spam.upper()
>>> spam
'HELLO WORLD!'
>>> spam = spam.lower()
>>> spam
'hello world!'
```

Note that these methods do not change the string itself but return new string values. If you want to change the original string, you have to call `upper()` or `lower()` on the string and then assign the new string to the variable where the original was stored. This is why you must use `spam = spam.upper()` to change the string in `spam` instead of simply `spam.upper()`. (This is just like if a variable `eggs` contains the value `10`. Writing `eggs + 3` does not change the value of `eggs`, but `eggs = eggs + 3` does.)

The `upper()` and `lower()` methods are helpful if you need to make a case-insensitive comparison. The strings `'great'` and `'GREat'` are not equal to each other. But in the following small program, it does not matter whether the user types `Great`, `GREAT`, or `grEAT`, because the string is first converted to lowercase.

```
print('How are you?')
feeling = input()
if feeling.lower() == 'great':
    print('I feel great too.')
else:
    print('I hope the rest of your day is good.')
```

When you run this program, the question is displayed, and entering a variation on `great`, such as `GREat`, will still give the output `I feel great too`. Adding code to your program to handle variations or mistakes in user input, such as inconsistent capitalization, will make your programs easier to use and less likely to fail.

```
How are you?
GREat
I feel great too.
```

The `isupper()` and `islower()` methods will return a Boolean `True` value if the string has at least one letter and all the letters are uppercase or

lowercase, respectively. Otherwise, the method returns `False`. Enter the following into the interactive shell, and notice what each method call returns:

```
>>> spam = 'Hello world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

Since the `upper()` and `lower()` string methods themselves return strings, you can call string methods on *those* returned string values as well. Expressions that do this will look like a chain of method calls. Enter the following into the interactive shell:

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

The isX String Methods

Along with `islower()` and `isupper()`, there are several string methods that have names beginning with the word *is*. These methods return a Boolean value that describes the nature of the string. Here are some common `is`*X* string methods:

- `isalpha()` returns `True` if the string consists only of letters and is not blank.
- `isalnum()` returns `True` if the string consists only of letters and numbers and is not blank.
- `isdecimal()` returns `True` if the string consists only of numeric characters and is not blank.

- `isspace()` returns `True` if the string consists only of spaces, tabs, and newlines and is not blank.
- `istitle()` returns `True` if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

Enter the following into the interactive shell:

```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> '    '.isspace()
True
>>> 'This Is Title Case'.istitle()
True
>>> 'This Is Title Case 123'.istitle()
True
>>> 'This Is not Title Case'.istitle()
False
>>> 'This Is NOT Title Case Either'.istitle()
False
```

The `is`*X* string methods are helpful when you need to validate user input. For example, the following program repeatedly asks users for their age and a password until they provide valid input. Open a new file editor window and enter this program, saving it as *validateInput.py*:

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')

while True:
    print('Select a new password (letters and numbers only):')
    password = input()
    if password.isalnum():
        break
    print('Passwords can only have letters and numbers.')
```

In the first `while` loop, we ask the user for their age and store their input in `age`. If `age` is a valid (decimal) value, we break out of this first `while` loop and move on to the second, which asks for a password. Otherwise, we inform the user that they need to enter a number and again ask them to

enter their age. In the second `while` loop, we ask for a password, store the user's input in `password`, and break out of the loop if the input was alphanumeric. If it wasn't, we're not satisfied so we tell the user the password needs to be alphanumeric and again ask them to enter a password.

When run, the program's output looks like this:

```
Enter your age:
forty two
Please enter a number for your age.
Enter your age:
42
Select a new password (letters and numbers only):
secr3t!
Passwords can only have letters and numbers.
Select a new password (letters and numbers only):
secr3t
```

Calling `isdecimal()` and `isalnum()` on variables, we're able to test whether the values stored in those variables are decimal or not, alphanumeric or not. Here, these tests help us reject the input `forty two` and accept `42`, and reject `secr3t!` and accept `secr3t`.

The startswith() and endswith() String Methods

The `startswith()` and `endswith()` methods return `True` if the string value they are called on begins or ends (respectively) with the string passed to the method; otherwise, they return `False`. Enter the following into the interactive shell:

```
>>> 'Hello world!'.startswith('Hello')
True
>>> 'Hello world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello world!'.startswith('Hello world!')
True
>>> 'Hello world!'.endswith('Hello world!')
True
```

These methods are useful alternatives to the `==` equals operator if you need to check only whether the first or last part of the string, rather than the whole thing, is equal to another string.

The join() and split() String Methods

The `join()` method is useful when you have a list of strings that need to be joined together into a single string value. The `join()` method is called on a

string, gets passed a list of strings, and returns a string. The returned string is the concatenation of each string in the passed-in list. For example, enter the following into the interactive shell:

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

Notice that the string `join()` calls on is inserted between each string of the list argument. For example, when `join(['cats', 'rats', 'bats'])` is called on the `', '` string, the returned string is 'cats, rats, bats'.

Remember that `join()` is called on a string value and is passed a list value. (It's easy to accidentally call it the other way around.) The `split()` method does the opposite: It's called on a string value and returns a list of strings. Enter the following into the interactive shell:

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

By default, the string `'My name is Simon'` is split wherever whitespace characters such as the space, tab, or newline characters are found. These whitespace characters are not included in the strings in the returned list. You can pass a delimiter string to the `split()` method to specify a different string to split upon. For example, enter the following into the interactive shell:

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

A common use of `split()` is to split a multiline string along the newline characters. Enter the following into the interactive shell:

```
>>> spam = '''Dear Alice,
How have you been? I am fine.
There is a container in the fridge
that is labeled "Milk Experiment".

Please do not drink it.
Sincerely,
Bob'''
>>> spam.split('\n')
['Dear Alice,', 'How have you been? I am fine.', 'There is a container in the
fridge', 'that is labeled "Milk Experiment".', '', 'Please do not drink it.',
'Sincerely,', 'Bob']
```

Passing `split()` the argument `'\n'` lets us split the multiline string stored in `spam` along the newlines and return a list in which each item corresponds to one line of the string.

Justifying Text with rjust(), ljust(), and center()

The `rjust()` and `ljust()` string methods return a padded version of the string they are called on, with spaces inserted to justify the text. The first argument to both methods is an integer length for the justified string. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(10)
'     Hello'
>>> 'Hello'.rjust(20)
'               Hello'
>>> 'Hello World'.rjust(20)
'         Hello World'
>>> 'Hello'.ljust(10)
'Hello     '
```

`'Hello'.rjust(10)` says that we want to right-justify `'Hello'` in a string of total length `10`. `'Hello'` is five characters, so five spaces will be added to its left, giving us a string of 10 characters with `'Hello'` justified right.

An optional second argument to `rjust()` and `ljust()` will specify a fill character other than a space character. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(20, '*')
'***************Hello'
>>> 'Hello'.ljust(20, '-')
'Hello---------------'
```

The `center()` string method works like `ljust()` and `rjust()` but centers the text rather than justifying it to the left or right. Enter the following into the interactive shell:

```
>>> 'Hello'.center(20)
'       Hello        '
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

These methods are especially useful when you need to print tabular data that has the correct spacing. Open a new file editor window and enter the following code, saving it as *picnicTable.py*:

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))
```

```
picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

In this program, we define a `printPicnic()` method that will take in a dictionary of information and use `center()`, `ljust()`, and `rjust()` to display that information in a neatly aligned table-like format.

The dictionary that we'll pass to `printPicnic()` is `picnicItems`. In `picnicItems`, we have 4 sandwiches, 12 apples, 4 cups, and 8000 cookies. We want to organize this information into two columns, with the name of the item on the left and the quantity on the right.

To do this, we decide how wide we want the left and right columns to be. Along with our dictionary, we'll pass these values to `printPicnic()`.

`printPicnic()` takes in a dictionary, a `leftWidth` for the left column of a table, and a `rightWidth` for the right column. It prints a title, `PICNIC ITEMS`, centered above the table. Then, it loops through the dictionary, printing each key-value pair on a line with the key justified left and padded by periods, and the value justified right and padded by spaces.

After defining `printPicnic()`, we define the dictionary `picnicItems` and call `printPicnic()` twice, passing it different widths for the left and right table columns.

When you run this program, the picnic items are displayed twice. The first time the left column is 12 characters wide, and the right column is 5 characters wide. The second time they are 20 and 6 characters wide, respectively.

```
---PICNIC ITEMS--
sandwiches..    4
apples......   12
cups........    4
cookies..... 8000
-------PICNIC ITEMS-------
sandwiches..........     4
apples.............    12
cups...............     4
cookies............  8000
```

Using `rjust()`, `ljust()`, and `center()` lets you ensure that strings are neatly aligned, even if you aren't sure how many characters long your strings are.

Removing Whitespace with strip(), rstrip(), and lstrip()

Sometimes you may want to strip off whitespace characters (space, tab, and newline) from the left side, right side, or both sides of a string. The `strip()` string method will return a new string without any whitespace

characters at the beginning or end. The `lstrip()` and `rstrip()` methods will remove whitespace characters from the left and right ends, respectively. Enter the following into the interactive shell:

```
>>> spam = '    Hello World    '
>>> spam.strip()
'Hello World'
>>> spam.lstrip()
'Hello World    '
>>> spam.rstrip()
'    Hello World'
```

Optionally, a string argument will specify which characters on the ends should be stripped. Enter the following into the interactive shell:

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')
'BaconSpamEggs'
```

Passing `strip()` the argument `'ampS'` will tell it to strip occurences of `a`, `m`, `p`, and capital `S` from the ends of the string stored in `spam`. The order of the characters in the string passed to `strip()` does not matter: `strip('ampS')` will do the same thing as `strip('mapS')` or `strip('Spam')`.

Copying and Pasting Strings with the pyperclip Module

The `pyperclip` module has `copy()` and `paste()` functions that can send text to and receive text from your computer's clipboard. Sending the output of your program to the clipboard will make it easy to paste it to an email, word processor, or some other software.

Pyperclip does not come with Python. To install it, follow the directions for installing third-party modules in Appendix A. After installing the `pyperclip` module, enter the following into the interactive shell:

```
>>> import pyperclip
>>> pyperclip.copy('Hello world!')
>>> pyperclip.paste()
'Hello world!'
```

Of course, if something outside of your program changes the clipboard contents, the `paste()` function will return it. For example, if I copied this sentence to the clipboard and then called `paste()`, it would look like this:

```
>>> pyperclip.paste()
'For example, if I copied this sentence to the clipboard and then called
paste(), it would look like this:'
```

RUNNING PYTHON SCRIPTS OUTSIDE OF IDLE

So far, you've been running your Python scripts using the interactive shell and file editor in IDLE. However, you won't want to go through the inconvenience of opening IDLE and the Python script each time you want to run a script. Fortunately, there are shortcuts you can set up to make running Python scripts easier. The steps are slightly different for Windows, OS X, and Linux, but each is described in Appendix B. Turn to Appendix B to learn how to run your Python scripts conveniently and be able to pass command line arguments to them. (You will not be able to pass command line arguments to your programs using IDLE.)

Project: Password Locker

You probably have accounts on many different websites. It's a bad habit to use the same password for each of them because if any of those sites has a security breach, the hackers will learn the password to all of your other accounts. It's best to use password manager software on your computer that uses one master password to unlock the password manager. Then you can copy any account password to the clipboard and paste it into the website's Password field.

The password manager program you'll create in this example isn't secure, but it offers a basic demonstration of how such programs work.

THE CHAPTER PROJECTS

This is the first "chapter project" of the book. From here on, each chapter will have projects that demonstrate the concepts covered in the chapter. The projects are written in a style that takes you from a blank file editor window to a full, working program. Just like with the interactive shell examples, don't only read the project sections—follow along on your computer!

Step 1: Program Design and Data Structures

You want to be able to run this program with a command line argument that is the account's name—for instance, *email* or *blog*. That account's password will be copied to the clipboard so that the user can paste it into a Password field. This way, the user can have long, complicated passwords without having to memorize them.

Open a new file editor window and save the program as *pw.py*. You need to start the program with a `#!` (*shebang*) line (see Appendix B) and should also write a comment that briefly describes the program. Since you want to associate each account's name with its password, you can store these as

strings in a dictionary. The dictionary will be the data structure that organizes your account and password data. Make your program look like the following:

```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8vO1if1MLZF3sdt',
             'luggage': '12345'}
```

Step 2: Handle Command Line Arguments

The command line arguments will be stored in the variable `sys.argv`. (See Appendix B for more information on how to use command line arguments in your programs.) The first item in the `sys.argv` list should always be a string containing the program's filename (`'pw.py'`), and the second item should be the first command line argument. For this program, this argument is the name of the account whose password you want. Since the command line argument is mandatory, you display a usage message to the user if they forget to add it (that is, if the `sys.argv` list has fewer than two values in it). Make your program look like the following:

```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8vO1if1MLZF3sdt',
             'luggage': '12345'}

import sys
if len(sys.argv) < 2:
    print('Usage: python pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1]    # first command line arg is the account name
```

Step 3: Copy the Right Password

Now that the account name is stored as a string in the variable `account`, you need to see whether it exists in the `PASSWORDS` dictionary as a key. If so, you want to copy the key's value to the clipboard using `pyperclip.copy()`. (Since you're using the `pyperclip` module, you need to import it.) Note that you don't actually *need* the `account` variable; you could just use `sys.argv[1]` everywhere `account` is used in this program. But a variable named `account` is much more readable than something cryptic like `sys.argv[1]`.

Make your program look like the following:

```
#! python3
# pw.py - An insecure password locker program.
```

```
PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8vO1if1MLZF3sdt',
             'luggage': '12345'}

import sys, pyperclip
if len(sys.argv) < 2:
    print('Usage: py pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1]    # first command line arg is the account name

if account in PASSWORDS:
    pyperclip.copy(PASSWORDS[account])
    print('Password for ' + account + ' copied to clipboard.')
else:
    print('There is no account named ' + account)
```

This new code looks in the PASSWORDS dictionary for the account name. If the account name is a key in the dictionary, we get the value corresponding to that key, copy it to the clipboard, and print a message saying that we copied the value. Otherwise, we print a message saying there's no account with that name.

That's the complete script. Using the instructions in Appendix B for launching command line programs easily, you now have a fast way to copy your account passwords to the clipboard. You will have to modify the PASSWORDS dictionary value in the source whenever you want to update the program with a new password.

Of course, you probably don't want to keep all your passwords in one place where anyone could easily copy them. But you can modify this program and use it to quickly copy regular text to the clipboard. Say you are sending out several emails that have many of the same stock paragraphs in common. You could put each paragraph as a value in the PASSWORDS dictionary (you'd probably want to rename the dictionary at this point), and then you would have a way to quickly select and copy one of many standard pieces of text to the clipboard.

On Windows, you can create a batch file to run this program with the WIN-R Run window. (For more about batch files, see Appendix B.) Type the following into the file editor and save the file as *pw.bat* in the *C:\Windows* folder:

```
@py.exe C:\Python34\pw.py %*
@pause
```

With this batch file created, running the password-safe program on Windows is just a matter of pressing WIN-R and typing pw <account name>.

Project: Adding Bullets to Wiki Markup

When editing a Wikipedia article, you can create a bulleted list by putting each list item on its own line and placing a star in front. But say you have a really large list that you want to add bullet points to. You could just type those stars at the beginning of each line, one by one. Or you could automate this task with a short Python script.

The *bulletPointAdder.py* script will get the text from the clipboard, add a star and space to the beginning of each line, and then paste this new text to the clipboard. For example, if I copied the following text (for the Wikipedia article "List of Lists of Lists") to the clipboard:

```
Lists of animals
Lists of aquarium life
Lists of biologists by author abbreviation
Lists of cultivars
```

and then ran the *bulletPointAdder.py* program, the clipboard would then contain the following:

```
* Lists of animals
* Lists of aquarium life
* Lists of biologists by author abbreviation
* Lists of cultivars
```

This star-prefixed text is ready to be pasted into a Wikipedia article as a bulleted list.

Step 1: Copy and Paste from the Clipboard

You want the *bulletPointAdder.py* program to do the following:

1. Paste text from the clipboard
2. Do something to it
3. Copy the new text to the clipboard

That second step is a little tricky, but steps 1 and 3 are pretty straightforward: They just involve the `pyperclip.copy()` and `pyperclip.paste()` functions. For now, let's just write the part of the program that covers steps 1 and 3. Enter the following, saving the program as *bulletPointAdder.py*:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()
```

```
# TODO: Separate lines and add stars.

pyperclip.copy(text)
```

The TODO comment is a reminder that you should complete this part of the program eventually. The next step is to actually implement that piece of the program.

Step 2: Separate the Lines of Text and Add the Star

The call to pyperclip.paste() returns all the text on the clipboard as one big string. If we used the "List of Lists of Lists" example, the string stored in text would look like this:

```
'Lists of animals\nLists of aquarium life\nLists of biologists by author
abbreviation\nLists of cultivars'
```

The \n newline characters in this string cause it to be displayed with multiple lines when it is printed or pasted from the clipboard. There are many "lines" in this one string value. You want to add a star to the start of each of these lines.

You could write code that searches for each \n newline character in the string and then adds the star just after that. But it would be easier to use the split() method to return a list of strings, one for each line in the original string, and then add the star to the front of each string in the list.

Make your program look like the following:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):    # loop through all indexes in the "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines" list

pyperclip.copy(text)
```

We split the text along its newlines to get a list in which each item is one line of the text. We store the list in lines and then loop through the items in lines. For each line, we add a star and a space to the start of the line. Now each string in lines begins with a star.

Step 3: Join the Modified Lines

The `lines` list now contains modified lines that start with stars. But `pyperclip.copy()` is expecting a single string value, not a list of string values. To make this single string value, pass `lines` into the `join()` method to get a single string joined from the list's strings. Make your program look like the following:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):    # loop through all indexes for "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines" list
text = '\n'.join(lines)
pyperclip.copy(text)
```

When this program is run, it replaces the text on the clipboard with text that has stars at the start of each line. Now the program is complete, and you can try running it with text copied to the clipboard.

Even if you don't need to automate this specific task, you might want to automate some other kind of text manipulation, such as removing trailing spaces from the end of lines or converting text to uppercase or lowercase. Whatever your needs, you can use the clipboard for input and output.

Summary

Text is a common form of data, and Python comes with many helpful string methods to process the text stored in string values. You will make use of indexing, slicing, and string methods in almost every Python program you write.

The programs you are writing now don't seem too sophisticated—they don't have graphical user interfaces with images and colorful text. So far, you're displaying text with `print()` and letting the user enter text with `input()`. However, the user can quickly enter large amounts of text through the clipboard. This ability provides a useful avenue for writing programs that manipulate massive amounts of text. These text-based programs might not have flashy windows or graphics, but they can get a lot of useful work done quickly.

Another way to manipulate large amounts of text is reading and writing files directly off the hard drive. You'll learn how to do this with Python in the next chapter.

Practice Questions

1. What are escape characters?
2. What do the `\n` and `\t` escape characters represent?
3. How can you put a `\` backslash character in a string?
4. The string value `"Howl's Moving Castle"` is a valid string. Why isn't it a problem that the single quote character in the word `Howl's` isn't escaped?
5. If you don't want to put `\n` in your string, how can you write a string with newlines in it?
6. What do the following expressions evaluate to?
   - `'Hello world!'[1]`
   - `'Hello world!'[0:5]`
   - `'Hello world!'[:5]`
   - `'Hello world!'[3:]`
7. What do the following expressions evaluate to?
   - `'Hello'.upper()`
   - `'Hello'.upper().isupper()`
   - `'Hello'.upper().lower()`
8. What do the following expressions evaluate to?
   - `'Remember, remember, the fifth of November.'.split()`
   - `'-'.join('There can be only one.'.split())`
9. What string methods can you use to right-justify, left-justify, and center a string?
10. How can you trim whitespace characters from the beginning or end of a string?

Practice Project

For practice, write a program that does the following.

Table Printer

Write a function named `printTable()` that takes a list of lists of strings and displays it in a well-organized table with each column right-justified. Assume that all the inner lists will contain the same number of strings. For example, the value could look like this:

```
tableData = [['apples', 'oranges', 'cherries', 'banana'],
             ['Alice', 'Bob', 'Carol', 'David'],
             ['dogs', 'cats', 'moose', 'goose']]
```

Your `printTable()` function would print the following:

```
   apples Alice  dogs
  oranges   Bob  cats
 cherries Carol moose
   banana David goose
```

Hint: Your code will first have to find the longest string in each of the inner lists so that the whole column can be wide enough to fit all the strings. You can store the maximum width of each column as a list of integers. The `printTable()` function can begin with `colWidths = [0] * len(tableData)`, which will create a list containing the same number of `0` values as the number of inner lists in `tableData`. That way, `colWidths[0]` can store the width of the longest string in `tableData[0]`, `colWidths[1]` can store the width of the longest string in `tableData[1]`, and so on. You can then find the largest value in the `colWidths` list to find out what integer width to pass to the `rjust()` string method.

PART II

AUTOMATING TASKS

7

PATTERN MATCHING WITH REGULAR EXPRESSIONS

You may be familiar with searching for text by pressing CTRL-F and typing in the words you're looking for. *Regular expressions* go one step further: They allow you to specify a *pattern* of text to search for. You may not know a business's exact phone number, but if you live in the United States or Canada, you know it will be three digits, followed by a hyphen, and then four more digits (and optionally, a three-digit area code at the start). This is how you, as a human, know a phone number when you see it: 415-555-1234 is a phone number, but 4,155,551,234 is not.

Regular expressions are helpful, but not many non-programmers know about them even though most modern text editors and word processors, such as Microsoft Word or OpenOffice, have find and find-and-replace features that can search based on regular expressions. Regular expressions are huge time-savers, not just for software users but also for

programmers. In fact, tech writer Cory Doctorow argues that even before teaching programming, we should be teaching regular expressions:

> "Knowing [regular expressions] can mean the difference between solving a problem in 3 steps and solving it in 3,000 steps. When you're a nerd, you forget that the problems you solve with a couple keystrokes can take other people days of tedious, error-prone work to slog through."[1]

In this chapter, you'll start by writing a program to find text patterns *without* using regular expressions and then see how to use regular expressions to make the code much less bloated. I'll show you basic matching with regular expressions and then move on to some more powerful features, such as string substitution and creating your own character classes. Finally, at the end of the chapter, you'll write a program that can automatically extract phone numbers and email addresses from a block of text.

Finding Patterns of Text Without Regular Expressions

Say you want to find a phone number in a string. You know the pattern: three numbers, a hyphen, three numbers, a hyphen, and four numbers. Here's an example: 415-555-4242.

Let's use a function named `isPhoneNumber()` to check whether a string matches this pattern, returning either `True` or `False`. Open a new file editor window and enter the following code; then save the file as *isPhoneNumber.py*:

```
def isPhoneNumber(text):
❶    if len(text) != 12:
         return False
     for i in range(0, 3):
❷        if not text[i].isdecimal():
             return False
❸    if text[3] != '-':
         return False
     for i in range(4, 7):
❹        if not text[i].isdecimal():
             return False
❺    if text[7] != '-':
         return False
     for i in range(8, 12):
❻        if not text[i].isdecimal():
             return False
❼    return True

print('415-555-4242 is a phone number:')
print(isPhoneNumber('415-555-4242'))
print('Moshi moshi is a phone number:')
print(isPhoneNumber('Moshi moshi'))
```

1. Cory Doctorow, "Here's what ICT should really teach kids: how to do regular expressions," *Guardian*, December 4, 2012, *http://www.theguardian.com/technology/2012/dec/04/ict-teach-kids-regular-expressions/*.

When this program is run, the output looks like this:

```
415-555-4242 is a phone number:
True
Moshi moshi is a phone number:
False
```

The `isPhoneNumber()` function has code that does several checks to see whether the string in `text` is a valid phone number. If any of these checks fail, the function returns `False`. First the code checks that the string is exactly 12 characters ❶. Then it checks that the area code (that is, the first three characters in `text`) consists of only numeric characters ❷. The rest of the function checks that the string follows the pattern of a phone number: The number must have the first hyphen after the area code ❸, three more numeric characters ❹, then another hyphen ❺, and finally four more numbers ❻. If the program execution manages to get past all the checks, it returns `True` ❼.

Calling `isPhoneNumber()` with the argument `'415-555-4242'` will return `True`. Calling `isPhoneNumber()` with `'Moshi moshi'` will return `False`; the first test fails because `'Moshi moshi'` is not 12 characters long.

You would have to add even more code to find this pattern of text in a larger string. Replace the last four `print()` function calls in *isPhoneNumber.py* with the following:

```
message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.'
for i in range(len(message)):
❶     chunk = message[i:i+12]
❷     if isPhoneNumber(chunk):
        print('Phone number found: ' + chunk)
print('Done')
```

When this program is run, the output will look like this:

```
Phone number found: 415-555-1011
Phone number found: 415-555-9999
Done
```

On each iteration of the `for` loop, a new chunk of 12 characters from `message` is assigned to the variable `chunk` ❶. For example, on the first iteration, `i` is `0`, and `chunk` is assigned `message[0:12]` (that is, the string `'Call me at 4'`). On the next iteration, `i` is `1`, and `chunk` is assigned `message[1:13]` (the string `'all me at 41'`).

You pass `chunk` to `isPhoneNumber()` to see whether it matches the phone number pattern ❷, and if so, you print the chunk.

Continue to loop through `message`, and eventually the 12 characters in `chunk` will be a phone number. The loop goes through the entire string, testing each 12-character piece and printing any `chunk` it finds that satisfies `isPhoneNumber()`. Once we're done going through `message`, we print `Done`.

While the string in `message` is short in this example, it could be millions of characters long and the program would still run in less than a second. A similar program that finds phone numbers using regular expressions would also run in less than a second, but regular expressions make it quicker to write these programs.

Finding Patterns of Text with Regular Expressions

The previous phone number–finding program works, but it uses a lot of code to do something limited: The `isPhoneNumber()` function is 17 lines but can find only one pattern of phone numbers. What about a phone number formatted like 415.555.4242 or (415) 555-4242? What if the phone number had an extension, like 415-555-4242 x99? The `isPhoneNumber()` function would fail to validate them. You could add yet more code for these additional patterns, but there is an easier way.

Regular expressions, called *regexes* for short, are descriptions for a pattern of text. For example, a `\d` in a regex stands for a digit character—that is, any single numeral 0 to 9. The regex `\d\d\d-\d\d\d-\d\d\d\d` is used by Python to match the same text the previous `isPhoneNumber()` function did: a string of three numbers, a hyphen, three more numbers, another hyphen, and four numbers. Any other string would not match the `\d\d\d-\d\d\d-\d\d\d\d` regex.

But regular expressions can be much more sophisticated. For example, adding a `3` in curly brackets (`{3}`) after a pattern is like saying, "Match this pattern three times." So the slightly shorter regex `\d{3}-\d{3}-\d{4}` also matches the correct phone number format.

Creating Regex Objects

All the regex functions in Python are in the `re` module. Enter the following into the interactive shell to import this module:

```
>>> import re
```

NOTE *Most of the examples that follow in this chapter will require the `re` module, so remember to import it at the beginning of any script you write or any time you restart IDLE. Otherwise, you'll get a `NameError: name 're' is not defined` error message.*

Passing a string value representing your regular expression to `re.compile()` returns a `Regex` pattern object (or simply, a `Regex` object).

To create a `Regex` object that matches the phone number pattern, enter the following into the interactive shell. (Remember that `\d` means "a digit character" and `\d\d\d-\d\d\d-\d\d\d\d` is the regular expression for the correct phone number pattern.)

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

Now the phoneNumRegex variable contains a Regex object.

PASSING RAW STRINGS TO RE.COMPILE()

Remember that escape characters in Python use the backslash (\). The string value '\n' represents a single newline character, not a backslash followed by a lowercase *n*. You need to enter the escape character \\ to print a single backslash. So '\\n' is the string that represents a backslash followed by a lowercase *n*. However, by putting an r before the first quote of the string value, you can mark the string as a *raw string*, which does not escape characters.

Since regular expressions frequently use backslashes in them, it is convenient to pass raw strings to the re.compile() function instead of typing extra backslashes. Typing r'\d\d\d-\d\d\d-\d\d\d\d' is much easier than typing '\\d\\d\\d-\\d\\d\\d-\\d\\d\\d\\d'.

Matching Regex Objects

A Regex object's search() method searches the string it is passed for any matches to the regex. The search() method will return None if the regex pattern is not found in the string. If the pattern *is* found, the search() method returns a Match object. Match objects have a group() method that will return the actual matched text from the searched string. (I'll explain groups shortly.) For example, enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> print('Phone number found: ' + mo.group())
Phone number found: 415-555-4242
```

The mo variable name is just a generic name to use for Match objects. This example might seem complicated at first, but it is much shorter than the earlier *isPhoneNumber.py* program and does the same thing.

Here, we pass our desired pattern to re.compile() and store the resulting Regex object in phoneNumRegex. Then we call search() on phoneNumRegex and pass search() the string we want to search for a match. The result of the search gets stored in the variable mo. In this example, we know that our pattern will be found in the string, so we know that a Match object will be returned. Knowing that mo contains a Match object and not the null value None, we can call group() on mo to return the match. Writing mo.group() inside our print statement displays the whole match, 415-555-4242.

Review of Regular Expression Matching

While there are several steps to using regular expressions in Python, each step is fairly simple.

1. Import the regex module with `import re`.
2. Create a `Regex` object with the `re.compile()` function. (Remember to use a raw string.)
3. Pass the string you want to search into the `Regex` object's `search()` method. This returns a `Match` object.
4. Call the `Match` object's `group()` method to return a string of the actual matched text.

NOTE *While I encourage you to enter the example code into the interactive shell, you should also make use of web-based regular expression testers, which can show you exactly how a regex matches a piece of text that you enter. I recommend the tester at* http://regexpal.com/.

More Pattern Matching with Regular Expressions

Now that you know the basic steps for creating and finding regular expression objects with Python, you're ready to try some of their more powerful pattern-matching capabilities.

Grouping with Parentheses

Say you want to separate the area code from the rest of the phone number. Adding parentheses will create *groups* in the regex: `(\d\d\d)-(\d\d\d-\d\d\d\d)`. Then you can use the `group()` match object method to grab the matching text from just one group.

The first set of parentheses in a regex string will be group `1`. The second set will be group `2`. By passing the integer `1` or `2` to the `group()` match object method, you can grab different parts of the matched text. Passing `0` or nothing to the `group()` method will return the entire matched text. Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242'
```

If you would like to retrieve all the groups at once, use the `groups()` method—note the plural form for the name.

```
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

Since `mo.groups()` returns a tuple of multiple values, you can use the multiple-assignment trick to assign each value to a separate variable, as in the previous `areaCode, mainNumber = mo.groups()` line.

Parentheses have a special meaning in regular expressions, but what do you do if you need to match a parenthesis in your text? For instance, maybe the phone numbers you are trying to match have the area code set in parentheses. In this case, you need to escape the (and) characters with a backslash. Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'(\(\d\d\d\)) (\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My phone number is (415) 555-4242.')
>>> mo.group(1)
'(415)'
>>> mo.group(2)
'555-4242'
```

The `\(` and `\)` escape characters in the raw string passed to `re.compile()` will match actual parenthesis characters.

Matching Multiple Groups with the Pipe

The | character is called a *pipe*. You can use it anywhere you want to match one of many expressions. For example, the regular expression `r'Batman|Tina Fey'` will match either `'Batman'` or `'Tina Fey'`.

When *both* Batman and Tina Fey occur in the searched string, the first occurrence of matching text will be returned as the `Match` object. Enter the following into the interactive shell:

```
>>> heroRegex = re.compile (r'Batman|Tina Fey')
>>> mo1 = heroRegex.search('Batman and Tina Fey.')
>>> mo1.group()
'Batman'

>>> mo2 = heroRegex.search('Tina Fey and Batman.')
>>> mo2.group()
'Tina Fey'
```

NOTE *You can find* all *matching occurrences with the `findall()` method that's discussed in "The `findall()` Method" on page 157.*

You can also use the pipe to match one of several patterns as part of your regex. For example, say you wanted to match any of the strings `'Batman'`, `'Batmobile'`, `'Batcopter'`, and `'Batbat'`. Since all these strings start with `Bat`, it would be nice if you could specify that prefix only once. This can be done with parentheses. Enter the following into the interactive shell:

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
>>> mo = batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'
>>> mo.group(1)
'mobile'
```

The method call `mo.group()` returns the full matched text `'Batmobile'`, while `mo.group(1)` returns just the part of the matched text inside the first parentheses group, `'mobile'`. By using the pipe character and grouping parentheses, you can specify several alternative patterns you would like your regex to match.

If you need to match an actual pipe character, escape it with a backslash, like `\|`.

Optional Matching with the Question Mark

Sometimes there is a pattern that you want to match only optionally. That is, the regex should find a match whether or not that bit of text is there. The `?` character flags the group that precedes it as an optional part of the pattern. For example, enter the following into the interactive shell:

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

The `(wo)?` part of the regular expression means that the pattern `wo` is an optional group. The regex will match text that has zero instances or one instance of *wo* in it. This is why the regex matches both `'Batwoman'` and `'Batman'`.

Using the earlier phone number example, you can make the regex look for phone numbers that do or do not have an area code. Enter the following into the interactive shell:

```
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
>>> mo1 = phoneRegex.search('My number is 415-555-4242')
>>> mo1.group()
'415-555-4242'
```

```
>>> mo2 = phoneRegex.search('My number is 555-4242')
>>> mo2.group()
'555-4242'
```

You can think of the ? as saying, "Match zero or one of the group preceding this question mark."

If you need to match an actual question mark character, escape it with \?.

Matching Zero or More with the Star

The * (called the *star* or *asterisk*) means "match zero or more"—the group that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again. Let's look at the Batman example again.

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'

>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

For 'Batman', the (wo)* part of the regex matches zero instances of wo in the string; for 'Batwoman', the (wo)* matches one instance of wo; and for 'Batwowowowoman', (wo)* matches four instances of wo.

If you need to match an actual star character, prefix the star in the regular expression with a backslash, *.

Matching One or More with the Plus

While * means "match zero or more," the + (or *plus*) means "match one or more." Unlike the star, which does not require its group to appear in the matched string, the group preceding a plus must appear *at least once*. It is not optional. Enter the following into the interactive shell, and compare it with the star regexes in the previous section:

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'

>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'
```

```
>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

The regex `Bat(wo)+man` will not match the string `'The Adventures of Batman'` because at least one `wo` is required by the plus sign.

If you need to match an actual plus sign character, prefix the plus sign with a backslash to escape it: `\+`.

Matching Specific Repetitions with Curly Brackets

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in curly brackets. For example, the regex `(Ha){3}` will match the string `'HaHaHa'`, but it will not match `'HaHa'`, since the latter has only two repeats of the `(Ha)` group.

Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the curly brackets. For example, the regex `(Ha){3,5}` will match `'HaHaHa'`, `'HaHaHaHa'`, and `'HaHaHaHaHa'`.

You can also leave out the first or second number in the curly brackets to leave the minimum or maximum unbounded. For example, `(Ha){3,}` will match three or more instances of the `(Ha)` group, while `(Ha){,5}` will match zero to five instances. Curly brackets can help make your regular expressions shorter. These two regular expressions match identical patterns:

```
(Ha){3}
(Ha)(Ha)(Ha)
```

And these two regular expressions also match identical patterns:

```
(Ha){3,5}
((Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha)(Ha))
```

Enter the following into the interactive shell:

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'

>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

Here, `(Ha){3}` matches `'HaHaHa'` but not `'Ha'`. Since it doesn't match `'Ha'`, `search()` returns `None`.

Greedy and Nongreedy Matching

Since `(Ha){3,5}` can match three, four, or five instances of `Ha` in the string `'HaHaHaHaHa'`, you may wonder why the `Match` object's call to `group()` in the

previous curly bracket example returns 'HaHaHaHaHa' instead of the shorter possibilities. After all, 'HaHaHa' and 'HaHaHaHa' are also valid matches of the regular expression (Ha){3,5}.

Python's regular expressions are *greedy* by default, which means that in ambiguous situations they will match the longest string possible. The *non-greedy* version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a question mark.

Enter the following into the interactive shell, and notice the difference between the greedy and nongreedy forms of the curly brackets searching the same string:

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'

>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

Note that the question mark can have two meanings in regular expressions: declaring a nongreedy match or flagging an optional group. These meanings are entirely unrelated.

The findall() Method

In addition to the search() method, Regex objects also have a findall() method. While search() will return a Match object of the *first* matched text in the searched string, the findall() method will return the strings of *every* match in the searched string. To see how search() returns a Match object only on the first instance of matching text, enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
>>> mo.group()
'415-555-9999'
```

On the other hand, findall() will not return a Match object but a list of strings—*as long as there are no groups in the regular expression*. Each string in the list is a piece of the searched text that matched the regular expression. Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

If there *are* groups in the regular expression, then findall() will return a list of tuples. Each tuple represents a found match, and its items are the

matched strings for each group in the regex. To see `findall()` in action, enter the following into the interactive shell (notice that the regular expression being compiled now has groups in parentheses):

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
[('415', '555', '1122'), ('212', '555', '0000')]
```

To summarize what the `findall()` method returns, remember the following:

1. When called on a regex with no groups, such as `\d\d\d-\d\d\d-\d\d\d\d`, the method `findall()` returns a list of string matches, such as `['415-555-9999', '212-555-0000']`.
2. When called on a regex that has groups, such as `(\d\d\d)-(\d\d\d)-(\d\d\d\d)`, the method `findall()` returns a list of tuples of strings (one string for each group), such as `[('415', '555', '1122'), ('212', '555', '0000')]`.

Character Classes

In the earlier phone number regex example, you learned that `\d` could stand for any numeric digit. That is, `\d` is shorthand for the regular expression `(0|1|2|3|4|5|6|7|8|9)`. There are many such *shorthand character classes*, as shown in Table 7-1.

Table 7-1: Shorthand Codes for Common Character Classes

Shorthand character class	Represents
`\d`	Any numeric digit from 0 to 9.
`\D`	Any character that is *not* a numeric digit from 0 to 9.
`\w`	Any letter, numeric digit, or the underscore character. (Think of this as matching "word" characters.)
`\W`	Any character that is *not* a letter, numeric digit, or the underscore character.
`\s`	Any space, tab, or newline character. (Think of this as matching "space" characters.)
`\S`	Any character that is *not* a space, tab, or newline.

Character classes are nice for shortening regular expressions. The character class `[0-5]` will match only the numbers `0` to `5`; this is much shorter than typing `(0|1|2|3|4|5)`.

For example, enter the following into the interactive shell:

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids, 7
swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans', '6
geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

The regular expression `\d+\s\w+` will match text that has one or more numeric digits (`\d+`), followed by a whitespace character (`\s`), followed by one or more letter/digit/underscore characters (`\w+`). The `findall()` method returns all matching strings of the regex pattern in a list.

Making Your Own Character Classes

There are times when you want to match a set of characters but the shorthand character classes (`\d`, `\w`, `\s`, and so on) are too broad. You can define your own character class using square brackets. For example, the character class `[aeiouAEIOU]` will match any vowel, both lowercase and uppercase. Enter the following into the interactive shell:

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')
>>> vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

You can also include ranges of letters or numbers by using a hyphen. For example, the character class `[a-zA-Z0-9]` will match all lowercase letters, uppercase letters, and numbers.

Note that inside the square brackets, the normal regular expression symbols are not interpreted as such. This means you do not need to escape the `.`, `*`, `?`, or `()` characters with a preceding backslash. For example, the character class `[0-5.]` will match digits `0` to `5` and a period. You do not need to write it as `[0-5\.]`.

By placing a caret character (`^`) just after the character class's opening bracket, you can make a *negative character class*. A negative character class will match all the characters that are *not* in the character class. For example, enter the following into the interactive shell:

```
>>> consonantRegex = re.compile(r'[^aeiouAEIOU]')
>>> consonantRegex.findall('RoboCop eats baby food. BABY FOOD.')
['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', '.', '
', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

Now, instead of matching every vowel, we're matching every character that isn't a vowel.

The Caret and Dollar Sign Characters

You can also use the caret symbol (`^`) at the start of a regex to indicate that a match must occur at the *beginning* of the searched text. Likewise, you can put a dollar sign (`$`) at the end of the regex to indicate the string must *end* with this regex pattern. And you can use the `^` and `$` together to indicate that the entire string must match the regex—that is, it's not enough for a match to be made on some subset of the string.

For example, the `r'^Hello'` regular expression string matches strings that begin with `'Hello'`. Enter the following into the interactive shell:

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello world!')
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> beginsWithHello.search('He said hello.') == None
True
```

The `r'\d$'` regular expression string matches strings that end with a numeric character from 0 to 9. Enter the following into the interactive shell:

```
>>> endsWithNumber = re.compile(r'\d$')
>>> endsWithNumber.search('Your number is 42')
<_sre.SRE_Match object; span=(16, 17), match='2'>
>>> endsWithNumber.search('Your number is forty two.') == None
True
```

The `r'^\d+$'` regular expression string matches strings that both begin and end with one or more numeric characters. Enter the following into the interactive shell:

```
>>> wholeStringIsNum = re.compile(r'^\d+$')
>>> wholeStringIsNum.search('1234567890')
<_sre.SRE_Match object; span=(0, 10), match='1234567890'>
>>> wholeStringIsNum.search('12345xyz67890') == None
True
>>> wholeStringIsNum.search('12  34567890') == None
True
```

The last two `search()` calls in the previous interactive shell example demonstrate how the entire string must match the regex if `^` and `$` are used.

I always confuse the meanings of these two symbols, so I use the mnemonic "Carrots cost dollars" to remind myself that the caret comes first and the dollar sign comes last.

The Wildcard Character

The `.` (or *dot*) character in a regular expression is called a *wildcard* and will match any character except for a newline. For example, enter the following into the interactive shell:

```
>>> atRegex = re.compile(r'.at')
>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

Remember that the dot character will match just one character, which is why the match for the text `flat` in the previous example matched only `lat`. To match an actual dot, escape the dot with a backslash: `\.`.

Matching Everything with Dot-Star

Sometimes you will want to match everything and anything. For example, say you want to match the string `'First Name:'`, followed by any and all text, followed by `'Last Name:'`, and then followed by anything again. You can use the dot-star (`.*`) to stand in for that "anything." Remember that the dot character means "any single character except the newline," and the star character means "zero or more of the preceding character."

Enter the following into the interactive shell:

```
>>> nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')
>>> mo = nameRegex.search('First Name: Al Last Name: Sweigart')
>>> mo.group(1)
'Al'
>>> mo.group(2)
'Sweigart'
```

The dot-star uses *greedy* mode: It will always try to match as much text as possible. To match any and all text in a *nongreedy* fashion, use the dot, star, and question mark (`.*?`). Like with curly brackets, the question mark tells Python to match in a nongreedy way.

Enter the following into the interactive shell to see the difference between the greedy and nongreedy versions:

```
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man>'

>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man> for dinner.>'
```

Both regexes roughly translate to "Match an opening angle bracket, followed by anything, followed by a closing angle bracket." But the string `'<To serve man> for dinner.>'` has two possible matches for the closing angle bracket. In the nongreedy version of the regex, Python matches the shortest possible string: `'<To serve man>'`. In the greedy version, Python matches the longest possible string: `'<To serve man> for dinner.>'`.

Matching Newlines with the Dot Character

The dot-star will match everything except a newline. By passing `re.DOTALL` as the second argument to `re.compile()`, you can make the dot character match *all* characters, including the newline character.

Enter the following into the interactive shell:

```
>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.'

>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

The regex `noNewlineRegex`, which did not have `re.DOTALL` passed to the `re.compile()` call that created it, will match everything only up to the first newline character, whereas `newlineRegex`, which *did* have `re.DOTALL` passed to `re.compile()`, matches everything. This is why the `newlineRegex.search()` call matches the full string, including its newline characters.

Review of Regex Symbols

This chapter covered a lot of notation, so here's a quick review of what you learned:

- The `?` matches zero or one of the preceding group.
- The `*` matches zero or more of the preceding group.
- The `+` matches one or more of the preceding group.
- The `{n}` matches exactly *n* of the preceding group.
- The `{n,}` matches *n* or more of the preceding group.
- The `{,m}` matches 0 to *m* of the preceding group.
- The `{n,m}` matches at least *n* and at most *m* of the preceding group.
- `{n,m}?` or `*?` or `+?` performs a nongreedy match of the preceding group.
- `^spam` means the string must begin with *spam*.
- `spam$` means the string must end with *spam*.
- The `.` matches any character, except newline characters.
- `\d`, `\w`, and `\s` match a digit, word, or space character, respectively.
- `\D`, `\W`, and `\S` match anything except a digit, word, or space character, respectively.
- `[abc]` matches any character between the brackets (such as *a*, *b*, or *c*).
- `[^abc]` matches any character that isn't between the brackets.

Case-Insensitive Matching

Normally, regular expressions match text with the exact casing you specify. For example, the following regexes match completely different strings:

```
>>> regex1 = re.compile('RoboCop')
>>> regex2 = re.compile('ROBOCOP')
>>> regex3 = re.compile('robOcop')
>>> regex4 = re.compile('RobocOp')
```

But sometimes you care only about matching the letters without worrying whether they're uppercase or lowercase. To make your regex case-insensitive, you can pass `re.IGNORECASE` or `re.I` as a second argument to `re.compile()`. Enter the following into the interactive shell:

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('RoboCop is part man, part machine, all cop.').group()
'RoboCop'

>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'

>>> robocop.search('Al, why does your programming book talk about robocop so much?').group()
'robocop'
```

Substituting Strings with the sub() Method

Regular expressions can not only find text patterns but can also substitute new text in place of those patterns. The `sub()` method for `Regex` objects is passed two arguments. The first argument is a string to replace any matches. The second is the string for the regular expression. The `sub()` method returns a string with the substitutions applied.

For example, enter the following into the interactive shell:

```
>>> namesRegex = re.compile(r'Agent \w+')
>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
'CENSORED gave the secret documents to CENSORED.'
```

Sometimes you may need to use the matched text itself as part of the substitution. In the first argument to `sub()`, you can type `\1`, `\2`, `\3`, and so on, to mean "Enter the text of group `1`, `2`, `3`, and so on, in the substitution."

For example, say you want to censor the names of the secret agents by showing just the first letters of their names. To do this, you could use the regex `Agent (\w)\w*` and pass `r'\1****'` as the first argument to `sub()`. The `\1` in that string will be replaced by whatever text was matched by group `1`—that is, the `(\w)` group of the regular expression.

```
>>> agentNamesRegex = re.compile(r'Agent (\w)\w*')
>>> agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent
Eve knew Agent Bob was a double agent.')
A**** told C**** that E**** knew B**** was a double agent.'
```

Managing Complex Regexes

Regular expressions are fine if the text pattern you need to match is simple. But matching complicated text patterns might require long, convoluted regular expressions. You can mitigate this by telling the `re.compile()` function to ignore whitespace and comments inside the regular expression string. This "verbose mode" can be enabled by passing the variable `re.VERBOSE` as the second argument to `re.compile()`.

Now instead of a hard-to-read regular expression like this:

```
phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4}
(\s*(ext|x|ext.)\s*\d{2,5})?)')
```

you can spread the regular expression over multiple lines with comments like this:

```
phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?           # area code
    (\s|-|\.)?                   # separator
    \d{3}                        # first 3 digits
    (\s|-|\.)                    # separator
    \d{4}                        # last 4 digits
    (\s*(ext|x|ext.)\s*\d{2,5})?  # extension
    )''', re.VERBOSE)
```

Note how the previous example uses the triple-quote syntax (`'''`) to create a multiline string so that you can spread the regular expression definition over many lines, making it much more legible.

The comment rules inside the regular expression string are the same as regular Python code: The `#` symbol and everything after it to the end of the line are ignored. Also, the extra spaces inside the multiline string for the regular expression are not considered part of the text pattern to be matched. This lets you organize the regular expression so it's easier to read.

Combining re.IGNORECASE, re.DOTALL, and re.VERBOSE

What if you want to use `re.VERBOSE` to write comments in your regular expression but also want to use `re.IGNORECASE` to ignore capitalization? Unfortunately, the `re.compile()` function takes only a single value as its second argument. You can get around this limitation by combining the `re.IGNORECASE`, `re.DOTALL`, and `re.VERBOSE` variables using the pipe character (`|`), which in this context is known as the *bitwise or* operator.

So if you want a regular expression that's case-insensitive *and* includes newlines to match the dot character, you would form your `re.compile()` call like this:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL)
```

All three options for the second argument will look like this:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERBOSE)
```

This syntax is a little old-fashioned and originates from early versions of Python. The details of the bitwise operators are beyond the scope of this book, but check out the resources at *http://nostarch.com/automatestuff/* for more information. You can also pass other options for the second argument; they're uncommon, but you can read more about them in the resources, too.

Project: Phone Number and Email Address Extractor

Say you have the boring task of finding every phone number and email address in a long web page or document. If you manually scroll through the page, you might end up searching for a long time. But if you had a program that could search the text in your clipboard for phone numbers and email addresses, you could simply press CTRL-A to select all the text, press CTRL-C to copy it to the clipboard, and then run your program. It could replace the text on the clipboard with just the phone numbers and email addresses it finds.

Whenever you're tackling a new project, it can be tempting to dive right into writing code. But more often than not, it's best to take a step back and consider the bigger picture. I recommend first drawing up a high-level plan for what your program needs to do. Don't think about the actual code yet—you can worry about that later. Right now, stick to broad strokes.

For example, your phone and email address extractor will need to do the following:

- Get the text off the clipboard.
- Find all phone numbers and email addresses in the text.
- Paste them onto the clipboard.

Now you can start thinking about how this might work in code. The code will need to do the following:

- Use the `pyperclip` module to copy and paste strings.
- Create two regexes, one for matching phone numbers and the other for matching email addresses.
- Find all matches, not just the first match, of both regexes.
- Neatly format the matched strings into a single string to paste.
- Display some kind of message if no matches were found in the text.

This list is like a road map for the project. As you write the code, you can focus on each of these steps separately. Each step is fairly manageable and expressed in terms of things you already know how to do in Python.

Step 1: Create a Regex for Phone Numbers

First, you have to create a regular expression to search for phone numbers. Create a new file, enter the following, and save it as *phoneAndEmail.py*:

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

import pyperclip, re

phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?              # area code
    (\s|-|\.)?                      # separator
    (\d{3})                         # first 3 digits
    (\s|-|\.)                       # separator
    (\d{4})                         # last 4 digits
    (\s*(ext|x|ext.)\s*(\d{2,5}))?  # extension
    )''', re.VERBOSE)

# TODO: Create email regex.

# TODO: Find matches in clipboard text.

# TODO: Copy results to the clipboard.
```

The `TODO` comments are just a skeleton for the program. They'll be replaced as you write the actual code.

The phone number begins with an *optional* area code, so the area code group is followed with a question mark. Since the area code can be just three digits (that is, `\d{3}`) *or* three digits within parentheses (that is, `\(\d{3}\)`), you should have a pipe joining those parts. You can add the regex comment `# Area code` to this part of the multiline string to help you remember what `(\d{3}|\(\d{3}\))?` is supposed to match.

The phone number separator character can be a space (`\s`), hyphen (`-`), or period (`.`), so these parts should also be joined by pipes. The next few parts of the regular expression are straightforward: three digits, followed by another separator, followed by four digits. The last part is an optional extension made up of any number of spaces followed by `ext`, `x`, or `ext.`, followed by two to five digits.

Step 2: Create a Regex for Email Addresses

You will also need a regular expression that can match email addresses. Make your program look like the following:

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.
```

```
import pyperclip, re

phoneRegex = re.compile(r'''(
--snip--

# Create email regex.
emailRegex = re.compile(r'''(
❶     [a-zA-Z0-9._%+-]+      # username
❷     @                      # @ symbol
❸     [a-zA-Z0-9.-]+         # domain name
      (\.[a-zA-Z]{2,4})      # dot-something
      )''', re.VERBOSE)

# TODO: Find matches in clipboard text.

# TODO: Copy results to the clipboard.
```

The username part of the email address ❶ is one or more characters that can be any of the following: lowercase and uppercase letters, numbers, a dot, an underscore, a percent sign, a plus sign, or a hyphen. You can put all of these into a character class: `[a-zA-Z0-9._%+-]`.

The domain and username are separated by an @ symbol ❷. The domain name ❸ has a slightly less permissive character class with only letters, numbers, periods, and hyphens: `[a-zA-Z0-9.-]`. And last will be the "dot-com" part (technically known as the *top-level domain*), which can really be dot-anything. This is between two and four characters.

The format for email addresses has a lot of weird rules. This regular expression won't match every possible valid email address, but it'll match almost any typical email address you'll encounter.

Step 3: Find All Matches in the Clipboard Text

Now that you have specified the regular expressions for phone numbers and email addresses, you can let Python's `re` module do the hard work of finding all the matches on the clipboard. The `pyperclip.paste()` function will get a string value of the text on the clipboard, and the `findall()` regex method will return a list of tuples.

Make your program look like the following:

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

import pyperclip, re

phoneRegex = re.compile(r'''(
--snip--

# Find matches in clipboard text.
text = str(pyperclip.paste())
```

```
❶ matches = []
❷ for groups in phoneRegex.findall(text):
      phoneNum = '-'.join([groups[1], groups[3], groups[5]])
      if groups[8] != '':
          phoneNum += ' x' + groups[8]
      matches.append(phoneNum)
❸ for groups in emailRegex.findall(text):
      matches.append(groups[0])

  # TODO: Copy results to the clipboard.
```

There is one tuple for each match, and each tuple contains strings for each group in the regular expression. Remember that group `0` matches the entire regular expression, so the group at index `0` of the tuple is the one you are interested in.

As you can see at ❶, you'll store the matches in a list variable named `matches`. It starts off as an empty list, and a couple `for` loops. For the email addresses, you append group `0` of each match ❸. For the matched phone numbers, you don't want to just append group `0`. While the program *detects* phone numbers in several formats, you want the phone number appended to be in a single, standard format. The `phoneNum` variable contains a string built from groups `1`, `3`, `5`, and `8` of the matched text ❷. (These groups are the area code, first three digits, last four digits, and extension.)

Step 4: Join the Matches into a String for the Clipboard

Now that you have the email addresses and phone numbers as a list of strings in `matches`, you want to put them on the clipboard. The `pyperclip.copy()` function takes only a single string value, not a list of strings, so you call the `join()` method on `matches`.

To make it easier to see that the program is working, let's print any matches you find to the terminal. And if no phone numbers or email addresses were found, the program should tell the user this.

Make your program look like the following:

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

--snip--
for groups in emailRegex.findall(text):
    matches.append(groups[0])

# Copy results to the clipboard.
if len(matches) > 0:
    pyperclip.copy('\n'.join(matches))
    print('Copied to clipboard:')
    print('\n'.join(matches))
else:
    print('No phone numbers or email addresses found.')
```

Running the Program

For an example, open your web browser to the No Starch Press contact page at *http://www.nostarch.com/contactus.htm*, press CTRL-A to select all the text on the page, and press CTRL-C to copy it to the clipboard. When you run this program, the output will look something like this:

```
Copied to clipboard:
800-420-7240
415-863-9900
415-863-9950
info@nostarch.com
media@nostarch.com
academic@nostarch.com
help@nostarch.com
```

Ideas for Similar Programs

Identifying patterns of text (and possibly substituting them with the `sub()` method) has many different potential applications.

- Find website URLs that begin with *http://* or *https://*.
- Clean up dates in different date formats (such as 3/14/2015, 03-14-2015, and 2015/3/14) by replacing them with dates in a single, standard format.
- Remove sensitive information such as Social Security or credit card numbers.
- Find common typos such as multiple spaces between words, accidentally accidentally repeated words, or multiple exclamation marks at the end of sentences. Those are annoying!!

Summary

While a computer can search for text quickly, it must be told precisely what to look for. Regular expressions allow you to specify the precise patterns of characters you are looking for. In fact, some word processing and spreadsheet applications provide find-and-replace features that allow you to search using regular expressions.

The `re` module that comes with Python lets you compile `Regex` objects. These values have several methods: `search()` to find a single match, `findall()` to find all matching instances, and `sub()` to do a find-and-replace substitution of text.

There's a bit more to regular expression syntax than is described in this chapter. You can find out more in the official Python documentation at *http://docs.python.org/3/library/re.html*. The tutorial website *http://www.regular-expressions.info/* is also a useful resource.

Now that you have expertise manipulating and matching strings, it's time to dive into how to read from and write to files on your computer's hard drive.

Practice Questions

1. What is the function that creates `Regex` objects?
2. Why are raw strings often used when creating `Regex` objects?
3. What does the `search()` method return?
4. How do you get the actual strings that match the pattern from a `Match` object?
5. In the regex created from `r'(\d\d\d)-(\d\d\d-\d\d\d\d)'`, what does group `0` cover? Group `1`? Group `2`?
6. Parentheses and periods have specific meanings in regular expression syntax. How would you specify that you want a regex to match actual parentheses and period characters?
7. The `findall()` method returns a list of strings or a list of tuples of strings. What makes it return one or the other?
8. What does the `|` character signify in regular expressions?
9. What two things does the `?` character signify in regular expressions?
10. What is the difference between the `+` and `*` characters in regular expressions?
11. What is the difference between `{3}` and `{3,5}` in regular expressions?
12. What do the `\d`, `\w`, and `\s` shorthand character classes signify in regular expressions?
13. What do the `\D`, `\W`, and `\S` shorthand character classes signify in regular expressions?
14. How do you make a regular expression case-insensitive?
15. What does the `.` character normally match? What does it match if `re.DOTALL` is passed as the second argument to `re.compile()`?
16. What is the difference between `.*` and `.*?`?
17. What is the character class syntax to match all numbers and lowercase letters?
18. If `numRegex = re.compile(r'\d+')`, what will `numRegex.sub('X', '12 drummers, 11 pipers, five rings, 3 hens')` return?
19. What does passing `re.VERBOSE` as the second argument to `re.compile()` allow you to do?
20. How would you write a regex that matches a number with commas for every three digits? It must match the following:
    - `'42'`
    - `'1,234'`
    - `'6,368,745'`

    but not the following:
    - `'12,34,567'` (which has only two digits between the commas)
    - `'1234'` (which lacks commas)

21. How would you write a regex that matches the full name of someone whose last name is Nakamoto? You can assume that the first name that comes before it will always be one word that begins with a capital letter. The regex must match the following:
    - `'Satoshi Nakamoto'`
    - `'Alice Nakamoto'`
    - `'RoboCop Nakamoto'`

    but not the following:
    - `'satoshi Nakamoto'` (where the first name is not capitalized)
    - `'Mr. Nakamoto'` (where the preceding word has a nonletter character)
    - `'Nakamoto'` (which has no first name)
    - `'Satoshi nakamoto'` (where Nakamoto is not capitalized)
22. How would you write a regex that matches a sentence where the first word is either *Alice*, *Bob*, or *Carol*; the second word is either *eats*, *pets*, or *throws*; the third word is *apples*, *cats*, or *baseballs*; and the sentence ends with a period? This regex should be case-insensitive. It must match the following:
    - `'Alice eats apples.'`
    - `'Bob pets cats.'`
    - `'Carol throws baseballs.'`
    - `'Alice throws Apples.'`
    - `'BOB EATS CATS.'`

    but not the following:
    - `'RoboCop eats apples.'`
    - `'ALICE THROWS FOOTBALLS.'`
    - `'Carol eats 7 cats.'`

Practice Projects

For practice, write programs to do the following tasks.

Strong Password Detection

Write a function that uses regular expressions to make sure the password string it is passed is strong. A strong password is defined as one that is at least eight characters long, contains both uppercase and lowercase characters, and has at least one digit. You may need to test the string against multiple regex patterns to validate its strength.

Regex Version of strip()

Write a function that takes a string and does the same thing as the `strip()` string method. If no other arguments are passed other than the string to strip, then whitespace characters will be removed from the beginning and end of the string. Otherwise, the characters specified in the second argument to the function will be removed from the string.

8

READING AND WRITING FILES

Variables are a fine way to store data while your program is running, but if you want your data to persist even after your program has finished, you need to save it to a file. You can think of a file's contents as a single string value, potentially gigabytes in size. In this chapter, you will learn how to use Python to create, read, and save files on the hard drive.

Files and File Paths

A file has two key properties: a *filename* (usually written as one word) and a *path*. The path specifies the location of a file on the computer. For example, there is a file on my Windows 7 laptop with the filename *projects.docx* in the path *C:\Users\asweigart\Documents*. The part of the filename after the last period is called the file's *extension* and tells you a file's type. *project.docx* is a Word document, and *Users*, *asweigart*, and *Documents* all refer to *folders* (also

called *directories*). Folders can contain files and other folders. For example, *project.docx* is in the *Documents* folder, which is inside the *asweigart* folder, which is inside the *Users* folder. Figure 8-1 shows this folder organization.

Figure 8-1: A file in a hierarchy of folders

The *C:* part of the path is the *root folder*, which contains all other folders. On Windows, the root folder is named *C:* and is also called the *C: drive*. On OS X and Linux, the root folder is */*. In this book, I'll be using the Windows-style root folder, *C:*. If you are entering the interactive shell examples on OS X or Linux, enter `/` instead.

Additional *volumes*, such as a DVD drive or USB thumb drive, will appear differently on different operating systems. On Windows, they appear as new, lettered root drives, such as *D:* or *E:*. On OS X, they appear as new folders under the */Volumes* folder. On Linux, they appear as new folders under the */mnt* ("mount") folder. Also note that while folder names and filenames are not case sensitive on Windows and OS X, they are case sensitive on Linux.

Backslash on Windows and Forward Slash on OS X and Linux

On Windows, paths are written using backslashes (\) as the separator between folder names. OS X and Linux, however, use the forward slash (/) as their path separator. If you want your programs to work on all operating systems, you will have to write your Python scripts to handle both cases.

Fortunately, this is simple to do with the `os.path.join()` function. If you pass it the string values of individual file and folder names in your path, `os.path.join()` will return a string with a file path using the correct path separators. Enter the following into the interactive shell:

```
>>> import os
>>> os.path.join('usr', 'bin', 'spam')
'usr\\bin\\spam'
```

I'm running these interactive shell examples on Windows, so `os.path.join('usr', 'bin', 'spam')` returned `'usr\\bin\\spam'`. (Notice that the backslashes are doubled because each backslash needs to be escaped by another backslash character.) If I had called this function on OS X or Linux, the string would have been `'usr/bin/spam'`.

The os.path.join() function is helpful if you need to create strings for filenames. These strings will be passed to several of the file-related functions introduced in this chapter. For example, the following example joins names from a list of filenames to the end of a folder's name:

```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
```

```
        print(os.path.join('C:\\Users\\asweigart', filename))
C:\Users\asweigart\accounts.txt
C:\Users\asweigart\details.csv
C:\Users\asweigart\invite.docx
```

The Current Working Directory

Every program that runs on your computer has a *current working directory*, or *cwd*. Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory. You can get the current working directory as a string value with the `os.getcwd()` function and change it with `os.chdir()`. Enter the following into the interactive shell:

```
>>> import os
>>> os.getcwd()
'C:\\Python34'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

Here, the current working directory is set to *C:\Python34*, so the filename *project.docx* refers to *C:\Python34\project.docx*. When we change the current working directory to *C:\Windows*, *project.docx* is interpreted as *C:\Windows\project.docx*.

Python will display an error if you try to change to a directory that does not exist.

```
>>> os.chdir('C:\\ThisFolderDoesNotExist')
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    os.chdir('C:\\ThisFolderDoesNotExist')
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:\\ThisFolderDoesNotExist'
```

NOTE *While folder is the more modern name for directory, note that* current working directory *(or just* working directory*) is the standard term, not current working folder.*

Absolute vs. Relative Paths

There are two ways to specify a file path.

- An *absolute path*, which always begins with the root folder
- A *relative path*, which is relative to the program's current working directory

There are also the *dot* (`.`) and *dot-dot* (`..`) folders. These are not real folders but special names that can be used in a path. A single period ("dot") for a folder name is shorthand for "this directory." Two periods ("dot-dot") means "the parent folder."

Figure 8-2 is an example of some folders and files. When the current working directory is set to *C:\bacon*, the relative paths for the other folders and files are set as they are in the figure.

Figure 8-2: The relative paths for folders and files in the working directory C:\bacon

The .\ at the start of a relative path is optional. For example, *.\spam.txt* and *spam.txt* refer to the same file.

Creating New Folders with os.makedirs()

Your programs can create new folders (directories) with the `os.makedirs()` function. Enter the following into the interactive shell:

```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

This will create not just the *C:\delicious* folder but also a *walnut* folder inside *C:\delicious* and a *waffles* folder inside *C:\delicious\walnut*. That is, `os.makedirs()` will create any necessary intermediate folders in order to ensure that the full path exists. Figure 8-3 shows this hierarchy of folders.

Figure 8-3: The result of `os.makedirs('C:\\delicious\\walnut\\waffles')`

The os.path Module

The `os.path` module contains many helpful functions related to filenames and file paths. For instance, you've already used `os.path.join()` to build paths in a way that will work on any operating system. Since `os.path` is a module inside the `os` module, you can import it by simply running `import os`. Whenever your programs need to work with files, folders, or file paths, you can refer to the short examples in this section. The full documentation for the `os.path` module is on the Python website at *http://docs.python.org/3/library/os.path.html*.

NOTE *Most of the examples that follow in this section will require the* `os` *module, so remember to import it at the beginning of any script you write and any time you restart IDLE. Otherwise, you'll get a* `NameError: name 'os' is not defined` *error message.*

Handling Absolute and Relative Paths

The `os.path` module provides functions for returning the absolute path of a relative path and for checking whether a given path is an absolute path.

- Calling `os.path.abspath(`*`path`*`)` will return a string of the absolute path of the argument. This is an easy way to convert a relative path into an absolute one.
- Calling `os.path.isabs(`*`path`*`)` will return `True` if the argument is an absolute path and `False` if it is a relative path.
- Calling `os.path.relpath(`*`path, start`*`)` will return a string of a relative path from the *`start`* path to *`path`*. If *`start`* is not provided, the current working directory is used as the start path.

Try these functions in the interactive shell:

```
>>> os.path.abspath('.')
'C:\\Python34'
>>> os.path.abspath('.\\Scripts')
'C:\\Python34\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

Since *C:\Python34* was the working directory when `os.path.abspath()` was called, the "single-dot" folder represents the absolute path `'C:\\Python34'`.

NOTE *Since your system probably has different files and folders on it than mine, you won't be able to follow every example in this chapter exactly. Still, try to follow along using folders that exist on your computer.*

Enter the following calls to os.path.relpath() into the interactive shell:

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> os.getcwd()
'C:\\Python34'
```

Calling os.path.dirname(*path*) will return a string of everything that comes before the last slash in the path argument. Calling os.path.basename(*path*) will return a string of everything that comes after the last slash in the path argument. The dir name and base name of a path are outlined in Figure 8-4.

Figure 8-4: The base name follows the last slash in a path and is the same as the filename. The dir name is everything before the last slash.

For example, enter the following into the interactive shell:

```
>>> path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'C:\\Windows\\System32'
```

If you need a path's dir name and base name together, you can just call os.path.split() to get a tuple value with these two strings, like so:

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.split(calcFilePath)
('C:\\Windows\\System32', 'calc.exe')
```

Notice that you could create the same tuple by calling os.path.dirname() and os.path.basename() and placing their return values in a tuple.

```
>>> (os.path.dirname(calcFilePath), os.path.basename(calcFilePath))
('C:\\Windows\\System32', 'calc.exe')
```

But os.path.split() is a nice shortcut if you need both values.

Also, note that os.path.split() does *not* take a file path and return a list of strings of each folder. For that, use the split() string method and split on the string in os.sep. Recall from earlier that the os.sep variable is set to the correct folder-separating slash for the computer running the program.

For example, enter the following into the interactive shell:

```
>>> calcFilePath.split(os.path.sep)
['C:', 'Windows', 'System32', 'calc.exe']
```

On OS X and Linux systems, there will be a blank string at the start of the returned list:

```
>>> '/usr/bin'.split(os.path.sep)
['', 'usr', 'bin']
```

The `split()` string method will work to return a list of each part of the path. It will work on any operating system if you pass it `os.path.sep`.

Finding File Sizes and Folder Contents

Once you have ways of handling file paths, you can then start gathering information about specific files and folders. The `os.path` module provides functions for finding the size of a file in bytes and the files and folders inside a given folder.

- Calling `os.path.getsize(`*`path`*`)` will return the size in bytes of the file in the *path* argument.
- Calling `os.listdir(`*`path`*`)` will return a list of filename strings for each file in the *path* argument. (Note that this function is in the `os` module, not `os.path`.)

Here's what I get when I try these functions in the interactive shell:

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
776192
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
--snip--
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

As you can see, the *calc.exe* program on my computer is 776,192 bytes in size, and I have a lot of files in *C:\Windows\system32*. If I want to find the total size of all the files in this directory, I can use `os.path.getsize()` and `os.listdir()` together.

```
>>> totalSize = 0
>>> for filename in os.listdir('C:\\Windows\\System32'):
      totalSize = totalSize + os.path.getsize(os.path.join('C:\\Windows\\System32', filename))

>>> print(totalSize)
1117846456
```

As I loop over each filename in the *C:\Windows\System32* folder, the `totalSize` variable is incremented by the size of each file. Notice how when I call `os.path.getsize()`, I use `os.path.join()` to join the folder name with the current filename. The integer that `os.path.getsize()` returns is added to the value of `totalSize`. After looping through all the files, I print `totalSize` to see the total size of the *C:\Windows\System32* folder.

Checking Path Validity

Many Python functions will crash with an error if you supply them with a path that does not exist. The `os.path` module provides functions to check whether a given path exists and whether it is a file or folder.

- Calling `os.path.exists(`*`path`*`)` will return `True` if the file or folder referred to in the argument exists and will return `False` if it does not exist.
- Calling `os.path.isfile(`*`path`*`)` will return `True` if the path argument exists and is a file and will return `False` otherwise.
- Calling `os.path.isdir(`*`path`*`)` will return `True` if the path argument exists and is a folder and will return `False` otherwise.

Here's what I get when I try these functions in the interactive shell:

```
>>> os.path.exists('C:\\Windows')
True
>>> os.path.exists('C:\\some_made_up_folder')
False
>>> os.path.isdir('C:\\Windows\\System32')
True
>>> os.path.isfile('C:\\Windows\\System32')
False
>>> os.path.isdir('C:\\Windows\\System32\\calc.exe')
False
>>> os.path.isfile('C:\\Windows\\System32\\calc.exe')
True
```

You can determine whether there is a DVD or flash drive currently attached to the computer by checking for it with the `os.path.exists()` function. For instance, if I wanted to check for a flash drive with the volume named *D:* on my Windows computer, I could do that with the following:

```
>>> os.path.exists('D:\\')
False
```

Oops! It looks like I forgot to plug in my flash drive.

The File Reading/Writing Process

Once you are comfortable working with folders and relative paths, you'll be able to specify the location of files to read and write. The functions covered in the next few sections will apply to plaintext files. *Plaintext files*

contain only basic text characters and do not include font, size, or color information. Text files with the *.txt* extension or Python script files with the *.py* extension are examples of plaintext files. These can be opened with Windows's Notepad or OS X's TextEdit application. Your programs can easily read the contents of plaintext files and treat them as an ordinary string value.

Binary files are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs. If you open a binary file in Notepad or TextEdit, it will look like scrambled nonsense, like in Figure 8-5.

Figure 8-5: The Windows `calc.exe` program opened in Notepad

Since every different type of binary file must be handled in its own way, this book will not go into reading and writing raw binary files directly. Fortunately, many modules make working with binary files easier—you will explore one of them, the `shelve` module, later in this chapter.

There are three steps to reading or writing files in Python.

1. Call the `open()` function to return a `File` object.
2. Call the `read()` or `write()` method on the `File` object.
3. Close the file by calling the `close()` method on the `File` object.

Opening Files with the open() Function

To open a file with the `open()` function, you pass it a string path indicating the file you want to open; it can be either an absolute or relative path. The `open()` function returns a `File` object.

Try it by creating a text file named *hello.txt* using Notepad or TextEdit. Type **`Hello world!`** as the content of this text file and save it in your user home folder. Then, if you're using Windows, enter the following into the interactive shell:

```
>>> helloFile = open('C:\\Users\\your_home_folder\\hello.txt')
```

If you're using OS X, enter the following into the interactive shell instead:

```
>>> helloFile = open('/Users/your_home_folder/hello.txt')
```

Make sure to replace *your_home_folder* with your computer username. For example, my username is *asweigart*, so I'd enter `'C:\\Users\\asweigart\\hello.txt'` on Windows.

Both these commands will open the file in "reading plaintext" mode, or *read mode* for short. When a file is opened in read mode, Python lets you only read data from the file; you can't write or modify it in any way. Read mode is the default mode for files you open in Python. But if you don't want to rely on Python's defaults, you can explicitly specify the mode by passing the string value `'r'` as a second argument to `open()`. So `open('/Users/asweigart/hello.txt', 'r')` and `open('/Users/asweigart/hello.txt')` do the same thing.

The call to `open()` returns a `File` object. A `File` object represents a file on your computer; it is simply another type of value in Python, much like the lists and dictionaries you're already familiar with. In the previous example, you stored the `File` object in the variable `helloFile`. Now, whenever you want to read from or write to the file, you can do so by calling methods on the `File` object in `helloFile`.

Reading the Contents of Files

Now that you have a `File` object, you can start reading from it. If you want to read the entire contents of a file as a string value, use the `File` object's `read()` method. Let's continue with the *hello.txt* `File` object you stored in `helloFile` . Enter the following into the interactive shell:

```
>>> helloContent = helloFile.read()
>>> helloContent
'Hello world!'
```

If you think of the contents of a file as a single large string value, the `read()` method returns the string that is stored in the file.

Alternatively, you can use the `readlines()` method to get a *list* of string values from the file, one string for each line of text. For example, create a file named *sonnet29.txt* in the same directory as *hello.txt* and write the following text in it:

```
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself and curse my fate,
```

Make sure to separate the four lines with line breaks. Then enter the following into the interactive shell:

```
>>> sonnetFile = open('sonnet29.txt')
>>> sonnetFile.readlines()
[When, in disgrace with fortune and men's eyes,\n', ' I all alone beweep my
outcast state,\n', And trouble deaf heaven with my bootless cries,\n', And
look upon myself and curse my fate,']
```

Note that each of the string values ends with a newline character, `\n` , except for the last line of the file. A list of strings is often easier to work with than a single large string value.

Writing to Files

Python allows you to write content to a file in a way similar to how the `print()` function "writes" strings to the screen. You can't write to a file you've opened in read mode, though. Instead, you need to open it in "write plaintext" mode or "append plaintext" mode, or *write mode* and *append mode* for short.

Write mode will overwrite the existing file and start from scratch, just like when you overwrite a variable's value with a new value. Pass `'w'` as the second argument to `open()` to open the file in write mode. Append mode, on the other hand, will append text to the end of the existing file. You can think of this as appending to a list in a variable, rather than overwriting the variable altogether. Pass `'a'` as the second argument to `open()` to open the file in append mode.

If the filename passed to `open()` does not exist, both write and append mode will create a new, blank file. After reading or writing a file, call the `close()` method before opening the file again.

Let's put these concepts together. Enter the following into the interactive shell:

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello world!
Bacon is not a vegetable.
```

First, we open *bacon.txt* in write mode. Since there isn't a *bacon.txt* yet, Python creates one. Calling `write()` on the opened file and passing `write()` the string argument `'Hello world! /n'` writes the string to the file and returns the number of characters written, including the newline. Then we close the file.

To add text to the existing contents of the file instead of replacing the string we just wrote, we open the file in append mode. We write `'Bacon is not a vegetable.'` to the file and close it. Finally, to print the file contents to the screen, we open the file in its default read mode, call `read()`, store the resulting `File` object in `content`, close the file, and print `content`.

Note that the `write()` method does not automatically add a newline character to the end of the string like the `print()` function does. You will have to add this character yourself.

Saving Variables with the shelve Module

You can save variables in your Python programs to binary shelf files using the `shelve` module. This way, your program can restore data to variables from the hard drive. The `shelve` module will let you add Save and Open features to your program. For example, if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load them the next time it is run.

Enter the following into the interactive shell:

```
>>> import shelve
>>> shelfFile = shelve.open('mydata')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelfFile['cats'] = cats
>>> shelfFile.close()
```

To read and write data using the `shelve` module, you first import `shelve`. Call `shelve.open()` and pass it a filename, and then store the returned shelf value in a variable. You can make changes to the shelf value as if it were a dictionary. When you're done, call `close()` on the shelf value. Here, our shelf value is stored in `shelfFile`. We create a list `cats` and write `shelfFile['cats'] = cats` to store the list in `shelfFile` as a value associated with the key `'cats'` (like in a dictionary). Then we call `close()` on `shelfFile`.

After running the previous code on Windows, you will see three new files in the current working directory: *mydata.bak*, *mydata.dat*, and *mydata.dir*. On OS X, only a single *mydata.db* file will be created.

These binary files contain the data you stored in your shelf. The format of these binary files is not important; you only need to know what the `shelve` module does, not how it does it. The module frees you from worrying about how to store your program's data to a file.

Your programs can use the `shelve` module to later reopen and retrieve the data from these shelf files. Shelf values don't have to be opened in read or write mode—they can do both once opened. Enter the following into the interactive shell:

```
>>> shelfFile = shelve.open('mydata')
>>> type(shelfFile)
<class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelfFile.close()
```

Here, we open the shelf files to check that our data was stored correctly. Entering `shelfFile['cats']` returns the same list that we stored earlier, so we know that the list is correctly stored, and we call `close()`.

Just like dictionaries, shelf values have `keys()` and `values()` methods that will return list-like values of the keys and values in the shelf. Since these methods return list-like values instead of true lists, you should pass them to the `list()` function to get them in list form. Enter the following into the interactive shell:

```
>>> shelfFile = shelve.open('mydata')
>>> list(shelfFile.keys())
['cats']
>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon']]
>>> shelfFile.close()
```

Plaintext is useful for creating files that you'll read in a text editor such as Notepad or TextEdit, but if you want to save data from your Python programs, use the `shelve` module.

Saving Variables with the pprint.pformat() Function

Recall from "Pretty Printing" on page 111 that the `pprint.pprint()` function will "pretty print" the contents of a list or dictionary to the screen, while the `pprint.pformat()` function will return this same text as a string instead of printing it. Not only is this string formatted to be easy to read, but it is also syntactically correct Python code. Say you have a dictionary stored in a variable and you want to save this variable and its contents for future use. Using `pprint.pformat()` will give you a string that you can write to *.py* file. This file will be your very own module that you can import whenever you want to use the variable stored in it.

For example, enter the following into the interactive shell:

```
>>> import pprint
>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
>>> fileObj.close()
```

Here, we import `pprint` to let us use `pprint.pformat()`. We have a list of dictionaries, stored in a variable `cats`. To keep the list in `cats` available even after we close the shell, we use `pprint.pformat()` to return it as a string. Once we have the data in `cats` as a string, it's easy to write the string to a file, which we'll call *myCats.py*.

The modules that an `import` statement imports are themselves just Python scripts. When the string from `pprint.pformat()` is saved to a *.py* file, the file is a module that can be imported just like any other.

And since Python scripts are themselves just text files with the *.py* file extension, your Python programs can even generate other Python programs. You can then import these files into scripts.

```
>>> import myCats
>>> myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
>>> myCats.cats[0]['name']
'Zophie'
```

The benefit of creating a *.py* file (as opposed to saving variables with the `shelve` module) is that because it is a text file, the contents of the file can be read and modified by anyone with a simple text editor. For most applications, however, saving data using the `shelve` module is the preferred way to save variables to a file. Only basic data types such as integers, floats, strings, lists, and dictionaries can be written to a file as simple text. `File` objects, for example, cannot be encoded as text.

Project: Generating Random Quiz Files

Say you're a geography teacher with 35 students in your class and you want to give a pop quiz on US state capitals. Alas, your class has a few bad eggs in it, and you can't trust the students not to cheat. You'd like to randomize the order of questions so that each quiz is unique, making it impossible for anyone to crib answers from anyone else. Of course, doing this by hand would be a lengthy and boring affair. Fortunately, you know some Python.

Here is what the program does:

- Creates 35 different quizzes.
- Creates 50 multiple-choice questions for each quiz, in random order.
- Provides the correct answer and three random wrong answers for each question, in random order.
- Writes the quizzes to 35 text files.
- Writes the answer keys to 35 text files.

This means the code will need to do the following:

- Store the states and their capitals in a dictionary.
- Call `open()`, `write()`, and `close()` for the quiz and answer key text files.
- Use `random.shuffle()` to randomize the order of the questions and multiple-choice options.

Step 1: Store the Quiz Data in a Dictionary

The first step is to create a skeleton script and fill it with your quiz data. Create a file named *randomQuizGenerator.py*, and make it look like the following:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

❶ import random

# The quiz data. Keys are states and values are their capitals.
❷ capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Phoenix',
'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado': 'Denver',
'Connecticut': 'Hartford', 'Delaware': 'Dover', 'Florida': 'Tallahassee',
'Georgia': 'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise', 'Illinois':
'Springfield', 'Indiana': 'Indianapolis', 'Iowa': 'Des Moines', 'Kansas':
'Topeka', 'Kentucky': 'Frankfort', 'Louisiana': 'Baton Rouge', 'Maine':
'Augusta', 'Maryland': 'Annapolis', 'Massachusetts': 'Boston', 'Michigan':
'Lansing', 'Minnesota': 'Saint Paul', 'Mississippi': 'Jackson', 'Missouri':
'Jefferson City', 'Montana': 'Helena', 'Nebraska': 'Lincoln', 'Nevada':
'Carson City', 'New Hampshire': 'Concord', 'New Jersey': 'Trenton', 'New
Mexico': 'Santa Fe', 'New York': 'Albany', 'North Carolina': 'Raleigh',
'North Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma City',
'Oregon': 'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Providence',
'South Carolina': 'Columbia', 'South Dakota': 'Pierre', 'Tennessee':
'Nashville', 'Texas': 'Austin', 'Utah': 'Salt Lake City', 'Vermont':
'Montpelier', 'Virginia': 'Richmond', 'Washington': 'Olympia', 'West
Virginia': 'Charleston', 'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}

# Generate 35 quiz files.
❸ for quizNum in range(35):
    # TODO: Create the quiz and answer key files.

    # TODO: Write out the header for the quiz.

    # TODO: Shuffle the order of the states.

    # TODO: Loop through all 50 states, making a question for each.
```

Since this program will be randomly ordering the questions and answers, you'll need to import the `random` module ❶ to make use of its functions. The `capitals` variable ❷ contains a dictionary with US states as keys and their capitals as values. And since you want to create 35 quizzes, the code that actually generates the quiz and answer key files (marked with `TODO` comments for now) will go inside a `for` loop that loops 35 times ❸. (This number can be changed to generate any number of quiz files.)

Step 2: Create the Quiz File and Shuffle the Question Order

Now it's time to start filling in those TODOs.

The code in the loop will be repeated 35 times—once for each quiz—so you have to worry about only one quiz at a time within the loop. First you'll create the actual quiz file. It needs to have a unique filename and should also have some kind of standard header in it, with places for the student to fill in a name, date, and class period. Then you'll need to get a list of states in randomized order, which can be used later to create the questions and answers for the quiz.

Add the following lines of code to *randomQuizGenerator.py*:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip--

# Generate 35 quiz files.
for quizNum in range(35):
    # Create the quiz and answer key files.
❶   quizFile = open('capitalsquiz%s.txt' % (quizNum + 1), 'w')
❷   answerKeyFile = open('capitalsquiz_answers%s.txt' % (quizNum + 1), 'w')

    # Write out the header for the quiz.
❸   quizFile.write('Name:\n\nDate:\n\nPeriod:\n\n')
    quizFile.write((' ' * 20) + 'State Capitals Quiz (Form %s)' % (quizNum + 1))
    quizFile.write('\n\n')

    # Shuffle the order of the states.
    states = list(capitals.keys())
❹   random.shuffle(states)

    # TODO: Loop through all 50 states, making a question for each.
```

The filenames for the quizzes will be *capitalsquiz<N>.txt*, where *<N>* is a unique number for the quiz that comes from `quizNum`, the `for` loop's counter. The answer key for *capitalsquiz<N>.txt* will be stored in a text file named *capitalsquiz_answers<N>.txt*. Each time through the loop, the `%s` placeholder in `'capitalsquiz%s.txt'` and `'capitalsquiz_answers%s.txt'` will be replaced by `(quizNum + 1)`, so the first quiz and answer key created will be *capitalsquiz1.txt* and *capitalsquiz_answers1.txt*. These files will be created with calls to the `open()` function at ❶ and ❷, with `'w'` as the second argument to open them in write mode.

The `write()` statements at ❸ create a quiz header for the student to fill out. Finally, a randomized list of US states is created with the help of the `random.shuffle()` function ❹, which randomly reorders the values in any list that is passed to it.

Step 3: Create the Answer Options

Now you need to generate the answer options for each question, which will be multiple choice from A to D. You'll need to create another `for` loop—this one to generate the content for each of the 50 questions on the quiz. Then there will be a third `for` loop nested inside to generate the multiple-choice options for each question. Make your code look like the following:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip--

    # Loop through all 50 states, making a question for each.
    for questionNum in range(50):

        # Get right and wrong answers.
❶       correctAnswer = capitals[states[questionNum]]
❷       wrongAnswers = list(capitals.values())
❸       del wrongAnswers[wrongAnswers.index(correctAnswer)]
❹       wrongAnswers = random.sample(wrongAnswers, 3)
❺       answerOptions = wrongAnswers + [correctAnswer]
❻       random.shuffle(answerOptions)

        # TODO: Write the question and answer options to the quiz file.

        # TODO: Write the answer key to a file.
```

The correct answer is easy to get—it's stored as a value in the `capitals` dictionary ❶. This loop will loop through the states in the shuffled `states` list, from `states[0]` to `states[49]`, find each state in `capitals`, and store that state's corresponding capital in `correctAnswer`.

The list of possible wrong answers is trickier. You can get it by duplicating *all* the values in the `capitals` dictionary ❷, deleting the correct answer ❸, and selecting three random values from this list ❹. The `random.sample()` function makes it easy to do this selection. Its first argument is the list you want to select from; the second argument is the number of values you want to select. The full list of answer options is the combination of these three wrong answers with the correct answers ❺. Finally, the answers need to be randomized ❻ so that the correct response isn't always choice D.

Step 4: Write Content to the Quiz and Answer Key Files

All that is left is to write the question to the quiz file and the answer to the answer key file. Make your code look like the following:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--snip--
```

```
    # Loop through all 50 states, making a question for each.
    for questionNum in range(50):
        --snip--

        # Write the question and the answer options to the quiz file.
        quizFile.write('%s. What is the capital of %s?\n' % (questionNum + 1,
            states[questionNum]))
❶       for i in range(4):
❷           quizFile.write('    %s. %s\n' % ('ABCD'[i], answerOptions[i]))
        quizFile.write('\n')

        # Write the answer key to a file.
❸       answerKeyFile.write('%s. %s\n' % (questionNum + 1, 'ABCD'[
           answerOptions.index(correctAnswer)]))
    quizFile.close()
    answerKeyFile.close()
```

A `for` loop that goes through integers `0` to `3` will write the answer options in the `answerOptions` list ❶. The expression `'ABCD'[i]` at ❷ treats the string `'ABCD'` as an array and will evaluate to `'A'`,`'B'`, `'C'`, and then `'D'` on each respective iteration through the loop.

In the final line ❸, the expression `answerOptions.index(correctAnswer)` will find the integer index of the correct answer in the randomly ordered answer options, and `'ABCD'[answerOptions.index(correctAnswer)]` will evaluate to the correct answer's letter to be written to the answer key file.

After you run the program, this is how your *capitalsquiz1.txt* file will look, though of course your questions and answer options may be different from those shown here, depending on the outcome of your `random.shuffle()` calls:

```
Name:

Date:

Period:

                    State Capitals Quiz (Form 1)

1. What is the capital of West Virginia?
    A. Hartford
    B. Santa Fe
    C. Harrisburg
    D. Charleston

2. What is the capital of Colorado?
    A. Raleigh
    B. Harrisburg
    C. Denver
    D. Lincoln

--snip--
```

The corresponding *capitalsquiz_answers1.txt* text file will look like this:

```
1. D
2. C
3. A
4. C
--snip--
```

Project: Multiclipboard

Say you have the boring task of filling out many forms in a web page or software with several text fields. The clipboard saves you from typing the same text over and over again. But only one thing can be on the clipboard at a time. If you have several different pieces of text that you need to copy and paste, you have to keep highlighting and copying the same few things over and over again.

You can write a Python program to keep track of multiple pieces of text. This "multiclipboard" will be named *mcb.pyw* (since "mcb" is shorter to type than "multiclipboard"). The *.pyw* extension means that Python won't show a Terminal window when it runs this program. (See Appendix B for more details.)

The program will save each piece of clipboard text under a keyword. For example, when you run `py mcb.pyw save spam`, the current contents of the clipboard will be saved with the keyword *spam*. This text can later be loaded to the clipboard again by running `py mcb.pyw spam`. And if the user forgets what keywords they have, they can run `py mcb.pyw list` to copy a list of all keywords to the clipboard.

Here's what the program does:

- The command line argument for the keyword is checked.
- If the argument is `save`, then the clipboard contents are saved to the keyword.
- If the argument is `list`, then all the keywords are copied to the clipboard.
- Otherwise, the text for the keyword is copied to the keyboard.

This means the code will need to do the following:

- Read the command line arguments from `sys.argv`.
- Read and write to the clipboard.
- Save and load to a shelf file.

If you use Windows, you can easily run this script from the Run… window by creating a batch file named *mcb.bat* with the following content:

```
@pyw.exe C:\Python34\mcb.pyw %*
```

Step 1: Comments and Shelf Setup

Let's start by making a skeleton script with some comments and basic setup. Make your code look like the following:

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
❶ # Usage: py.exe mcb.pyw save <keyword> - Saves clipboard to keyword.
#        py.exe mcb.pyw <keyword> - Loads keyword to clipboard.
#        py.exe mcb.pyw list - Loads all keywords to clipboard.

❷ import shelve, pyperclip, sys

❸ mcbShelf = shelve.open('mcb')

# TODO: Save clipboard content.

# TODO: List keywords and load content.

mcbShelf.close()
```

It's common practice to put general usage information in comments at the top of the file ❶. If you ever forget how to run your script, you can always look at these comments for a reminder. Then you import your modules ❷. Copying and pasting will require the `pyperclip` module, and reading the command line arguments will require the `sys` module. The `shelve` module will also come in handy: Whenever the user wants to save a new piece of clipboard text, you'll save it to a shelf file. Then, when the user wants to paste the text back to their clipboard, you'll open the shelf file and load it back into your program. The shelf file will be named with the prefix *mcb* ❸.

Step 2: Save Clipboard Content with a Keyword

The program does different things depending on whether the user wants to save text to a keyword, load text into the clipboard, or list all the existing keywords. Let's deal with that first case. Make your code look like the following:

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip--

# Save clipboard content.
❶ if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
❷         mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
❸     # TODO: List keywords and load content.

mcbShelf.close()
```

If the first command line argument (which will always be at index `1` of the `sys.argv` list) is `'save'` ❶, the second command line argument is the keyword for the current content of the clipboard. The keyword will be used as the key for `mcbShelf`, and the value will be the text currently on the clipboard ❷.

If there is only one command line argument, you will assume it is either `'list'` or a keyword to load content onto the clipboard. You will implement that code later. For now, just put a `TODO` comment there ❸.

Step 3: List Keywords and Load a Keyword's Content

Finally, let's implement the two remaining cases: The user wants to load clipboard text in from a keyword, or they want a list of all available keywords. Make your code look like the following:

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
--snip--

# Save clipboard content.
if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
        mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
    # List keywords and load content.
❶   if sys.argv[1].lower() == 'list':
❷       pyperclip.copy(str(list(mcbShelf.keys())))
    elif sys.argv[1] in mcbShelf:
❸       pyperclip.copy(mcbShelf[sys.argv[1]])

mcbShelf.close()
```

If there is only one command line argument, first let's check whether it's `'list'` ❶. If so, a string representation of the list of shelf keys will be copied to the clipboard ❷. The user can paste this list into an open text editor to read it.

Otherwise, you can assume the command line argument is a keyword. If this keyword exists in the `mcbShelf` shelf as a key, you can load the value onto the clipboard ❸.

And that's it! Launching this program has different steps depending on what operating system your computer uses. See Appendix B for details for your operating system.

Recall the password locker program you created in Chapter 6 that stored the passwords in a dictionary. Updating the passwords required changing the source code of the program. This isn't ideal because average users don't feel comfortable changing source code to update their software. Also, every time you modify the source code to a program, you run the risk of accidentally introducing new bugs. By storing the data for a program in a different place than the code, you can make your programs easier for others to use and more resistant to bugs.

Summary

Files are organized into folders (also called directories), and a path describes the location of a file. Every program running on your computer has a current working directory, which allows you to specify file paths relative to the current location instead of always typing the full (or absolute) path. The `os.path` module has many functions for manipulating file paths.

Your programs can also directly interact with the contents of text files. The `open()` function can open these files to read in their contents as one large string (with the `read()` method) or as a list of strings (with the `readlines()` method). The `open()` function can open files in write or append mode to create new text files or add to existing text files, respectively.

In previous chapters, you used the clipboard as a way of getting large amounts of text into a program, rather than typing it all in. Now you can have your programs read files directly from the hard drive, which is a big improvement, since files are much less volatile than the clipboard.

In the next chapter, you will learn how to handle the files themselves, by copying them, deleting them, renaming them, moving them, and more.

Practice Questions

1. What is a relative path relative to?
2. What does an absolute path start with?
3. What do the `os.getcwd()` and `os.chdir()` functions do?
4. What are the `.` and `..` folders?
5. In *C:\bacon\eggs\spam.txt*, which part is the dir name, and which part is the base name?
6. What are the three "mode" arguments that can be passed to the `open()` function?
7. What happens if an existing file is opened in write mode?
8. What is the difference between the `read()` and `readlines()` methods?
9. What data structure does a shelf value resemble?

Practice Projects

For practice, design and write the following programs.

Extending the Multiclipboard

Extend the multiclipboard program in this chapter so that it has a `delete <keyword>` command line argument that will delete a keyword from the shelf. Then add a `delete` command line argument that will delete *all* keywords.

Mad Libs

Create a Mad Libs program that reads in text files and lets the user add their own text anywhere the word *ADJECTIVE*, *NOUN*, *ADVERB*, or *VERB* appears in the text file. For example, a text file may look like this:

```
The ADJECTIVE panda walked to the NOUN and then VERB. A nearby NOUN was
unaffected by these events.
```

The program would find these occurrences and prompt the user to replace them.

```
Enter an adjective:
silly
Enter a noun:
chandelier
Enter a verb:
screamed
Enter a noun:
pickup truck
```

The following text file would then be created:

```
The silly panda walked to the chandelier and then screamed. A nearby pickup
truck was unaffected by these events.
```

The results should be printed to the screen and saved to a new text file.

Regex Search

Write a program that opens all *.txt* files in a folder and searches for any line that matches a user-supplied regular expression. The results should be printed to the screen.

9

ORGANIZING FILES

In the previous chapter, you learned how to create and write to new files in Python. Your programs can also organize preexisting files on the hard drive. Maybe you've had the experience of going through a folder full of dozens, hundreds, or even thousands of files and copying, renaming, moving, or compressing them all by hand. Or consider tasks such as these:

- Making copies of all PDF files (and *only* the PDF files) in every subfolder of a folder
- Removing the leading zeros in the filenames for every file in a folder of hundreds of files named *spam001.txt*, *spam002.txt*, *spam003.txt*, and so on
- Compressing the contents of several folders into one ZIP file (which could be a simple backup system)

All this boring stuff is just begging to be automated in Python. By programming your computer to do these tasks, you can transform it into a quick-working file clerk who never makes mistakes.

As you begin working with files, you may find it helpful to be able to quickly see what the extension (*.txt*, *.pdf*, *.jpg*, and so on) of a file is. With OS X and Linux, your file browser most likely shows extensions automatically. With Windows, file extensions may be hidden by default. To show extensions, go to **Start ▸ Control Panel ▸ Appearance and Personalization ▸ Folder Options**. On the View tab, under Advanced Settings, uncheck the **Hide extensions for known file types** checkbox.

The shutil Module

The `shutil` (or shell utilities) module has functions to let you copy, move, rename, and delete files in your Python programs. To use the `shutil` functions, you will first need to use `import shutil`.

Copying Files and Folders

The `shutil` module provides functions for copying files, as well as entire folders.

Calling `shutil.copy(`*`source, destination`*`)` will copy the file at the path *`source`* to the folder at the path *`destination`*. (Both *`source`* and *`destination`* are strings.) If *`destination`* is a filename, it will be used as the new name of the copied file. This function returns a string of the path of the copied file.

Enter the following into the interactive shell to see how `shutil.copy()` works:

```
>>> import shutil, os
>>> os.chdir('C:\\')
❶ >>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
'C:\\delicious\\spam.txt'
❷ >>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')
'C:\\delicious\\eggs2.txt'
```

The first `shutil.copy()` call copies the file at *C:\spam.txt* to the folder *C:\delicious*. The return value is the path of the newly copied file. Note that since a folder was specified as the destination ❶, the original *spam.txt* filename is used for the new, copied file's filename. The second `shutil.copy()` call ❷ also copies the file at *C:\eggs.txt* to the folder *C:\delicious* but gives the copied file the name *eggs2.txt*.

While `shutil.copy()` will copy a single file, `shutil.copytree()` will copy an entire folder and every folder and file contained in it. Calling `shutil.copytree(`*`source, destination`*`)` will copy the folder at the path *`source`*, along with all of its files and subfolders, to the folder at the path *`destination`*. The *`source`* and *`destination`* parameters are both strings. The function returns a string of the path of the copied folder.

Enter the following into the interactive shell:

```
>>> import shutil, os
>>> os.chdir('C:\\')
>>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
'C:\\bacon_backup'
```

The `shutil.copytree()` call creates a new folder named *bacon_backup* with the same content as the original *bacon* folder. You have now safely backed up your precious, precious bacon.

Moving and Renaming Files and Folders

Calling `shutil.move(`*`source, destination`*`)` will move the file or folder at the path *`source`* to the path *`destination`* and will return a string of the absolute path of the new location.

If *`destination`* points to a folder, the *`source`* file gets moved into *`destination`* and keeps its current filename. For example, enter the following into the interactive shell:

```
>>> import shutil
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs\\bacon.txt'
```

Assuming a folder named *eggs* already exists in the *C:* directory, this `shutil.move()` calls says, "Move *C:\bacon.txt* into the folder *C:\eggs*."

If there had been a *bacon.txt* file already in *C:\eggs*, it would have been overwritten. Since it's easy to accidentally overwrite files in this way, you should take some care when using `move()`.

The *`destination`* path can also specify a filename. In the following example, the *`source`* file is moved *and* renamed.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\new_bacon.txt')
'C:\\eggs\\new_bacon.txt'
```

This line says, "Move *C:\bacon.txt* into the folder *C:\eggs*, and while you're at it, rename that *bacon.txt* file to *new_bacon.txt*."

Both of the previous examples worked under the assumption that there was a folder *eggs* in the *C:* directory. But if there is no *eggs* folder, then `move()` will rename *bacon.txt* to a file named *eggs*.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs'
```

Here, `move()` can't find a folder named *eggs* in the *C:* directory and so assumes that *`destination`* must be specifying a filename, not a folder. So the *bacon.txt* text file is renamed to *eggs* (a text file without the *.txt* file extension)—probably not what you wanted! This can be a tough-to-spot bug in

your programs since the move() call can happily do something that might be quite different from what you were expecting. This is yet another reason to be careful when using move().

Finally, the folders that make up the destination must already exist, or else Python will throw an exception. Enter the following into the interactive shell:

```
>>> shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
Traceback (most recent call last):
  File "C:\Python34\lib\shutil.py", line 521, in move
    os.rename(src, real_dst)
FileNotFoundError: [WinError 3] The system cannot find the path specified:
'spam.txt' -> 'c:\\does_not_exist\\eggs\\ham'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<pyshell#29>", line 1, in <module>
    shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
  File "C:\Python34\lib\shutil.py", line 533, in move
    copy2(src, real_dst)
  File "C:\Python34\lib\shutil.py", line 244, in copy2
    copyfile(src, dst, follow_symlinks=follow_symlinks)
  File "C:\Python34\lib\shutil.py", line 108, in copyfile
    with open(dst, 'wb') as fdst:
FileNotFoundError: [Errno 2] No such file or directory: 'c:\\does_not_exist\\
eggs\\ham'
```

Python looks for *eggs* and *ham* inside the directory *does_not_exist*. It doesn't find the nonexistent directory, so it can't move *spam.txt* to the path you specified.

Permanently Deleting Files and Folders

You can delete a single file or a single empty folder with functions in the os module, whereas to delete a folder and all of its contents, you use the shutil module.

- Calling os.unlink(*path*) will delete the file at *path*.
- Calling os.rmdir(*path*) will delete the folder at *path*. This folder must be empty of any files or folders.
- Calling shutil.rmtree(*path*) will remove the folder at *path*, and all files and folders it contains will also be deleted.

Be careful when using these functions in your programs! It's often a good idea to first run your program with these calls commented out and with print() calls added to show the files that would be deleted. Here is

a Python program that was intended to delete files that have the *.txt* file extension but has a typo (highlighted in bold) that causes it to delete *.rxt* files instead:

```
import os
for filename in os.listdir():
    if filename.endswith('.rxt'):
        os.unlink(filename)
```

If you had any important files ending with *.rxt*, they would have been accidentally, permanently deleted. Instead, you should have first run the program like this:

```
import os
for filename in os.listdir():
    if filename.endswith('.rxt'):
        #os.unlink(filename)
        print(filename)
```

Now the `os.unlink()` call is commented, so Python ignores it. Instead, you will print the filename of the file that would have been deleted. Running this version of the program first will show you that you've accidentally told the program to delete *.rxt* files instead of *.txt* files.

Once you are certain the program works as intended, delete the `print(filename)` line and uncomment the `os.unlink(filename)` line. Then run the program again to actually delete the files.

Safe Deletes with the send2trash Module

Since Python's built-in `shutil.rmtree()` function irreversibly deletes files and folders, it can be dangerous to use. A much better way to delete files and folders is with the third-party `send2trash` module. You can install this module by running `pip install send2trash` from a Terminal window. (See Appendix A for a more in-depth explanation of how to install third-party modules.)

Using `send2trash` is much safer than Python's regular delete functions, because it will send folders and files to your computer's trash or recycle bin instead of permanently deleting them. If a bug in your program deletes something with `send2trash` you didn't intend to delete, you can later restore it from the recycle bin.

After you have installed `send2trash`, enter the following into the interactive shell:

```
>>> import send2trash
>>> baconFile = open('bacon.txt', 'a')   # creates the file
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> send2trash.send2trash('bacon.txt')
```

In general, you should always use the `send2trash.send2trash()` function to delete files and folders. But while sending files to the recycle bin lets you recover them later, it will not free up disk space like permanently deleting them does. If you want your program to free up disk space, use the `os` and `shutil` functions for deleting files and folders. Note that the `send2trash()` function can only send files to the recycle bin; it cannot pull files out of it.

Walking a Directory Tree

Say you want to rename every file in some folder and also every file in every subfolder of that folder. That is, you want to walk through the directory tree, touching each file as you go. Writing a program to do this could get tricky; fortunately, Python provides a function to handle this process for you.

Let's look at the *C:\delicious* folder with its contents, shown in Figure 9-1.

Figure 9-1: An example folder that contains three folders and four files

Here is an example program that uses the `os.walk()` function on the directory tree from Figure 9-1:

```
import os

for folderName, subfolders, filenames in os.walk('C:\\delicious'):
    print('The current folder is ' + folderName)

    for subfolder in subfolders:
        print('SUBFOLDER OF ' + folderName + ': ' + subfolder)
```

```
    for filename in filenames:
        print('FILE INSIDE ' + folderName + ': '+ filename)

    print('')
```

The os.walk() function is passed a single string value: the path of a folder. You can use os.walk() in a for loop statement to walk a directory tree, much like how you can use the range() function to walk over a range of numbers. Unlike range(), the os.walk() function will return three values on each iteration through the loop:

1. A string of the current folder's name
2. A list of strings of the folders in the current folder
3. A list of strings of the files in the current folder

(By current folder, I mean the folder for the current iteration of the for loop. The current working directory of the program is *not* changed by os.walk().)

Just like you can choose the variable name i in the code for i in range(10):, you can also choose the variable names for the three values listed earlier. I usually use the names foldername, subfolders, and filenames.

When you run this program, it will output the following:

```
The current folder is C:\delicious
SUBFOLDER OF C:\delicious: cats
SUBFOLDER OF C:\delicious: walnut
FILE INSIDE C:\delicious: spam.txt

The current folder is C:\delicious\cats
FILE INSIDE C:\delicious\cats: catnames.txt
FILE INSIDE C:\delicious\cats: zophie.jpg

The current folder is C:\delicious\walnut
SUBFOLDER OF C:\delicious\walnut: waffles

The current folder is C:\delicious\walnut\waffles
FILE INSIDE C:\delicious\walnut\waffles: butter.txt.
```

Since os.walk() returns lists of strings for the subfolder and filename variables, you can use these lists in their own for loops. Replace the print() function calls with your own custom code. (Or if you don't need one or both of them, remove the for loops.)

Compressing Files with the zipfile Module

You may be familiar with ZIP files (with the *.zip* file extension), which can hold the compressed contents of many other files. Compressing a file reduces its size, which is useful when transferring it over the Internet. And

since a ZIP file can also contain multiple files and subfolders, it's a handy way to package several files into one. This single file, called an *archive file*, can then be, say, attached to an email.

Your Python programs can both create and open (or *extract*) ZIP files using functions in the `zipfile` module. Say you have a ZIP file named *example.zip* that has the contents shown in Figure 9-2.

You can download this ZIP file from *http://nostarch.com/automatestuff/* or just follow along using a ZIP file already on your computer.

Figure 9-2: The contents of example.zip

Reading ZIP Files

To read the contents of a ZIP file, first you must create a `ZipFile` object (note the capital letters *Z* and *F*). `ZipFile` objects are conceptually similar to the `File` objects you saw returned by the `open()` function in the previous chapter: They are values through which the program interacts with the file. To create a `ZipFile` object, call the `zipfile.ZipFile()` function, passing it a string of the *.zip* file's filename. Note that `zipfile` is the name of the Python module, and `ZipFile()` is the name of the function.

For example, enter the following into the interactive shell:

```
>>> import zipfile, os
>>> os.chdir('C:\\')    # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
>>> exampleZip.namelist()
['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']
>>> spamInfo = exampleZip.getinfo('spam.txt')
>>> spamInfo.file_size
13908
>>> spamInfo.compress_size
3828
❶ >>> 'Compressed file is %sx smaller!' % (round(spamInfo.file_size / spamInfo
.compress_size, 2))
'Compressed file is 3.63x smaller!'
>>> exampleZip.close()
```

A `ZipFile` object has a `namelist()` method that returns a list of strings for all the files and folders contained in the ZIP file. These strings can be passed to the `getinfo()` `ZipFile` method to return a `ZipInfo` object about that particular file. `ZipInfo` objects have their own attributes, such as `file_size` and `compress_size` in bytes, which hold integers of the original file size and compressed file size, respectively. While a `ZipFile` object represents an entire archive file, a `ZipInfo` object holds useful information about a *single file* in the archive.

The command at ❶ calculates how efficiently *example.zip* is compressed by dividing the original file size by the compressed file size and prints this information using a string formatted with `%s`.

Extracting from ZIP Files

The `extractall()` method for `ZipFile` objects extracts all the files and folders from a ZIP file into the current working directory.

```
>>> import zipfile, os
>>> os.chdir('C:\\')    # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
❶ >>> exampleZip.extractall()
>>> exampleZip.close()
```

After running this code, the contents of *example.zip* will be extracted to *C:*. Optionally, you can pass a folder name to `extractall()` to have it extract the files into a folder other than the current working directory. If the folder passed to the `extractall()` method does not exist, it will be created. For instance, if you replaced the call at ❶ with `exampleZip.extractall('C:\\delicious')`, the code would extract the files from *example.zip* into a newly created *C:\delicious* folder.

The `extract()` method for `ZipFile` objects will extract a single file from the ZIP file. Continue the interactive shell example:

```
>>> exampleZip.extract('spam.txt')
'C:\\spam.txt'
>>> exampleZip.extract('spam.txt', 'C:\\some\\new\\folders')
'C:\\some\\new\\folders\\spam.txt'
>>> exampleZip.close()
```

The string you pass to `extract()` must match one of the strings in the list returned by `namelist()`. Optionally, you can pass a second argument to `extract()` to extract the file into a folder other than the current working directory. If this second argument is a folder that doesn't yet exist, Python will create the folder. The value that `extract()` returns is the absolute path to which the file was extracted.

Creating and Adding to ZIP Files

To create your own compressed ZIP files, you must open the `ZipFile` object in *write mode* by passing `'w'` as the second argument. (This is similar to opening a text file in write mode by passing `'w'` to the `open()` function.)

When you pass a path to the `write()` method of a `ZipFile` object, Python will compress the file at that path and add it into the ZIP file. The `write()` method's first argument is a string of the filename to add. The second argument is the *compression type* parameter, which tells the computer what algorithm it should use to compress the files; you can always just set this value to `zipfile.ZIP_DEFLATED`. (This specifies the *deflate* compression algorithm, which works well on all types of data.) Enter the following into the interactive shell:

```
>>> import zipfile
>>> newZip = zipfile.ZipFile('new.zip', 'w')
>>> newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)
>>> newZip.close()
```

This code will create a new ZIP file named *new.zip* that has the compressed contents of *spam.txt*.

Keep in mind that, just as with writing to files, write mode will erase all existing contents of a ZIP file. If you want to simply add files to an existing ZIP file, pass `'a'` as the second argument to `zipfile.ZipFile()` to open the ZIP file in *append mode*.

Project: Renaming Files with American-Style Dates to European-Style Dates

Say your boss emails you thousands of files with American-style dates (MM-DD-YYYY) in their names and needs them renamed to European-style dates (DD-MM-YYYY). This boring task could take all day to do by hand! Let's write a program to do it instead.

Here's what the program does:

- It searches all the filenames in the current working directory for American-style dates.
- When one is found, it renames the file with the month and day swapped to make it European-style.

This means the code will need to do the following:

- Create a regex that can identify the text pattern of American-style dates.
- Call `os.listdir()` to find all the files in the working directory.
- Loop over each filename, using the regex to check whether it has a date.
- If it has a date, rename the file with `shutil.move()`.

For this project, open a new file editor window and save your code as *renameDates.py*.

Step 1: Create a Regex for American-Style Dates

The first part of the program will need to import the necessary modules and create a regex that can identify MM-DD-YYYY dates. The to-do comments will remind you what's left to write in this program. Typing them as `TODO` makes them easy to find using IDLE's CTRL-F find feature. Make your code look like the following:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.

❶ import shutil, os, re

# Create a regex that matches files with the American date format.
❷ datePattern = re.compile(r"""^(.*?) # all text before the date
    ((0|1)?\d)-                       # one or two digits for the month
```

```
    ((0|1|2|3)?\d)-                # one or two digits for the day
    ((19|20)\d\d)                  # four digits for the year
    (.*?)$                         # all text after the date
❸   """, re.VERBOSE)

# TODO: Loop over the files in the working directory.

# TODO: Skip files without a date.

# TODO: Get the different parts of the filename.

# TODO: Form the European-style filename.

# TODO: Get the full, absolute file paths.

# TODO: Rename the files.
```

From this chapter, you know the `shutil.move()` function can be used to rename files: Its arguments are the name of the file to rename and the new filename. Because this function exists in the `shutil` module, you must import that module ❶.

But before renaming the files, you need to identify which files you want to rename. Filenames with dates such as *spam4-4-1984.txt* and *01-03-2014eggs.zip* should be renamed, while filenames without dates such as *littlebrother.epub* can be ignored.

You can use a regular expression to identify this pattern. After importing the `re` module at the top, call `re.compile()` to create a `Regex` object ❷. Passing `re.VERBOSE` for the second argument ❸ will allow whitespace and comments in the regex string to make it more readable.

The regular expression string begins with `^(.*?)` to match any text at the beginning of the filename that might come before the date. The `((0|1)?\d)` group matches the month. The first digit can be either `0` or `1`, so the regex matches `12` for December but also `02` for February. This digit is also optional so that the month can be `04` or `4` for April. The group for the day is `((0|1|2|3)?\d)` and follows similar logic; `3`, `03`, and `31` are all valid numbers for days. (Yes, this regex will accept some invalid dates such as `4-31-2014`, `2-29-2013`, and `0-15-2014`. Dates have a lot of thorny special cases that can be easy to miss. But for simplicity, the regex in this program works well enough.)

While 1885 is a valid year, you can just look for years in the 20th or 21st century. This will keep your program from accidentally matching nondate filenames with a date-like format, such as *10-10-1000.txt*.

The `(.*?)$` part of the regex will match any text that comes after the date.

Step 2: Identify the Date Parts from the Filenames

Next, the program will have to loop over the list of filename strings returned from `os.listdir()` and match them against the regex. Any files that do not

have a date in them should be skipped. For filenames that have a date, the matched text will be stored in several variables. Fill in the first three TODOs in your program with the following code:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.

--snip--

# Loop over the files in the working directory.
for amerFilename in os.listdir('.'):
    mo = datePattern.search(amerFilename)

    # Skip files without a date.
❶   if mo == None:
❷       continue

❸   # Get the different parts of the filename.
    beforePart = mo.group(1)
    monthPart  = mo.group(2)
    dayPart    = mo.group(4)
    yearPart   = mo.group(6)
    afterPart  = mo.group(8)

--snip--
```

If the Match object returned from the search() method is None ❶, then the filename in amerFilename does not match the regular expression. The continue statement ❷ will skip the rest of the loop and move on to the next filename.

Otherwise, the various strings matched in the regular expression groups are stored in variables named beforePart, monthPart, dayPart, yearPart, and afterPart ❸. The strings in these variables will be used to form the European-style filename in the next step.

To keep the group numbers straight, try reading the regex from the beginning and count up each time you encounter an opening parenthesis. Without thinking about the code, just write an outline of the regular expression. This can help you visualize the groups. For example:

```
datePattern = re.compile(r"""^(1) # all text before the date
    (2 (3) )-                     # one or two digits for the month
    (4 (5) )-                     # one or two digits for the day
    (6 (7) )                      # four digits for the year
    (8)$                          # all text after the date
    """, re.VERBOSE)
```

Here, the numbers 1 through 8 represent the groups in the regular expression you wrote. Making an outline of the regular expression, with just the parentheses and group numbers, can give you a clearer understanding of your regex before you move on with the rest of the program.

Step 3: Form the New Filename and Rename the Files

As the final step, concatenate the strings in the variables made in the previous step with the European-style date: The date comes before the month. Fill in the three remaining TODOs in your program with the following code:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.

--snip--

    # Form the European-style filename.
❶   euroFilename = beforePart + dayPart + '-' + monthPart + '-' + yearPart +
                   afterPart

    # Get the full, absolute file paths.
    absWorkingDir = os.path.abspath('.')
    amerFilename = os.path.join(absWorkingDir, amerFilename)
    euroFilename = os.path.join(absWorkingDir, euroFilename)

    # Rename the files.
❷   print('Renaming "%s" to "%s"...' % (amerFilename, euroFilename))
❸   #shutil.move(amerFilename, euroFilename)   # uncomment after testing
```

Store the concatenated string in a variable named euroFilename ❶. Then, pass the original filename in amerFilename and the new euroFilename variable to the shutil.move() function to rename the file ❸.

This program has the shutil.move() call commented out and instead prints the filenames that will be renamed ❷. Running the program like this first can let you double-check that the files are renamed correctly. Then you can uncomment the shutil.move() call and run the program again to actually rename the files.

Ideas for Similar Programs

There are many other reasons why you might want to rename a large number of files.

- To add a prefix to the start of the filename, such as adding *spam_* to rename *eggs.txt* to *spam_eggs.txt*
- To change filenames with European-style dates to American-style dates
- To remove the zeros from files such as *spam0042.txt*

Project: Backing Up a Folder into a ZIP File

Say you're working on a project whose files you keep in a folder named *C:\AlsPythonBook*. You're worried about losing your work, so you'd like to create ZIP file "snapshots" of the entire folder. You'd like to keep different versions, so you want the ZIP file's filename to increment each time it is made; for example, *AlsPythonBook_1.zip*, *AlsPythonBook_2.zip*,

AlsPythonBook_3.zip, and so on. You could do this by hand, but it is rather annoying, and you might accidentally misnumber the ZIP files' names. It would be much simpler to run a program that does this boring task for you.

For this project, open a new file editor window and save it as *backupToZip.py*.

Step 1: Figure Out the ZIP File's Name

The code for this program will be placed into a function named `backupToZip()`. This will make it easy to copy and paste the function into other Python programs that need this functionality. At the end of the program, the function will be called to perform the backup. Make your program look like this:

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

❶ import zipfile, os

def backupToZip(folder):
    # Backup the entire contents of "folder" into a ZIP file.

    folder = os.path.abspath(folder)   # make sure folder is absolute

    # Figure out the filename this code should use based on
    # what files already exist.
❷   number = 1
❸   while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1

❹   # TODO: Create the ZIP file.

    # TODO: Walk the entire folder tree and compress the files in each folder.
    print('Done.')

backupToZip('C:\\delicious')
```

Do the basics first: Add the shebang (`#!`) line, describe what the program does, and import the `zipfile` and `os` modules ❶.

Define a `backupToZip()` function that takes just one parameter, `folder`. This parameter is a string path to the folder whose contents should be backed up. The function will determine what filename to use for the ZIP file it will create; then the function will create the file, walk the `folder` folder, and add each of the subfolders and files to the ZIP file. Write `TODO` comments for these steps in the source code to remind yourself to do them later ❹.

The first part, naming the ZIP file, uses the base name of the absolute path of `folder`. If the folder being backed up is *C:\delicious*, the ZIP file's name should be *delicious_N.zip*, where $N = 1$ is the first time you run the program, $N = 2$ is the second time, and so on.

You can determine what *N* should be by checking whether *delicious_1.zip* already exists, then checking whether *delicious_2.zip* already exists, and so on. Use a variable named `number` for *N* ❷, and keep incrementing it inside the loop that calls `os.path.exists()` to check whether the file exists ❸. The first nonexistent filename found will cause the loop to `break`, since it will have found the filename of the new zip.

Step 2: Create the New ZIP File

Next let's create the ZIP file. Make your program look like the following:

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

--snip--
    while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1

    # Create the ZIP file.
    print('Creating %s...' % (zipFilename))
❶   backupZip = zipfile.ZipFile(zipFilename, 'w')

    # TODO: Walk the entire folder tree and compress the files in each folder.
    print('Done.')

backupToZip('C:\\delicious')
```

Now that the new ZIP file's name is stored in the `zipFilename` variable, you can call `zipfile.ZipFile()` to actually create the ZIP file ❶. Be sure to pass `'w'` as the second argument so that the ZIP file is opened in write mode.

Step 3: Walk the Directory Tree and Add to the ZIP File

Now you need to use the `os.walk()` function to do the work of listing every file in the folder and its subfolders. Make your program look like the following:

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

--snip--

    # Walk the entire folder tree and compress the files in each folder.
❶   for foldername, subfolders, filenames in os.walk(folder):
        print('Adding files in %s...' % (foldername))
        # Add the current folder to the ZIP file.
❷       backupZip.write(foldername)
```

```
        # Add all the files in this folder to the ZIP file.
❸       for filename in filenames:
            newBase / os.path.basename(folder) + '_'
            if filename.startswith(newBase) and filename.endswith('.zip')
                continue   # don't backup the backup ZIP files
            backupZip.write(os.path.join(foldername, filename))
    backupZip.close()
    print('Done.')


backupToZip('C:\\delicious')
```

You can use `os.walk()` in a `for` loop ❶, and on each iteration it will return the iteration's current folder name, the subfolders in that folder, and the filenames in that folder.

In the `for` loop, the folder is added to the ZIP file ❷. The nested `for` loop can go through each filename in the `filenames` list ❸. Each of these is added to the ZIP file, except for previously made backup ZIPs.

When you run this program, it will produce output that will look something like this:

```
Creating delicious_1.zip...
Adding files in C:\delicious...
Adding files in C:\delicious\cats...
Adding files in C:\delicious\waffles...
Adding files in C:\delicious\walnut...
Adding files in C:\delicious\walnut\waffles...
Done.
```

The second time you run it, it will put all the files in *C:\delicious* into a ZIP file named *delicious_2.zip*, and so on.

Ideas for Similar Programs

You can walk a directory tree and add files to compressed ZIP archives in several other programs. For example, you can write programs that do the following:

- Walk a directory tree and archive just files with certain extensions, such as *.txt* or *.py*, and nothing else
- Walk a directory tree and archive every file except the *.txt* and *.py* ones
- Find the folder in a directory tree that has the greatest number of files or the folder that uses the most disk space

Summary

Even if you are an experienced computer user, you probably handle files manually with the mouse and keyboard. Modern file explorers make it easy to work with a few files. But sometimes you'll need to perform a task that would take hours using your computer's file explorer.

The os and shutil modules offer functions for copying, moving, renaming, and deleting files. When deleting files, you might want to use the send2trash module to move files to the recycle bin or trash rather than permanently deleting them. And when writing programs that handle files, it's a good idea to comment out the code that does the actual copy/move/rename/delete and add a print() call instead so you can run the program and verify exactly what it will do.

Often you will need to perform these operations not only on files in one folder but also on every folder in that folder, every folder in those folders, and so on. The os.walk() function handles this trek across the folders for you so that you can concentrate on what your program needs to do with the files in them.

The zipfile module gives you a way of compressing and extracting files in *.zip* archives through Python. Combined with the file-handling functions of os and shutil, zipfile makes it easy to package up several files from anywhere on your hard drive. These *.zip* files are much easier to upload to websites or send as email attachments than many separate files.

Previous chapters of this book have provided source code for you to copy. But when you write your own programs, they probably won't come out perfectly the first time. The next chapter focuses on some Python modules that will help you analyze and debug your programs so that you can quickly get them working correctly.

Practice Questions

1. What is the difference between shutil.copy() and shutil.copytree()?
2. What function is used to rename files?
3. What is the difference between the delete functions in the send2trash and shutil modules?
4. ZipFile objects have a close() method just like File objects' close() method. What ZipFile method is equivalent to File objects' open() method?

Practice Projects

For practice, write programs to do the following tasks.

Selective Copy

Write a program that walks through a folder tree and searches for files with a certain file extension (such as *.pdf* or *.jpg*). Copy these files from whatever location they are in to a new folder.

Deleting Unneeded Files

It's not uncommon for a few unneeded but humongous files or folders to take up the bulk of the space on your hard drive. If you're trying to free up

room on your computer, you'll get the most bang for your buck by deleting the most massive of the unwanted files. But first you have to find them.

Write a program that walks through a folder tree and searches for exceptionally large files or folders—say, ones that have a file size of more than 100MB. (Remember, to get a file's size, you can use `os.path.getsize()` from the `os` module.) Print these files with their absolute path to the screen.

Filling in the Gaps

Write a program that finds all files with a given prefix, such as *spam001.txt*, *spam002.txt*, and so on, in a single folder and locates any gaps in the numbering (such as if there is a *spam001.txt* and *spam003.txt* but no *spam002.txt*). Have the program rename all the later files to close this gap.

As an added challenge, write another program that can insert gaps into numbered files so that a new file can be added.

10

DEBUGGING

Now that you know enough to write more complicated programs, you may start finding not-so-simple bugs in them. This chapter covers some tools and techniques for finding the root cause of bugs in your program to help you fix bugs faster and with less effort.

To paraphrase an old joke among programmers, "Writing code accounts for 90 percent of programming. Debugging code accounts for the other 90 percent."

Your computer will do only what you tell it to do; it won't read your mind and do what you *intended* it to do. Even professional programmers create bugs all the time, so don't feel discouraged if your program has a problem.

Fortunately, there are a few tools and techniques to identify what exactly your code is doing and where it's going wrong. First, you will look at logging and assertions, two features that can help you detect bugs early. In general, the earlier you catch bugs, the easier they will be to fix.

Second, you will look at how to use the debugger. The debugger is a feature of IDLE that executes a program one instruction at a time, giving you a chance to inspect the values in variables while your code runs, and track how the values change over the course of your program. This is much slower than running the program at full speed, but it is helpful to see the actual values in a program while it runs, rather than deducing what the values might be from the source code.

Raising Exceptions

Python raises an exception whenever it tries to execute invalid code. In Chapter 3, you read about how to handle Python's exceptions with `try` and `except` statements so that your program can recover from exceptions that you anticipated. But you can also raise your own exceptions in your code. Raising an exception is a way of saying, "Stop running the code in this function and move the program execution to the `except` statement."

Exceptions are raised with a `raise` statement. In code, a `raise` statement consists of the following:

- The `raise` keyword
- A call to the `Exception()` function
- A string with a helpful error message passed to the `Exception()` function

For example, enter the following into the interactive shell:

```
>>> raise Exception('This is the error message.')
Traceback (most recent call last):
  File "<pyshell#191>", line 1, in <module>
    raise Exception('This is the error message.')
Exception: This is the error message.
```

If there are no `try` and `except` statements covering the `raise` statement that raised the exception, the program simply crashes and displays the exception's error message.

Often it's the code that calls the function, not the fuction itself, that knows how to handle an expection. So you will commonly see a `raise` statement inside a function and the `try` and `except` statements in the code calling the function. For example, open a new file editor window, enter the following code, and save the program as *boxPrint.py*:

```
def boxPrint(symbol, width, height):
    if len(symbol) != 1:
❶       raise Exception('Symbol must be a single character string.')
    if width <= 2:
❷       raise Exception('Width must be greater than 2.')
    if height <= 2:
❸       raise Exception('Height must be greater than 2.')
```

```
    print(symbol * width)
    for i in range(height - 2):
        print(symbol + (' ' * (width - 2)) + symbol)
    print(symbol * width)

for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
    try:
        boxPrint(sym, w, h)
❹   except Exception as err:
❺       print('An exception happened: ' + str(err))
```

Here we've defined a `boxPrint()` function that takes a character, a width, and a height, and uses the character to make a little picture of a box with that width and height. This box shape is printed to the console.

Say we want the character to be a single character, and the width and height to be greater than 2. We add `if` statements to raise exceptions if these requirements aren't satisfied. Later, when we call `boxPrint()` with various arguments, our `try/except` will handle invalid arguments.

This program uses the `except Exception as err` form of the `except` statement ❹. If an `Exception` object is returned from `boxPrint()` ❶❷❸, this `except` statement will store it in a variable named `err`. The `Exception` object can then be converted to a string by passing it to `str()` to produce a user-friendly error message ❺. When you run this *boxPrint.py*, the output will look like this:

```
****
*  *
*  *
****
OOOOOOOOOOOOOOOOOOOO
O                  O
O                  O
O                  O
OOOOOOOOOOOOOOOOOOOO
An exception happened: Width must be greater than 2.
An exception happened: Symbol must be a single character string.
```

Using the `try` and `except` statements, you can handle errors more gracefully instead of letting the entire program crash.

Getting the Traceback as a String

When Python encounters an error, it produces a treasure trove of error information called the *traceback*. The traceback includes the error message, the line number of the line that caused the error, and the sequence of the function calls that led to the error. This sequence of calls is called the *call stack*.

Open a new file editor window in IDLE, enter the following program, and save it as *errorExample.py*:

```
def spam():
    bacon()
```

```
def bacon():
    raise Exception('This is the error message.')

spam()
```

When you run *errorExample.py*, the output will look like this:

```
Traceback (most recent call last):
  File "errorExample.py", line 7, in <module>
    spam()
  File "errorExample.py", line 2, in spam
    bacon()
  File "errorExample.py", line 5, in bacon
    raise Exception('This is the error message.')
Exception: This is the error message.
```

From the traceback, you can see that the error happened on line 5, in the `bacon()` function. This particular call to `bacon()` came from line 2, in the `spam()` function, which in turn was called on line 7. In programs where functions can be called from multiple places, the call stack can help you determine which call led to the error.

The traceback is displayed by Python whenever a raised exception goes unhandled. But you can also obtain it as a string by calling `traceback.format_exc()`. This function is useful if you want the information from an exception's traceback but also want an `except` statement to gracefully handle the exception. You will need to import Python's `traceback` module before calling this function.

For example, instead of crashing your program right when an exception occurs, you can write the traceback information to a log file and keep your program running. You can look at the log file later, when you're ready to debug your program. Enter the following into the interactive shell:

```
>>> import traceback
>>> try:
        raise Exception('This is the error message.')
except:
        errorFile = open('errorInfo.txt', 'w')
        errorFile.write(traceback.format_exc())
        errorFile.close()
        print('The traceback info was written to errorInfo.txt.')

116
The traceback info was written to errorInfo.txt.
```

The `116` is the return value from the `write()` method, since 116 characters were written to the file. The traceback text was written to *errorInfo.txt*.

```
Traceback (most recent call last):
  File "<pyshell#28>", line 2, in <module>
Exception: This is the error message.
```

Assertions

An *assertion* is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by `assert` statements. If the sanity check fails, then an `AssertionError` exception is raised. In code, an `assert` statement consists of the following:

- The `assert` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A comma
- A string to display when the condition is `False`

For example, enter the following into the interactive shell:

```
>>> podBayDoorStatus = 'open'
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
>>> podBayDoorStatus = 'I\'m sorry, Dave. I\'m afraid I can't do that.''
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
AssertionError: The pod bay doors need to be "open".
```

Here we've set `podBayDoorStatus` to `'open'`, so from now on, we fully expect the value of this variable to be `'open'`. In a program that uses this variable, we might have written a lot of code under the assumption that the value is `'open'`—code that depends on its being `'open'` in order to work as we expect. So we add an assertion to make sure we're right to assume `podBayDoorStatus` is `'open'`. Here, we include the message `'The pod bay doors need to be "open".'` so it'll be easy to see what's wrong if the assertion fails.

Later, say we make the obvious mistake of assigning `podBayDoorStatus` another value, but don't notice it among many lines of code. The assertion catches this mistake and clearly tells us what's wrong.

In plain English, an `assert` statement says, "I assert that this condition holds true, and if not, there is a bug somewhere in the program." Unlike exceptions, your code should *not* handle `assert` statements with `try` and `except`; if an `assert` fails, your program *should* crash. By failing fast like this, you shorten the time between the original cause of the bug and when you first notice the bug. This will reduce the amount of code you will have to check before finding the code that's causing the bug.

Assertions are for programmer errors, not user errors. For errors that can be recovered from (such as a file not being found or the user entering invalid data), raise an exception instead of detecting it with an `assert` statement.

Using an Assertion in a Traffic Light Simulation

Say you're building a traffic light simulation program. The data structure representing the stoplights at an intersection is a dictionary with

keys 'ns' and 'ew', for the stoplights facing north-south and east-west, respectively. The values at these keys will be one of the strings 'green', 'yellow', or 'red'. The code would look something like this:

```
market_2nd = {'ns': 'green', 'ew': 'red'}
mission_16th = {'ns': 'red', 'ew': 'green'}
```

These two variables will be for the intersections of Market Street and 2nd Street, and Mission Street and 16th Street. To start the project, you want to write a switchLights() function, which will take an intersection dictionary as an argument and switch the lights.

At first, you might think that switchLights() should simply switch each light to the next color in the sequence: Any 'green' values should change to 'yellow', 'yellow' values should change to 'red', and 'red' values should change to 'green'. The code to implement this idea might look like this:

```
def switchLights(stoplight):
    for key in stoplight.keys():
        if stoplight[key] == 'green':
            stoplight[key] = 'yellow'
        elif stoplight[key] == 'yellow':
            stoplight[key] = 'red'
        elif stoplight[key] == 'red':
            stoplight[key] = 'green'

switchLights(market_2nd)
```

You may already see the problem with this code, but let's pretend you wrote the rest of the simulation code, thousands of lines long, without noticing it. When you finally do run the simulation, the program doesn't crash—but your virtual cars do!

Since you've already written the rest of the program, you have no idea where the bug could be. Maybe it's in the code simulating the cars or in the code simulating the virtual drivers. It could take hours to trace the bug back to the switchLights() function.

But if while writing switchLights() you had added an assertion to check that *at least one of the lights is always red*, you might have included the following at the bottom of the function:

```
assert 'red' in stoplight.values(), 'Neither light is red! ' + str(stoplight)
```

With this assertion in place, your program would crash with this error message:

```
Traceback (most recent call last):
  File "carSim.py", line 14, in <module>
    switchLights(market_2nd)
  File "carSim.py", line 13, in switchLights
    assert 'red' in stoplight.values(), 'Neither light is red! ' + str(stoplight)
❶ AssertionError: Neither light is red! {'ns': 'yellow', 'ew': 'green'}
```

The important line here is the `AssertionError` ❶. While your program crashing is not ideal, it immediately points out that a sanity check failed: Neither direction of traffic has a red light, meaning that traffic could be going both ways. By failing fast early in the program's execution, you can save yourself a lot of future debugging effort.

Disabling Assertions

Assertions can be disabled by passing the `-O` option when running Python. This is good for when you have finished writing and testing your program and don't want it to be slowed down by performing sanity checks (although most of the time `assert` statements do not cause a noticeable speed difference). Assertions are for development, not the final product. By the time you hand off your program to someone else to run, it should be free of bugs and not require the sanity checks. See Appendix B for details about how to launch your probably-not-insane programs with the `-O` option.

Logging

If you've ever put a `print()` statement in your code to output some variable's value while your program is running, you've used a form of *logging* to debug your code. Logging is a great way to understand what's happening in your program and in what order its happening. Python's `logging` module makes it easy to create a record of custom messages that you write. These log messages will describe when the program execution has reached the logging function call and list any variables you have specified at that point in time. On the other hand, a missing log message indicates a part of the code was skipped and never executed.

Using the logging Module

To enable the `logging` module to display log messages on your screen as your program runs, copy the following to the top of your program (but under the `#! python` shebang line):

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -  %(levelname)s
-  %(message)s')
```

You don't need to worry too much about how this works, but basically, when Python logs an event, it creates a `LogRecord` object that holds information about that event. The `logging` module's `basicConfig()` function lets you specify what details about the `LogRecord` object you want to see and how you want those details displayed.

Say you wrote a function to calculate the *factorial* of a number. In mathematics, factorial 4 is 1 × 2 × 3 × 4, or 24. Factorial 7 is 1 × 2 × 3 × 4 × 5 × 6 × 7, or 5,040. Open a new file editor window and enter the following code. It has a bug in it, but you will also enter several log messages to help yourself figure out what is going wrong. Save the program as *factorialLog.py*.

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -  %(levelname)s
-  %(message)s')
logging.debug('Start of program')

def factorial(n):
    logging.debug('Start of factorial(%s%%)'  % (n))
    total = 1
    for i in range(n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(%s%%)'  % (n))
    return total

print(factorial(5))
logging.debug('End of program')
```

Here, we use the `logging.debug()` function when we want to print log information. This `debug()` function will call `basicConfig()`, and a line of information will be printed. This information will be in the format we specified in `basicConfig()` and will include the messages we passed to `debug()`. The `print(factorial(5))` call is part of the original program, so the result is displayed even if logging messages are disabled.

The output of this program looks like this:

```
2015-05-23 16:20:12,664 - DEBUG - Start of program
2015-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
2015-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
2015-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
2015-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
2015-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
2015-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
2015-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
2015-05-23 16:20:12,680 - DEBUG - End of factorial(5)
0
2015-05-23 16:20:12,684 - DEBUG - End of program
```

The `factorial()` function is returning `0` as the factorial of `5`, which isn't right. The `for` loop should be multiplying the value in `total` by the numbers from `1` to `5`. But the log messages displayed by `logging.debug()` show that the `i` variable is starting at `0` instead of `1`. Since zero times anything is zero, the rest of the iterations also have the wrong value for `total`. Logging messages provide a trail of breadcrumbs that can help you figure out when things started to go wrong.

Change the `for i in range(n + 1):` line to `for i in range(1, n + 1):`, and run the program again. The output will look like this:

```
2015-05-23 17:13:40,650 - DEBUG - Start of program
2015-05-23 17:13:40,651 - DEBUG - Start of factorial(5)
2015-05-23 17:13:40,651 - DEBUG - i is 1, total is 1
2015-05-23 17:13:40,654 - DEBUG - i is 2, total is 2
2015-05-23 17:13:40,656 - DEBUG - i is 3, total is 6
```

```
2015-05-23 17:13:40,659 - DEBUG - i is 4, total is 24
2015-05-23 17:13:40,661 - DEBUG - i is 5, total is 120
2015-05-23 17:13:40,661 - DEBUG - End of factorial(5)
120
2015-05-23 17:13:40,666 - DEBUG - End of program
```

The `factorial(5)` call correctly returns `120`. The log messages showed what was going on inside the loop, which led straight to the bug.

You can see that the `logging.debug()` calls printed out not just the strings passed to them but also a timestamp and the word *DEBUG*.

Don't Debug with print()

Typing `import logging` and `logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')` is somewhat unwieldy. You may want to use `print()` calls instead, but don't give in to this temptation! Once you're done debugging, you'll end up spending a lot of time removing `print()` calls from your code for each log message. You might even accidentally remove some `print()` calls that were being used for nonlog messages. The nice thing about log messages is that you're free to fill your program with as many as you like, and you can always disable them later by adding a single `logging.disable(logging.CRITICAL)` call. Unlike `print()`, the `logging` module makes it easy to switch between showing and hiding log messages.

Log messages are intended for the programmer, not the user. The user won't care about the contents of some dictionary value you need to see to help with debugging; use a log message for something like that. For messages that the user will want to see, like *File not found* or *Invalid input, please enter a number*, you should use a `print()` call. You don't want to deprive the user of useful information after you've disabled log messages.

Logging Levels

Logging levels provide a way to categorize your log messages by importance. There are five logging levels, described in Table 10-1 from least to most important. Messages can be logged at each level using a different logging function.

Table 10-1: Logging Levels in Python

Level	Logging Function	Description
DEBUG	`logging.debug()`	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
INFO	`logging.info()`	Used to record information on general events in your program or confirm that things are working at their point in the program.
WARNING	`logging.warning()`	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.

(*continued*)

Table 10-1 (*continued*)

Level	Logging Function	Description
ERROR	`logging.error()`	Used to record an error that caused the program to fail to do something.
CRITICAL	`logging.critical()`	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

Your logging message is passed as a string to these functions. The logging levels are suggestions. Ultimately, it is up to you to decide which category your log message falls into. Enter the following into the interactive shell:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s -  %(message)s')
>>> logging.debug('Some debugging details.')
2015-05-18 19:04:26,901 - DEBUG - Some debugging details.
>>> logging.info('The logging module is working.')
2015-05-18 19:04:35,569 - INFO - The logging module is working.
>>> logging.warning('An error message is about to be logged.')
2015-05-18 19:04:56,843 - WARNING - An error message is about to be logged.
>>> logging.error('An error has occurred.')
2015-05-18 19:05:07,737 - ERROR - An error has occurred.
>>> logging.critical('The program is unable to recover!')
2015-05-18 19:05:45,794 - CRITICAL - The program is unable to recover!
```

The benefit of logging levels is that you can change what priority of logging message you want to see. Passing `logging.DEBUG` to the `basicConfig()` function's `level` keyword argument will show messages from all the logging levels (DEBUG being the lowest level). But after developing your program some more, you may be interested only in errors. In that case, you can set `basicConfig()`'s `level` argument to `logging.ERROR`. This will show only ERROR and CRITICAL messages and skip the DEBUG, INFO, and WARNING messages.

Disabling Logging

After you've debugged your program, you probably don't want all these log messages cluttering the screen. The `logging.disable()` function disables these so that you don't have to go into your program and remove all the logging calls by hand. You simply pass `logging.disable()` a logging level, and it will suppress all log messages at that level or lower. So if you want to disable logging entirely, just add `logging.disable(logging.CRITICAL)` to your program. For example, enter the following into the interactive shell:

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s -
%(levelname)s -  %(message)s')
```

```
>>> logging.critical('Critical error! Critical error!')
2015-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!
>>> logging.disable(logging.CRITICAL)
>>> logging.critical('Critical error! Critical error!')
>>> logging.error('Error! Error!')
```

Since `logging.disable()` will disable all messages after it, you will probably want to add it near the `import logging` line of code in your program. This way, you can easily find it to comment out or uncomment that call to enable or disable logging messages as needed.

Logging to a File

Instead of displaying the log messages to the screen, you can write them to a text file. The `logging.basicConfig()` function takes a `filename` keyword argument, like so:

```
import logging
logging.basicConfig(filename='myProgramLog.txt', level=logging.DEBUG, format='
%(asctime)s -  %(levelname)s -  %(message)s')
```

The log messages will be saved to *myProgramLog.txt*. While logging messages are helpful, they can clutter your screen and make it hard to read the program's output. Writing the logging messages to a file will keep your screen clear and store the messages so you can read them after running the program. You can open this text file in any text editor, such as Notepad or TextEdit.

IDLE's Debugger

The *debugger* is a feature of IDLE that allows you to execute your program one line at a time. The debugger will run a single line of code and then wait for you to tell it to continue. By running your program "under the debugger" like this, you can take as much time as you want to examine the values in the variables at any given point during the program's lifetime. This is a valuable tool for tracking down bugs.

To enable IDLE's debugger, click **Debug ▸ Debugger** in the interactive shell window. This will bring up the Debug Control window, which looks like Figure 10-1.

When the Debug Control window appears, select all four of the **Stack**, **Locals**, **Source**, and **Globals** checkboxes so that the window shows the full set of debug information. While the Debug Control window is displayed, any time you run a program from the file editor, the debugger will pause execution before the first instruction and display the following:

- The line of code that is about to be executed
- A list of all local variables and their values
- A list of all global variables and their values

Figure 10-1: The Debug Control window

You'll notice that in the list of global variables there are several variables you haven't defined, such as `__builtins__`, `__doc__`, `__file__`, and so on. These are variables that Python automatically sets whenever it runs a program. The meaning of these variables is beyond the scope of this book, and you can comfortably ignore them.

The program will stay paused until you press one of the five buttons in the Debug Control window: Go, Step, Over, Out, or Quit.

Go

Clicking the Go button will cause the program to execute normally until it terminates or reaches a *breakpoint*. (Breakpoints are described later in this chapter.) If you are done debugging and want the program to continue normally, click the **Go** button.

Step

Clicking the Step button will cause the debugger to execute the next line of code and then pause again. The Debug Control window's list of global and local variables will be updated if their values change. If the next line of code is a function call, the debugger will "step into" that function and jump to the first line of code of that function.

Over

Clicking the Over button will execute the next line of code, similar to the Step button. However, if the next line of code is a function call, the Over button will "step over" the code in the function. The function's code will be executed at full speed, and the debugger will pause as soon as the function call returns. For example, if the next line of code is a `print()` call, you don't

really care about code inside the built-in `print()` function; you just want the string you pass it printed to the screen. For this reason, using the Over button is more common than the Step button.

Out

Clicking the Out button will cause the debugger to execute lines of code at full speed until it returns from the current function. If you have stepped into a function call with the Step button and now simply want to keep executing instructions until you get back out, click the **Out** button to "step out" of the current function call.

Quit

If you want to stop debugging entirely and not bother to continue executing the rest of the program, click the **Quit** button. The Quit button will immediately terminate the program. If you want to run your program normally again, select **Debug ▸ Debugger** again to disable the debugger.

Debugging a Number Adding Program

Open a new file editor window and enter the following code:

```
print('Enter the first number to add:')
first = input()
print('Enter the second number to add:')
second = input()
print('Enter the third number to add:')
third = input()
print('The sum is ' + first + second + third)
```

Save it as *buggyAddingProgram.py* and run it first without the debugger enabled. The program will output something like this:

```
Enter the first number to add:
5
Enter the second number to add:
3
Enter the third number to add:
42
The sum is 5342
```

The program hasn't crashed, but the sum is obviously wrong. Let's enable the Debug Control window and run it again, this time under the debugger.

When you press F5 or select **Run ▸ Run Module** (with **Debug ▸ Debugger** enabled and all four checkboxes on the Debug Control window checked), the program starts in a paused state on line 1. The debugger will always pause on the line of code it is about to execute. The Debug Control window will look like Figure 10-2.

Figure 10-2: The Debug Control window when the program first starts under the debugger

Click the **Over** button once to execute the first `print()` call. You should use Over instead of Step here, since you don't want to step into the code for the `print()` function. The Debug Control window will update to line 2, and line 2 in the file editor window will be highlighted, as shown in Figure 10-3. This shows you where the program execution currently is.

Figure 10-3: The Debug Control window after clicking Over

Click **Over** again to execute the `input()` function call, and the buttons in the Debug Control window will disable themselves while IDLE waits for you to type something for the `input()` call into the interactive shell window. Enter **5** and press Return. The Debug Control window buttons will be reenabled.

Keep clicking **Over**, entering **3** and **42** as the next two numbers, until the debugger is on line 7, the final `print()` call in the program. The Debug Control window should look like Figure 10-4. You can see in the Globals section that the first, second, and third variables are set to string values `'5'`, `'3'`, and `'42'` instead of integer values `5`, `3`, and `42`. When the last line is executed, these strings are concatenated instead of added together, causing the bug.

Figure 10-4: The Debug Control window on the last line. The variables are set to strings, causing the bug.

Stepping through the program with the debugger is helpful but can also be slow. Often you'll want the program to run normally until it reaches a certain line of code. You can configure the debugger to do this with breakpoints.

Breakpoints

A *breakpoint* can be set on a specific line of code and forces the debugger to pause whenever the program execution reaches that line. Open a new file editor window and enter the following program, which simulates flipping a coin 1,000 times. Save it as *coinFlip.py*.

```
import random
heads = 0
for i in range(1, 1001):
❶    if random.randint(0, 1) == 1:
         heads = heads + 1
     if i == 500:
❷        print('Halfway done!')
print('Heads came up ' + str(heads) + ' times.')
```

The `random.randint(0, 1)` call ❶ will return `0` half of the time and `1` the other half of the time. This can be used to simulate a 50/50 coin flip where `1` represents heads. When you run this program without the debugger, it quickly outputs something like the following:

```
Halfway done!
Heads came up 490 times.
```

If you ran this program under the debugger, you would have to click the Over button thousands of times before the program terminated. If you were interested in the value of `heads` at the halfway point of the program's execution, when 500 of 1000 coin flips have been completed, you could instead just set a breakpoint on the line `print('Halfway done!')` ❷. To set a breakpoint, right-click the line in the file editor and select **Set Breakpoint**, as shown in Figure 10-5.

Figure 10-5: Setting a breakpoint

You don't want to set a breakpoint on the `if` statement line, since the `if` statement is executed on every single iteration through the loop. By setting the breakpoint on the code in the `if` statement, the debugger breaks only when the execution enters the `if` clause.

The line with the breakpoint will be highlighted in yellow in the file editor. When you run the program under the debugger, it will start in a paused state at the first line, as usual. But if you click Go, the program will run at full speed until it reaches the line with the breakpoint set on it. You can then click Go, Over, Step, or Out to continue as normal.

If you want to remove a breakpoint, right-click the line in the file editor and select **Clear Breakpoint** from the menu. The yellow highlighting will go away, and the debugger will not break on that line in the future.

Summary

Assertions, exceptions, logging, and the debugger are all valuable tools to find and prevent bugs in your program. Assertions with the Python `assert` statement are a good way to implement "sanity checks" that give you an early warning when a necessary condition doesn't hold true. Assertions are only for errors that the program shouldn't try to recover from and should fail fast. Otherwise, you should raise an exception.

An exception can be caught and handled by the `try` and `except` statements. The `logging` module is a good way to look into your code while it's running and is much more convenient to use than the `print()` function because of its different logging levels and ability to log to a text file.

The debugger lets you step through your program one line at a time. Alternatively, you can run your program at normal speed and have the debugger pause execution whenever it reaches a line with a breakpoint set. Using the debugger, you can see the state of any variable's value at any point during the program's lifetime.

These debugging tools and techniques will help you write programs that work. Accidentally introducing bugs into your code is a fact of life, no matter how many years of coding experience you have.

Practice Questions

1. Write an `assert` statement that triggers an `AssertionError` if the variable `spam` is an integer less than `10`.
2. Write an `assert` statement that triggers an `AssertionError` if the variables `eggs` and `bacon` contain strings that are the same as each other, even if their cases are different (that is, `'hello'` and `'hello'` are considered the same, and `'goodbye'` and `'GOODbye'` are also considered the same).
3. Write an `assert` statement that *always* triggers an `AssertionError`.
4. What are the two lines that your program must have in order to be able to call `logging.debug()`?
5. What are the two lines that your program must have in order to have `logging.debug()` send a logging message to a file named *programLog.txt*?
6. What are the five logging levels?
7. What line of code can you add to disable all logging messages in your program?
8. Why is using logging messages better than using `print()` to display the same message?
9. What are the differences between the Step, Over, and Out buttons in the Debug Control window?

10. After you click Go in the Debug Control window, when will the debugger stop?
11. What is a breakpoint?
12. How do you set a breakpoint on a line of code in IDLE?

Practice Project

For practice, write a program that does the following.

Debugging Coin Toss

The following program is meant to be a simple coin toss guessing game. The player gets two guesses (it's an easy game). However, the program has several bugs in it. Run through the program a few times to find the bugs that keep the program from working correctly.

```
import random
guess = ''
while guess not in ('heads', 'tails'):
    print('Guess the coin toss! Enter heads or tails:')
    guess = input()
toss = random.randint(0, 1) # 0 is tails, 1 is heads
if toss == guess:
    print('You got it!')
else:
    print('Nope! Guess again!')
    guesss = input()
    if toss == guess:
        print('You got it!')
    else:
        print('Nope. You are really bad at this game.')
```

11

WEB SCRAPING

In those rare, terrifying moments when I'm without Wi-Fi, I realize just how much of what I do on the computer is really what I do on the Internet. Out of sheer habit I'll find myself trying to check email, read friends' Twitter feeds, or answer the question, "Did Kurtwood Smith have any major roles before he was in the original 1987 *Robo@op*?"[1]

Since so much work on a computer involves going on the Internet, it'd be great if your programs could get online. *Web scraping* is the term for using a program to download and process content from the Web. For example, Google runs many web scraping programs to index web pages for its search engine. In this chapter, you will learn about several modules that make it easy to scrape web pages in Python.

1. The answer is no.

webbrowser Comes with Python and opens a browser to a specific page.

Requests Downloads files and web pages from the Internet.

Beautiful Soup Parses HTML, the format that web pages are written in.

Selenium Launches and controls a web browser. Selenium is able to fill in forms and simulate mouse clicks in this browser.

Project: mapIt.py with the webbrowser Module

The `webbrowser` module's `open()` function can launch a new browser to a specified URL. Enter the following into the interactive shell:

```
>>> import webbrowser
>>> webbrowser.open('http://inventwithpython.com/')
```

A web browser tab will open to the URL *http://inventwithpython.com/*. This is about the only thing the `webbrowser` module can do. Even so, the `open()` function does make some interesting things possible. For example, it's tedious to copy a street address to the clipboard and bring up a map of it on Google Maps. You could take a few steps out of this task by writing a simple script to automatically launch the map in your browser using the contents of your clipboard. This way, you only have to copy the address to a clipboard and run the script, and the map will be loaded for you.

This is what your program does:

- Gets a street address from the command line arguments or clipboard.
- Opens the web browser to the Google Maps page for the address.

This means your code will need to do the following:

- Read the command line arguments from `sys.argv`.
- Read the clipboard contents.
- Call the `webbrowser.open()` function to open the web browser.

Open a new file editor window and save it as *mapIt.py*.

Step 1: Figure Out the URL

Based on the instructions in Appendix B, set up *mapIt.py* so that when you run it from the command line, like so . . .

```
C:\> mapit 870 Valencia St, San Francisco, CA 94110
```

. . . the script will use the command line arguments instead of the clipboard. If there are no command line arguments, then the program will know to use the contents of the clipboard.

First you need to figure out what URL to use for a given street address. When you load *http://maps.google.com/* in the browser and search for an address, the URL in the address bar looks something like this: *https://www.google.com/maps/place/870+Valencia+St/@37.7590311,-122.4215096,17z/data=!3m1!4b1!4m2!3m1!1s0x808f7e3dadc07a37:0xc86b0b2bb93b73d8*.

The address is in the URL, but there's a lot of additional text there as well. Websites often add extra data to URLs to help track visitors or customize sites. But if you try just going to *https://www.google.com/maps/place/870+Valencia+St+San+Francisco+CA/*, you'll find that it still brings up the correct page. So your program can be set to open a web browser to `'https://www.google.com/maps/place/`*`your_address_string`*`'` (where *`your_address_string`* is the address you want to map).

Step 2: Handle the Command Line Arguments

Make your code look like this:

```
#! python3
# mapIt.py - Launches a map in the browser using an address from the
# command line or clipboard.

import webbrowser, sys
if len(sys.argv) > 1:
    # Get address from command line.
    address = ' '.join(sys.argv[1:])

# TODO: Get address from clipboard.
```

After the program's `#!` shebang line, you need to import the `webbrowser` module for launching the browser and import the `sys` module for reading the potential command line arguments. The `sys.argv` variable stores a list of the program's filename and command line arguments. If this list has more than just the filename in it, then `len(sys.argv)` evaluates to an integer greater than `1`, meaning that command line arguments have indeed been provided.

Command line arguments are usually separated by spaces, but in this case, you want to interpret all of the arguments as a single string. Since `sys.argv` is a list of strings, you can pass it to the `join()` method, which returns a single string value. You don't want the program name in this string, so instead of `sys.argv`, you should pass `sys.argv[1:]` to chop off the first element of the array. The final string that this expression evaluates to is stored in the `address` variable.

If you run the program by entering this into the command line . . .

```
mapit 870 Valencia St, San Francisco, CA 94110
```

. . . the `sys.argv` variable will contain this list value:

```
['mapIt.py', '870', 'Valencia', 'St, ', 'San', 'Francisco, ', 'CA', '94110']
```

The `address` variable will contain the string `'870 Valencia St, San Francisco, CA 94110'`.

Step 3: Handle the Clipboard Content and Launch the Browser

Make your code look like the following:

```
#! python3
# mapIt.py - Launches a map in the browser using an address from the
# command line or clipboard.

import webbrowser, sys, pyperclip
if len(sys.argv) > 1:
    # Get address from command line.
    address = ' '.join(sys.argv[1:])
else:
    # Get address from clipboard.
    address = pyperclip.paste()

webbrowser.open('https://www.google.com/maps/place/' + address)
```

If there are no command line arguments, the program will assume the address is stored on the clipboard. You can get the clipboard content with `pyperclip.paste()` and store it in a variable named `address`. Finally, to launch a web browser with the Google Maps URL, call `webbrowser.open()`.

While some of the programs you write will perform huge tasks that save you hours, it can be just as satisfying to use a program that conveniently saves you a few seconds each time you perform a common task, such as getting a map of an address. Table 11-1 compares the steps needed to display a map with and without *mapIt.py*.

Table 11-1: Getting a Map with and Without *mapIt.py*

Manually getting a map	Using *mapIt.py*
Highlight the address.	Highlight the address.
Copy the address.	Copy the address.
Open the web browser.	Run *mapIt.py*.
Go to *http://maps.google.com/*.	
Click the address text field.	
Paste the address.	
Press ENTER.	

See how *mapIt.py* makes this task less tedious?

Ideas for Similar Programs

As long as you have a URL, the `webbrowser` module lets users cut out the step of opening the browser and directing themselves to a website. Other programs could use this functionality to do the following:

- Open all links on a page in separate browser tabs.
- Open the browser to the URL for your local weather.
- Open several social network sites that you regularly check.

Downloading Files from the Web with the requests Module

The `requests` module lets you easily download files from the Web without having to worry about complicated issues such as network errors, connection problems, and data compression. The `requests` module doesn't come with Python, so you'll have to install it first. From the command line, run **`pip install requests`**. (Appendix A has additional details on how to install third-party modules.)

The `requests` module was written because Python's `urllib2` module is too complicated to use. In fact, take a permanent marker and black out this entire paragraph. Forget I ever mentioned `urllib2`. If you need to download things from the Web, just use the `requests` module.

Next, do a simple test to make sure the `requests` module installed itself correctly. Enter the following into the interactive shell:

```
>>> import requests
```

If no error messages show up, then the `requests` module has been successfully installed.

Downloading a Web Page with the requests.get() Function

The `requests.get()`function takes a string of a URL to download. By calling `type()` on `requests.get()`'s return value, you can see that it returns a `Response` object, which contains the response that the web server gave for your request. I'll explain the `Response` object in more detail later, but for now, enter the following into the interactive shell while your computer is connected to the Internet:

```
  >>> import requests
❶ >>> res = requests.get('http://www.gutenberg.org/cache/epub/1112/pg1112.txt')
  >>> type(res)
  <class 'requests.models.Response'>
❷ >>> res.status_code == requests.codes.ok
  True
  >>> len(res.text)
  178981
  >>> print(res.text[:250])
  The Project Gutenberg EBook of Romeo and Juliet, by William Shakespeare

  This eBook is for the use of anyone anywhere at no cost and with
  almost no restrictions whatsoever.  You may copy it, give it away or
  re-use it under the terms of the Proje
```

The URL goes to a text web page for the entire play of *Romeo and Juliet*, provided by Project Gutenberg ❶. You can tell that the request for this web page succeeded by checking the `status_code` attribute of the `Response` object.

If it is equal to the value of `requests.codes.ok`, then everything went fine ❷. (Incidentally, the status code for "OK" in the HTTP protocol is 200. You may already be familiar with the 404 status code for "Not Found.")

If the request succeeded, the downloaded web page is stored as a string in the `Response` object's `text` variable. This variable holds a large string of the entire play; the call to `len(res.text)` shows you that it is more than 178,000 characters long. Finally, calling `print(res.text[:250])` displays only the first 250 characters.

Checking for Errors

As you've seen, the `Response` object has a `status_code` attribute that can be checked against `requests.codes.ok` to see whether the download succeeded. A simpler way to check for success is to call the `raise_for_status()` method on the `Response` object. This will raise an exception if there was an error downloading the file and will do nothing if the download succeeded. Enter the following into the interactive shell:

```
>>> res = requests.get('http://inventwithpython.com/page_that_does_not_exist')
>>> res.raise_for_status()
Traceback (most recent call last):
  File "<pyshell#138>", line 1, in <module>
    res.raise_for_status()
  File "C:\Python34\lib\site-packages\requests\models.py", line 773, in raise_for_status
    raise HTTPError(http_error_msg, response=self)
requests.exceptions.HTTPError: 404 Client Error: Not Found
```

The `raise_for_status()` method is a good way to ensure that a program halts if a bad download occurs. This is a good thing: You want your program to stop as soon as some unexpected error happens. If a failed download *isn't* a deal breaker for your program, you can wrap the `raise_for_status()` line with `try` and `except` statements to handle this error case without crashing.

```
import requests
res = requests.get('http://inventwithpython.com/page_that_does_not_exist')
try:
    res.raise_for_status()
except Exception as exc:
    print('There was a problem: %s' % (exc))
```

This `raise_for_status()` method call causes the program to output the following:

```
There was a problem: 404 Client Error: Not Found
```

Always call `raise_for_status()` after calling `requests.get()`. You want to be sure that the download has actually worked before your program continues.

Saving Downloaded Files to the Hard Drive

From here, you can save the web page to a file on your hard drive with the standard `open()` function and `write()` method. There are some slight differences, though. First, you must open the file in *write binary* mode by passing the string `'wb'` as the second argument to `open()`. Even if the page is in plaintext (such as the *Romeo and Juliet* text you downloaded earlier), you need to write binary data instead of text data in order to maintain the *Unicode encoding* of the text.

> **UNICODE ENCODINGS**
>
> Unicode encodings are beyond the scope of this book, but you can learn more about them from these web pages:
>
> - Joel on Software: The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!): *http://www.joelonsoftware.com/articles/Unicode.html*
> - Pragmatic Unicode: *http://nedbatchelder.com/text/unipain.html*

To write the web page to a file, you can use a `for` loop with the `Response` object's `iter_content()` method.

```
>>> import requests
>>> res = requests.get('http://www.gutenberg.org/cache/epub/1112/pg1112.txt')
>>> res.raise_for_status()
>>> playFile = open('RomeoAndJuliet.txt', 'wb')
>>> for chunk in res.iter_content(100000):
        playFile.write(chunk)

100000
78981
>>> playFile.close()
```

The `iter_content()` method returns "chunks" of the content on each iteration through the loop. Each chunk is of the *bytes* data type, and you get to specify how many bytes each chunk will contain. One hundred thousand bytes is generally a good size, so pass `100000` as the argument to `iter_content()`.

The file *RomeoAndJuliet.txt* will now exist in the current working directory. Note that while the filename on the website was *pg1112.txt*, the file on your hard drive has a different filename. The `requests` module simply handles downloading the contents of web pages. Once the page is downloaded, it is simply data in your program. Even if you were to lose your Internet connection after downloading the web page, all the page data would still be on your computer.

The `write()` method returns the number of bytes written to the file. In the previous example, there were 100,000 bytes in the first chunk, and the remaining part of the file needed only 78,981 bytes.

To review, here's the complete process for downloading and saving a file:

1. Call `requests.get()` to download the file.
2. Call `open()` with `'wb'` to create a new file in write binary mode.
3. Loop over the `Response` object's `iter_content()` method.
4. Call `write()` on each iteration to write the content to the file.
5. Call `close()` to close the file.

That's all there is to the `requests` module! The `for` loop and `iter_content()` stuff may seem complicated compared to the `open()`/`write()`/`close()` workflow you've been using to write text files, but it's to ensure that the `requests` module doesn't eat up too much memory even if you download massive files. You can learn about the `requests` module's other features from *http://requests.readthedocs.org/*.

HTML

Before you pick apart web pages, you'll learn some HTML basics. You'll also see how to access your web browser's powerful developer tools, which will make scraping information from the Web much easier.

Resources for Learning HTML

Hypertext Markup Language (HTML) is the format that web pages are written in. This chapter assumes you have some basic experience with HTML, but if you need a beginner tutorial, I suggest one of the following sites:

- *http://htmldog.com/guides/html/beginner/*
- *http://www.codecademy.com/tracks/web/*
- *https://developer.mozilla.org/en-US/learn/html/*

A Quick Refresher

In case it's been a while since you've looked at any HTML, here's a quick overview of the basics. An HTML file is a plaintext file with the *.html* file extension. The text in these files is surrounded by *tags*, which are words enclosed in angle brackets. The tags tell the browser how to format the web page. A starting tag and closing tag can enclose some text to form an *element*. The *text* (or *inner HTML*) is the content between the starting and closing tags. For example, the following HTML will display *Hello world!* in the browser, with *Hello* in bold:

```
<strong>Hello</strong> world!
```

This HTML will look like Figure 11-1 in a browser.

Figure 11-1: Hello world! *rendered in the browser*

The opening `<strong>` tag says that the enclosed text will appear in bold. The closing `</strong>` tags tells the browser where the end of the bold text is.

There are many different tags in HTML. Some of these tags have extra properties in the form of *attributes* within the angle brackets. For example, the `<a>` tag encloses text that should be a link. The URL that the text links to is determined by the `href` attribute. Here's an example:

```
Al's free <a href="http://inventwithpython.com">Python books</a>.
```

This HTML will look like Figure 11-2 in a browser.

Figure 11-2: The link rendered in the browser

Some elements have an `id` attribute that is used to uniquely identify the element in the page. You will often instruct your programs to seek out an element by its `id` attribute, so figuring out an element's `id` attribute using the browser's developer tools is a common task in writing web scraping programs.

Viewing the Source HTML of a Web Page

You'll need to look at the HTML source of the web pages that your programs will work with. To do this, right-click (or CTRL-click on OS X) any web page in your web browser, and select **View Source** or **View page source** to see the HTML text of the page (see Figure 11-3). This is the text your browser actually receives. The browser knows how to display, or *render*, the web page from this HTML.

Figure 11-3: Viewing the source of a web page

I highly recommend viewing the source HTML of some of your favorite sites. It's fine if you don't fully understand what you are seeing when you look at the source. You won't need HTML mastery to write simple web scraping programs—after all, you won't be writing your own websites. You just need enough knowledge to pick out data from an existing site.

Opening Your Browser's Developer Tools

In addition to viewing a web page's source, you can look through a page's HTML using your browser's developer tools. In Chrome and Internet Explorer for Windows, the developer tools are already installed, and you can press F12 to make them appear (see Figure 11-4). Pressing F12 again will make the developer tools disappear. In Chrome, you can also bring up the developer tools by selecting View ▸Developer ▸Developer Tools. In OS X, pressing ⌘-OPTION-I will open Chrome's Developer Tools.

Figure 11-4: The Developer Tools window in the Chrome browser

In Firefox, you can bring up the Web Developer Tools Inspector by pressing CTRL-SHIFT-C on Windows and Linux or by pressing ⌘-OPTION-C on OS X. The layout is almost identical to Chrome's developer tools.

In Safari, open the Preferences window, and on the Advanced pane check the **Show Develop menu in the menu bar** option. After it has been enabled, you can bring up the developer tools by pressing ⌘-OPTION-I.

After enabling or installing the developer tools in your browser, you can right-click any part of the web page and select **Inspect Element** from the context menu to bring up the HTML responsible for that part of the page. This will be helpful when you begin to parse HTML for your web scraping programs.

DON'T USE REGULAR EXPRESSIONS TO PARSE HTML

Locating a specific piece of HTML in a string seems like a perfect case for regular expressions. However, I advise you against it. There are many different ways that HTML can be formatted and still be considered valid HTML, but trying to capture all these possible variations in a regular expression can be tedious and error prone. A module developed specifically for parsing HTML, such as Beautiful Soup, will be less likely to result in bugs.

You can find an extended argument for why you shouldn't to parse HTML with regular expressions at *http://stackoverflow.com/a/1732454/1893164/*.

Using the Developer Tools to Find HTML Elements

Once your program has downloaded a web page using the `requests` module, you will have the page's HTML content as a single string value. Now you need to figure out which part of the HTML corresponds to the information on the web page you're interested in.

This is where the browser's developer tools can help. Say you want to write a program to pull weather forecast data from *http://weather.gov/*. Before writing any code, do a little research. If you visit the site and search for the 94105 ZIP code, the site will take you to a page showing the forecast for that area.

What if you're interested in scraping the temperature information for that ZIP code? Right-click where it is on the page (or CONTROL-click on OS X) and select **Inspect Element** from the context menu that appears. This will bring up the Developer Tools window, which shows you the HTML that produces this particular part of the web page. Figure 11-5 shows the developer tools open to the HTML of the temperature.

Figure 11-5: Inspecting the element that holds the temperature text with the developer tools

From the developer tools, you can see that the HTML responsible for the temperature part of the web page is `<p class="myforecast-current-lrg">57°F</p>`. This is exactly what you were looking for! It seems that the temperature information is contained inside a `<p>` element with the `myforecast-current-lrg` class. Now that you know what you're looking for, the `BeautifulSoup` module will help you find it in the string.

Parsing HTML with the BeautifulSoup Module

Beautiful Soup is a module for extracting information from an HTML page (and is much better for this purpose than regular expressions). The `BeautifulSoup` module's name is `bs4` (for Beautiful Soup, version 4). To install it, you will need to run `pip install beautifulsoup4` from the command line. (Check out Appendix A for instructions on installing third-party modules.) While `beautifulsoup4` is the name used for installation, to import Beautiful Soup you run `import bs4`.

For this chapter, the Beautiful Soup examples will *parse* (that is, analyze and identify the parts of) an HTML file on the hard drive. Open a new file editor window in IDLE, enter the following, and save it as *example.html*. Alternatively, download it from *http://nostarch.com/automatestuff/*.

```
<!-- This is the example.html example file. -->

<html><head><title>The Website Title</title></head>
<body>
<p>Download my <strong>Python</strong> book from <a href="http://
inventwithpython.com">my website</a>.</p>
<p class="slogan">Learn Python the easy way!</p>
<p>By <span id="author">Al Sweigart</span></p>
</body></html>
```

As you can see, even a simple HTML file involves many different tags and attributes, and matters quickly get confusing with complex websites. Thankfully, Beautiful Soup makes working with HTML much easier.

Creating a BeautifulSoup Object from HTML

The `bs4.BeautifulSoup()` function needs to be called with a string containing the HTML it will parse. The `bs4.BeautifulSoup()` function returns is a `BeautifulSoup` object. Enter the following into the interactive shell while your computer is connected to the Internet:

```
>>> import requests, bs4
>>> res = requests.get('http://nostarch.com')
>>> res.raise_for_status()
>>> noStarchSoup = bs4.BeautifulSoup(res.text)
>>> type(noStarchSoup)
<class 'bs4.BeautifulSoup'>
```

This code uses `requests.get()` to download the main page from the No Starch Press website and then passes the `text` attribute of the response to `bs4.BeautifulSoup()`. The `BeautifulSoup` object that it returns is stored in a variable named `noStarchSoup`.

You can also load an HTML file from your hard drive by passing a `File` object to `bs4.BeautifulSoup()`. Enter the following into the interactive shell (make sure the *example.html* file is in the working directory):

```
>>> exampleFile = open('example.html')
>>> exampleSoup = bs4.BeautifulSoup(exampleFile)
>>> type(exampleSoup)
<class 'bs4.BeautifulSoup'>
```

Once you have a `BeautifulSoup` object, you can use its methods to locate specific parts of an HTML document.

Finding an Element with the select() Method

You can retrieve a web page element from a `BeautifulSoup` object by calling the `select()`method and passing a string of a CSS *selector* for the element you are looking for. Selectors are like regular expressions: They specify a pattern to look for, in this case, in HTML pages instead of general text strings.

A full discussion of CSS selector syntax is beyond the scope of this book (there's a good selector tutorial in the resources at *http://nostarch.com/automatestuff/*), but here's a short introduction to selectors. Table 11-2 shows examples of the most common CSS selector patterns.

Table 11-2: Examples of CSS Selectors

Selector passed to the `select()` method	Will match . . .
`soup.select('div')`	All elements named `<div>`
`soup.select('#author')`	The element with an `id` attribute of `author`
`soup.select('.notice')`	All elements that use a CSS `class` attribute named `notice`
`soup.select('div span')`	All elements named `<span>` that are within an element named `<div>`
`soup.select('div > span')`	All elements named `<span>` that are *directly* within an element named `<div>`, with no other element in between
`soup.select('input[name]')`	All elements named `<input>` that have a `name` attribute with any value
`soup.select('input[type="button"]')`	All elements named `<input>` that have an attribute named `type` with value `button`

The various selector patterns can be combined to make sophisticated matches. For example, `soup.select('p #author')` will match any element that has an `id` attribute of `author`, as long as it is also inside a `<p>` element.

The `select()` method will return a list of `Tag` objects, which is how Beautiful Soup represents an HTML element. The list will contain one `Tag` object for every match in the `BeautifulSoup` object's HTML. Tag values can be passed to the `str()` function to show the HTML tags they represent.

Tag values also have an attrs attribute that shows all the HTML attributes of the tag as a dictionary. Using the *example.html* file from earlier, enter the following into the interactive shell:

```
>>> import bs4
>>> exampleFile = open('example.html')
>>> exampleSoup = bs4.BeautifulSoup(exampleFile.read())
>>> elems = exampleSoup.select('#author')
>>> type(elems)
<class 'list'>
>>> len(elems)
1
>>> type(elems[0])
<class 'bs4.element.Tag'>
>>> elems[0].getText()
'Al Sweigart'
>>> str(elems[0])
'<span id="author">Al Sweigart</span>'
>>> elems[0].attrs
{'id': 'author'}
```

This code will pull the element with id="author" out of our example HTML. We use select('#author') to return a list of all the elements with id="author". We store this list of **Tag** objects in the variable elems, and len(elems) tells us there is one **Tag** object in the list; there was one match. Calling getText() on the element returns the element's text, or inner HTML. The text of an element is the content between the opening and closing tags: in this case, 'Al Sweigart'.

Passing the element to str() returns a string with the starting and closing tags and the element's text. Finally, attrs gives us a dictionary with the element's attribute, 'id', and the value of the id attribute, 'author'.

You can also pull all the <p> elements from the BeautifulSoup object. Enter this into the interactive shell:

```
>>> pElems = exampleSoup.select('p')
>>> str(pElems[0])
'<p>Download my <strong>Python</strong> book from <a href="http://
inventwithpython.com">my website</a>.</p>'
>>> pElems[0].getText()
'Download my Python book from my website.'
>>> str(pElems[1])
'<p class="slogan">Learn Python the easy way!</p>'
>>> pElems[1].getText()
'Learn Python the easy way!'
>>> str(pElems[2])
'<p>By <span id="author">Al Sweigart</span></p>'
>>> pElems[2].getText()
'By Al Sweigart'
```

This time, select() gives us a list of three matches, which we store in pElems. Using str() on pElems[0], pElems[1], and pElems[2] shows you each element as a string, and using getText() on each element shows you its text.

Getting Data from an Element's Attributes

The `get()` method for `Tag` objects makes it simple to access attribute values from an element. The method is passed a string of an attribute name and returns that attribute's value. Using *example.html*, enter the following into the interactive shell:

```
>>> import bs4
>>> soup = bs4.BeautifulSoup(open('example.html'))
>>> spanElem = soup.select('span')[0]
>>> str(spanElem)
'<span id="author">Al Sweigart</span>'
>>> spanElem.get('id')
'author'
>>> spanElem.get('some_nonexistent_addr') == None
True
>>> spanElem.attrs
{'id': 'author'}
```

Here we use `select()` to find any `<span>` elements and then store the first matched element in `spanElem`. Passing the attribute name `'id'` to `get()` returns the attribute's value, `'author'`.

Project: "I'm Feeling Lucky" Google Search

Whenever I search a topic on Google, I don't look at just one search result at a time. By middle-clicking a search result link (or clicking while holding CTRL), I open the first several links in a bunch of new tabs to read later. I search Google often enough that this workflow—opening my browser, searching for a topic, and middle-clicking several links one by one—is tedious. It would be nice if I could simply type a search term on the command line and have my computer automatically open a browser with all the top search results in new tabs. Let's write a script to do this.

This is what your program does:

- Gets search keywords from the command line arguments.
- Retrieves the search results page.
- Opens a browser tab for each result.

This means your code will need to do the following:

- Read the command line arguments from `sys.argv`.
- Fetch the search result page with the `requests` module.
- Find the links to each search result.
- Call the `webbrowser.open()` function to open the web browser.

Open a new file editor window and save it as *lucky.py*.

Step 1: Get the Command Line Arguments and Request the Search Page

Before coding anything, you first need to know the URL of the search result page. By looking at the browser's address bar after doing a Google search, you can see that the result page has a URL like *https://www.google.com/search?q=SEARCH_TERM_HERE*. The `requests` module can download this page and then you can use Beautiful Soup to find the search result links in the HTML. Finally, you'll use the `webbrowser` module to open those links in browser tabs.

Make your code look like the following:

```
#! python3
# lucky.py - Opens several Google search results.

import requests, sys, webbrowser, bs4

print('Googling...')    # display text while downloading the Google page
res = requests.get('http://google.com/search?q=' + ' '.join(sys.argv[1:]))
res.raise_for_status()

# TODO: Retrieve top search result links.

# TODO: Open a browser tab for each result.
```

The user will specify the search terms using command line arguments when they launch the program. These arguments will be stored as strings in a list in `sys.argv`.

Step 2: Find All the Results

Now you need to use Beautiful Soup to extract the top search result links from your downloaded HTML. But how do you figure out the right selector for the job? For example, you can't just search for all `<a>` tags, because there are lots of links you don't care about in the HTML. Instead, you must inspect the search result page with the browser's developer tools to try to find a selector that will pick out only the links you want.

After doing a Google search for *Beautiful Soup*, you can open the browser's developer tools and inspect some of the link elements on the page. They look incredibly complicated, something like this: `<a href="/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0CCgQFjAA&url=http%3A%2F%2Fwww.crummy.com%2Fsoftware%2FBeautifulSoup%2F&ei=LHBVU_XDD9KVyAShmYDwCw&usg=AFQjCNHAxwplurFOBqg5cehWQEVKi-TuLQ&sig2=sdZu6WVlBlVSDrwhtworMA" onmousedown="return rwt(this,'','','','1','AFQjCNHAxwplurFOBqg5cehWQEVKi-TuLQ','sdZu6WVlBlVSDrwhtworMA','0CCgQFjAA','','',event)" data-href="http://www.crummy.com/software/BeautifulSoup/"><em>Beautiful Soup</em>: We called him Tortoise because he taught us.</a>`.

It doesn't matter that the element looks incredibly complicated. You just need to find the pattern that all the search result links have. But this `<a>` element doesn't have anything that easily distinguishes it from the nonsearch result `<a>` elements on the page.

Make your code look like the following:

```
#! python3
# lucky.py - Opens several google search results.

import requests, sys, webbrowser, bs4

--snip--

# Retrieve top search result links.
soup = bs4.BeautifulSoup(res.text)

# Open a browser tab for each result.
linkElems = soup.select('.r a')
```

If you look up a little from the <a> element, though, there is an element like this: <h3 class="r">. Looking through the rest of the HTML source, it looks like the r class is used only for search result links. You don't have to know what the CSS class r is or what it does. You're just going to use it as a marker for the <a> element you are looking for. You can create a BeautifulSoup object from the downloaded page's HTML text and then use the selector '.r a' to find all <a> elements that are within an element that has the r CSS class.

Step 3: Open Web Browsers for Each Result

Finally, we'll tell the program to open web browser tabs for our results. Add the following to the end of your program:

```
#! python3
# lucky.py - Opens several google search results.

import requests, sys, webbrowser, bs4

--snip--

# Open a browser tab for each result.
linkElems = soup.select('.r a')
numOpen = min(5, len(linkElems))
for i in range(numOpen):
    webbrowser.open('http://google.com' + linkElems[i].get('href'))
```

By default, you open the first five search results in new tabs using the webbrowser module. However, the user may have searched for something that turned up fewer than five results. The soup.select() call returns a list of all the elements that matched your '.r a' selector, so the number of tabs you want to open is either 5 or the length of this list (whichever is smaller).

The built-in Python function min() returns the smallest of the integer or float arguments it is passed. (There is also a built-in max() function that

returns the largest argument it is passed.) You can use `min()` to find out whether there are fewer than five links in the list and store the number of links to open in a variable named `numOpen`. Then you can run through a `for` loop by calling `range(numOpen)`.

On each iteration of the loop, you use `webbrowser.open()` to open a new tab in the web browser. Note that the `href` attribute's value in the returned `<a>` elements do not have the initial `http://google.com` part, so you have to concatenate that to the `href` attribute's string value.

Now you can instantly open the first five Google results for, say, *Python programming tutorials* by running `lucky python programming tutorials` on the command line! (See Appendix B for how to easily run programs on your operating system.)

Ideas for Similar Programs

The benefit of tabbed browsing is that you can easily open links in new tabs to peruse later. A program that automatically opens several links at once can be a nice shortcut to do the following:

- Open all the product pages after searching a shopping site such as Amazon
- Open all the links to reviews for a single product
- Open the result links to photos after performing a search on a photo site such as Flickr or Imgur

Project: Downloading All XKCD Comics

Blogs and other regularly updating websites usually have a front page with the most recent post as well as a Previous button on the page that takes you to the previous post. Then that post will also have a Previous button, and so on, creating a trail from the most recent page to the first post on the site. If you wanted a copy of the site's content to read when you're not online, you could manually navigate over every page and save each one. But this is pretty boring work, so let's write a program to do it instead.

XKCD is a popular geek webcomic with a website that fits this structure (see Figure 11-6). The front page at *http://xkcd.com/* has a Prev button that guides the user back through prior comics. Downloading each comic by hand would take forever, but you can write a script to do this in a couple of minutes.

Here's what your program does:

- Loads the XKCD home page.
- Saves the comic image on that page.
- Follows the Previous Comic link.
- Repeats until it reaches the first comic.

Figure 11-6: XKCD, "a webcomic of romance, sarcasm, math, and language"

This means your code will need to do the following:

- Download pages with the `requests` module.
- Find the URL of the comic image for a page using Beautiful Soup.
- Download and save the comic image to the hard drive with `iter_content()`.
- Find the URL of the Previous Comic link, and repeat.

Open a new file editor window and save it as *downloadXkcd.py*.

Step 1: Design the Program

If you open the browser's developer tools and inspect the elements on the page, you'll find the following:

- The URL of the comic's image file is given by the `href` attribute of an `<img>` element.
- The `<img>` element is inside a `<div id="comic">` element.
- The Prev button has a `rel` HTML attribute with the value `prev`.
- The first comic's Prev button links to the *http://xkcd.com/#* URL, indicating that there are no more previous pages.

Make your code look like the following:

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

url = 'http://xkcd.com'              # starting url
os.makedirs('xkcd', exist_ok=True)   # store comics in ./xkcd
```

```
while not url.endswith('#'):
    # TODO: Download the page.

    # TODO: Find the URL of the comic image.

    # TODO: Download the image.

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

You'll have a `url` variable that starts with the value `'http://xkcd.com'` and repeatedly update it (in a `for` loop) with the URL of the current page's Prev link. At every step in the loop, you'll download the comic at `url`. You'll know to end the loop when `url` ends with `'#'`.

You will download the image files to a folder in the current working directory named *xkcd*. The call `os.makedirs()` ensures that this folder exists, and the `exist_ok=True` keyword argument prevents the function from throwing an exception if this folder already exists. The rest of the code is just comments that outline the rest of your program.

Step 2: Download the Web Page

Let's implement the code for downloading the page. Make your code look like the following:

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

url = 'http://xkcd.com'               # starting url
os.makedirs('xkcd', exist_ok=True)    # store comics in ./xkcd
while not url.endswith('#'):
    # Download the page.
    print('Downloading page %s...' % url)
    res = requests.get(url)
    res.raise_for_status()

    soup = bs4.BeautifulSoup(res.text)

    # TODO: Find the URL of the comic image.

    # TODO: Download the image.

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

First, print `url` so that the user knows which URL the program is about to download; then use the `requests` module's `request.get()` function to download it. As always, you immediately call the `Response` object's `raise_for_status()` method to throw an exception and end the program if something went wrong with the download. Otherwise, you create a `BeautifulSoup` object from the text of the downloaded page.

Step 3: Find and Download the Comic Image

Make your code look like the following:

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

--snip--

    # Find the URL of the comic image.
    comicElem = soup.select('#comic img')
    if comicElem == []:
        print('Could not find comic image.')
    else:
        comicUrl = comicElem[0].get('src')
        # Download the image.
        print('Downloading image %s...' % (comicUrl))
        res = requests.get(comicUrl)
        res.raise_for_status()

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

From inspecting the XKCD home page with your developer tools, you know that the `<img>` element for the comic image is inside a `<div>` element with the `id` attribute set to `comic`, so the selector `'#comic img'` will get you the correct `<img>` element from the `BeautifulSoup` object.

A few XKCD pages have special content that isn't a simple image file. That's fine; you'll just skip those. If your selector doesn't find any elements, then `soup.select('#comic img')` will return a blank list. When that happens, the program can just print an error message and move on without downloading the image.

Otherwise, the selector will return a list containing one `<img>` element. You can get the `src` attribute from this `<img>` element and pass it to `requests.get()` to download the comic's image file.

Step 4: Save the Image and Find the Previous Comic

Make your code look like the following:

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

--snip--

        # Save the image to ./xkcd.
        imageFile = open(os.path.join('xkcd', os.path.basename(comicUrl)), 'wb')
        for chunk in res.iter_content(100000):
            imageFile.write(chunk)
        imageFile.close()

    # Get the Prev button's url.
    prevLink = soup.select('a[rel="prev"]')[0]
    url = 'http://xkcd.com' + prevLink.get('href')

print('Done.')
```

At this point, the image file of the comic is stored in the `res` variable. You need to write this image data to a file on the hard drive.

You'll need a filename for the local image file to pass to `open()`. The `comicUrl` will have a value like `'http://imgs.xkcd.com/comics/heartbleed_explanation.png'`—which you might have noticed looks a lot like a file path. And in fact, you can call `os.path.basename()` with `comicUrl`, and it will return just the last part of the URL, `'heartbleed_explanation.png'`. You can use this as the filename when saving the image to your hard drive. You join this name with the name of your `xkcd` folder using `os.path.join()` so that your program uses backslashes (`\`) on Windows and forward slashes (`/`) on OS X and Linux. Now that you finally have the filename, you can call `open()` to open a new file in `'wb'` "write binary" mode.

Remember from earlier in this chapter that to save files you've downloaded using Requests, you need to loop over the return value of the `iter_content()` method. The code in the `for` loop writes out chunks of the image data (at most 100,000 bytes each) to the file and then you close the file. The image is now saved to your hard drive.

Afterward, the selector `'a[rel="prev"]'` identifies the `<a>` element with the `rel` attribute set to `prev`, and you can use this `<a>` element's `href` attribute to get the previous comic's URL, which gets stored in `url`. Then the `while` loop begins the entire download process again for this comic.

The output of this program will look like this:

```
Downloading page http://xkcd.com...
Downloading image http://imgs.xkcd.com/comics/phone_alarm.png...
Downloading page http://xkcd.com/1358/...
```

```
Downloading image http://imgs.xkcd.com/comics/nro.png...
Downloading page http://xkcd.com/1357/...
Downloading image http://imgs.xkcd.com/comics/free_speech.png...
Downloading page http://xkcd.com/1356/...
Downloading image http://imgs.xkcd.com/comics/orbital_mechanics.png...
Downloading page http://xkcd.com/1355/...
Downloading image http://imgs.xkcd.com/comics/airplane_message.png...
Downloading page http://xkcd.com/1354/...
Downloading image http://imgs.xkcd.com/comics/heartbleed_explanation.png...
--snip--
```

This project is a good example of a program that can automatically follow links in order to scrape large amounts of data from the Web. You can learn about Beautiful Soup's other features from its documentation at *http://www.crummy.com/software/BeautifulSoup/bs4/doc/*.

Ideas for Similar Programs

Downloading pages and following links are the basis of many web crawling programs. Similar programs could also do the following:

- Back up an entire site by following all of its links.
- Copy all the messages off a web forum.
- Duplicate the catalog of items for sale on an online store.

The `requests` and `BeautifulSoup` modules are great as long as you can figure out the URL you need to pass to `requests.get()`. However, sometimes this isn't so easy to find. Or perhaps the website you want your program to navigate requires you to log in first. The `selenium` module will give your programs the power to perform such sophisticated tasks.

Controlling the Browser with the selenium Module

The `selenium` module lets Python directly control the browser by programmatically clicking links and filling in login information, almost as though there is a human user interacting with the page. Selenium allows you to interact with web pages in a much more advanced way than Requests and Beautiful Soup; but because it launches a web browser, it is a bit slower and hard to run in the background if, say, you just need to download some files from the Web.

Appendix A has more detailed steps on installing third-party modules.

Starting a Selenium-Controlled Browser

For these examples, you'll need the Firefox web browser. This will be the browser that you control. If you don't already have Firefox, you can download it for free from *http://getfirefox.com/*.

Importing the modules for Selenium is slightly tricky. Instead of `import selenium`, you need to run `from selenium import webdriver`. (The exact reason why the `selenium` module is set up this way is beyond the scope of this book.) After that, you can launch the Firefox browser with Selenium. Enter the following into the interactive shell:

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> type(browser)
<class 'selenium.webdriver.firefox.webdriver.WebDriver'>
>>> browser.get('http://inventwithpython.com')
```

You'll notice when `webdriver.Firefox()` is called, the Firefox web browser starts up. Calling `type()` on the value `webdriver.Firefox()` reveals it's of the `WebDriver` data type. And calling `browser.get('http://inventwithpython.com')` directs the browser to *http://inventwithpython.com/*. Your browser should look something like Figure 11-7.

Figure 11-7: After calling `webdriver.Firefox()` and `get()` in IDLE, the Firefox browser appears.

Finding Elements on the Page

`WebDriver` objects have quite a few methods for finding elements on a page. They are divided into the `find_element_*` and `find_elements_*` methods. The `find_element_*` methods return a single `WebElement` object, representing the first element on the page that matches your query. The `find_elements_*` methods return a list of `WebElement_*` objects for *every* matching element on the page.

Table 11-3 shows several examples of `find_element_*` and `find_elements_*` methods being called on a `WebDriver` object that's stored in the variable `browser`.

Table 11-3: Selenium's `WebDriver` Methods for Finding Elements

Method name	WebElement object/list returned
`browser.find_element_by_class_name(name)` `browser.find_elements_by_class_name(name)`	Elements that use the CSS class *name*
`browser.find_element_by_css_selector(selector)` `browser.find_elements_by_css_selector(selector)`	Elements that match the CSS *selector*
`browser.find_element_by_id(id)` `browser.find_elements_by_id(id)`	Elements with a matching *id* attribute value
`browser.find_element_by_link_text(text)` `browser.find_elements_by_link_text(text)`	`<a>` elements that completely match the *text* provided
`browser.find_element_by_partial_link_text(text)` `browser.find_elements_by_partial_link_text(text)`	`<a>` elements that contain the *text* provided
`browser.find_element_by_name(name)` `browser.find_elements_by_name(name)`	Elements with a matching *name* attribute value
`browser.find_element_by_tag_name(name)` `browser.find_elements_by_tag_name(name)`	Elements with a matching tag *name* (case insensitive; an `<a>` element is matched by `'a'` and `'A'`)

Except for the `*_by_tag_name()` methods, the arguments to all the methods are case sensitive. If no elements exist on the page that match what the method is looking for, the `selenium` module raises a `NoSuchElement` exception. If you do not want this exception to crash your program, add `try` and `except` statements to your code.

Once you have the `WebElement` object, you can find out more about it by reading the attributes or calling the methods in Table 11-4.

Table 11-4: WebElement Attributes and Methods

Attribute or method	Description
`tag_name`	The tag name, such as `'a'` for an `<a>` element
`get_attribute(name)`	The value for the element's `name` attribute
`text`	The text within the element, such as `'hello'` in `<span>hello</span>`
`clear()`	For text field or text area elements, clears the text typed into it
`is_displayed()`	Returns `True` if the element is visible; otherwise returns `False`
`is_enabled()`	For input elements, returns `True` if the element is enabled; otherwise returns `False`
`is_selected()`	For checkbox or radio button elements, returns `True` if the element is selected; otherwise returns `False`
`location`	A dictionary with keys `'x'` and `'y'` for the position of the element in the page

For example, open a new file editor and enter the following program:

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://inventwithpython.com')
```

```
try:
    elem = browser.find_element_by_class_name('bookcover')
    print('Found <%s> element with that class name!' % (elem.tag_name))
except:
    print('Was not able to find an element with that name.')
```

Here we open Firefox and direct it to a URL. On this page, we try to find elements with the class name `'bookcover'`, and if such an element is found, we print its tag name using the `tag_name` attribute. If no such element was found, we print a different message.

This program will output the following:

```
Found <img> element with that class name!
```

We found an element with the class name `'bookcover'` and the tag name `'img'`.

Clicking the Page

`WebElement` objects returned from the `find_element_*` and `find_elements_*` methods have a `click()` method that simulates a mouse click on that element. This method can be used to follow a link, make a selection on a radio button, click a Submit button, or trigger whatever else might happen when the element is clicked by the mouse. For example, enter the following into the interactive shell:

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> browser.get('http://inventwithpython.com')
>>> linkElem = browser.find_element_by_link_text('Read It Online')
>>> type(linkElem)
<class 'selenium.webdriver.remote.webelement.WebElement'>
>>> linkElem.click()    # follows the "Read It Online" link
```

This opens Firefox to *http://inventwithpython.com/*, gets the `WebElement` object for the `<a>` element with the text *Read It Online*, and then simulates clicking that `<a>` element. It's just like if you clicked the link yourself; the browser then follows that link.

Filling Out and Submitting Forms

Sending keystrokes to text fields on a web page is a matter of finding the `<input>` or `<textarea>` element for that text field and then calling the `send_keys()` method. For example, enter the following into the interactive shell:

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> browser.get('http://gmail.com')
>>> emailElem = browser.find_element_by_id('Email')
>>> emailElem.send_keys('not_my_real_email@gmail.com')
>>> passwordElem = browser.find_element_by_id('Passwd')
```

```
>>> passwordElem.send_keys('12345')
>>> passwordElem.submit()
```

As long as Gmail hasn't changed the `id` of the Username and Password text fields since this book was published, the previous code will fill in those text fields with the provided text. (You can always use the browser's inspector to verify the `id`.) Calling the `submit()` method on any element will have the same result as clicking the Submit button for the form that element is in. (You could have just as easily called `emailElem.submit()`, and the code would have done the same thing.)

Sending Special Keys

Selenium has a module for keyboard keys that are impossible to type into a string value, which function much like escape characters. These values are stored in attributes in the `selenium.webdriver.common.keys` module. Since that is such a long module name, it's much easier to run `from selenium.webdriver.common.keys import Keys` at the top of your program; if you do, then you can simply write `Keys` anywhere you'd normally have to write `selenium.webdriver.common.keys`. Table 11-5 lists the commonly used `Keys` variables.

Table 11-5: Commonly Used Variables in the `selenium.webdriver.common.keys` Module

Attributes	Meanings
`Keys.DOWN`, `Keys.UP`, `Keys.LEFT`, `Keys.RIGHT`	The keyboard arrow keys
`Keys.ENTER`, `Keys.RETURN`	The ENTER and RETURN keys
`Keys.HOME`, `Keys.END`, `Keys.PAGE_DOWN`, `Keys.PAGE_UP`	The HOME, END, PAGEDOWN, and PAGEUP keys
`Keys.ESCAPE`, `Keys.BACK_SPACE`, `Keys.DELETE`	The ESC, BACKSPACE, and DELETE keys
`Keys.F1`, `Keys.F2`, . . . , `Keys.F12`	The F1 to F12 keys at the top of the keyboard
`Keys.TAB`	The TAB key

For example, if the cursor is not currently in a text field, pressing the HOME and END keys will scroll the browser to the top and bottom of the page, respectively. Enter the following into the interactive shell, and notice how the `send_keys()` calls scroll the page:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.common.keys import Keys
>>> browser = webdriver.Firefox()
>>> browser.get('http://nostarch.com')
>>> htmlElem = browser.find_element_by_tag_name('html')
>>> htmlElem.send_keys(Keys.END)     # scrolls to bottom
>>> htmlElem.send_keys(Keys.HOME)    # scrolls to top
```

The `<html>` tag is the base tag in HTML files: The full content of the HTML file is enclosed within the `<html>` and `</html>` tags. Calling `browser.find_element_by_tag_name('html')` is a good place to send keys to the general web page. This would be useful if, for example, new content is loaded once you've scrolled to the bottom of the page.

Clicking Browser Buttons

Selenium can simulate clicks on various browser buttons as well through the following methods:

`browser.back()` Clicks the Back button.

`browser.forward()` Clicks the Forward button.

`browser.refresh()` Clicks the Refresh/Reload button.

`browser.quit()` Clicks the Close Window button.

More Information on Selenium

Selenium can do much more beyond the functions described here. It can modify your browser's cookies, take screenshots of web pages, and run custom JavaScript. To learn more about these features, you can visit the Selenium documentation at *http://selenium-python.readthedocs.org/*.

Summary

Most boring tasks aren't limited to the files on your computer. Being able to programmatically download web pages will extend your programs to the Internet. The `requests` module makes downloading straightforward, and with some basic knowledge of HTML concepts and selectors, you can utilize the `BeautifulSoup` module to parse the pages you download.

But to fully automate any web-based tasks, you need direct control of your web browser through the `selenium` module. The `selenium` module will allow you to log in to websites and fill out forms automatically. Since a web browser is the most common way to send and receive information over the Internet, this is a great ability to have in your programmer toolkit.

Practice Questions

1. Briefly describe the differences between the `webbrowser`, `requests`, `BeautifulSoup`, and `selenium` modules.
2. What type of object is returned by `requests.get()`? How can you access the downloaded content as a string value?
3. What Requests method checks that the download worked?
4. How can you get the HTTP status code of a Requests response?
5. How do you save a Requests response to a file?

6. What is the keyboard shortcut for opening a browser's developer tools?
7. How can you view (in the developer tools) the HTML of a specific element on a web page?
8. What is the CSS selector string that would find the element with an `id` attribute of `main`?
9. What is the CSS selector string that would find the elements with a CSS class of `highlight`?
10. What is the CSS selector string that would find all the `<div>` elements inside another `<div>` element?
11. What is the CSS selector string that would find the `<button>` element with a `value` attribute set to `favorite`?
12. Say you have a Beautiful Soup `Tag` object stored in the variable `spam` for the element `<div>Hello world!</div>`. How could you get a string `'Hello world!'` from the `Tag` object?
13. How would you store all the attributes of a Beautiful Soup `Tag` object in a variable named `linkElem`?
14. Running `import selenium` doesn't work. How do you properly import the `selenium` module?
15. What's the difference between the `find_element_*` and `find_elements_*` methods?
16. What methods do Selenium's `WebElement` objects have for simulating mouse clicks and keyboard keys?
17. You could call `send_keys(Keys.ENTER)` on the Submit button's `WebElement` object, but what is an easier way to submit a form with Selenium?
18. How can you simulate clicking a browser's Forward, Back, and Refresh buttons with Selenium?

Practice Projects

For practice, write programs to do the following tasks.

Command Line Emailer

Write a program that takes an email address and string of text on the command line and then, using Selenium, logs into your email account and sends an email of the string to the provided address. (You might want to set up a separate email account for this program.)

This would be a nice way to add a notification feature to your programs. You could also write a similar program to send messages from a Facebook or Twitter account.

Image Site Downloader

Write a program that goes to a photo-sharing site like Flickr or Imgur, searches for a category of photos, and then downloads all the resulting images. You could write a program that works with any photo site that has a search feature.

2048

2048 is a simple game where you combine tiles by sliding them up, down, left, or right with the arrow keys. You can actually get a fairly high score by repeatedly sliding in an up, right, down, and left pattern over and over again. Write a program that will open the game at *https://gabrielecirulli.github.io/2048/* and keep sending up, right, down, and left keystrokes to automatically play the game.

Link Verification

Write a program that, given the URL of a web page, will attempt to download every linked page on the page. The program should flag any pages that have a 404 "Not Found" status code and print them out as broken links.

12

WORKING WITH EXCEL SPREADSHEETS

Excel is a popular and powerful spreadsheet application for Windows. The `openpyxl` module allows your Python programs to read and modify Excel spreadsheet files. For example, you might have the boring task of copying certain data from one spreadsheet and pasting it into another one. Or you might have to go through thousands of rows and pick out just a handful of them to make small edits based on some criteria. Or you might have to look through hundreds of spreadsheets of department budgets, searching for any that are in the red. These are exactly the sort of boring, mindless spreadsheet tasks that Python can do for you.

Although Excel is proprietary software from Microsoft, there are free alternatives that run on Windows, OS X, and Linux. Both LibreOffice Calc and OpenOffice Calc work with Excel's *.xlsx* file format for spreadsheets, which means the `openpyxl` module can work on spreadsheets from these applications as well. You can download the software from *https://www.libreoffice.org/* and *http://www.openoffice.org/*, respectively. Even if you already have

Excel installed on your computer, you may find these programs easier to use. The screenshots in this chapter, however, are all from Excel 2010 on Windows 7.

Excel Documents

First, let's go over some basic definitions: An Excel spreadsheet document is called a *workbook*. A single workbook is saved in a file with the *.xlsx* extension. Each workbook can contain multiple *sheets* (also called *worksheets*). The sheet the user is currently viewing (or last viewed before closing Excel) is called the *active sheet*.

Each sheet has *columns* (addressed by letters starting at *A*) and *rows* (addressed by numbers starting at 1). A box at a particular column and row is called a *cell*. Each cell can contain a number or text value. The grid of cells with data makes up a sheet.

Installing the openpyxl Module

Python does not come with OpenPyXL, so you'll have to install it. Follow the instructions for installing third-party modules in Appendix A; the name of the module is `openpyxl`. To test whether it is installed correctly, enter the following into the interactive shell:

```
>>> import openpyxl
```

If the module was correctly installed, this should produce no error messages. Remember to import the `openpyxl` module before running the interactive shell examples in this chapter, or you'll get a `NameError: name 'openpyxl' is not defined` error.

This book covers version 2.1.4 of OpenPyXL, but new versions are regularly released by the OpenPyXL team. Don't worry, though: New versions should stay backward compatible with the instructions in this book for quite some time. If you have a newer version and want to see what additional features may be available to you, you can check out the full documentation for OpenPyXL at *http://openpyxl.readthedocs.org/*.

Reading Excel Documents

The examples in this chapter will use a spreadsheet named *example.xlsx* stored in the root folder. You can either create the spreadsheet yourself or download it from *http://nostarch.com/automatestuff/*. Figure 12-1 shows the tabs for the three default sheets named *Sheet1*, *Sheet2*, and *Sheet3* that Excel automatically provides for new workbooks. (The number of default sheets created may vary between operating systems and spreadsheet programs.)

Figure 12-1: The tabs for a workbook's sheets are in the lower-left corner of Excel.

Sheet 1 in the example file should look like Table 12-1. (If you didn't download *example.xlsx* from the website, you should enter this data into the sheet yourself.)

Table 12-1: The *example.xlsx* Spreadsheet

	A	B	C
1	4/5/2015 1:34:02 PM	Apples	73
2	4/5/2015 3:41:23 AM	Cherries	85
3	4/6/2015 12:46:51 PM	Pears	14
4	4/8/2015 8:59:43 AM	Oranges	52
5	4/10/2015 2:07:00 AM	Apples	152
6	4/10/2015 6:10:37 PM	Bananas	23
7	4/10/2015 2:40:46 AM	Strawberries	98

Now that we have our example spreadsheet, let's see how we can manipulate it with the `openpyxl` module.

Opening Excel Documents with OpenPyXL

Once you've imported the `openpyxl` module, you'll be able to use the `openpyxl.load_workbook()` function. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> type(wb)
<class 'openpyxl.workbook.workbook.Workbook'>
```

The `openpyxl.load_workbook()` function takes in the filename and returns a value of the `workbook` data type. This `Workbook` object represents the Excel file, a bit like how a `File` object represents an opened text file.

Remember that *example.xlsx* needs to be in the current working directory in order for you to work with it. You can find out what the current working directory is by importing `os` and using `os.getcwd()`, and you can change the current working directory using `os.chdir()`.

Getting Sheets from the Workbook

You can get a list of all the sheet names in the workbook by calling the `get_sheet_names()` method. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> wb.get_sheet_names()
['Sheet1', 'Sheet2', 'Sheet3']
>>> sheet = wb.get_sheet_by_name('Sheet3')
>>> sheet
<Worksheet "Sheet3">
>>> type(sheet)
<class 'openpyxl.worksheet.worksheet.Worksheet'>
>>> sheet.title
'Sheet3'
>>> anotherSheet = wb.get_active_sheet()
>>> anotherSheet
<Worksheet "Sheet1">
```

Each sheet is represented by a `Worksheet` object, which you can obtain by passing the sheet name string to the `get_sheet_by_name()` workbook method. Finally, you can call the `get_active_sheet()` method of a `Workbook` object to get the workbook's active sheet. The active sheet is the sheet that's on top when the workbook is opened in Excel. Once you have the `Worksheet` object, you can get its name from the `title` attribute.

Getting Cells from the Sheets

Once you have a `Worksheet` object, you can access a `Cell` object by its name. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> sheet['A1']
<Cell Sheet1.A1>
>>> sheet['A1'].value
datetime.datetime(2015, 4, 5, 13, 34, 2)
>>> c = sheet['B1']
>>> c.value
'Apples'
>>> 'Row ' + str(c.row) + ', Column ' + c.column + ' is ' + c.value
'Row 1, Column B is Apples'
>>> 'Cell ' + c.coordinate + ' is ' + c.value
'Cell B1 is Apples'
>>> sheet['C1'].value
73
```

The `Cell` object has a `value` attribute that contains, unsurprisingly, the value stored in that cell. `Cell` objects also have `row`, `column`, and `coordinate` attributes that provide location information for the cell.

Here, accessing the `value` attribute of our `Cell` object for cell B1 gives us the string `'Apples'`. The `row` attribute gives us the integer `1`, the `column` attribute gives us `'B'`, and the `coordinate` attribute gives us `'B1'`.

OpenPyXL will automatically interpret the dates in column A and return them as `datetime` values rather than strings. The `datetime` data type is explained further in Chapter 16.

Specifying a column by letter can be tricky to program, especially because after column Z, the columns start by using two letters: AA, AB, AC, and so on. As an alternative, you can also get a cell using the sheet's `cell()` method and passing integers for its `row` and `column` keyword arguments. The first row or column integer is `1`, not `0`. Continue the interactive shell example by entering the following:

```
>>> sheet.cell(row=1, column=2)
<Cell Sheet1.B1>
>>> sheet.cell(row=1, column=2).value
'Apples'
>>> for i in range(1, 8, 2):
        print(i, sheet.cell(row=i, column=2).value)

1 Apples
3 Pears
5 Apples
7 Strawberries
```

As you can see, using the sheet's `cell()` method and passing it `row=1` and `column=2` gets you a `Cell` object for cell `B1`, just like specifying `sheet['B1']` did. Then, using the `cell()` method and its keyword arguments, you can write a `for` loop to print the values of a series of cells.

Say you want to go down column B and print the value in every cell with an odd row number. By passing `2` for the `range()` function's "step" parameter, you can get cells from every second row (in this case, all the odd-numbered rows). The `for` loop's `i` variable is passed for the `row` keyword argument to the `cell()` method, while `2` is always passed for the `column` keyword argument. Note that the integer `2`, not the string `'B'`, is passed.

You can determine the size of the sheet with the `Worksheet` object's `get_highest_row()` and `get_highest_column()` methods. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> sheet.get_highest_row()
7
>>> sheet.get_highest_column()
3
```

Note that the `get_highest_column()` method returns an integer rather than the letter that appears in Excel.

Converting Between Column Letters and Numbers

To convert from letters to numbers, call the `openpyxl.cell.column_index_from_string()` function. To convert from numbers to letters, call the `openpyxl.cell.get_column_letter()` function. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> from openpyxl.cell import get_column_letter, column_index_from_string
>>> get_column_letter(1)
'A'
>>> get_column_letter(2)
'B'
>>> get_column_letter(27)
'AA'
>>> get_column_letter(900)
'AHP'
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> get_column_letter(sheet.get_highest_column())
'C'
>>> column_index_from_string('A')
1
>>> column_index_from_string('AA')
27
```

After you import these two functions from the `openpyxl.cell` module, you can call `get_column_letter()` and pass it an integer like 27 to figure out what the letter name of the 27th column is. The function `column_index_string()` does the reverse: You pass it the letter name of a column, and it tells you what number that column is. You don't need to have a workbook loaded to use these functions. If you want, you can load a workbook, get a `Worksheet` object, and call a `Worksheet` object method like `get_highest_column()` to get an integer. Then, you can pass that integer to `get_column_letter()`.

Getting Rows and Columns from the Sheets

You can slice `Worksheet` objects to get all the `Cell` objects in a row, column, or rectangular area of the spreadsheet. Then you can loop over all the cells in the slice. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> tuple(sheet['A1':'C3'])
((<Cell Sheet1.A1>, <Cell Sheet1.B1>, <Cell Sheet1.C1>), (<Cell Sheet1.A2>,
<Cell Sheet1.B2>, <Cell Sheet1.C2>), (<Cell Sheet1.A3>, <Cell Sheet1.B3>,
<Cell Sheet1.C3>))
❶ >>> for rowOfCellObjects in sheet['A1':'C3']:
❷         for cellObj in rowOfCellObjects:
              print(cellObj.coordinate, cellObj.value)
          print('--- END OF ROW ---')
```

```
A1 2015-04-05 13:34:02
B1 Apples
C1 73
--- END OF ROW ---
A2 2015-04-05 03:41:23
B2 Cherries
C2 85
--- END OF ROW ---
A3 2015-04-06 12:46:51
B3 Pears
C3 14
--- END OF ROW ---
```

Here, we specify that we want the `Cell` objects in the rectangular area from A1 to C3, and we get a `Generator` object containing the `Cell` objects in that area. To help us visualize this `Generator` object, we can use `tuple()` on it to display its `Cell` objects in a tuple.

This tuple contains three tuples: one for each row, from the top of the desired area to the bottom. Each of these three inner tuples contains the `Cell` objects in one row of our desired area, from the leftmost cell to the right. So overall, our slice of the sheet contains all the `Cell` objects in the area from A1 to C3, starting from the top-left cell and ending with the bottom-right cell.

To print the values of each cell in the area, we use two `for` loops. The outer `for` loop goes over each row in the slice ❶. Then, for each row, the nested `for` loop goes through each cell in that row ❷.

To access the values of cells in a particular row or column, you can also use a `Worksheet` object's `rows` and `columns` attribute. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_active_sheet()
>>> sheet.columns[1]
(<Cell Sheet1.B1>, <Cell Sheet1.B2>, <Cell Sheet1.B3>, <Cell Sheet1.B4>,
<Cell Sheet1.B5>, <Cell Sheet1.B6>, <Cell Sheet1.B7>)
>>> for cellObj in sheet.columns[1]:
        print(cellObj.value)

Apples
Cherries
Pears
Oranges
Apples
Bananas
Strawberries
```

Using the `rows` attribute on a `Worksheet` object will give you a tuple of tuples. Each of these inner tuples represents a row, and contains the `Cell` objects in that row. The `columns` attribute also gives you a tuple of tuples, with each of the inner tuples containing the `Cell` objects in a particular

column. For *example.xlsx*, since there are 7 rows and 3 columns, `rows` gives us a tuple of 7 tuples (each containing 3 `Cell` objects), and `columns` gives us a tuple of 3 tuples (each containing 7 `Cell` objects).

To access one particular tuple, you can refer to it by its index in the larger tuple. For example, to get the tuple that represents column B, you use `sheet.columns[1]`. To get the tuple containing the `Cell` objects in column A, you'd use `sheet.columns[0]`. Once you have a tuple representing one row or column, you can loop through its `Cell` objects and print their values.

Workbooks, Sheets, Cells

As a quick review, here's a rundown of all the functions, methods, and data types involved in reading a cell out of a spreadsheet file:

1. Import the `openpyxl` module.
2. Call the `openpyxl.load_workbook()` function.
3. Get a `Workbook` object.
4. Call the `get_active_sheet()` or `get_sheet_by_name()` workbook method.
5. Get a `Worksheet` object.
6. Use indexing or the `cell()` sheet method with `row` and `column` keyword arguments.
7. Get a `Cell` object.
8. Read the `Cell` object's `value` attribute.

Project: Reading Data from a Spreadsheet

Say you have a spreadsheet of data from the 2010 US Census and you have the boring task of going through its thousands of rows to count both the total population and the number of census tracts for each county. (A census tract is simply a geographic area defined for the purposes of the census.) Each row represents a single census tract. We'll name the spreadsheet file *censuspopdata.xlsx*, and you can download it from *http://nostarch.com/automatestuff/*. Its contents look like Figure 12-2.

	A	B	C	D	E
1	**CensusTract**	**State**	**County**	**POP2010**	
9841	06075010500	CA	San Francisco	2685	
9842	06075010600	CA	San Francisco	3894	
9843	06075010700	CA	San Francisco	5592	
9844	06075010800	CA	San Francisco	4578	
9845	06075010900	CA	San Francisco	4320	
9846	06075011000	CA	San Francisco	4827	
9847	06075011100	CA	San Francisco	5164	

Population by Census Tract

Ready

Figure 12-2: The censuspopdata.xlsx *spreadsheet*

Even though Excel can calculate the sum of multiple selected cells, you'd still have to select the cells for each of the 3,000-plus counties. Even if it takes just a few seconds to calculate a county's population by hand, this would take hours to do for the whole spreadsheet.

In this project, you'll write a script that can read from the census spreadsheet file and calculate statistics for each county in a matter of seconds.

This is what your program does:

- Reads the data from the Excel spreadsheet.
- Counts the number of census tracts in each county.
- Counts the total population of each county.
- Prints the results.

This means your code will need to do the following:

- Open and read the cells of an Excel document with the `openpyxl` module.
- Calculate all the tract and population data and store it in a data structure.
- Write the data structure to a text file with the *.py* extension using the `pprint` module.

Step 1: Read the Spreadsheet Data

There is just one sheet in the *censuspopdata.xlsx* spreadsheet, named `'Population by Census Tract'`, and each row holds the data for a single census tract. The columns are the tract number (A), the state abbreviation (B), the county name (C), and the population of the tract (D).

Open a new file editor window and enter the following code. Save the file as *readCensusExcel.py*.

```
#! python3
# readCensusExcel.py - Tabulates population and number of census tracts for
# each county.

❶ import openpyxl, pprint
  print('Opening workbook...')
❷ wb = openpyxl.load_workbook('censuspopdata.xlsx')
❸ sheet = wb.get_sheet_by_name('Population by Census Tract')
  countyData = {}

  # TODO: Fill in countyData with each county's population and tracts.
  print('Reading rows...')
❹ for row in range(2, sheet.get_highest_row() + 1):
      # Each row in the spreadsheet has data for one census tract.
      state  = sheet['B' + str(row)].value
      county = sheet['C' + str(row)].value
      pop    = sheet['D' + str(row)].value

  # TODO: Open a new text file and write the contents of countyData to it.
```

This code imports the `openpyxl` module, as well as the `pprint` module that you'll use to print the final county data ❶. Then it opens the *censuspopdata.xlsx* file ❷, gets the sheet with the census data ❸, and begins iterating over its rows ❹.

Note that you've also created a variable named `countyData`, which will contain the populations and number of tracts you calculate for each county. Before you can store anything in it, though, you should determine exactly how you'll structure the data inside it.

Step 2: Populate the Data Structure

The data structure stored in `countyData` will be a dictionary with state abbreviations as its keys. Each state abbreviation will map to another dictionary, whose keys are strings of the county names in that state. Each county name will in turn map to a dictionary with just two keys, `'tracts'` and `'pop'`. These keys map to the number of census tracts and population for the county. For example, the dictionary will look similar to this:

```
{'AK': {'Aleutians East': {'pop': 3141, 'tracts': 1},
        'Aleutians West': {'pop': 5561, 'tracts': 2},
        'Anchorage': {'pop': 291826, 'tracts': 55},
        'Bethel': {'pop': 17013, 'tracts': 3},
        'Bristol Bay': {'pop': 997, 'tracts': 1},
        --snip--
```

If the previous dictionary were stored in `countyData`, the following expressions would evaluate like this:

```
>>> countyData['AK']['Anchorage']['pop']
291826
>>> countyData['AK']['Anchorage']['tracts']
55
```

More generally, the `countyData` dictionary's keys will look like this:

```
countyData[state abbrev][county]['tracts']
countyData[state abbrev][county]['pop']
```

Now that you know how `countyData` will be structured, you can write the code that will fill it with the county data. Add the following code to the bottom of your program:

```
#! python 3
# readCensusExcel.py - Tabulates population and number of census tracts for
# each county.

--snip--
```

```
    for row in range(2, sheet.get_highest_row() + 1):
        # Each row in the spreadsheet has data for one census tract.
        state  = sheet['B' + str(row)].value
        county = sheet['C' + str(row)].value
        pop    = sheet['D' + str(row)].value

        # Make sure the key for this state exists.
❶       countyData.setdefault(state, {})
        # Make sure the key for this county in this state exists.
❷       countyData[state].setdefault(county, {'tracts': 0, 'pop': 0})

        # Each row represents one census tract, so increment by one.
❸       countyData[state][county]['tracts'] += 1
        # Increase the county pop by the pop in this census tract.
❹       countyData[state][county]['pop'] += int(pop)

    # TODO: Open a new text file and write the contents of countyData to it.
```

The last two lines of code perform the actual calculation work, incrementing the value for `tracts` ❸ and increasing the value for `pop` ❹ for the current county on each iteration of the `for` loop.

The other code is there because you cannot add a county dictionary as the value for a state abbreviation key until the key itself exists in `countyData`. (That is, `countyData['AK']['Anchorage']['tracts'] += 1` will cause an error if the `'AK'` key doesn't exist yet.) To make sure the state abbreviation key exists in your data structure, you need to call the `setdefault()` method to set a value if one does not already exist for `state` ❶.

Just as the `countyData` dictionary needs a dictionary as the value for each state abbreviation key, each of *those* dictionaries will need its own dictionary as the value for each county key ❷. And each of *those* dictionaries in turn will need keys `'tracts'` and `'pop'` that start with the integer value `0`. (If you ever lose track of the dictionary structure, look back at the example dictionary at the start of this section.)

Since `setdefault()` will do nothing if the key already exists, you can call it on every iteration of the `for` loop without a problem.

Step 3: Write the Results to a File

After the `for` loop has finished, the `countyData` dictionary will contain all of the population and tract information keyed by county and state. At this point, you could program more code to write this to a text file or another Excel spreadsheet. For now, let's just use the `pprint.pformat()` function to write the `countyData` dictionary value as a massive string to a file named *census2010.py*. Add the following code to the bottom of your program (making sure to keep it unindented so that it stays outside the `for` loop):

```
#! python 3
# readCensusExcel.py - Tabulates population and number of census tracts for
# each county.
```

```
--snip--

for row in range(2, sheet.get_highest_row() + 1):
--snip--

# Open a new text file and write the contents of countyData to it.
print('Writing results...')
resultFile = open('census2010.py', 'w')
resultFile.write('allData = ' + pprint.pformat(countyData))
resultFile.close()
print('Done.')
```

The `pprint.pformat()` function produces a string that itself is formatted as valid Python code. By outputting it to a text file named *census2010.py*, you've generated a Python program from your Python program! This may seem complicated, but the advantage is that you can now import *census2010.py* just like any other Python module. In the interactive shell, change the current working directory to the folder with your newly created *census2010.py* file (on my laptop, this is *C:\Python34*), and then import it:

```
>>> import os
>>> os.chdir('C:\\Python34')
>>> import census2010
>>> census2010.allData['AK']['Anchorage']
{'pop': 291826, 'tracts': 55}
>>> anchoragePop = census2010.allData['AK']['Anchorage']['pop']
>>> print('The 2010 population of Anchorage was ' + str(anchoragePop))
The 2010 population of Anchorage was 291826
```

The *readCensusExcel.py* program was throwaway code: Once you have its results saved to *census2010.py*, you won't need to run the program again. Whenever you need the county data, you can just run `import census2010`.

Calculating this data by hand would have taken hours; this program did it in a few seconds. Using OpenPyXL, you will have no trouble extracting information that is saved to an Excel spreadsheet and performing calculations on it. You can download the complete program from *http://nostarch.com/automatestuff/*.

Ideas for Similar Programs

Many businesses and offices use Excel to store various types of data, and it's not uncommon for spreadsheets to become large and unwieldy. Any program that parses an Excel spreadsheet has a similar structure: It loads the spreadsheet file, preps some variables or data structures, and then loops through each of the rows in the spreadsheet. Such a program could do the following:

- Compare data across multiple rows in a spreadsheet.
- Open multiple Excel files and compare data between spreadsheets.

- Check whether a spreadsheet has blank rows or invalid data in any cells and alert the user if it does.
- Read data from a spreadsheet and use it as the input for your Python programs.

Writing Excel Documents

OpenPyXL also provides ways of writing data, meaning that your programs can create and edit spreadsheet files. With Python, it's simple to create spreadsheets with thousands of rows of data.

Creating and Saving Excel Documents

Call the `openpyxl.Workbook()` function to create a new, blank `Workbook` object. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> wb.get_sheet_names()
['Sheet']
>>> sheet = wb.get_active_sheet()
>>> sheet.title
'Sheet'
>>> sheet.title = 'Spam Bacon Eggs Sheet'
>>> wb.get_sheet_names()
['Spam Bacon Eggs Sheet']
```

The workbook will start off with a single sheet named *Sheet*. You can change the name of the sheet by storing a new string in its `title` attribute.

Any time you modify the `Workbook` object or its sheets and cells, the spreadsheet file will not be saved until you call the `save()` workbook method. Enter the following into the interactive shell (with *example.xlsx* in the current working directory):

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_active_sheet()
>>> sheet.title = 'Spam Spam Spam'
>>> wb.save('example_copy.xlsx')
```

Here, we change the name of our sheet. To save our changes, we pass a filename as a string to the `save()` method. Passing a different filename than the original, such as `'example_copy.xlsx'`, saves the changes to a copy of the spreadsheet.

Whenever you edit a spreadsheet you've loaded from a file, you should always save the new, edited spreadsheet to a different filename than the original. That way, you'll still have the original spreadsheet file to work with in case a bug in your code caused the new, saved file to have incorrect or corrupt data.

Creating and Removing Sheets

Sheets can be added to and removed from a workbook with the `create_sheet()` and `remove_sheet()` methods. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> wb.get_sheet_names()
['Sheet']
>>> wb.create_sheet()
<Worksheet "Sheet1">
>>> wb.get_sheet_names()
['Sheet', 'Sheet1']
>>> wb.create_sheet(index=0, title='First Sheet')
<Worksheet "First Sheet">
>>> wb.get_sheet_names()
['First Sheet', 'Sheet', 'Sheet1']
>>> wb.create_sheet(index=2, title='Middle Sheet')
<Worksheet "Middle Sheet">
>>> wb.get_sheet_names()
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
```

The `create_sheet()` method returns a new `Worksheet` object named `SheetX`, which by default is set to be the last sheet in the workbook. Optionally, the index and name of the new sheet can be specified with the `index` and `title` keyword arguments.

Continue the previous example by entering the following:

```
>>> wb.get_sheet_names()
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
>>> wb.remove_sheet(wb.get_sheet_by_name('Middle Sheet'))
>>> wb.remove_sheet(wb.get_sheet_by_name('Sheet1'))
>>> wb.get_sheet_names()
['First Sheet', 'Sheet']
```

The `remove_sheet()` method takes a `Worksheet` object, not a string of the sheet name, as its argument. If you know only the name of a sheet you want to remove, call `get_sheet_by_name()` and pass its return value into `remove_sheet()`.

Remember to call the `save()` method to save the changes after adding sheets to or removing sheets from the workbook.

Writing Values to Cells

Writing values to cells is much like writing values to keys in a dictionary. Enter this into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_sheet_by_name('Sheet')
>>> sheet['A1'] = 'Hello world!'
>>> sheet['A1'].value
'Hello world!'
```

If you have the cell's coordinate as a string, you can use it just like a dictionary key on the `Worksheet` object to specify which cell to write to.

Project: Updating a Spreadsheet

In this project, you'll write a program to update cells in a spreadsheet of produce sales. Your program will look through the spreadsheet, find specific kinds of produce, and update their prices. Download this spreadsheet from *http://nostarch.com/automatestuff/*. Figure 12-3 shows what the spreadsheet looks like.

	A	B	C	D	E
1	PRODUCE	COST PER POUND	POUNDS SOLD	TOTAL	
2	Potatoes	0.86	21.6	18.58	
3	Okra	2.26	38.6	87.24	
4	Fava beans	2.69	32.8	88.23	
5	Watermelon	0.66	27.3	18.02	
6	Garlic	1.19	4.9	5.83	
7	Parsnips	2.27	1.1	2.5	
8	Asparagus	2.49	37.9	94.37	
9	Avocados	3.23	9.2	29.72	
10	Celery	3.07	28.9	88.72	
11	Okra	2.26	40	90.4	

Sheet

Ready 100%

Figure 12-3: A spreadsheet of produce sales

Each row represents an individual sale. The columns are the type of produce sold (A), the cost per pound of that produce (B), the number of pounds sold (C), and the total revenue from the sale (D). The TOTAL column is set to the Excel formula *=ROUND(B3*C3, 2)*, which multiplies the cost per pound by the number of pounds sold and rounds the result to the nearest cent. With this formula, the cells in the TOTAL column will automatically update themselves if there is a change in column B or C.

Now imagine that the prices of garlic, celery, and lemons were entered incorrectly, leaving you with the boring task of going through thousands of rows in this spreadsheet to update the cost per pound for any garlic, celery, and lemon rows. You can't do a simple find-and-replace for the price because there might be other items with the same price that you don't want to mistakenly "correct." For thousands of rows, this would take hours to do by hand. But you can write a program that can accomplish this in seconds.

Your program does the following:

- Loops over all the rows.
- If the row is for garlic, celery, or lemons, changes the price.

This means your code will need to do the following:

- Open the spreadsheet file.
- For each row, check whether the value in column A is `Celery`, `Garlic`, or `Lemon`.
- If it is, update the price in column B.
- Save the spreadsheet to a new file (so that you don't lose the old spreadsheet, just in case).

Step 1: Set Up a Data Structure with the Update Information

The prices that you need to update are as follows:

Celery	1.19
Garlic	3.07
Lemon	1.27

You could write code like this:

```
if produceName == 'Celery':
    cellObj = 1.19
if produceName == 'Garlic':
    cellObj = 3.07
if produceName == 'Lemon':
    cellObj = 1.27
```

Having the produce and updated price data hardcoded like this is a bit inelegant. If you needed to update the spreadsheet again with different prices or different produce, you would have to change a lot of the code. Every time you change code, you risk introducing bugs.

A more flexible solution is to store the corrected price information in a dictionary and write your code to use this data structure. In a new file editor window, enter the following code:

```
#! python3
# updateProduce.py - Corrects costs in produce sales spreadsheet.

import openpyxl

wb = openpyxl.load_workbook('produceSales.xlsx')
sheet = wb.get_sheet_by_name('Sheet')

# The produce types and their updated prices
PRICE_UPDATES = {'Garlic': 3.07,
                 'Celery': 1.19,
                 'Lemon': 1.27}

# TODO: Loop through the rows and update the prices.
```

Save this as *updateProduce.py*. If you need to update the spreadsheet again, you'll need to update only the `PRICE_UPDATES` dictionary, not any other code.

Step 2: Check All Rows and Update Incorrect Prices

The next part of the program will loop through all the rows in the spreadsheet. Add the following code to the bottom of *updateProduce.py*:

```
#! python3
# updateProduce.py - Corrects costs in produce sales spreadsheet.

--snip--

# Loop through the rows and update the prices.
❶ for rowNum in range(2, sheet.get_highest_row()):    # skip the first row
❷     produceName = sheet.cell(row=rowNum, column=1).value
❸     if produceName in PRICE_UPDATES:
          sheet.cell(row=rowNum, column=2).value = PRICE_UPDATES[produceName]

❹ wb.save('updatedProduceSales.xlsx')
```

We loop through the rows starting at row 2, since row 1 is just the header ❶. The cell in column 1 (that is, column A) will be stored in the variable `produceName` ❷. If `produceName` exists as a key in the `PRICE_UPDATES` dictionary ❸, then you know this is a row that must have its price corrected. The correct price will be in `PRICE_UPDATES[produceName]`.

Notice how clean using `PRICE_UPDATES` makes the code. Only one `if` statement, rather than code like `if produceName == 'Garlic':` , is necessary for every type of produce to update. And since the code uses the `PRICE_UPDATES` dictionary instead of hardcoding the produce names and updated costs into the `for` loop, you modify only the `PRICE_UPDATES` dictionary and not the code if the produce sales spreadsheet needs additional changes.

After going through the entire spreadsheet and making changes, the code saves the `Workbook` object to *updatedProduceSales.xlsx* ❹. It doesn't overwrite the old spreadsheet just in case there's a bug in your program and the updated spreadsheet is wrong. After checking that the updated spreadsheet looks right, you can delete the old spreadsheet.

You can download the complete source code for this program from *http://nostarch.com/automatestuff/*.

Ideas for Similar Programs

Since many office workers use Excel spreadsheets all the time, a program that can automatically edit and write Excel files could be really useful. Such a program could do the following:

- Read data from one spreadsheet and write it to parts of other spreadsheets.
- Read data from websites, text files, or the clipboard and write it to a spreadsheet.
- Automatically "clean up" data in spreadsheets. For example, it could use regular expressions to read multiple formats of phone numbers and edit them to a single, standard format.

Setting the Font Style of Cells

Styling certain cells, rows, or columns can help you emphasize important areas in your spreadsheet. In the produce spreadsheet, for example, your program could apply bold text to the potato, garlic, and parsnip rows. Or perhaps you want to italicize every row with a cost per pound greater than $5. Styling parts of a large spreadsheet by hand would be tedious, but your programs can do it instantly.

To customize font styles in cells, important, import the `Font()` and `Style()` functions from the `openpyxl.styles` module.

```
from openpyxl.styles import Font, Style
```

This allows you to type `Font()` instead of `openpyxl.styles.Font()`. (See "Importing Modules" on page 57 to review this style of `import` statement.)

Here's an example that creates a new workbook and sets cell A1 to have a 24-point, italicized font. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> from openpyxl.styles import Font, Style
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_sheet_by_name('Sheet')
❶ >>> italic24Font = Font(size=24, italic=True)
❷ >>> styleObj = Style(font=italic24Font)
❸ >>> sheet['A'].style/styleObj
>>> sheet['A1'] = 'Hello world!'
>>> wb.save('styled.xlsx')
```

OpenPyXL represents the collection of style settings for a cell with a `Style` object, which is stored in the `Cell` object's `style` attribute. A cell's style can be set by assigning the `Style` object to the `style` attribute.

In this example, `Font(size=24, italic=True)` returns a `Font` object, which is stored in `italic24Font` ❶. The keyword arguments to `Font()`, `size` and `italic`, configure the `Font` object's `style` attributes. This `Font` object is then passed into the `Style(font=italic24Font)` call, which returns the value you stored in `styleObj` ❷. And when `styleObj` is assigned to the cell's `style` attribute ❸, all that font styling information gets applied to cell A1.

Font Objects

The `style` attributes in `Font` objects affect how the text in cells is displayed. To set font style attributes, you pass keyword arguments to `Font()`. Table 12-2 shows the possible keyword arguments for the `Font()` function.

Table 12-2: Keyword Arguments for Font `style` Attributes

Keyword argument	Data type	Description
`name`	String	The font name, such as `'Calibri'` or `'Times New Roman'`
`size`	Integer	The point size
`bold`	Boolean	`True`, for bold font
`italic`	Boolean	`True`, for italic font

You can call `Font()` to create a `Font` object and store that `Font` object in a variable. You then pass that to `Style()`, store the resulting `Style` object in a variable, and assign that variable to a `Cell` object's `style` attribute. For example, this code creates various font styles:

```
>>> import openpyxl
>>> from openpyxl.styles import Font, Style
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_sheet_by_name('Sheet')

>>> fontObj1 = Font(name='Times New Roman', bold=True)
>>> styleObj1 = Style(font=fontObj1)
>>> sheet['A1'].style/styleObj
>>> sheet['A1'] = 'Bold Times New Roman'

>>> fontObj2 = Font(size=24, italic=True)
>>> styleObj2 = Style(font=fontObj2)
>>> sheet['B3'].style/styleObj
>>> sheet['B3'] = '24 pt Italic'

>>> wb.save('styles.xlsx')
```

Here, we store a `Font` object in `fontObj1` and use it to create a `Style` object, which we store in `styleObj1`, and then set the A1 `Cell` object's `style` attribute to `styleObj`. We repeat the process with another `Font` object and `Style` object to set the style of a second cell. After you run this code, the styles of the A1 and B3 cells in the spreadsheet will be set to custom font styles, as shown in Figure 12-4.

Figure 12-4: A spreadsheet with custom font styles

For cell A1, we set the font name to `'Times New Roman'` and set `bold` to `True`, so our text appears in bold Times New Roman. We didn't specify a size, so the `openpyxl` default, 11, is used. In cell B3, our text is italic, with a size of 24; we didn't specify a font name, so the `openpyxl` default, Calibri, is used.

Formulas

Formulas, which begin with an equal sign, can configure cells to contain values calculated from other cells. In this section, you'll use the `openpyxl` module to programmatically add formulas to cells, just like any normal value. For example:

```
>>> sheet['B9'] = '=SUM(B1:B8)'
```

This will store *=SUM(B1:B8)* as the value in cell B9. This sets the B9 cell to a formula that calculates the sum of values in cells B1 to B8. You can see this in action in Figure 12-5.

Figure 12-5: Cell B9 contains the formula =SUM(B1:B8)*, which adds the cells B1 to B8.*

A formula is set just like any other text value in a cell. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_active_sheet()
>>> sheet['A1'] = 200
>>> sheet['A2'] = 300
>>> sheet['A3'] = '=SUM(A1:A2)'
>>> wb.save('writeFormula.xlsx')
```

The cells in A1 and A2 are set to 200 and 300, respectively. The value in cell A3 is set to a formula that sums the values in A1 and A2. When the spreadsheet is opened in Excel, A3 will display its value as 500.

You can also read the formula in a cell just as you would any value. However, if you want to see the result of the *calculation* for the formula instead of the literal formula, you must pass `True` for the `data_only` keyword argument to `load_workbook()`. This means a `Workbook` object can show either the formulas or the result of the formulas but not both. (But you can have multiple `Workbook` objects loaded for the same spreadsheet file.) Enter the following into the interactive shell to see the difference between loading a workbook with and without the `data_only` keyword argument:

```
>>> import openpyxl
>>> wbFormulas = openpyxl.load_workbook('writeFormula.xlsx')
>>> sheet = wbFormulas.get_active_sheet()
>>> sheet['A3'].value
'=SUM(A1:A2)'

>>> wbDataOnly = openpyxl.load_workbook('writeFormula.xlsx', data_only=True)
>>> sheet = wbDataOnly.get_active_sheet()
>>> sheet['A3'].value
500
```

Here, when `load_workbook()` is called with `data_only=True`, the A3 cell shows 500, the result of the *=SUM(A1:A2)* formula, rather than the text of the formula.

Excel formulas offer a level of programmability for spreadsheets but can quickly become unmanageable for complicated tasks. For example, even if you're deeply familiar with Excel formulas, it's a headache to try to decipher what *=IFERROR(TRIM(IF(LEN(VLOOKUP(F7, Sheet2!$A$1:$B$10000, 2, FALSE))>0,SUBSTITUTE(VLOOKUP(F7, Sheet2!$A$1:$B$10000, 2, FALSE), " ", ""),""")), "")* actually does. Python code is much more readable.

Adjusting Rows and Columns

In Excel, adjusting the sizes of rows and columns is as easy as clicking and dragging the edges of a row or column header. But if you need to set a row or column's size based on its cells' contents or if you want to set sizes in a large number of spreadsheet files, it will be much quicker to write a Python program to do it.

Rows and columns can also be hidden entirely from view. Or they can be "frozen" in place so that they are always visible on the screen and appear on every page when the spreadsheet is printed (which is handy for headers).

Setting Row Height and Column Width

`Worksheet` objects have `row_dimensions` and `column_dimensions` attributes that control row heights and column widths. Enter this into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_active_sheet()
>>> sheet['A1'] = 'Tall row'
>>> sheet['B2'] = 'Wide column'
>>> sheet.row_dimensions[1].height = 70
>>> sheet.column_dimensions['B'].width = 20
>>> wb.save('dimensions.xlsx')
```

A sheet's `row_dimensions` and `column_dimensions` are dictionary-like values; `row_dimensions` contains `RowDimension` objects and `column_dimensions` contains `ColumnDimension` objects. In `row_dimensions`, you can access one of the objects using the number of the row (in this case, 1 or 2). In `column_dimensions`, you can access one of the objects using the letter of the column (in this case, A or B).

The *dimensions.xlsx* spreadsheet looks like Figure 12-6.

Figure 12-6: Row 1 and column B set to larger heights and widths

Once you have the `RowDimension` object, you can set its height. Once you have the `ColumnDimension` object, you can set its width. The row height can be set to an integer or float value between `0` and `409`. This value represents the height measured in *points*, where one point equals 1/72 of an inch. The default row height is 12.75. The column width can be set to an integer or float value between `0` and `255`. This value represents the number of characters at the default font size (11 point) that can be displayed in the cell. The default column width is 8.43 characters. Columns with widths of `0` or rows with heights of `0` are hidden from the user.

Merging and Unmerging Cells

A rectangular area of cells can be merged into a single cell with the `merge_cells()` sheet method. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
```

```
>>> sheet = wb.get_active_sheet()
>>> sheet.merge_cells('A1:D3')
>>> sheet['A1'] = 'Twelve cells merged together.'
>>> sheet.merge_cells('C5:D5')
>>> sheet['C5'] = 'Two merged cells.'
>>> wb.save('merged.xlsx')
```

The argument to `merge_cells()` is a single string of the top-left and bottom-right cells of the rectangular area to be merged: `'A1:D3'` merges 12 cells into a single cell. To set the value of these merged cells, simply set the value of the top-left cell of the merged group.

When you run this code, *merged.xlsx* will look like Figure 12-7.

Figure 12-7: Merged cells in a spreadsheet

To unmerge cells, call the `unmerge_cells()` sheet method. Enter this into the interactive shell.

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('merged.xlsx')
>>> sheet = wb.get_active_sheet()
>>> sheet.unmerge_cells('A1:D3')
>>> sheet.unmerge_cells('C5:D5')
>>> wb.save('merged.xlsx')
```

If you save your changes and then take a look at the spreadsheet, you'll see that the merged cells have gone back to being individual cells.

Freeze Panes

For spreadsheets too large to be displayed all at once, it's helpful to "freeze" a few of the top rows or leftmost columns onscreen. Frozen column or row headers, for example, are always visible to the user even as they scroll through the spreadsheet. These are known as *freeze panes*. In OpenPyXL, each `Worksheet` object has a `freeze_panes` attribute that can be set to a `Cell` object or a string of a cell's coordinates. Note that all rows above and all columns to the left of this cell will be frozen, but the row and column of the cell itself will not be frozen.

To unfreeze all panes, set `freeze_panes` to `None` or `'A1'`. Table 12-3 shows which rows and columns will be frozen for some example settings of `freeze_panes`.

Table 12-3: Frozen Pane Examples

freeze_panes setting	Rows and columns frozen
`sheet.freeze_panes = 'A2'`	Row 1
`sheet.freeze_panes = 'B1'`	Column A
`sheet.freeze_panes = 'C1'`	Columns A and B
`sheet.freeze_panes = 'C2'`	Row 1 and columns A and B
`sheet.freeze_panes = 'A1'` or `sheet.freeze_panes = None`	No frozen panes

Make sure you have the produce sales spreadsheet from *http://nostarch.com/automatestuff/*. Then enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('produceSales.xlsx')
>>> sheet = wb.get_active_sheet()
>>> sheet.freeze_panes = 'A2'
>>> wb.save('freezeExample.xlsx')
```

If you set the `freeze_panes` attribute to `'A2'`, row 1 will always be viewable, no matter where the user scrolls in the spreadsheet. You can see this in Figure 12-8.

Figure 12-8: With `freeze_panes` set to `'A2'`, row 1 is always visible even as the user scrolls down.

Charts

OpenPyXL supports creating bar, line, scatter, and pie charts using the data in a sheet's cells. To make a chart, you need to do the following:

1. Create a `Reference` object from a rectangular selection of cells.
2. Create a `Series` object by passing in the `Reference` object.
3. Create a `Chart` object.

4. Append the `Series` object to the `Chart` object.
5. Optionally, set the `drawing.top`, `drawing.left`, `drawing.width`, and `drawing.height` variables of the `Chart` object.
6. Add the `Chart` object to the `Worksheet` object.

The `Reference` object requires some explaining. `Reference` objects are created by calling the `openpyxl.charts.Reference()` function and passing three arguments:

1. The `Worksheet` object containing your chart data.
2. A tuple of two integers, representing the top-left cell of the rectangular selection of cells containing your chart data: The first integer in the tuple is the row, and the second is the column. Note that `1` is the first row, not `0`.
3. A tuple of two integers, representing the bottom-right cell of the rectangular selection of cells containing your chart data: The first integer in the tuple is the row, and the second is the column.

Figure 12-9 shows some sample coordinate arguments.

Figure 12-9: From left to right: (1, 1), (10, 1); (3, 2), (6, 4); (5, 3), (5, 3)

Enter this interactive shell example to create a bar chart and add it to the spreadsheet:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_active_sheet()
>>> for i in range(1, 11):          # create some data in column A
        sheet['A' + str(i)] = i

>>> refObj = openpyxl.charts.Reference(sheet, (1, 1), (10, 1))

>>> seriesObj = openpyxl.charts.Series(refObj, title='First series')

>>> chartObj = openpyxl.charts.BarChart()
>>> chartObj.append(seriesObj)
```

```
>>> chartObj.drawing.top = 50       # set the position
>>> chartObj.drawing.left = 100
>>> chartObj.drawing.width = 300    # set the size
>>> chartObj.drawing.height = 200

>>> sheet.add_chart(chartObj)
>>> wb.save('sampleChart.xlsx')
```

This produces a spreadsheet that looks like Figure 12-10.

Figure 12-10: A spreadsheet with a chart added

We've created a bar chart by calling `openpyxl.charts.BarChart()`. You can also create line charts, scatter charts, and pie charts by calling `openpyxl.charts.LineChart()`, `openpyxl.charts.ScatterChart()`, and `openpyxl.charts.PieChart()`.

Unfortunately, in the current version of OpenPyXL (2.1.4), the `load_workbook()` function does not load charts in Excel files. Even if the Excel file has charts, the loaded `Workbook` object will not include them. If you load a `Workbook` object and immediately save it to the same *.xlsx* filename, you will effectively remove the charts from it.

Summary

Often the hard part of processing information isn't the processing itself but simply getting the data in the right format for your program. But once you have your spreadsheet loaded into Python, you can extract and manipulate its data much faster than you could by hand.

You can also generate spreadsheets as output from your programs. So if colleagues need your text file or PDF of thousands of sales contacts transferred to a spreadsheet file, you won't have to tediously copy and paste it all into Excel.

Equipped with the `openpyxl` module and some programming knowledge, you'll find processing even the biggest spreadsheets a piece of cake.

Practice Questions

For the following questions, imagine you have a `Workbook` object in the variable `wb`, a `Worksheet` object in `sheet`, a `Cell` object in `cell`, a `Comment` object in `comm`, and an `Image` object in `img`.

1. What does the `openpyxl.load_workbook()` function return?
2. What does the `get_sheet_names()` workbook method return?
3. How would you retrieve the `Worksheet` object for a sheet named `'Sheet1'`?
4. How would you retrieve the `Worksheet` object for the workbook's active sheet?
5. How would you retrieve the value in the cell C5?
6. How would you set the value in the cell C5 to `"Hello"`?
7. How would you retrieve the cell's row and column as integers?
8. What do the `get_highest_column()` and `get_highest_row()` sheet methods return, and what is the data type of these return values?
9. If you needed to get the integer index for column `'M'`, what function would you need to call?
10. If you needed to get the string name for column `14`, what function would you need to call?
11. How can you retrieve a tuple of all the `Cell` objects from A1 to F1?
12. How would you save the workbook to the filename *example.xlsx*?
13. How do you set a formula in a cell?
14. If you want to retrieve the result of a cell's formula instead of the cell's formula itself, what must you do first?
15. How would you set the height of row 5 to 100?
16. How would you hide column C?
17. Name a few features that OpenPyXL 2.1.4 does not load from a spreadsheet file.
18. What is a freeze pane?
19. What five functions and methods do you have to call to create a bar chart?

Practice Projects

For practice, write programs that perform the following tasks.

Multiplication Table Maker

Create a program *multiplicationTable.py* that takes a number *N* from the command line and creates an *N*×*N* multiplication table in an Excel spreadsheet. For example, when the program is run like this:

```
py multiplicationTable.py 6
```

. . . it should create a spreadsheet that looks like Figure 12-11.

	A	B	C	D	E	F	G	H
1		**1**	**2**	**3**	**4**	**5**	**6**	
2	**1**	1	2	3	4	5	6	
3	**2**	2	4	6	8	10	12	
4	**3**	3	6	9	12	15	18	
5	**4**	4	8	12	16	20	24	
6	**5**	5	10	15	20	25	30	
7	**6**	6	12	18	24	30	36	
8								
9								

Figure 12-11: A multiplication table generated in a spreadsheet

Row 1 and column A should be used for labels and should be in bold.

Blank Row Inserter

Create a program *blankRowInserter.py* that takes two integers and a filename string as command line arguments. Let's call the first integer *N* and the second integer *M*. Starting at row *N*, the program should insert *M* blank rows into the spreadsheet. For example, when the program is run like this:

```
python blankRowInserter.py 3 2 myProduce.xlsx
```

. . . the "before" and "after" spreadsheets should look like Figure 12-12.

A1 fx Potatoes

	A	B	C	D	E	F
1	Potatoes	Celery	Ginger	Yellow pep	Green bea	Fava be
2	Okra	Okra	Corn	Garlic	Tomatoes	Yellow
3	Fava bean	Spinach	Grapefruit	Grapes	Apricots	Papaya
4	Watermel	Cucumber	Ginger	Watermel	Red onion	Butterr
5	Garlic	Apricots	Eggplant	Cherries	Strawberri	Apricot
6	Parsnips	Okra	Cucumber	Apples	Grapes	Avocac
7	Asparagus	Fava bean	Green cabl	Grapefruit	Ginger	Butterr
8	Avocados	Watermel	Eggplant	Grapes	Strawberri	Celery

A1 fx Potatoes

	A	B	C	D	E	F
1	Potatoes	Celery	Ginger	Yellow pep	Green bea	Fava be
2	Okra	Okra	Corn	Garlic	Tomatoes	Yellow
3						
4						
5	Fava bean	Spinach	Grapefruit	Grapes	Apricots	Papaya
6	Watermel	Cucumber	Ginger	Watermel	Red onion	Butterr
7	Garlic	Apricots	Eggplant	Cherries	Strawberri	Apricot
8	Parsnips	Okra	Cucumber	Apples	Grapes	Avocac

Figure 12-12: Before (left) and after (right) the two blank rows are inserted at row 3

You can write this program by reading in the contents of the spreadsheet. Then, when writing out the new spreadsheet, use a `for` loop to copy the first *N* lines. For the remaining lines, add *M* to the row number in the output spreadsheet.

Spreadsheet Cell Inverter

Write a program to invert the row and column of the cells in the spreadsheet. For example, the value at row 5, column 3 will be at row 3, column 5 (and vice versa). This should be done for all cells in the spreadsheet. For example, the "before" and "after" spreadsheets would look something like Figure 12-13.

A1 ITEM

	A	B	C	D	E	F	G	H	I	J
1	ITEM	SOLD								
2	Eggplant	334								
3	Cucumber	252								
4	Green cabl	238								
5	Eggplant	516								
6	Garlic	98								
7	Parsnips	16								
8	Asparagus	335								
9	Avocados	84								
10										

A1 ITEM

	A	B	C	D	E	F	G	H	I	J
1	ITEM	Eggplant	Cucumber	Green cabl	Eggplant	Garlic	Parsnips	Asparagus	Avocados	
2	SOLD	334	252	238	516	98	16	335	84	
3										
4										
5										
6										
7										
8										
9										
10										

Figure 12-13: The spreadsheet before (top) and after (bottom) inversion

You can write this program by using nested `for` loops to read in the spreadsheet's data into a list of lists data structure. This data structure could have `sheetData[x][y]` for the cell at column `x` and row `y`. Then, when writing out the new spreadsheet, use `sheetData[y][x]` for the cell at column `x` and row `y`.

Text Files to Spreadsheet

Write a program to read in the contents of several text files (you can make the text files yourself) and insert those contents into a spreadsheet, with one line of text per row. The lines of the first text file will be in the cells of column A, the lines of the second text file will be in the cells of column B, and so on.

Use the `readlines()` `File` object method to return a list of strings, one string per line in the file. For the first file, output the first line to column 1, row 1. The second line should be written to column 1, row 2, and so on. The next file that is read with `readlines()` will be written to column 2, the next file to column 3, and so on.

Spreadsheet to Text Files

Write a program that performs the tasks of the previous program in reverse order: The program should open a spreadsheet and write the cells of column A into one text file, the cells of column B into another text file, and so on.

13

WORKING WITH PDF AND WORD DOCUMENTS

PDF and Word documents are binary files, which makes them much more complex than plaintext files. In addition to text, they store lots of font, color, and layout information. If you want your programs to read or write to PDFs or Word documents, you'll need to do more than simply pass their filenames to `open()`.

Fortunately, there are Python modules that make it easy for you to interact with PDFs and Word documents. This chapter will cover two such modules: PyPDF2 and Python-Docx.

PDF Documents

PDF stands for *Portable Document Format* and uses the *.pdf* file extension. Although PDFs support many features, this chapter will focus on the two things you'll be doing most often with them: reading text content from PDFs and crafting new PDFs from existing documents.

The module you'll use to work with PDFs is PyPDF2. To install it, run `pip install PyPDF2` from the command line. This module name is case sensitive, so make sure the *y* is lowercase and everything else is uppercase. (Check out Appendix A for full details about installing third-party modules.) If the module was installed correctly, running `import PyPDF2` in the interactive shell shouldn't display any errors.

THE PROBLEMATIC PDF FORMAT

While PDF files are great for laying out text in a way that's easy for people to print and read, they're not straightforward for software to parse into plaintext. As such, PyPDF2 might make mistakes when extracting text from a PDF and may even be unable to open some PDFs at all. There isn't much you can do about this, unfortunately. PyPDF2 may simply be unable to work with some of your particular PDF files. That said, I haven't found any PDF files so far that can't be opened with PyPDF2.

Extracting Text from PDFs

PyPDF2 does not have a way to extract images, charts, or other media from PDF documents, but it can extract text and return it as a Python string. To start learning how PyPDF2 works, we'll use it on the example PDF shown in Figure 13-1.

Figure 13-1: The PDF page that we will be extracting text from

Download this PDF from *http://nostarch.com/automatestuff/*, and enter the following into the interactive shell:

```
>>> import PyPDF2
>>> pdfFileObj = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
❶ >>> pdfReader.numPages
19
❷ >>> pageObj = pdfReader.getPage(0)
❸ >>> pageObj.extractText()
'OOFFFFIICCIIAALL  BBOOAARRDD  MMIINNUUTTEESS   Meeting of March 7, 2015
\n     The Board of Elementary and Secondary Education shall provide leadership
and create policies for education that expand opportunities for children,
empower families and communities, and advance Louisiana in an increasingly
competitive global market. BOARD  of ELEMENTARY and  SECONDARY EDUCATION  '
```

First, import the `PyPDF2` module. Then open *meetingminutes.pdf* in read binary mode and store it in `pdfFileObj`. To get a `PdfFileReader` object that represents this PDF, call `PyPDF2.PdfFileReader()` and pass it `pdfFileObj`. Store this `PdfFileReader` object in `pdfReader`.

The total number of pages in the document is stored in the `numPages` attribute of a `PdfFileReader` object ❶. The example PDF has 19 pages, but let's extract text from only the first page.

To extract text from a page, you need to get a `Page` object, which represents a single page of a PDF, from a `PdfFileReader` object. You can get a `Page` object by calling the `getPage()` method ❷ on a `PdfFileReader` object and passing it the page number of the page you're interested in—in our case, 0.

PyPDF2 uses a *zero-based index* for getting pages: The first page is page 0, the second is page 1, and so on. This is always the case, even if pages are numbered differently within the document. For example, say your PDF is a three-page excerpt from a longer report, and its pages are numbered 42, 43, and 44. To get the first page of this document, you would want to call `pdfReader.getPage(0)`, not `getPage(42)` or `getPage(1)`.

Once you have your `Page` object, call its `extractText()` method to return a string of the page's text ❸. The text extraction isn't perfect: The text *Charles E. "Chas" Roemer, President* from the PDF is absent from the string returned by `extractText()`, and the spacing is sometimes off. Still, this approximation of the PDF text content may be good enough for your program.

Decrypting PDFs

Some PDF documents have an encryption feature that will keep them from being read until whoever is opening the document provides a password. Enter the following into the interactive shell with the PDF you downloaded, which has been encrypted with the password *rosebud*:

```
>>> import PyPDF2
>>> pdfReader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb'))
❶ >>> pdfReader.isEncrypted
True
```

```
>>> pdfReader.getPage(0)
❷ Traceback (most recent call last):
  File "<pyshell#173>", line 1, in <module>
    pdfReader.getPage()
  --snip--
  File "C:\Python34\lib\site-packages\PyPDF2\pdf.py", line 1173, in getObject
    raise utils.PdfReadError("file has not been decrypted")
PyPDF2.utils.PdfReadError: file has not been decrypted
❸ >>> pdfReader.decrypt('rosebud')
1
>>> pageObj = pdfReader.getPage(0)
```

All `PdfFileReader` objects have an `isEncrypted` attribute that is `True` if the PDF is encrypted and `False` if it isn't ❶. Any attempt to call a function that reads the file before it has been decrypted with the correct password will result in an error ❷.

To read an encrypted PDF, call the `decrypt()` function and pass the password as a string ❸. After you call `decrypt()` with the correct password, you'll see that calling `getPage()` no longer causes an error. If given the wrong password, the `decrypt()` function will return `0` and `getPage()` will continue to fail. Note that the `decrypt()` method decrypts only the `PdfFileReader` object, not the actual PDF file. After your program terminates, the file on your hard drive remains encrypted. Your program will have to call `decrypt()` again the next time it is run.

Creating PDFs

PyPDF2's counterpart to `PdfFileReader` objects is `PdfFileWriter` objects, which can create new PDF files. But PyPDF2 cannot write arbitrary text to a PDF like Python can do with plaintext files. Instead, PyPDF2's PDF-writing capabilities are limited to copying pages from other PDFs, rotating pages, overlaying pages, and encrypting files.

PyPDF2 doesn't allow you to directly edit a PDF. Instead, you have to create a new PDF and then copy content over from an existing document. The examples in this section will follow this general approach:

1. Open one or more existing PDFs (the source PDFs) into `PdfFileReader` objects.
2. Create a new `PdfFileWriter` object.
3. Copy pages from the `PdfFileReader` objects into the `PdfFileWriter` object.
4. Finally, use the `PdfFileWriter` object to write the output PDF.

Creating a `PdfFileWriter` object creates only a value that represents a PDF document in Python. It doesn't create the actual PDF file. For that, you must call the PdfFileWriter's `write()` method.

The `write()` method takes a regular `File` object that has been opened in *write-binary* mode. You can get such a `File` object by calling Python's `open()` function with two arguments: the string of what you want the PDF's filename to be and `'wb'` to indicate the file should be opened in write-binary mode.

If this sounds a little confusing, don't worry—you'll see how this works in the following code examples.

Copying Pages

You can use PyPDF2 to copy pages from one PDF document to another. This allows you to combine multiple PDF files, cut unwanted pages, or reorder pages.

Download *meetingminutes.pdf* and *meetingminutes2.pdf* from *http://nostarch.com/automatestuff/* and place the PDFs in the current working directory. Enter the following into the interactive shell:

```
  >>> import PyPDF2
  >>> pdf1File = open('meetingminutes.pdf', 'rb')
  >>> pdf2File = open('meetingminutes2.pdf', 'rb')
❶ >>> pdf1Reader = PyPDF2.PdfFileReader(pdf1File)
❷ >>> pdf2Reader = PyPDF2.PdfFileReader(pdf2File)
❸ >>> pdfWriter = PyPDF2.PdfFileWriter()

  >>> for pageNum in range(pdf1Reader.numPages):
❹         pageObj = pdf1Reader.getPage(pageNum)
❺         pdfWriter.addPage(pageObj)

  >>> for pageNum in range(pdf2Reader.numPages):
❹         pageObj = pdf2Reader.getPage(pageNum)
❺         pdfWriter.addPage(pageObj)

❻ >>> pdfOutputFile = open('combinedminutes.pdf', 'wb')
  >>> pdfWriter.write(pdfOutputFile)
  >>> pdfOutputFile.close()
  >>> pdf1File.close()
  >>> pdf2File.close()
```

Open both PDF files in read binary mode and store the two resulting `File` objects in `pdf1File` and `pdf2File`. Call `PyPDF2.PdfFileReader()` and pass it `pdf1File` to get a `PdfFileReader` object for *meetingminutes.pdf* ❶. Call it again and pass it `pdf2File` to get a `PdfFileReader` object for *meetingminutes2.pdf* ❷. Then create a new `PdfFileWriter` object, which represents a blank PDF document ❸.

Next, copy all the pages from the two source PDFs and add them to the `PdfFileWriter` object. Get the `Page` object by calling `getPage()` on a `PdfFileReader` object ❹. Then pass that `Page` object to your PdfFileWriter's `addPage()` method ❺. These steps are done first for `pdf1Reader` and then

again for `pdf2Reader`. When you're done copying pages, write a new PDF called *combinedminutes.pdf* by passing a `File` object to the PdfFileWriter's `write()` method ❻.

NOTE *PyPDF2 cannot insert pages in the middle of a `PdfFileWriter` object; the `addPage()` method will only add pages to the end.*

You have now created a new PDF file that combines the pages from *meetingminutes.pdf* and *meetingminutes2.pdf* into a single document. Remember that the `File` object passed to `PyPDF2.PdfFileReader()` needs to be opened in read-binary mode by passing `'rb'` as the second argument to `open()`. Likewise, the `File` object passed to `PyPDF2.PdfFileWriter()` needs to be opened in write-binary mode with `'wb'`.

Rotating Pages

The pages of a PDF can also be rotated in 90-degree increments with the `rotateClockwise()` and `rotateCounterClockwise()` methods. Pass one of the integers `90`, `180`, or `270` to these methods. Enter the following into the interactive shell, with the *meetingminutes.pdf* file in the current working directory:

```
>>> import PyPDF2
>>> minutesFile = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
❶ >>> page = pdfReader.getPage(0)
❷ >>> page.rotateClockwise(90)
{'/Contents': [IndirectObject(961, 0), IndirectObject(962, 0),
--snip--
}
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> pdfWriter.addPage(page)
❸ >>> resultPdfFile = open('rotatedPage.pdf', 'wb')
>>> pdfWriter.write(resultPdfFile)
>>> resultPdfFile.close()
>>> minutesFile.close()
```

Here we use `getPage(0)` to select the first page of the PDF ❶, and then we call `rotateClockwise(90)` on that page ❷. We write a new PDF with the rotated page and save it as *rotatedPage.pdf* ❸.

The resulting PDF will have one page, rotated 90 degrees clockwise, as in Figure 13-2. The return values from `rotateClockwise()` and `rotateCounterClockwise()` contain a lot of information that you can ignore.

Figure 13-2: The rotatedPage.pdf *file with the page rotated 90 degrees clockwise*

Overlaying Pages

PyPDF2 can also overlay the contents of one page over another, which is useful for adding a logo, timestamp, or watermark to a page. With Python, it's easy to add watermarks to multiple files and only to pages your program specifies.

Download *watermark.pdf* from *http://nostarch.com/automatestuff/* and place the PDF in the current working directory along with *meetingminutes.pdf*. Then enter the following into the interactive shell:

```
  >>> import PyPDF2
  >>> minutesFile = open('meetingminutes.pdf', 'rb')
❶ >>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
❷ >>> minutesFirstPage = pdfReader.getPage(0)
❸ >>> pdfWatermarkReader = PyPDF2.PdfFileReader(open('watermark.pdf', 'rb'))
❹ >>> minutesFirstPage.mergePage(pdfWatermarkReader.getPage(0))
❺ >>> pdfWriter = PyPDF2.PdfFileWriter()
❻ >>> pdfWriter.addPage(minutesFirstPage)

❼ >>> for pageNum in range(1, pdfReader.numPages):
          pageObj = pdfReader.getPage(pageNum)
          pdfWriter.addPage(pageObj)
```

```
>>> resultPdfFile = open('watermarkedCover.pdf', 'wb')
>>> pdfWriter.write(resultPdfFile)
>>> minutesFile.close()
>>> resultPdfFile.close()
```

Here we make a `PdfFileReader` object of *meetingminutes.pdf* ❶. We call `getPage(0)` to get a `Page` object for the first page and store this object in `minutesFirstPage` ❷. We then make a `PdfFileReader` object for *watermark.pdf* ❸ and call `mergePage()` on `minutesFirstPage` ❹. The argument we pass to `mergePage()` is a `Page` object for the first page of *watermark.pdf*.

Now that we've called `mergePage()` on `minutesFirstPage`, `minutesFirstPage` represents the watermarked first page. We make a `PdfFileWriter` object ❺ and add the watermarked first page ❻. Then we loop through the rest of the pages in *meetingminutes.pdf* and add them to the `PdfFileWriter` object ❼. Finally, we open a new PDF called *watermarkedCover.pdf* and write the contents of the PdfFileWriter to the new PDF.

Figure 13-3 shows the results. Our new PDF, *watermarkedCover.pdf*, has all the contents of the *meetingminutes.pdf*, and the first page is watermarked.

Figure 13-3: The original PDF (left), the watermark PDF (center), and the merged PDF (right)

Encrypting PDFs

A `PdfFileWriter` object can also add encryption to a PDF document. Enter the following into the interactive shell:

```
>>> import PyPDF2
>>> pdfFile = open('meetingminutes.pdf', 'rb')
>>> pdfReader = PyPDF2.PdfFileReader(pdfFile)
>>> pdfWriter = PyPDF2.PdfFileWriter()
>>> for pageNum in range(pdfReader.numPages):
        pdfWriter.addPage(pdfReader.getPage(pageNum))

❶ >>> pdfWriter.encrypt('swordfish')
>>> resultPdf = open('encryptedminutes.pdf', 'wb')
>>> pdfWriter.write(resultPdf)
>>> resultPdf.close()
```

Before calling the `write()` method to save to a file, call the `encrypt()` method and pass it a password string ❶. PDFs can have a *user password* (allowing you to view the PDF) and an *owner password* (allowing you to set permissions for printing, commenting, extracting text, and other features). The user password and owner password are the first and second arguments to `encrypt()`, respectively. If only one string argument is passed to `encrypt()`, it will be used for both passwords.

In this example, we copied the pages of *meetingminutes.pdf* to a `PdfFileWriter` object. We encrypted the PdfFileWriter with the password *swordfish*, opened a new PDF called *encryptedminutes.pdf*, and wrote the contents of the PdfFileWriter to the new PDF. Before anyone can view *encryptedminutes.pdf*, they'll have to enter this password. You may want to delete the original, unencrypted *meetingminutes.pdf* file after ensuring its copy was correctly encrypted.

Project: Combining Select Pages from Many PDFs

Say you have the boring job of merging several dozen PDF documents into a single PDF file. Each of them has a cover sheet as the first page, but you don't want the cover sheet repeated in the final result. Even though there are lots of free programs for combining PDFs, many of them simply merge entire files together. Let's write a Python program to customize which pages you want in the combined PDF.

At a high level, here's what the program will do:

- Find all PDF files in the current working directory.
- Sort the filenames so the PDFs are added in order.
- Write each page, excluding the first page, of each PDF to the output file.

In terms of implementation, your code will need to do the following:

- Call `os.listdir()` to find all the files in the working directory and remove any non-PDF files.
- Call Python's `sort()` list method to alphabetize the filenames.
- Create a `PdfFileWriter` object for the output PDF.
- Loop over each PDF file, creating a `PdfFileReader` object for it.
- Loop over each page (except the first) in each PDF file.
- Add the pages to the output PDF.
- Write the output PDF to a file named *allminutes.pdf*.

For this project, open a new file editor window and save it as *combinePdfs.py*.

Step 1: Find All PDF Files

First, your program needs to get a list of all files with the *.pdf* extension in the current working directory and sort them. Make your code look like the following:

```
#! python3
# combinePdfs.py - Combines all the PDFs in the current working directory into
# into a single PDF.

❶ import PyPDF2, os

# Get all the PDF filenames.
pdfFiles = []
for filename in os.listdir('.'):
    if filename.endswith('.pdf'):
❷       pdfFiles.append(filename)
❸ pdfFiles.sort(key/str.lower)

❹ pdfWriter = PyPDF2.PdfFileWriter()

# TODO: Loop through all the PDF files.

# TODO: Loop through all the pages (except the first) and add them.

# TODO: Save the resulting PDF to a file.
```

After the shebang line and the descriptive comment about what the program does, this code imports the `os` and `PyPDF2` modules ❶. The `os.listdir('.')` call will return a list of every file in the current working directory. The code loops over this list and adds only those files with the *.pdf* extension to `pdfFiles` ❷. Afterward, this list is sorted in alphabetical order with the `key/str.lower` keyword argument to `sort()` ❸.

A `PdfFileWriter` object is created to hold the combined PDF pages ❹. Finally, a few comments outline the rest of the program.

Step 2: Open Each PDF

Now the program must read each PDF file in `pdfFiles`. Add the following to your program:

```
#! python3
# combinePdfs.py - Combines all the PDFs in the current working directory into
# a single PDF.

import PyPDF2, os

# Get all the PDF filenames.
pdfFiles = []
--snip--
```

```
# Loop through all the PDF files.
for filename in pdfFiles:
    pdfFileObj = open(filename, 'rb')
    pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
    # TODO: Loop through all the pages (except the first) and add them.

# TODO: Save the resulting PDF to a file.
```

For each PDF, the loop opens a filename in read-binary mode by calling `open()` with `'rb'` as the second argument. The `open()` call returns a `File` object, which gets passed to `PyPDF2.PdfFileReader()` to create a `PdfFileReader` object for that PDF file.

Step 3: Add Each Page

For each PDF, you'll want to loop over every page except the first. Add this code to your program:

```
#! python3
# combinePdfs.py - Combines all the PDFs in the current working directory into
# a single PDF.

import PyPDF2, os

--snip--

# Loop through all the PDF files.
for filename in pdfFiles:
--snip--
    # Loop through all the pages (except the first) and add them.
❶   for pageNum in range(1, pdfReader.numPages):
        pageObj = pdfReader.getPage(pageNum)
        pdfWriter.addPage(pageObj)

# TODO: Save the resulting PDF to a file.
```

The code inside the `for` loop copies each `Page` object individually to the `PdfFileWriter` object. Remember, you want to skip the first page. Since PyPDF2 considers `0` to be the first page, your loop should start at `1` ❶ and then go up to, but not include, the integer in `pdfReader.numPages`.

Step 4: Save the Results

After these nested `for` loops are done looping, the `pdfWriter` variable will contain a `PdfFileWriter` object with the pages for all the PDFs combined. The last step is to write this content to a file on the hard drive. Add this code to your program:

```
#! python3
# combinePdfs.py - Combines all the PDFs in the current working directory into
# a single PDF.
```

```
import PyPDF2, os

--snip--

# Loop through all the PDF files.
for filename in pdfFiles:
--snip--
    # Loop through all the pages (except the first) and add them.
    for pageNum in range(1, pdfReader.numPages):
    --snip--

# Save the resulting PDF to a file.
pdfOutput = open('allminutes.pdf', 'wb')
pdfWriter.write(pdfOutput)
pdfOutput.close()
```

Passing `'wb'` to `open()` opens the output PDF file, *allminutes.pdf*, in write-binary mode. Then, passing the resulting `File` object to the `write()` method creates the actual PDF file. A call to the `close()` method finishes the program.

Ideas for Similar Programs

Being able to create PDFs from the pages of other PDFs will let you make programs that can do the following:

- Cut out specific pages from PDFs.
- Reorder pages in a PDF.
- Create a PDF from only those pages that have some specific text, identified by `extractText()`.

Word Documents

Python can create and modify Word documents, which have the *.docx* file extension, with the `python-docx` module. You can install the module by running `pip install python-docx`. (Appendix A has full details on installing third-party modules.)

NOTE *When using pip to first install Python-Docx, be sure to install* `python-docx`*, not* `docx`*. The installation name* `docx` *is for a different module that this book does not cover. However, when you are going to import the* `python-docx` *module, you'll need to run* `import docx`*, not* `import python-docx`*.*

If you don't have Word, LibreOffice Writer and OpenOffice Writer are both free alternative applications for Windows, OS X, and Linux that can be used to open *.docx* files. You can download them from *https://www.libreoffice.org* and *http://openoffice.org*, respectively. The full documentation for Python-Docx is available at *https://python-docx.readthedocs.org/*. Although there is a version of Word for OS X, this chapter will focus on Word for Windows.

Compared to plaintext, *.docx* files have a lot of structure. This structure is represented by three different data types in Python-Docx. At the highest level, a `Document` object represents the entire document. The `Document` object contains a list of `Paragraph` objects for the paragraphs in the document. (A new paragraph begins whenever the user presses ENTER or RETURN while typing in a Word document.) Each of these `Paragraph` objects contains a list of one or more `Run` objects. The single-sentence paragraph in Figure 13-4 has four runs.

Figure 13-4: The `Run` objects identified in a `Paragraph` object

The text in a Word document is more than just a string. It has font, size, color, and other styling information associated with it. A *style* in Word is a collection of these attributes. A `Run` object is a contiguous run of text with the same style. A new `Run` object is needed whenever the text style changes.

Reading Word Documents

Let's experiment with the `python-docx` module. Download *demo.docx* from *http://nostarch.com/automatestuff/* and save the document to the working directory. Then enter the following into the interactive shell:

```
>>> import docx
❶ >>> doc = docx.Document('demo.docx')
❷ >>> len(doc.paragraphs)
7
❸ >>> doc.paragraphs[0].text
'Document Title'
❹ >>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
❺ >>> len(doc.paragraphs[1].runs)
4
❻ >>> doc.paragraphs[1].runs[0].text
'A plain paragraph with some '
❼ >>> doc.paragraphs[1].runs[1].text
'bold'
❽ >>> doc.paragraphs[1].runs[2].text
' and some '
❾ >>> doc.paragraphs[1].runs[3].text
'italic'
```

At ❶, we open a *.docx* file in Python, call `docx.Document()`, and pass the filename *demo.docx*. This will return a `Document` object, which has a `paragraphs` attribute that is a list of `Paragraph` objects. When we call `len()` on `doc.paragraphs`, it returns `7`, which tells us that there are seven `Paragraph` objects in this document ❷. Each of these `Paragraph` objects has a `text`

attribute that contains a string of the text in that paragraph (without the style information). Here, the first `text` attribute contains `'DocumentTitle'` ❸, and the second contains `'A plain paragraph with some bold and some italic'` ❹.

Each `Paragraph` object also has a `runs` attribute that is a list of `Run` objects. `Run` objects also have a `text` attribute, containing just the text in that particular run. Let's look at the `text` attributes in the second `Paragraph` object, `'A plain paragraph with some bold and some italic'`. Calling `len()` on this `Paragraph` object tells us that there are four `Run` objects ❺. The first run object contains `'A plain paragraph with some '` ❻. Then, the text change to a bold style, so `'bold'` starts a new `Run` object ❼. The text returns to an unbolded style after that, which results in a third `Run` object, `' and some '` ❽. Finally, the fourth and last `Run` object contains `'italic'` in an italic style ❾.

With Python-Docx, your Python programs will now be able to read the text from a *.docx* file and use it just like any other string value.

Getting the Full Text from a .docx File

If you care only about the text, not the styling information, in the Word document, you can use the `getText()` function. It accepts a filename of a *.docx* file and returns a single string value of its text. Open a new file editor window and enter the following code, saving it as *readDocx.py*:

```
#! python3

import docx

def getText(filename):
    doc = docx.Document(filename)
    fullText = []
    for para in doc.paragraphs:
        fullText.append(para.text)
    return '\n'.join(fullText)
```

The `getText()` function opens the Word document, loops over all the `Paragraph` objects in the `paragraphs` list, and then appends their text to the list in `fullText`. After the loop, the strings in `fullText` are joined together with newline characters.

The *readDocx.py* program can be imported like any other module. Now if you just need the text from a Word document, you can enter the following:

```
>>> import readDocx
>>> print(readDocx.getText('demo.docx'))
Document Title
A plain paragraph with some bold and some italic
Heading, level 1
Intense quote
first item in unordered list
first item in ordered list
```

You can also adjust `getText()` to modify the string before returning it. For example, to indent each paragraph, replace the `append()` call in *readDocx.py* with this:

```
fullText.append('  ' + para.text)
```

To add a double space in between paragraphs, change the `join()` call code to this:

```
return '\n\n'.join(fullText)
```

As you can see, it takes only a few lines of code to write functions that will read a *.docx* file and return a string of its content to your liking.

Styling Paragraph and Run Objects

In Word for Windows, you can see the styles by pressing CTRL-ALT-SHIFT-S to display the Styles pane, which looks like Figure 13-5. On OS X, you can view the Styles pane by clicking the **View ▸ Styles** menu item.

Figure 13-5: Display the Styles pane by pressing CTRL-ALT-SHIFT-S on Windows.

Word and other word processors use styles to keep the visual presentation of similar types of text consistent and easy to change. For example, perhaps you want to set body paragraphs in 11-point, Times New Roman, left-justified, ragged-right text. You can create a style with these settings and

assign it to all body paragraphs. Then, if you later want to change the presentation of all body paragraphs in the document, you can just change the style, and all those paragraphs will be automatically updated.

For Word documents, there are three types of styles: *Paragraph styles* can be applied to `Paragraph` objects, *character styles* can be applied to `Run` objects, and *linked styles* can be applied to both kinds of objects. You can give both `Paragraph` and `Run` objects styles by setting their `style` attribute to a string. This string should be the name of a style. If `style` is set to `None`, then there will be no style associated with the `Paragraph` or `Run` object.

The string values for the default Word styles are as follows:

```
'Normal'          'Heading5'        'ListBullet'       'ListParagraph'
'BodyText'        'Heading6'        'ListBullet2'      'MacroText'
'BodyText2'       'Heading7'        'ListBullet3'      'NoSpacing'
'BodyText3'       'Heading8'        'ListContinue'     'Quote'
'Caption'         'Heading9'        'ListContinue2'    'Subtitle'
'Heading1'        'IntenseQuote'    'ListContinue3'    'TOCHeading'
'Heading2'        'List'            'ListNumber'       'Title'
'Heading3'        'List2'           'ListNumber2'
'Heading4'        'List3'           'ListNumber3'
```

When setting the `style` attribute, do not use spaces in the style name. For example, while the style name may be Subtle Emphasis, you should set the `style` attribute to the string value `'SubtleEmphasis'` instead of `'Subtle Emphasis'`. Including spaces will cause Word to misread the style name and not apply it.

When using a linked style for a `Run` object, you will need to add `'Char'` to the end of its name. For example, to set the Quote linked style for a `Paragraph` object, you would use `paragraphObj.style = 'Quote'`, but for a `Run` object, you would use `runObj.style = 'QuoteChar'`.

In the current version of Python-Docx (0.7.4), the only styles that can be used are the default Word styles and the styles in the opened *.docx*. New styles cannot be created—though this may change in future versions of Python-Docx.

Creating Word Documents with Nondefault Styles

If you want to create Word documents that use styles beyond the default ones, you will need to open Word to a blank Word document and create the styles yourself by clicking the **New Style** button at the bottom of the Styles pane (Figure 13-6 shows this on Windows).

This will open the Create New Style from Formatting dialog, where you can enter the new style. Then, go back into the interactive shell and open this blank Word document with `docx.Document()`, using it as the base for your Word document. The name you gave this style will now be available to use with Python-Docx.

Figure 13-6: The New Style button (left) and the Create New Style from Formatting dialog (right)

Run Attributes

Runs can be further styled using text attributes. Each attribute can be set to one of three values: `True` (the attribute is always enabled, no matter what other styles are applied to the run), `False` (the attribute is always disabled), or `None` (defaults to whatever the run's style is set to).

Table 13-1 lists the text attributes that can be set on `Run` objects.

Table 13-1: `Run` Object text Attributes

Attribute	Description
`bold`	The text appears in bold.
`italic`	The text appears in italic.
`underline`	The text is underlined.
`strike`	The text appears with strikethrough.
`double_strike`	The text appears with double strikethrough.
`all_caps`	The text appears in capital letters.
`small_caps`	The text appears in capital letters, with lowercase letters two points smaller.
`shadow`	The text appears with a shadow.
`outline`	The text appears outlined rather than solid.
`rtl`	The text is written right-to-left.
`imprint`	The text appears pressed into the page.
`emboss`	The text appears raised off the page in relief.

For example, to change the styles of *demo.docx*, enter the following into the interactive shell:

```
>>> doc = docx.Document('demo.docx')
>>> doc.paragraphs[0].text
'Document Title'
>>> doc.paragraphs[0].style
'Title'
>>> doc.paragraphs[0].style = 'Normal'
>>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
>>> (doc.paragraphs[1].runs[0].text, doc.paragraphs[1].runs[1].text, doc.
paragraphs[1].runs[2].text, doc.paragraphs[1].runs[3].text)
('A plain paragraph with some ', 'bold', ' and some ', 'italic')
>>> doc.paragraphs[1].runs[0].style = 'QuoteChar'
>>> doc.paragraphs[1].runs[1].underline = True
>>> doc.paragraphs[1].runs[3].underline = True
>>> doc.save('restyled.docx')
```

Here, we use the `text` and `style` attributes to easily see what's in the paragraphs in our document. We can see that it's simple to divide a paragraph into runs and access each run individiaully. So we get the first, second, and fourth runs in the second paragraph, style each run, and save the results to a new document.

The words *Document Title* at the top of *restyled.docx* will have the Normal style instead of the Title style, the `Run` object for the text *A plain paragraph with some* will have the QuoteChar style, and the two `Run` objects for the words *bold* and *italic* will have their `underline` attributes set to `True`. Figure 13-7 shows how the styles of paragraphs and runs look in *restyled.docx*.

Figure 13-7: The restyled.docx *file*

You can find more complete documentation on Python-Docx's use of styles at *https://python-docx.readthedocs.org/en/latest/user/styles.html*.

Writing Word Documents

Enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')
<docx.text.Paragraph object at 0x0000000003B56F60>
>>> doc.save('helloworld.docx')
```

To create your own *.docx* file, call `docx.Document()` to return a new, blank Word `Document` object. The `add_paragraph()` document method adds a new paragraph of text to the document and returns a reference to the `Paragraph` object that was added. When you're done adding text, pass a filename string to the `save()` document method to save the `Document` object to a file.

This will create a file named *helloworld.docx* in the current working directory that, when opened, looks like Figure 13-8.

Figure 13-8: The Word document created using `add_paragraph('Hello world!')`

You can add paragraphs by calling the `add_paragraph()` method again with the new paragraph's text. Or to add text to the end of an existing paragraph, you can call the paragraph's `add_run()` method and pass it a string. Enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')
<docx.text.Paragraph object at 0x000000000366AD30>
>>> paraObj1 = doc.add_paragraph('This is a second paragraph.')
>>> paraObj2 = doc.add_paragraph('This is a yet another paragraph.')
>>> paraObj1.add_run(' This text is being added to the second paragraph.')
<docx.text.Run object at 0x0000000003A2C860>
>>> doc.save('multipleParagraphs.docx')
```

The resulting document will look like Figure 13-9. Note that the text *This text is being added to the second paragraph.* was added to the `Paragraph` object in `paraObj1`, which was the second paragraph added to `doc`. The `add_paragraph()` and `add_run()` functions return paragraph and `Run` objects, respectively, to save you the trouble of extracting them as a separate step.

Keep in mind that as of Python-Docx version 0.5.3, new `Paragraph` objects can be added only to the end of the document, and new `Run` objects can be added only to the end of a `Paragraph` object.

The `save()` method can be called again to save the additional changes you've made.

Figure 13-9: The document with multiple `Paragraph` and `Run` objects added

Both `add_paragraph()` and `add_run()` accept an optional second argument that is a string of the `Paragraph` or `Run` object's style. For example:

```
>>> doc.add_paragraph('Hello world!', 'Title')
```

This line adds a paragraph with the text *Hello world!* in the Title style.

Adding Headings

Calling `add_heading()` adds a paragraph with one of the heading styles. Enter the following into the interactive shell:

```
>>> doc = docx.Document()
>>> doc.add_heading('Header 0', 0)
<docx.text.Paragraph object at 0x00000000036CB3C8>
>>> doc.add_heading('Header 1', 1)
<docx.text.Paragraph object at 0x00000000036CB630>
>>> doc.add_heading('Header 2', 2)
<docx.text.Paragraph object at 0x00000000036CB828>
>>> doc.add_heading('Header 3', 3)
<docx.text.Paragraph object at 0x00000000036CB2E8>
>>> doc.add_heading('Header 4', 4)
<docx.text.Paragraph object at 0x00000000036CB3C8>
>>> doc.save('headings.docx')
```

The arguments to `add_heading()` are a string of the heading text and an integer from `0` to `4`. The integer `0` makes the heading the Title style, which is used for the top of the document. Integers `1` to `4` are for various heading levels, with `1` being the main heading and `4` the lowest subheading. The `add_heading()` function returns a `Paragraph` object to save you the step of extracting it from the `Document` object as a separate step.

The resulting *headings.docx* file will look like Figure 13-10.

Figure 13-10: The headings.docx *document with headings 0 to 4*

Adding Line and Page Breaks

To add a line break (rather than starting a whole new paragraph), you can call the `add_break()` method on the `Run` object you want to have the break appear after. If you want to add a page break instead, you need to pass the value `docx.text.WD_BREAK.PAGE` as a lone argument to `add_break()`, as is done in the middle of the following example:

```
>>> doc = docx.Document()
>>> doc.add_paragraph('This is on the first page!')
<docx.text.Paragraph object at 0x0000000003785518>
❶ >>> doc.paragraphs[0].runs[0].add_break(docx.text.WD_BREAK.PAGE)
>>> doc.add_paragraph('This is on the second page!')
<docx.text.Paragraph object at 0x00000000037855F8>
>>> doc.save('twoPage.docx')
```

This creates a two-page Word document with *This is on the first page!* on the first page and *This is on the second page!* on the second. Even though there was still plenty of space on the first page after the text *This is on the first page!*, we forced the next paragraph to begin on a new page by inserting a page break after the first run of the first paragraph ❶.

Adding Pictures

`Document` objects have an `add_picture()` method that will let you add an image to the end of the document. Say you have a file *zophie.png* in the current working directory. You can add *zophie.png* to the end of your document with a width of 1 inch and height of 4 centimeters (Word can use both imperial and metric units) by entering the following:

```
>>> doc.add_picture('zophie.png', width=docx.shared.Inches(1),
height=docx.shared.Cm(4))
<docx.shape.InlineShape object at 0x00000000036C7D30>
```

The first argument is a string of the image's filename. The optional `width` and `height` keyword arguments will set the width and height of the image in the document. If left out, the width and height will default to the normal size of the image.

You'll probably prefer to specify an image's height and width in familiar units such as inches and centimeters, so you can use the `docx.shared.Inches()` and `docx.shared.Cm()` functions when you're specifying the `width` and `height` keyword arguments.

Summary

Text information isn't just for plaintext files; in fact, it's pretty likely that you deal with PDFs and Word documents much more often. You can use the `PyPDF2` module to read and write PDF documents. Unfortunately, reading text from PDF documents might not always result in a perfect translation to a string because of the complicated PDF file format, and some PDFs might not be readable at all. In these cases, you're out of luck unless future updates to PyPDF2 support additional PDF features.

Word documents are more reliable, and you can read them with the `python-docx` module. You can manipulate text in Word documents via `Paragraph` and `Run` objects. These objects can also be given styles, though they must be from the default set of styles or styles already in the document. You can add new paragraphs, headings, breaks, and pictures to the document, though only to the end.

Many of the limitations that come with working with PDFs and Word documents are because these formats are meant to be nicely displayed for human readers, rather than easy to parse by software. The next chapter takes a look at two other common formats for storing information: JSON and CSV files. These formats are designed to be used by computers, and you'll see that Python can work with these formats much more easily.

Practice Questions

1. A string value of the PDF filename is *not* passed to the `PyPDF2.PdfFileReader()` function. What do you pass to the function instead?
2. What modes do the `File` objects for `PdfFileReader()` and `PdfFileWriter()` need to be opened in?
3. How do you acquire a `Page` object for page 5 from a `PdfFileReader` object?
4. What `PdfFileReader` variable stores the number of pages in the PDF document?
5. If a `PdfFileReader` object's PDF is encrypted with the password `swordfish`, what must you do before you can obtain `Page` objects from it?
6. What methods do you use to rotate a page?
7. What method returns a `Document` object for a file named *demo.docx*?

8. What is the difference between a `Paragraph` object and a `Run` object?
9. How do you obtain a list of `Paragraph` objects for a `Document` object that's stored in a variable named `doc`?
10. What type of object has `bold`, `underline`, `italic`, `strike`, and `outline` variables?
11. What is the difference between setting the `bold` variable to `True`, `False`, or `None`?
12. How do you create a `Document` object for a new Word document?
13. How do you add a paragraph with the text `'Hello there!'` to a `Document` object stored in a variable named `doc`?
14. What integers represent the levels of headings available in Word documents?

Practice Projects

For practice, write programs that do the following.

PDF Paranoia

Using the `os.walk()` function from Chapter 9, write a script that will go through every PDF in a folder (and its subfolders) and encrypt the PDFs using a password provided on the command line. Save each encrypted PDF with an *_encrypted.pdf* suffix added to the original filename. Before deleting the original file, have the program attempt to read and decrypt the file to ensure that it was encrypted correctly.

Then, write a program that finds all encrypted PDFs in a folder (and its subfolders) and creates a decrypted copy of the PDF using a provided password. If the password is incorrect, the program should print a message to the user and continue to the next PDF.

Custom Invitations as Word Documents

Say you have a text file of guest names. This *guests.txt* file has one name per line, as follows:

```
Prof. Plum
Miss Scarlet
Col. Mustard
Al Sweigart
RoboCop
```

Write a program that would generate a Word document with custom invitations that look like Figure 13-11.

Since Python-Docx can use only those styles that already exist in the Word document, you will have to first add these styles to a blank Word file and then open that file with Python-Docx. There should be one invitation per page in the resulting Word document, so call `add_break()` to add a page break after the last paragraph of each invitation. This way, you will need to open only one Word document to print all of the invitations at once.

Figure 13-11: The Word document generated by your custom invite script

You can download a sample *guests.txt* file from *http://nostarch.com/automatestuff/*.

Brute-Force PDF Password Breaker

Say you have an encrypted PDF that you have forgotten the password to, but you remember it was a single English word. Trying to guess your forgotten password is quite a boring task. Instead you can write a program that will decrypt the PDF by trying every possible English word until it finds one that works. This is called a *brute-force password attack*. Download the text file *dictionary.txt* from *http://nostarch.com/automatestuff/*. This *dictionary file* contains over 44,000 English words with one word per line.

Using the file-reading skills you learned in Chapter 8, create a list of word strings by reading this file. Then loop over each word in this list, passing it to the `decrypt()` method. If this method returns the integer `0`, the password was wrong and your program should continue to the next password. If `decrypt()` returns `1`, then your program should break out of the loop and print the hacked password. You should try both the uppercase and lowercase form of each word. (On my laptop, going through all 88,000 uppercase and lowercase words from the dictionary file takes a couple of minutes. This is why you shouldn't use a simple English word for your passwords.)

14

WORKING WITH CSV FILES AND JSON DATA

In Chapter 13, you learned how to extract text from PDF and Word documents. These files were in a binary format, which required special Python modules to access their data. CSV and JSON files, on the other hand, are just plaintext files. You can view them in a text editor, such as IDLE's file editor. But Python also comes with the special `csv` and `json` modules, each providing functions to help you work with these file formats.

CSV stands for "comma-separated values," and CSV files are simplified spreadsheets stored as plaintext files. Python's `csv` module makes it easy to parse CSV files.

JSON (pronounced "JAY-sawn" or "Jason"—it doesn't matter how because either way people will say you're pronouncing it wrong) is a format that stores information as JavaScript source code in plaintext files.

(JSON is short for JavaScript Object Notation.) You don't need to know the JavaScript programming language to use JSON files, but the JSON format is useful to know because it's used in many web applications.

The csv Module

Each line in a CSV file represents a row in the spreadsheet, and commas separate the cells in the row. For example, the spreadsheet *example.xlsx* from *http://nostarch.com/automatestuff/* would look like this in a CSV file:

```
4/5/2015 13:34,Apples,73
4/5/2015 3:41,Cherries,85
4/6/2015 12:46,Pears,14
4/8/2015 8:59,Oranges,52
4/10/2015 2:07,Apples,152
4/10/2015 18:10,Bananas,23
4/10/2015 2:40,Strawberries,98
```

I will use this file for this chapter's interactive shell examples. You can download *example.csv* from *http://nostarch.com/automatestuff/* or enter the text into a text editor and save it as *example.csv*.

CSV files are simple, lacking many of the features of an Excel spreadsheet. For example, CSV files

- Don't have types for their values—everything is a string
- Don't have settings for font size or color
- Don't have multiple worksheets
- Can't specify cell widths and heights
- Can't have merged cells
- Can't have images or charts embedded in them

The advantage of CSV files is simplicity. CSV files are widely supported by many types of programs, can be viewed in text editors (including IDLE's file editor), and are a straightforward way to represent spreadsheet data. The CSV format is exactly as advertised: It's just a text file of comma-separated values.

Since CSV files are just text files, you might be tempted to read them in as a string and then process that string using the techniques you learned in Chapter 8. For example, since each cell in a CSV file is separated by a comma, maybe you could just call the `split()` method on each line of text to get the values. But not every comma in a CSV file represents the boundary between two cells. CSV files also have their own set of escape characters to allow commas and other characters to be included *as part of the values*. The `split()` method doesn't handle these escape characters. Because of these potential pitfalls, you should always use the `csv` module for reading and writing CSV files.

Reader Objects

To read data from a CSV file with the `csv` module, you need to create a `Reader` object. A `Reader` object lets you iterate over lines in the CSV file. Enter the following into the interactive shell, with *example.csv* in the current working directory:

```
❶ >>> import csv
❷ >>> exampleFile = open('example.csv')
❸ >>> exampleReader = csv.reader(exampleFile)
❹ >>> exampleData = list(exampleReader)
❺ >>> exampleData
[['4/5/2015 13:34', 'Apples', '73'], ['4/5/2015 3:41', 'Cherries', '85'],
['4/6/2015 12:46', 'Pears', '14'], ['4/8/2015 8:59', 'Oranges', '52'],
['4/10/2015 2:07', 'Apples', '152'], ['4/10/2015 18:10', 'Bananas', '23'],
['4/10/2015 2:40', 'Strawberries', '98']]
```

The `csv` module comes with Python, so we can import it ❶ without having to install it first.

To read a CSV file with the `csv` module, first open it using the `open()` function ❷, just as you would any other text file. But instead of calling the `read()` or `readlines()` method on the `File` object that `open()` returns, pass it to the `csv.reader()` function ❸. This will return a `Reader` object for you to use. Note that you don't pass a filename string directly to the `csv.reader()` function.

The most direct way to access the values in the `Reader` object is to convert it to a plain Python list by passing it to `list()` ❹. Using `list()` on this `Reader` object returns a list of lists, which you can store in a variable like `exampleData`. Entering `exampleData` in the shell displays the list of lists ❺.

Now that you have the CSV file as a list of lists, you can access the value at a particular row and column with the expression `exampleData[row][col]`, where `row` is the index of one of the lists in `exampleData`, and `col` is the index of the item you want from that list. Enter the following into the interactive shell:

```
>>> exampleData[0][0]
'4/5/2015 13:34'
>>> exampleData[0][1]
'Apples'
>>> exampleData[0][2]
'73'
>>> exampleData[1][1]
'Cherries'
>>> exampleData[6][1]
'Strawberries'
```

`exampleData[0][0]` goes into the first list and gives us the first string, `exampleData[0][2]` goes into the first list and gives us the third string, and so on.

Reading Data from Reader Objects in a for Loop

For large CSV files, you'll want to use the `Reader` object in a `for` loop. This avoids loading the entire file into memory at once. For example, enter the following into the interactive shell:

```
>>> import csv
>>> exampleFile = open('example.csv')
>>> exampleReader = csv.reader(exampleFile)
>>> for row in exampleReader:
        print('Row #' + str(exampleReader.line_num) + ' ' + str(row))

Row #1 ['4/5/2015 13:34', 'Apples', '73']
Row #2 ['4/5/2015 3:41', 'Cherries', '85']
Row #3 ['4/6/2015 12:46', 'Pears', '14']
Row #4 ['4/8/2015 8:59', 'Oranges', '52']
Row #5 ['4/10/2015 2:07', 'Apples', '152']
Row #6 ['4/10/2015 18:10', 'Bananas', '23']
Row #7 ['4/10/2015 2:40', 'Strawberries', '98']
```

After you import the `csv` module and make a `Reader` object from the CSV file, you can loop through the rows in the `Reader` object. Each row is a list of values, with each value representing a cell.

The `print()` function call prints the number of the current row and the contents of the row. To get the row number, use the `Reader` object's `line_num` variable, which contains the number of the current line.

The `Reader` object can be looped over only once. To reread the CSV file, you must call `csv.reader` to create a `Reader` object.

Writer Objects

A `Writer` object lets you write data to a CSV file. To create a `Writer` object, you use the `csv.writer()` function. Enter the following into the interactive shell:

```
  >>> import csv
❶ >>> outputFile = open('output.csv', 'w', newline='')
❷ >>> outputWriter = csv.writer(outputFile)
  >>> outputWriter.writerow(['spam', 'eggs', 'bacon', 'ham'])
  21
  >>> outputWriter.writerow(['Hello, world!', 'eggs', 'bacon', 'ham'])
  32
  >>> outputWriter.writerow([1, 2, 3.141592, 4])
  16
  >>> outputFile.close()
```

First, call `open()` and pass it `'w'` to open a file in write mode ❶. This will create the object you can then pass to `csv.writer()` ❷ to create a `Writer` object.

On Windows, you'll also need to pass a blank string for the `open()` function's `newline` keyword argument. For technical reasons beyond the scope of this book, if you forget to set the `newline` argument, the rows in *output.csv* will be double-spaced, as shown in Figure 14-1.

	A	B	C	D	E	F	G
1	42	2	3	4	5	6	7
2							
3	2	4	6	8	10	12	14
4							
5	3	6	9	12	15	18	21
6							
7	4	8	12	16	20	24	28
8							
9	5	10	15	20	25	30	35
10							

Figure 14-1: If you forget the `newline=''` keyword argument in `open()`, the CSV file will be double-spaced.

The `writerow()` method for `Writer` objects takes a list argument. Each value in the list is placed in its own cell in the output CSV file. The return value of `writerow()` is the number of characters written to the file for that row (including newline characters).

This code produces an *output.csv* file that looks like this:

```
spam,eggs,bacon,ham
"Hello, world!",eggs,bacon,ham
1,2,3.141592,4
```

Notice how the `Writer` object automatically escapes the comma in the value `'Hello, world!'` with double quotes in the CSV file. The `csv` module saves you from having to handle these special cases yourself.

The delimiter and lineterminator Keyword Arguments

Say you want to separate cells with a tab character instead of a comma and you want the rows to be double-spaced. You could enter something like the following into the interactive shell:

```
>>> import csv
>>> csvFile = open('example.tsv', 'w', newline='')
❶ >>> csvWriter = csv.writer(csvFile, delimiter='\t', lineterminator='\n\n')
>>> csvWriter.writerow(['apples', 'oranges', 'grapes'])
24
>>> csvWriter.writerow(['eggs', 'bacon', 'ham'])
17
>>> csvWriter.writerow(['spam', 'spam', 'spam', 'spam', 'spam', 'spam'])
32
>>> csvFile.close()
```

This changes the delimiter and line terminator characters in your file. The *delimiter* is the character that appears between cells on a row. By default, the delimiter for a CSV file is a comma. The *line terminator* is the character that comes at the end of a row. By default, the line terminator is a newline. You can change characters to different values by using the `delimiter` and `lineterminator` keyword arguments with `csv.writer()`.

Passing `delimeter='\t'` and `lineterminator='\n\n'` ❶ changes the character between cells to a tab and the character between rows to two newlines. We then call `writerow()` three times to give us three rows.

This produces a file named *example.tsv* with the following contents:

```
apples  oranges grapes

eggs    bacon   ham

spam    spam    spam    spam    spam    spam
```

Now that our cells are separated by tabs, we're using the file extension *.tsv*, for tab-separated values.

Project: Removing the Header from CSV Files

Say you have the boring job of removing the first line from several hundred CSV files. Maybe you'll be feeding them into an automated process that requires just the data and not the headers at the top of the columns. You *could* open each file in Excel, delete the first row, and resave the file—but that would take hours. Let's write a program to do it instead.

The program will need to open every file with the *.csv* extension in the current working directory, read in the contents of the CSV file, and rewrite the contents without the first row to a file of the same name. This will replace the old contents of the CSV file with the new, headless contents.

WARNING *As always, whenever you write a program that modifies files, be sure to back up the files, first just in case your program does not work the way you expect it to. You don't want to accidentally erase your original files.*

At a high level, the program must do the following:

- Find all the CSV files in the current working directory.
- Read in the full contents of each file.
- Write out the contents, skipping the first line, to a new CSV file.

At the code level, this means the program will need to do the following:

- Loop over a list of files from `os.listdir()`, skipping the non-CSV files.
- Create a CSV `Reader` object and read in the contents of the file, using the `line_num` attribute to figure out which line to skip.
- Create a CSV `Writer` object and write out the read-in data to the new file.

For this project, open a new file editor window and save it as *removeCsvHeader.py*.

Step 1: Loop Through Each CSV File

The first thing your program needs to do is loop over a list of all CSV filenames for the current working directory. Make your *removeCsvHeader.py* look like this:

```
#! python3
# removeCsvHeader.py - Removes the header from all CSV files in the current
# working directory.

import csv, os

os.makedirs('headerRemoved', exist_ok=True)

# Loop through every file in the current working directory.
for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
❶       continue    # skip non-csv files

    print('Removing header from ' + csvFilename + '...')

    # TODO: Read the CSV file in (skipping first row).

    # TODO: Write out the CSV file.
```

The `os.makedirs()` call will create a `headerRemoved` folder where all the headless CSV files will be written. A `for` loop on `os.listdir('.')` gets you partway there, but it will loop over *all* files in the working directory, so you'll need to add some code at the start of the loop that skips filenames that don't end with `.csv`. The `continue` statement ❶ makes the `for` loop move on to the next filename when it comes across a non-CSV file.

Just so there's *some* output as the program runs, print out a message saying which CSV file the program is working on. Then, add some `TODO` comments for what the rest of the program should do.

Step 2: Read in the CSV File

The program doesn't remove the first line from the CSV file. Rather, it creates a new copy of the CSV file without the first line. Since the copy's filename is the same as the original filename, the copy will overwrite the original.

The program will need a way to track whether it is currently looping on the first row. Add the following to *removeCsvHeader.py*.

```
#! python3
# removeCsvHeader.py - Removes the header from all CSV files in the current
# working directory.

--snip--
```

```
    # Read the CSV file in (skipping first row).
    csvRows = []
    csvFileObj = open(csvFilename)
    readerObj = csv.reader(csvFileObj)
    for row in readerObj:
        if readerObj.line_num == 1:
            continue    # skip first row
        csvRows.append(row)
    csvFileObj.close()

    # TODO: Write out the CSV file.
```

The `Reader` object's `line_num` attribute can be used to determine which line in the CSV file it is currently reading. Another `for` loop will loop over the rows returned from the CSV `Reader` object, and all rows but the first will be appended to `csvRows`.

As the `for` loop iterates over each row, the code checks whether `readerObj.line_num` is set to `1`. If so, it executes a `continue` to move on to the next row without appending it to `csvRows`. For every row afterward, the condition will be always be `False`, and the row will be appended to `csvRows`.

Step 3: Write Out the CSV File Without the First Row

Now that `csvRows` contains all rows but the first row, the list needs to be written out to a CSV file in the *headerRemoved* folder. Add the following to *removeCsvHeader.py*:

```
#! python3
# removeCsvHeader.py - Removes the header from all CSV files in the current
# working directory.
--snip--

# Loop through every file in the current working directory.
❶ for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
        continue    # skip non-CSV files

    --snip--

    # Write out the CSV file.
    csvFileObj = open(os.path.join('headerRemoved', csvFilename), 'w',
                 newline='')
    csvWriter = csv.writer(csvFileObj)
    for row in csvRows:
        csvWriter.writerow(row)
    csvFileObj.close()
```

The CSV `Writer` object will write the list to a CSV file in `headerRemoved` using `csvFilename` (which we also used in the CSV reader). This will overwrite the original file.

Once we create the `Writer` object, we loop over the sublists stored in `csvRows` and write each sublist to the file.

After the code is executed, the outer `for` loop ❶ will loop to the next filename from `os.listdir('.')`. When that loop is finished, the program will be complete.

To test your program, download *removeCsvHeader.zip* from *http://nostarch.com/automatestuff/* and unzip it to a folder. Run the *removeCsvHeader.py* program in that folder. The output will look like this:

```
Removing header from NAICS_data_1048.csv...
Removing header from NAICS_data_1218.csv...
--snip--
Removing header from NAICS_data_9834.csv...
Removing header from NAICS_data_9986.csv...
```

This program should print a filename each time it strips the first line from a CSV file.

Ideas for Similar Programs

The programs that you could write for CSV files are similar to the kinds you could write for Excel files, since they're both spreadsheet files. You could write programs to do the following:

- Compare data between different rows in a CSV file or between multiple CSV files.
- Copy specific data from a CSV file to an Excel file, or vice versa.
- Check for invalid data or formatting mistakes in CSV files and alert the user to these errors.
- Read data from a CSV file as input for your Python programs.

JSON and APIs

JavaScript Object Notation is a popular way to format data as a single human-readable string. JSON is the native way that JavaScript programs write their data structures and usually resembles what Python's `pprint()` function would produce. You don't need to know JavaScript in order to work with JSON-formatted data.

Here's an example of data formatted as JSON:

```
{"name": "Zophie", "isCat": true,
 "miceCaught": 0, "napsTaken": 37.5,
 "felineIQ": null}
```

JSON is useful to know, because many websites offer JSON content as a way for programs to interact with the website. This is known as providing an *application programming interface (API)*. Accessing an API is the same

as accessing any other web page via a URL. The difference is that the data returned by an API is formatted (with JSON, for example) for machines; APIs aren't easy for people to read.

Many websites make their data available in JSON format. Facebook, Twitter, Yahoo, Google, Tumblr, Wikipedia, Flickr, Data.gov, Reddit, IMDb, Rotten Tomatoes, LinkedIn, and many other popular sites offer APIs for programs to use. Some of these sites require registration, which is almost always free. You'll have to find documentation for what URLs your program needs to request in order to get the data you want, as well as the general format of the JSON data structures that are returned. This documentation should be provided by whatever site is offering the API; if they have a "Developers" page, look for the documentation there.

Using APIs, you could write programs that do the following:

- Scrape raw data from websites. (Accessing APIs is often more convenient than downloading web pages and parsing HTML with Beautiful Soup.)
- Automatically download new posts from one of your social network accounts and post them to another account. For example, you could take your Tumblr posts and post them to Facebook.
- Create a "movie encyclopedia" for your personal movie collection by pulling data from IMDb, Rotten Tomatoes, and Wikipedia and putting it into a single text file on your computer.

You can see some examples of JSON APIs in the resources at *http://nostarch.com/automatestuff/*.

The json Module

Python's `json` module handles all the details of translating between a string with JSON data and Python values for the `json.loads()` and `json.dumps()` functions. JSON can't store *every* kind of Python value. It can contain values of only the following data types: strings, integers, floats, Booleans, lists, dictionaries, and `NoneType`. JSON cannot represent Python-specific objects, such as `File` objects, CSV `Reader` or `Writer` objects, `Regex` objects, or Selenium `WebElement` objects.

Reading JSON with the loads() Function

To translate a string containing JSON data into a Python value, pass it to the `json.loads()` function. (The name means "load string," not "loads.") Enter the following into the interactive shell:

```
>>> stringOfJsonData = '{"name": "Zophie", "isCat": true, "miceCaught": 0,
"felineIQ": null}'
>>> import json
>>> jsonDataAsPythonValue = json.loads(stringOfJsonData)
>>> jsonDataAsPythonValue
{'isCat': True, 'miceCaught': 0, 'name': 'Zophie', 'felineIQ': None}
```

After you import the `json` module, you can call `loads()` and pass it a string of JSON data. Note that JSON strings always use double quotes. It will return that data as a Python dictionary. Python dictionaries are not ordered, so the key-value pairs may appear in a different order when you print `jsonDataAsPythonValue`.

Writing JSON with the dumps() Function

The `json.dumps()` function (which means "dump string," not "dumps") will translate a Python value into a string of JSON-formatted data. Enter the following into the interactive shell:

```
>>> pythonValue = {'isCat': True, 'miceCaught': 0, 'name': 'Zophie',
'felineIQ': None}
>>> import json
>>> stringOfJsonData = json.dumps(pythonValue)
>>> stringOfJsonData
'{"isCat": true, "felineIQ": null, "miceCaught": 0, "name": "Zophie" }'
```

The value can only be one of the following basic Python data types: dictionary, list, integer, float, string, Boolean, or `None`.

Project: Fetching Current Weather Data

Checking the weather seems fairly trivial: Open your web browser, click the address bar, type the URL to a weather website (or search for one and then click the link), wait for the page to load, look past all the ads, and so on.

Actually, there are a lot of boring steps you could skip if you had a program that downloaded the weather forecast for the next few days and printed it as plaintext. This program uses the `requests` module from Chapter 11 to download data from the Web.

Overall, the program does the following:

- Reads the requested location from the command line.
- Downloads JSON weather data from OpenWeatherMap.org.
- Converts the string of JSON data to a Python data structure.
- Prints the weather for today and the next two days.

So the code will need to do the following:

- Join strings in `sys.argv` to get the location.
- Call `requests.get()` to download the weather data.
- Call `json.loads()` to convert the JSON data to a Python data structure.
- Print the weather forecast.

For this project, open a new file editor window and save it as *quickWeather.py*.

Step 1: Get Location from the Command Line Argument

The input for this program will come from the command line. Make *quickWeather.py* look like this:

```
#! python3
# quickWeather.py - Prints the weather for a location from the command line.

import json, requests, sys

# Compute location from command line arguments.
if len(sys.argv) < 2:
    print('Usage: quickWeather.py location')
    sys.exit()
location = ' '.join(sys.argv[1:])

# TODO: Download the JSON data from OpenWeatherMap.org's API.

# TODO: Load JSON data into a Python variable.
```

In Python, command line arguments are stored in the `sys.argv` list. After the `#!` shebang line and `import` statements, the program will check that there is more than one command line argument. (Recall that `sys.argv` will always have at least one element, `sys.argv[0]`, which contains the Python script's filename.) If there is only one element in the list, then the user didn't provide a location on the command line, and a "usage" message will be provided to the user before the program ends.

Command line arguments are split on spaces. The command line argument `San Francisco, CA` would make `sys.argv` hold `['quickWeather.py', 'San', 'Francisco,', 'CA']`. Therefore, call the `join()` method to join all the strings except for the first in `sys.argv`. Store this joined string in a variable named `location`.

Step 2: Download the JSON Data

OpenWeatherMap.org provides real-time weather information in JSON format. Your program simply has to download the page at *http://api.openweathermap.org/data/2.5/forecast/daily?q=<Location>&cnt=3*, where *<Location>* is the name of the city whose weather you want. Add the following to *quickWeather.py*.

```
#! python3
# quickWeather.py - Prints the weather for a location from the command line.

--snip--

# Download the JSON data from OpenWeatherMap.org's API.
url ='http://api.openweathermap.org/data/2.5/forecast/daily?q=%s&cnt=3' % (location)
response = requests.get(url)
response.raise_for_status()

# TODO: Load JSON data into a Python variable.
```

We have `location` from our command line arguments. To make the URL we want to access, we use the `%s` placeholder and insert whatever string is stored in `location` into that spot in the URL string. We store the result in `url` and pass `url` to `requests.get()`. The `requests.get()` call returns a `Response` object, which you can check for errors by calling `raise_for_status()`. If no exception is raised, the downloaded text will be in `response.text`.

Step 3: Load JSON Data and Print Weather

The `response.text` member variable holds a large string of JSON-formatted data. To convert this to a Python value, call the `json.loads()` function. The JSON data will look something like this:

```
{'city': {'coord': {'lat': 37.7771, 'lon': -122.42},
          'country': 'United States of America',
          'id': '5391959',
          'name': 'San Francisco',
          'population': 0},
 'cnt': 3,
 'cod': '200',
 'list': [{'clouds': 0,
           'deg': 233,
           'dt': 1402344000,
           'humidity': 58,
           'pressure': 1012.23,
           'speed': 1.96,
           'temp': {'day': 302.29,
                    'eve': 296.46,
                    'max': 302.29,
                    'min': 289.77,
                    'morn': 294.59,
                    'night': 289.77},
           'weather': [{'description': 'sky is clear',
                        'icon': '01d',
--snip--
```

You can see this data by passing `weatherData` to `pprint.pprint()`. You may want to check *http://openweathermap.org/* for more documentation on what these fields mean. For example, the online documentation will tell you that the `302.29` after `'day'` is the daytime temperature in Kelvin, not Celsius or Fahrenheit.

The weather descriptions you want are after `'main'` and `'description'`. To neatly print them out, add the following to *quickWeather.py*.

```
! python3
# quickWeather.py - Prints the weather for a location from the command line.

--snip--

# Load JSON data into a Python variable.
weatherData = json.loads(response.text)
```

```
# Print weather descriptions.
❶ w = weatherData['list']
print('Current weather in %s:' % (location))
print(w[0]['weather'][0]['main'], '-', w[0]['weather'][0]['description'])
print()
print('Tomorrow:')
print(w[1]['weather'][0]['main'], '-', w[1]['weather'][0]['description'])
print()
print('Day after tomorrow:')
print(w[2]['weather'][0]['main'], '-', w[2]['weather'][0]['description'])
```

Notice how the code stores `weatherData['list']` in the variable `w` to save you some typing ❶. You use `w[0]`, `w[1]`, and `w[2]` to retrieve the dictionaries for today, tomorrow, and the day after tomorrow's weather, respectively. Each of these dictionaries has a `'weather'` key, which contains a list value. You're interested in the first list item, a nested dictionary with several more keys, at index 0. Here, we print the values stored in the `'main'` and `'description'` keys, separated by a hyphen.

When this program is run with the command line argument `quickWeather.py San Francisco, CA`, the output looks something like this:

```
Current weather in San Francisco, CA:
Clear - sky is clear

Tomorrow:
Clouds - few clouds

Day after tomorrow:
Clear - sky is clear
```

(The weather is one of the reasons I like living in San Francisco!)

Ideas for Similar Programs

Accessing weather data can form the basis for many types of programs. You can create similar programs to do the following:

- Collect weather forecasts for several campsites or hiking trails to see which one will have the best weather.
- Schedule a program to regularly check the weather and send you a frost alert if you need to move your plants indoors. (Chapter 15 covers scheduling, and Chapter 16 explains how to send email.)
- Pull weather data from multiple sites to show all at once, or calculate and show the average of the multiple weather predictions.

Summary

CSV and JSON are common plaintext formats for storing data. They are easy for programs to parse while still being human readable, so they are often used for simple spreadsheets or web app data. The `csv` and `json` modules greatly simplify the process of reading and writing to CSV and JSON files.

The last few chapters have taught you how to use Python to parse information from a wide variety of file formats. One common task is taking data from a variety of formats and parsing it for the particular information you need. These tasks are often specific to the point that commercial software is not optimally helpful. By writing your own scripts, you can make the computer handle large amounts of data presented in these formats.

In Chapter 15, you'll break away from data formats and learn how to make your programs communicate with you by sending emails and text messages.

Practice Questions

1. What are some features Excel spreadsheets have that CSV spreadsheets don't?
2. What do you pass to `csv.reader()` and `csv.writer()` to create `Reader` and `Writer` objects?
3. What modes do `File` objects for reader and `Writer` objects need to be opened in?
4. What method takes a list argument and writes it to a CSV file?
5. What do the `delimiter` and `lineterminator` keyword arguments do?
6. What function takes a string of JSON data and returns a Python data structure?
7. What function takes a Python data structure and returns a string of JSON data?

Practice Project

For practice, write a program that does the following.

Excel-to-CSV Converter

Excel can save a spreadsheet to a CSV file with a few mouse clicks, but if you had to convert hundreds of Excel files to CSVs, it would take hours of clicking. Using the `openpyxl` module from Chapter 12, write a program that reads all the Excel files in the current working directory and outputs them as CSV files.

A single Excel file might contain multiple sheets; you'll have to create one CSV file per *sheet*. The filenames of the CSV files should be *<excel filename>_<sheet title>.csv*, where *<excel filename>* is the filename of the Excel file without the file extension (for example, `'spam_data'`, not `'spam_data.xlsx'`) and *<sheet title>* is the string from the `Worksheet` object's `title` variable.

This program will involve many nested `for` loops. The skeleton of the program will look something like this:

```
for excelFile in os.listdir('.'):
    # Skip non-xlsx files, load the workbook object.
    for sheetName in wb.get_sheet_names():
        # Loop through every sheet in the workbook.
        sheet = wb.get_sheet_by_name(sheetName)

        # Create the CSV filename from the Excel filename and sheet title.
        # Create the csv.writer object for this CSV file.

        # Loop through every row in the sheet.
        for rowNum in range(1, sheet.get_highest_row() + 1):
            rowData = []    # append each cell to this list
            # Loop through each cell in the row.
            for colNum in range(1, sheet.get_highest_column() + 1):
                # Append each cell's data to rowData.

            # Write the rowData list to the CSV file.

        csvFile.close()
```

Download the ZIP file *excelSpreadsheets.zip* from *http://nostarch.com/automatestuff/*, and unzip the spreadsheets into the same directory as your program. You can use these as the files to test the program on.

15

KEEPING TIME, SCHEDULING TASKS, AND LAUNCHING PROGRAMS

Running programs while you're sitting at your computer is fine, but it's also useful to have programs run without your direct supervision. Your computer's clock can schedule programs to run code at some specified time and date or at regular intervals. For example, your program could scrape a website every hour to check for changes or do a CPU-intensive task at 4 AM while you sleep. Python's `time` and `datetime` modules provide these functions.

You can also write programs that launch other programs on a schedule by using the `subprocess` and `threading` modules. Often, the fastest way to program is to take advantage of applications that other people have already written.

The time Module

Your computer's system clock is set to a specific date, time, and time zone. The built-in `time` module allows your Python programs to read the system clock for the current time. The `time.time()` and `time.sleep()` functions are the most useful in the `time` module.

The time.time() Function

The *Unix epoch* is a time reference commonly used in programming: 12 AM on January 1, 1970, Coordinated Universal Time (UTC). The `time.time()` function returns the number of seconds since that moment as a float value. (Recall that a float is just a number with a decimal point.) This number is called an *epoch timestamp*. For example, enter the following into the interactive shell:

```
>>> import time
>>> time.time()
1425063955.068649
```

Here I'm calling `time.time()` on February 27, 2015, at 11:05 Pacific Standard Time, or 7:05 PM UTC. The return value is how many seconds have passed between the Unix epoch and the moment `time.time()` was called.

NOTE *The interactive shell examples will yield dates and times for when I wrote this chapter in February 2015. Unless you're a time traveler, your dates and times will be different.*

Epoch timestamps can be used to *profile* code, that is, measure how long a piece of code takes to run. If you call `time.time()` at the beginning of the code block you want to measure and again at the end, you can subtract the first timestamp from the second to find the elapsed time between those two calls. For example, open a new file editor window and enter the following program:

```
  import time
❶ def calcProd():
      # Calculate the product of the first 100,000 numbers.
      product = 1
      for i in range(1, 100000):
          product = product * i
      return product

❷ startTime = time.time()
  prod = calcProd()
❸ endTime = time.time()
❹ print('The result is %s digits long.' % (len(str(prod))))
❺ print('Took %s seconds to calculate.' % (endTime - startTime))
```

At ❶, we define a function `calcProd()` to loop through the integers from 1 to 99,999 and return their product. At ❷, we call `time.time()` and store it in `startTime`. Right after calling `calcProd()`, we call `time.time()` again and store it in `endTime` ❸. We end by printing the length of the product returned by `calcProd()` ❹ and how long it took to run `calcProd()` ❺.

Save this program as *calcProd.py* and run it. The output will look something like this:

```
The result is 456569 digits long.
Took 2.844162940979004 seconds to calculate.
```

NOTE *Another way to profile your code is to use the `cProfile.run()` function, which provides a much more informative level of detail than the simple `time.time()` technique. The `cProfile.run()` function is explained at* https://docs.python.org/3/library/profile.html.

The time.sleep() Function

If you need to pause your program for a while, call the `time.sleep()` function and pass it the number of seconds you want your program to stay paused. Enter the following into the interactive shell:

```
  >>> import time
  >>> for i in range(3):
❶         print('Tick')
❷         time.sleep(1)
❸         print('Tock')
❹         time.sleep(1)

  Tick
  Tock
  Tick
  Tock
  Tick
  Tock
❺ >>> time.sleep(5)
```

The `for` loop will print `Tick` ❶, pause for one second ❷, print `Tock` ❸, pause for one second ❹, print `Tick`, pause, and so on until `Tick` and `Tock` have each been printed three times.

The `time.sleep()` function will *block*—that is, it will not return and release your program to execute other code—until after the number of seconds you passed to `time.sleep()` has elapsed. For example, if you enter `time.sleep(5)` ❺, you'll see that the next prompt (`>>>`) doesn't appear until five seconds have passed.

Be aware that pressing CTRL-C will not interrupt `time.sleep()` calls in IDLE. IDLE waits until the entire pause is over before raising the `KeyboardInterrupt` exception. To work around this problem, instead of having a single `time.sleep(30)` call to pause for 30 seconds, use a `for` loop to make 30 calls to `time.sleep(1)`.

```
>>> for i in range(30):
    time.sleep(1)
```

If you press CTRL-C sometime during these 30 seconds, you should see the `KeyboardInterrupt` exception thrown right away.

Rounding Numbers

When working with times, you'll often encounter float values with many digits after the decimal. To make these values easier to work with, you can shorten them with Python's built-in `round()` function, which rounds a float to the precision you specify. Just pass in the number you want to round, plus an optional second argument representing how many digits after the decimal point you want to round it to. If you omit the second argument, `round()` rounds your number to the nearest whole integer. Enter the following into the interactive shell:

```
>>> import time
>>> now = time.time()
>>> now
1425064108.017826
>>> round(now, 2)
1425064108.02
>>> round(now, 4)
1425064108.0178
>>> round(now)
1425064108
```

After importing `time` and storing `time.time()` in `now`, we call `round(now, 2)` to round `now` to two digits after the decimal, `round(now, 4)` to round to four digits after the decimal, and `round(now)` to round to the nearest integer.

Project: Super Stopwatch

Say you want to track how much time you spend on boring tasks you haven't automated yet. You don't have a physical stopwatch, and it's surprisingly difficult to find a free stopwatch app for your laptop or smartphone that isn't covered in ads and doesn't send a copy of your browser history to marketers. (It says it can do this in the license agreement you agreed to. You did read the license agreement, didn't you?) You can write a simple stopwatch program yourself in Python.

At a high level, here's what your program will do:

- Track the amount of time elapsed between presses of the ENTER key, with each key press starting a new "lap" on the timer.
- Print the lap number, total time, and lap time.

This means your code will need to do the following:

- Find the current time by calling `time.time()` and store it as a timestamp at the start of the program, as well as at the start of each lap.
- Keep a lap counter and increment it every time the user presses ENTER.
- Calculate the elapsed time by subtracting timestamps.
- Handle the `KeyboardInterrupt` exception so the user can press CTRL-C to quit.

Open a new file editor window and save it as *stopwatch.py*.

Step 1: Set Up the Program to Track Times

The stopwatch program will need to use the current time, so you'll want to import the `time` module. Your program should also print some brief instructions to the user before calling `input()`, so the timer can begin after the user presses ENTER. Then the code will start tracking lap times.

Enter the following code into the file editor, writing a `TODO` comment as a placeholder for the rest of the code:

```
#! python3
# stopwatch.py - A simple stopwatch program.

import time

# Display the program's instructions.
print('Press ENTER to begin. Afterwards, press ENTER to "click" the stopwatch.
Press Ctrl-C to quit.')
input()                    # press Enter to begin
print('Started.')
startTime = time.time()    # get the first lap's start time
lastTime = startTime
lapNum = 1

# TODO: Start tracking the lap times.
```

Now that we've written the code to display the instructions, start the first lap, note the time, and set our lap count to 1.

Step 2: Track and Print Lap Times

Now let's write the code to start each new lap, calculate how long the previous lap took, and calculate the total time elapsed since starting the stopwatch. We'll display the lap time and total time and increase the lap count for each new lap. Add the following code to your program:

```
#! python3
# stopwatch.py - A simple stopwatch program.

import time

--snip--
```

```
  # Start tracking the lap times.
❶ try:
❷    while True:
         input()
❸        lapTime = round(time.time() - lastTime, 2)
❹        totalTime = round(time.time() - startTime, 2)
❺        print('Lap #%s: %s (%s)' % (lapNum, totalTime, lapTime), end='')
         lapNum += 1
         lastTime = time.time() # reset the last lap time
❻ except KeyboardInterrupt:
     # Handle the Ctrl-C exception to keep its error message from displaying.
     print('\nDone.')
```

If the user presses CTRL-C to stop the stopwatch, the `KeyboardInterrupt` exception will be raised, and the program will crash if its execution is not a `try` statement. To prevent crashing, we wrap this part of the program in a `try` statement ❶. We'll handle the exception in the `except` clause ❻, so when CTRL-C is pressed and the exception is raised, the program execution moves to the except clause to print `Done`, instead of the `KeyboardInterrupt` error message. Until this happens, the execution is inside an infinite loop ❷ that calls `input()` and waits until the user presses ENTER to end a lap. When a lap ends, we calculate how long the lap took by subtracting the start time of the lap, `lastTime`, from the current time, `time.time()` ❸. We calculate the total time elapsed by subtracting the overall start time of the stopwatch, `startTime`, from the current time ❹.

Since the results of these time calculations will have many digits after the decimal point (such as `4.766272783279419`), we use the `round()` function to round the float value to two digits at ❸ and ❹.

At ❺, we print the lap number, total time elapsed, and the lap time. Since the user pressing ENTER for the `input()` call will print a newline to the screen, pass `end=''` to the `print()` function to avoid double-spacing the output. After printing the lap information, we get ready for the next lap by adding 1 to the count `lapNum` and setting `lastTime` to the current time, which is the start time of the next lap.

Ideas for Similar Programs

Time tracking opens up several possibilities for your programs. Although you can download apps to do some of these things, the benefit of writing programs yourself is that they will be free and not bloated with ads and useless features. You could write similar programs to do the following:

- Create a simple timesheet app that records when you type a person's name and uses the current time to clock them in or out.
- Add a feature to your program to display the elapsed time since a process started, such as a download that uses the `requests` module. (See Chapter 11.)
- Intermittently check how long a program has been running and offer the user a chance to cancel tasks that are taking too long.

The datetime Module

The time module is useful for getting a Unix epoch timestamp to work with. But if you want to display a date in a more convenient format, or do arithmetic with dates (for example, figuring out what date was 205 days ago or what date is 123 days from now), you should use the datetime module.

The datetime module has its own datetime data type. datetime values represent a specific moment in time. Enter the following into the interactive shell:

```
   >>> import datetime
❶ >>> datetime.datetime.now()
❷ datetime.datetime(2015, 2, 27, 11, 10, 49, 55, 53)
❸ >>> dt = datetime.datetime(2015, 10, 21, 16, 29, 0)
❹ >>> dt.year, dt.month, dt.day
   (2015, 10, 21)
❺ >>> dt.hour, dt.minute, dt.second
   (16, 29, 0)
```

Calling datetime.datetime.now() ❶ returns a datetime object ❷ for the current date and time, according to your computer's clock. This object includes the year, month, day, hour, minute, second, and microsecond of the current moment. You can also retrieve a datetime object for a specific moment by using the datetime.datetime() function ❸, passing it integers representing the year, month, day, hour, and second of the moment you want. These integers will be stored in the datetime object's year, month, day ❹, hour, minute, and second ❺ attributes.

A Unix epoch timestamp can be converted to a datetime object with the datetime.datetime.fromtimestamp() function. The date and time of the datetime object will be converted for the local time zone. Enter the following into the interactive shell:

```
>>> datetime.datetime.fromtimestamp(1000000)
datetime.datetime(1970, 1, 12, 5, 46, 40)
>>> datetime.datetime.fromtimestamp(time.time())
datetime.datetime(2015, 2, 27, 11, 13, 0, 604980)
```

Calling datetime.datetime.fromtimestamp() and passing it 1000000 returns a datetime object for the moment 1,000,000 seconds after the Unix epoch. Passing time.time(), the Unix epoch timestamp for the current moment, returns a datetime object for the current moment. So the expressions datetime.datetime.now() and datetime.datetime.fromtimestamp(time.time()) do the same thing; they both give you a datetime object for the present moment.

NOTE *These examples were entered on a computer set to Pacific Standard Time. If you're in another time zone, your results will look different.*

datetime objects can be compared with each other using comparison operators to find out which one precedes the other. The later datetime object is the "greater" value. Enter the following into the interactive shell:

```
❶ >>> halloween2015 = datetime.datetime(2015, 10, 31, 0, 0, 0)
❷ >>> newyears2016 = datetime.datetime(2016, 1, 1, 0, 0, 0)
  >>> oct31_2015 = datetime.datetime(2015, 10, 31, 0, 0, 0)
❸ >>> halloween2015 == oct31_2015
  True
❹ >>> halloween2015 > newyears2016
  False
❺ >>> newyears2016 > halloween2015
  True
  >>> newyears2016 != oct31_2015
  True
```

Make a datetime object for the first moment (midnight) of October 31, 2015 and store it in halloween2015 ❶. Make a datetime object for the first moment of January 1, 2016 and store it in newyears2016 ❷. Then make another object for midnight on October 31, 2015 and store it in oct31_2015. Comparing halloween2015 and oct31_2015 shows that they're equal ❸. Comparing newyears2016 and halloween 2015 shows that newyears2016 is greater (later) than halloween2015 ❹ ❺.

The timedelta Data Type

The datetime module also provides a timedelta data type, which represents a *duration* of time rather than a *moment* in time. Enter the following into the interactive shell:

```
❶ >>> delta = datetime.timedelta(days=11, hours=10, minutes=9, seconds=8)
❷ >>> delta.days, delta.seconds, delta.microseconds
  (11, 36548, 0)
  >>> delta.total_seconds()
  986948.0
  >>> str(delta)
  '11 days, 10:09:08'
```

To create a timedelta object, use the datetime.timedelta() function. The datetime.timedelta() function takes keyword arguments weeks, days, hours, minutes, seconds, milliseconds, and microseconds. There is no month or year keyword argument because "a month" or "a year" is a variable amount of time depending on the particular month or year. A timedelta object has the total duration represented in days, seconds, and microseconds. These numbers are stored in the days, seconds, and microseconds attributes, respectively. The total_seconds() method will return the duration in number of seconds alone. Passing a timedelta object to str() will return a nicely formatted, human-readable string representation of the object.

In this example, we pass keyword arguments to `datetime.delta()` to specify a duration of 11 days, 10 hours, 9 minutes, and 8 seconds, and store the returned `timedelta` object in `delta` ❶. This `timedelta` object's `days` attributes stores `11`, and its `seconds` attribute stores `36548` (10 hours, 9 minutes, and 8 seconds, expressed in seconds) ❷. Calling `total_seconds()` tells us that 11 days, 10 hours, 9 minutes, and 8 seconds is 986,948 seconds. Finally, passing the `timedelta` object to `str()` returns a string clearly explaning the duration.

The arithmetic operators can be used to perform *date arithmetic* on `datetime` values. For example, to calculate the date 1,000 days from now, enter the following into the interactive shell:

```
>>> dt = datetime.datetime.now()
>>> dt
datetime.datetime(2015, 2, 27, 18, 38, 50, 636181)
>>> thousandDays = datetime.timedelta(days=1000)
>>> dt + thousandDays
datetime.datetime(2017, 11, 23, 18, 38, 50, 636181)
```

First, make a `datetime` object for the current moment and store it in `dt`. Then make a `timedelta` object for a duration of 1,000 days and store it in `thousandDays`. Add `dt` and `thousandDays` together to get a `datetime` object for the date 1,000 days from now. Python will do the date arithmetic to figure out that 1,000 days after February 27, 2015, will be November 23, 2017. This is useful because when you calculate 1,000 days from a given date, you have to remember how many days are in each month and factor in leap years and other tricky details. The `datetime` module handles all of this for you.

`timedelta` objects can be added or subtracted with `datetime` objects or other `timedelta` objects using the `+` and `-` operators. A `timedelta` object can be multiplied or divided by integer or float values with the `*` and `/` operators. Enter the following into the interactive shell:

```
❶ >>> oct21st = datetime.datetime(2015, 10, 21, 16, 29, 0)
❷ >>> aboutThirtyYears = datetime.timedelta(days=365 * 30)
  >>> oct21st
  datetime.datetime(2015, 10, 21, 16, 29)
  >>> oct21st - aboutThirtyYears
  datetime.datetime(1985, 10, 28, 16, 29)
  >>> oct21st - (2 * aboutThirtyYears)
  datetime.datetime(1955, 11, 5, 16, 29)
```

Here we make a `datetime` object for October 21, 2015 ❶ and a `timedelta` object for a duration of about 30 years (we're assuming 365 days for each of those years) ❷. Subtracting `aboutThirtyYears` from `oct21st` gives us a `datetime` object for the date 30 years before October 21, 2015. Subtracting `2 * aboutThirtyYears` from `oct21st` returns a `datetime` object for the date 60 years before October 21, 2015.

Pausing Until a Specific Date

The `time.sleep()` method lets you pause a program for a certain number of seconds. By using a `while` loop, you can pause your programs until a specific date. For example, the following code will continue to loop until Halloween 2016:

```
import datetime
import time
halloween2016 = datetime.datetime(2016, 10, 31, 0, 0, 0)
while datetime.datetime.now() < halloween2016:
    time.sleep(1)
```

The `time.sleep(1)` call will pause your Python program so that the computer doesn't waste CPU processing cycles simply checking the time over and over. Rather, the `while` loop will just check the condition once per second and continue with the rest of the program after Halloween 2016 (or whenever you program it to stop).

Converting datetime Objects into Strings

Epoch timestamps and `datetime` objects aren't very friendly to the human eye. Use the `strftime()` method to display a `datetime` object as a string. (The *f* in the name of the `strftime()` function stands for *format*.)

The `strftime()` method uses directives similar to Python's string formatting. Table 15-1 has a full list of `strftime()` directives.

Table 15-1: `strftime()` Directives

strftime directive	Meaning
`%Y`	Year with century, as in `'2014'`
`%y`	Year without century, `'00'` to `'99'` (1970 to 2069)
`%m`	Month as a decimal number, `'01'` to `'12'`
`%B`	Full month name, as in `'November'`
`%b`	Abbreviated month name, as in `'Nov'`
`%d`	Day of the month, `'01'` to `'31'`
`%j`	Day of the year, `'001'` to `'366'`
`%w`	Day of the week, `'0'` (Sunday) to `'6'` (Saturday)
`%A`	Full weekday name, as in `'Monday'`
`%a`	Abbreviated weekday name, as in `'Mon'`
`%H`	Hour (24-hour clock), `'00'` to `'23'`
`%I`	Hour (12-hour clock), `'01'` to `'12'`
`%M`	Minute, `'00'` to `'59'`
`%S`	Second, `'00'` to `'59'`
`%p`	`'AM'` or `'PM'`
`%%`	Literal `'%'` character

Pass strrftime() a custom format string containing formatting directives (along with any desired slashes, colons, and so on), and strftime() will return the datetime object's information as a formatted string. Enter the following into the interactive shell:

```
>>> oct21st = datetime.datetime(2015, 10, 21, 16, 29, 0)
>>> oct21st.strftime('%Y/%m/%d %H:%M:%S')
'2015/10/21 16:29:00'
>>> oct21st.strftime('%I:%M %p')
'04:29 PM'
>>> oct21st.strftime("%B of '%y")
"October of '15"
```

Here we have a datetime object for October 21, 2015 at 4:29 PM, stored in oct21st. Passing strftime() the custom format string '%Y/%m/%d %H:%M:%S' returns a string containing 2015, 10, and 21 separated by slahes and 16, 29, and 00 separated by colons. Passing '%I:%M% p' returns '04:29 PM', and passing "%B of '%y" returns "October of '15". Note that strftime() doesn't begin with datetime.datetime.

Converting Strings into datetime Objects

If you have a string of date information, such as '2015/10/21 16:29:00' or 'October 21, 2015', and need to convert it to a datetime object, use the datetime.datetime.strptime() function. The strptime() function is the inverse of the strftime() method. A custom format string using the same directives as strftime() must be passed so that strptime() knows how to parse and understand the string. (The *p* in the name of the strptime() function stands for *parse*.)

Enter the following into the interactive shell:

```
❶ >>> datetime.datetime.strptime('October 21, 2015', '%B %d, %Y')
datetime.datetime(2015, 10, 21, 0, 0)
>>> datetime.datetime.strptime('2015/10/21 16:29:00', '%Y/%m/%d %H:%M:%S')
datetime.datetime(2015, 10, 21, 16, 29)
>>> datetime.datetime.strptime("October of '15", "%B of '%y")
datetime.datetime(2015, 10, 1, 0, 0)
>>> datetime.datetime.strptime("November of '63", "%B of '%y")
datetime.datetime(2063, 11, 1, 0, 0)
```

To get a datetime object from the string 'October 21, 2015', pass 'October 21, 2015' as the first argument to strptime() and the custom format string that corresponds to 'October 21, 2015' as the second argument ❶. The string with the date information must match the custom format string exactly, or Python will raise a ValueError exception.

Review of Python's Time Functions

Dates and times in Python can involve quite a few different data types and functions. Here's a review of the three different types of values used to represent time:

- A Unix epoch timestamp (used by the `time` module) is a float or integer value of the number of seconds since 12 AM on January 1, 1970, UTC.
- A `datetime` object (of the `datetime` module) has integers stored in the attributes `year`, `month`, `day`, `hour`, `minute`, and `second`.
- A `timedelta` object (of the `datetime` module) represents a time duration, rather than a specific moment.

Here's a review of time functions and their parameters and return values:

- The `time.time()` function returns an epoch timestamp float value of the current moment.
- The `time.sleep(`*`seconds`*`)` function stops the program for the amount of seconds specified by the *`seconds`* argument.
- The `datetime.datetime(`*`year, month, day, hour, minute, second`*`)` function returns a `datetime` object of the moment specified by the arguments. If *`hour`*, *`minute`*, or *`second`* arguments are not provided, they default to `0`.
- The `datetime.datetime.now()` function returns a `datetime` object of the current moment.
- The `datetime.datetime.fromtimestamp(`*`epoch`*`)` function returns a `datetime` object of the moment represented by the *`epoch`* timestamp argument.
- The `datetime.timedelta(`*`weeks, days, hours, minutes, seconds, milliseconds, microseconds`*`)` function returns a `timedelta` object representing a duration of time. The function's keyword arguments are all optional and do not include *`month`* or *`year`*.
- The `total_seconds()` method for `timedelta` objects returns the number of seconds the `timedelta` object represents.
- The `strftime(`*`format`*`)` method returns a string of the time represented by the `datetime` object in a custom format that's based on the *`format`* string. See Table 15-1 for the format details.
- The `datetime.datetime.strptime(`*`time_string, format`*`)` function returns a `datetime` object of the moment specified by *`time_string`*, parsed using the *`format`* string argument. See Table 15-1 for the format details.

Multithreading

To introduce the concept of multithreading, let's look at an example situation. Say you want to schedule some code to run after a delay or at a specific time. You could add code like the following at the start of your program:

```
import time, datetime

startTime = datetime.datetime(2029, 10, 31, 0, 0, 0)
while datetime.datetime.now() < startTime:
    time.sleep(1)

print('Program now starting on Halloween 2029')
--snip--
```

This code designates a start time of October 31, 2029, and keeps calling `time.sleep(1)` until the start time arrives. Your program cannot do anything while waiting for the loop of `time.sleep()` calls to finish; it just sits around until Halloween 2029. This is because Python programs by default have a single *thread* of execution.

To understand what a thread of execution is, remember the Chapter 2 discussion of flow control, when you imagined the execution of a program as placing your finger on a line of code in your program and moving to the next line or wherever it was sent by a flow control statement. A *single-threaded* program has only one finger. But a *multithreaded* program has multiple fingers. Each finger still moves to the next line of code as defined by the flow control statements, but the fingers can be at different places in the program, executing different lines of code at the same time. (All of the programs in this book so far have been single threaded.)

Rather than having all of your code wait until the `time.sleep()` function finishes, you can execute the delayed or scheduled code in a separate thread using Python's `threading` module. The separate thread will pause for the `time.sleep` calls. Meanwhile, your program can do other work in the original thread.

To make a separate thread, you first need to make a `Thread` object by calling the `threading.Thread()` function. Enter the following code in a new file and save it as *threadDemo.py*:

```
import threading, time
print('Start of program.')

❶ def takeANap():
    time.sleep(5)
    print('Wake up!')

❷ threadObj = threading.Thread(target=takeANap)
❸ threadObj.start()

print('End of program.')
```

At ❶, we define a function that we want to use in a new thread. To create a `Thread` object, we call `threading.Thread()` and pass it the keyword argument `target=takeANap` ❷. This means the function we want to call in the new thread is `takeANap()`. Notice that the keyword argument is `target=takeANap`, not `target=takeANap()`. This is because you want to pass the `takeANap()` function itself as the argument, not call `takeANap()` and pass its return value.

After we store the `Thread` object created by `threading.Thread()` in `threadObj`, we call `threadObj.start()` ❸ to create the new thread and start executing the target function in the new thread. When this program is run, the output will look like this:

```
Start of program.
End of program.
Wake up!
```

This can be a bit confusing. If `print('End of program.')` is the last line of the program, you might think that it should be the last thing printed. The reason `Wake up!` comes after it is that when `threadObj.start()` is called, the target function for `threadObj` is run in a new thread of execution. Think of it as a second finger appearing at the start of the `takeANap()` function. The main thread continues to `print('End of program.')`. Meanwhile, the new thread that has been executing the `time.sleep(5)` call, pauses for 5 seconds. After it wakes from its 5-second nap, it prints `'Wake up!'` and then returns from the `takeANap()` function. Chronologically, `'Wake up!'` is the last thing printed by the program.

Normally a program terminates when the last line of code in the file has run (or the `sys.exit()` function is called). But *threadDemo.py* has two threads. The first is the original thread that began at the start of the program and ends after `print('End of program.')`. The second thread is created when `threadObj.start()` is called, begins at the start of the `takeANap()` function, and ends after `takeANap()` returns.

A Python program will not terminate until all its threads have terminated. When you ran *threadDemo.py*, even though the original thread had terminated, the second thread was still executing the `time.sleep(5)` call.

Passing Arguments to the Thread's Target Function

If the target function you want to run in the new thread takes arguments, you can pass the target function's arguments to `threading.Thread()`. For example, say you wanted to run this `print()` call in its own thread:

```
>>> print('Cats', 'Dogs', 'Frogs', sep=' & ')
Cats & Dogs & Frogs
```

This `print()` call has three regular arguments, `'Cats'`, `'Dogs'`, and `'Frogs'`, and one keyword argument, `sep=' & '`. The regular arguments can be passed

as a list to the `args` keyword argument in `threading.Thread()`. The keyword argument can be specified as a dictionary to the `kwargs` keyword argument in `threading.Thread()`.

Enter the following into the interactive shell:

```
>>> import threading
>>> threadObj = threading.Thread(target=print, args=['Cats', 'Dogs', 'Frogs'],
kwargs={'sep': ' & '})
>>> threadObj.start()
Cats & Dogs & Frogs
```

To make sure the arguments `'Cats'`, `'Dogs'`, and `'Frogs'` get passed to `print()` in the new thread, we pass `args=['Cats', 'Dogs', 'Frogs']` to `threading.Thread()`. To make sure the keyword argument `sep=' & '` gets passed to `print()` in the new thread, we pass `kwargs={'sep': '& '}` to `threading.Thread()`.

The `threadObj.start()` call will create a new thread to call the `print()` function, and it will pass `'Cats'`, `'Dogs'`, and `'Frogs'` as arguments and `' & '` for the `sep` keyword argument.

This is an incorrect way to create the new thread that calls `print()`:

```
threadObj = threading.Thread(target=print('Cats', 'Dogs', 'Frogs', sep=' & '))
```

What this ends up doing is calling the `print()` function and passing its return value (`print()`'s return value is always `None`) as the `target` keyword argument. It *doesn't* pass the `print()` function itself. When passing arguments to a function in a new thread, use the `threading.Thread()` function's `args` and `kwargs` keyword arguments.

Concurrency Issues

You can easily create several new threads and have them all running at the same time. But multiple threads can also cause problems called *concurrency issues*. These issues happen when threads read and write variables at the same time, causing the threads to trip over each other. Concurrency issues can be hard to reproduce consistently, making them hard to debug.

Multithreaded programming is its own wide subject and beyond the scope of this book. What you have to keep in mind is this: To avoid concurrency issues, never let multiple threads read or write the same variables. When you create a new `Thread` object, make sure its target function uses only local variables in that function. This will avoid hard-to-debug concurrency issues in your programs.

NOTE *A beginner's tutorial on multithreaded programming is available at* http://nostarch.com/automatestuff/.

Project: Multithreaded XKCD Downloader

In Chapter 11, you wrote a program that downloaded all of the XKCD comic strips from the XKCD website. This was a single-threaded program: It downloaded one comic at a time. Much of the program's running time was spent establishing the network connection to begin the download and writing the downloaded images to the hard drive. If you have a broadband Internet connection, your single-threaded program wasn't fully utilizing the available bandwidth.

A multithreaded program that has some threads downloading comics while others are establishing connections and writing the comic image files to disk uses your Internet connection more efficiently and downloads the collection of comics more quickly. Open a new file editor window and save it as *multidownloadXkcd.py*. You will modify this program to add multithreading. The completely modified source code is available to download from *http://nostarch.com/automatestuff/*.

Step 1: Modify the Program to Use a Function

This program will mostly be the same downloading code from Chapter 11, so I'll skip the explanation for the Requests and BeautifulSoup code. The main changes you need to make are importing the `threading` module and making a `downloadXkcd()` function, which takes starting and ending comic numbers as parameters.

For example, calling `downloadXkcd(140, 280)` would loop over the downloading code to download the comics at *http://xkcd.com/140*, *http://xkcd.com/141*, *http://xkcd.com/142*, and so on, up to *http://xkcd.com/279*. Each thread that you create will call `downloadXkcd()` and pass a different range of comics to download.

Add the following code to your *multidownloadXkcd.py* program:

```
#! python3
# multidownloadXkcd.py - Downloads XKCD comics using multiple threads.

import requests, os, bs4, threading
❶ os.makedirs('xkcd', exist_ok=True)    # store comics in ./xkcd

❷ def downloadXkcd(startComic, endComic):
❸     for urlNumber in range(startComic, endComic):
          # Download the page.
          print('Downloading page http://xkcd.com/%s...' % (urlNumber))
❹         res = requests.get('http://xkcd.com/%s' % (urlNumber))
          res.raise_for_status()

❺         soup = bs4.BeautifulSoup(res.text)

          # Find the URL of the comic image.
❻         comicElem = soup.select('#comic img')
          if comicElem == []:
              print('Could not find comic image.')
```

```
        else:
❼           comicUrl = comicElem[0].get('src')
            # Download the image.
            print('Downloading image %s...' % (comicUrl))
❽           res = requests.get(comicUrl)
            res.raise_for_status()

            # Save the image to ./xkcd.
            imageFile = open(os.path.join('xkcd', os.path.basename(comicUrl)), 'wb')
            for chunk in res.iter_content(100000):
                imageFile.write(chunk)
            imageFile.close()

# TODO: Create and start the Thread objects.
# TODO: Wait for all threads to end.
```

After importing the modules we need, we make a directory to store comics in ❶ and start defining `downloadxkcd()` ❷. We loop through all the numbers in the specified range ❸ and download each page ❹. We use Beautiful Soup to look through the HTML of each page ❺ and find the comic image ❻. If no comic image is found on a page, we print a message. Otherwise, we get the URL of the image ❼ and download the image ❽. Finally, we save the image to the directory we created.

Step 2: Create and Start Threads

Now that we've defined `downloadXkcd()`, we'll create the multiple threads that each call `downloadXkcd()` to download different ranges of comics from the XKCD website. Add the following code to *multidownloadXkcd.py* after the `downloadXkcd()` function definition:

```
#! python3
# multidownloadXkcd.py - Downloads XKCD comics using multiple threads.

--snip--

# Create and start the Thread objects.
downloadThreads = []             # a list of all the Thread objects
for i in range(0, 1400, 100):    # loops 14 times, creates 14 threads
    downloadThread = threading.Thread(target=downloadXkcd, args=(i, i + 99))
    downloadThreads.append(downloadThread)
    downloadThread.start()
```

First we make an empy list `downloadThreads`; the list will help us keep track of the many `Thread` objects we'll create. Then we start our `for` loop. Each time through the loop, we create a `Thread` object with `threading.Thread()`, append the `Thread` object to the list, and call `start()` to start running `downloadXkcd()` in the new thread. Since the `for` loop sets the `i` variable from `0` to `1400` at steps of `100`, `i` will be set to `0` on the first iteration, `100` on the second iteration, `200` on the third, and so on. Since we pass `args=(i, i + 99)` to `threading.Thread()`, the two arguments passed to `downloadXkcd()` will be `0` and `99` on the first iteration, `100` and `199` on the second iteration, `200` and `299` on the third, and so on.

As the Thread object's start() method is called and the new thread begins to run the code inside downloadXkcd(), the main thread will continue to the next iteration of the for loop and create the next thread.

Step 3: Wait for All Threads to End

The main thread moves on as normal while the other threads we create download comics. But say there's some code you don't want to run in the main thread until all the threads have completed. Calling a Thread object's join() method will block until that thread has finished. By using a for loop to iterate over all the Thread objects in the downloadThreads list, the main thread can call the join() method on each of the other threads. Add the following to the bottom of your program:

```
#! python3
# multidownloadXkcd.py - Downloads XKCD comics using multiple threads.

--snip--

# Wait for all threads to end.
for downloadThread in downloadThreads:
    downloadThread.join()
print('Done.')
```

The 'Done.' string will not be printed until all of the join() calls have returned. If a Thread object has already completed when its join() method is called, then the method will simply return immediately. If you wanted to extend this program with code that runs only after all of the comics downloaded, you could replace the print('Done.') line with your new code.

Launching Other Programs from Python

Your Python program can start other programs on your computer with the Popen() function in the built-in subprocess module. (The *P* in the name of the Popen() function stands for *process*.) If you have multiple instances of an application open, each of those instances is a separate process of the same program. For example, if you open multiple windows of your web browser at the same time, each of those windows is a different process of the web browser program. See Figure 15-1 for an example of multiple calculator processes open at once.

Every process can have multiple threads. Unlike threads, a process cannot directly read and write another process's variables. If you think of a multithreaded program as having multiple fingers following source code, then having multiple processes of the same program open is like having a friend with a separate copy of the program's source code. You are both independently executing the same program.

If you want to start an external program from your Python script, pass the program's filename to subprocess.Popen(). (On Windows, right-click the application's **Start** menu item and select **Properties** to view the application's

filename. On OS X, CTRL-click the application and select **Show Package Contents** to find the path to the executable file.) The `Popen()` function will then immediately return. Keep in mind that the launched program is not run in the same thread as your Python program.

Figure 15-1: Six running processes of the same calculator program

On a Windows computer, enter the following into the interactive shell:

```
>>> import subprocess
>>> subprocess.Popen('C:\\Windows\\System32\\calc.exe')
<subprocess.Popen object at 0x0000000003055A58>
```

On Ubuntu Linux, you would enter the following:

```
>>> import subprocess
>>> subprocess.Popen('/usr/bin/gnome-calculator')
<subprocess.Popen object at 0x7f2bcf93b20>
```

On OS X, the process is slightly different. See "Opening Files with Default Applications" on page 355.

The return value is a `Popen` object, which has two useful methods: `poll()` and `wait()`.

You can think of the `poll()` method as asking your friend if she's finished running the code you gave her. The `poll()` method will return `None` if the process is still running at the time `poll()` is called. If the program has terminated, it will return the process's integer *exit code*. An exit code is used

to indicate whether the process terminated without errors (an exit code of 0) or whether an error caused the process to terminate (a nonzero exit code—generally 1, but it may vary depending on the program).

The wait() method is like waiting for your friend to finish working on her code before you keep working on yours. The wait() method will block until the launched process has terminated. This is helpful if you want your program to pause until the user finishes with the other program. The return value of wait() is the process's integer exit code.

On Windows, enter the following into the interactive shell. Note that the wait() call will block until you quit the launched calculator program.

```
❶ >>> calcProc = subprocess.Popen('c:\\Windows\\System32\\calc.exe')
❷ >>> calcProc.poll() == None
  True
❸ >>> calcProc.wait()
  0
  >>> calcProc.poll()
  0
```

Here we open a calculator process ❶. While it's still running, we check if poll() returns None ❷. It should, as the process is still running. Then we close the calculator program and call wait() on the terminated process ❸. wait() and poll() now return 0, indicating that the process terminated without errors.

Passing Command Line Arguments to Popen()

You can pass command line arguments to processes you create with Popen(). To do so, you pass a list as the sole argument to Popen(). The first string in this list will be the executable filename of the program you want to launch; all the subsequent strings will be the command line arguments to pass to the program when it starts. In effect, this list will be the value of sys.argv for the launched program.

Most applications with a graphical user interface (GUI) don't use command line arguments as extensively as command line–based or terminal-based programs do. But most GUI applications will accept a single argument for a file that the applications will immediately open when they start. For example, if you're using Windows, create a simple text file called *C:\hello.txt* and then enter the following into the interactive shell:

```
>>> subprocess.Popen(['C:\\Windows\\notepad.exe', 'C:\\hello.txt'])
<subprocess.Popen object at 0x00000000032DCEB8>
```

This will not only launch the Notepad application but also have it immediately open the *C:\hello.txt* file.

Task Scheduler, launchd, and cron

If you are computer savvy, you may know about Task Scheduler on Windows, launchd on OS X, or the cron scheduler on Linux. These well-documented

and reliable tools all allow you to schedule applications to launch at specific times. If you'd like to learn more about them, you can find links to tutorials at *http://nostarch.com/automatestuff/*.

Using your operating system's built-in scheduler saves you from writing your own clock-checking code to schedule your programs. However, use the `time.sleep()` function if you just need your program to pause briefly. Or instead of using the operating system's scheduler, your code can loop until a certain date and time, calling `time.sleep(1)` each time through the loop.

Opening Websites with Python

The `webbrowser.open()` function can launch a web browser from your program to a specific website, rather than opening the browser application with `subprocess.Popen()`. See "Project: *mapIt.py* with the `webbrowser` Module" on page 234 for more details.

Running Other Python Scripts

You can launch a Python script from Python just like any other application. You just have to pass the *python.exe* executable to `Popen()` and the filename of the *.py* script you want to run as its argument. For example, the following would run the *hello.py* script from Chapter 1:

```
>>> subprocess.Popen(['C:\\python34\\python.exe', 'hello.py'])
<subprocess.Popen object at 0x000000000331CF28>
```

Pass `Popen()` a list containing a string of the Python executable's path and a string of the script's filename. If the script you're launching needs command line arguments, add them to the list after the script's filename. The location of the Python executable on Windows is *C:\python34\python.exe*. On OS X, it is */Library/Frameworks/Python.framework/Versions/3.3/bin/python3*. On Linux, it is */usr/bin/python3*.

Unlike importing the Python program as a module, when your Python program launches another Python program, the two are run in separate processes and will not be able to share each other's variables.

Opening Files with Default Applications

Double-clicking a *.txt* file on your computer will automatically launch the application associated with the *.txt* file extension. Your computer will have several of these file extension associations set up already. Python can also open files this way with `Popen()`.

Each operating system has a program that performs the equivalent of double-clicking a document file to open it. On Windows, this is the `start` program. On OS X, this is the `open` program. On Ubuntu Linux, this is the `see` program. Enter the following into the interactive shell, passing `'start'`, `'open'`, or `'see'` to `Popen()` depending on your system:

```
>>> fileObj = open('hello.txt', 'w')
>>> fileObj.write('Hello world!')
12
```

```
>>> fileObj.close()
>>> import subprocess
>>> subprocess.Popen(['start', 'hello.txt'], shell=True)
```

Here we write `Hello world!` to a new *hello.txt* file. Then we call `Popen()`, passing it a list containing the program name (in this example, `'start'` for Windows) and the filename. We also pass the `shell=True` keyword argument, which is needed only on Windows. The operating system knows all of the file associations and can figure out that it should launch, say, *Notepad.exe* to handle the *hello.txt* file.

On OS X, the `open` program is used for opening both document files and programs. Enter the following into the interactive shell if you have a Mac:

```
>>> subprocess.Popen(['open', '/Applications/Calculator.app/'])
<subprocess.Popen object at 0x10202ff98>
```

The Calculator app should open.

THE UNIX PHILOSOPHY

Programs well designed to be launched by other programs become more powerful than their code alone. The *Unix philosophy* is a set of software design principles established by the programmers of the Unix operating system (on which the modern Linux and OS X are built). It says that it's better to write small, limited-purpose programs that can interoperate, rather than large, feature-rich applications. The smaller programs are easier to understand, and by being interoperable, they can be the building blocks of much more powerful applications.

Smartphone apps follow this approach as well. If your restaurant app needs to display directions to a café, the developers didn't reinvent the wheel by writing their own map code. The restaurant app simply launches a map app while passing it the café's address, just as your Python code would call a function and pass it arguments.

The Python programs you've been writing in this book mostly fit the Unix philosophy, especially in one important way: They use command line arguments rather than `input()` function calls. If all the information your program needs can be supplied up front, it is preferable to have this information passed as command line arguments rather than waiting for the user to type it in. This way, the command line arguments can be entered by a human user or supplied by another program. This interoperable approach will make your programs reusable as part of another program.

The sole exception is that you don't want passwords passed as command line arguments, since the command line may record them as part of its command history feature. Instead, your program should call the `input()` function when it needs you to enter a password.

You can read more about Unix philosophy at *https://en.wikipedia.org/wiki/Unix_philosophy/*.

Project: Simple Countdown Program

Just like it's hard to find a simple stopwatch application, it can be hard to find a simple countdown application. Let's write a countdown program that plays an alarm at the end of the countdown.

At a high level, here's what your program will do:

- Count down from 60.
- Play a sound file (*alarm.wav*) when the countdown reaches zero.

This means your code will need to do the following:

- Pause for one second in between displaying each number in the countdown by calling `time.sleep()`.
- Call `subprocess.Popen()` to open the sound file with the default application.

Open a new file editor window and save it as *countdown.py*.

Step 1: Count Down

This program will require the `time` module for the `time.sleep()` function and the `subprocess` module for the `subprocess.Popen()` function. Enter the following code and save the file as *countdown.py*:

```
#! python3
# countdown.py - A simple countdown script.

import time, subprocess

❶ timeLeft = 60
  while timeLeft > 0:
❷     print(timeLeft, end='')
❸     time.sleep(1)
❹     timeLeft = timeLeft - 1

# TODO: At the end of the countdown, play a sound file.
```

After importing `time` and `subprocess`, make a variable called `timeLeft` to hold the number of seconds left in the countdown ❶. It can start at 60—or you can change the value here to whatever you need or even have it get set from a command line argument.

In a `while` loop, you display the remaining count ❷, pause for one second ❸, and then decrement the `timeLeft` variable ❹ before the loop starts over again. The loop will keep looping as long as `timeLeft` is greater than `0`. After that, the countdown will be over.

Step 2: Play the Sound File

While there are third-party modules to play sound files of various formats, the quick and easy way is to just launch whatever application the user already

uses to play sound files. The operating system will figure out from the *.wav* file extension which application it should launch to play the file. This *.wav* file could easily be some other sound file format, such as *.mp3* or *.ogg*.

You can use any sound file that is on your computer to play at the end of the countdown, or you can download *alarm.wav* from *http://nostarch.com/automatestuff/*.

Add the following to your code:

```
#! python3
# countdown.py - A simple countdown script.

import time, subprocess

--snip--

# At the end of the countdown, play a sound file.
subprocess.Popen(['start', 'alarm.wav'], shell=True)
```

After the `while` loop finishes, *alarm.wav* (or the sound file you choose) will play to notify the user that the countdown is over. On Windows, be sure to include `'start'` in the list you pass to `Popen()` and pass the keyword argument `shell=True`. On OS X, pass `'open'` instead of `'start'` and remove `shell=True`.

Instead of playing a sound file, you could save a text file somewhere with a message like *Break time is over!* and use `Popen()` to open it at the end of the countdown. This will effectively create a pop-up window with a message. Or you could use the `webbrowser.open()` function to open a specific website at the end of the countdown. Unlike some free countdown application you'd find online, your own countdown program's alarm can be anything you want!

Ideas for Similar Programs

A countdown is a simple delay before continuing the program's execution. This can also be used for other applications and features, such as the following:

- Use `time.sleep()` to give the user a chance to press CTRL-C to cancel an action, such as deleting files. Your program can print a "Press CTRL-C to cancel" message and then handle any `KeyboardInterrupt` exceptions with `try` and `except` statements.
- For a long-term countdown, you can use `timedelta` objects to measure the number of days, hours, minutes, and seconds until some point (a birthday? an anniversary?) in the future.

Summary

The Unix epoch (January 1, 1970, at midnight, UTC) is a standard reference time for many programming languages, including Python. While the `time.time()` function module returns an epoch timestamp (that is, a float value of the number of seconds since the Unix epoch), the `datetime` module is better for performing date arithmetic and formatting or parsing strings with date information.

The `time.sleep()` function will block (that is, not return) for a certain number of seconds. It can be used to add pauses to your program. But if you want to schedule your programs to start at a certain time, the instructions at *http://nostarch.com/automatestuff/* can tell you how to use the scheduler already provided by your operating system.

The `threading` module is used to create multiple threads, which is useful when you need to download multiple files or do other tasks simultaneously. But make sure the thread reads and writes only local variables, or you might run into concurrency issues.

Finally, your Python programs can launch other applications with the `subprocess.Popen()` function. Command line arguments can be passed to the `Popen()` call to open specific documents with the application. Alternatively, you can use the `start`, `open`, or `see` program with `Popen()` to use your computer's file associations to automatically figure out which application to use to open a document. By using the other applications on your computer, your Python programs can leverage their capabilities for your automation needs.

Practice Questions

1. What is the Unix epoch?
2. What function returns the number of seconds since the Unix epoch?
3. How can you pause your program for exactly 5 seconds?
4. What does the `round()` function return?
5. What is the difference between a `datetime` object and a `timedelta` object?
6. Say you have a function named `spam()`. How can you call this function and run the code inside it in a separate thread?
7. What should you do to avoid concurrency issues with multiple threads?
8. How can you have your Python program run the *calc.exe* program located in the *C:\Windows\System32* folder?

Practice Projects

For practice, write programs that do the following.

Prettified Stopwatch

Expand the stopwatch project from this chapter so that it uses the `rjust()` and `ljust()` string methods to "prettify" the output. (These methods were covered in Chapter 6.) Instead of output such as this:

```
Lap #1: 3.56 (3.56)
Lap #2: 8.63 (5.07)
Lap #3: 17.68 (9.05)
Lap #4: 19.11 (1.43)
```

. . . the output will look like this:

```
Lap # 1:   3.56 (  3.56)
Lap # 2:   8.63 (  5.07)
Lap # 3:  17.68 (  9.05)
Lap # 4:  19.11 (  1.43)
```

Note that you will need string versions of the `lapNum`, `lapTime`, and `totalTime` integer and float variables in order to call the string methods on them.

Next, use the `pyperclip` module introduced in Chapter 6 to copy the text output to the clipboard so the user can quickly paste the output to a text file or email.

Scheduled Web Comic Downloader

Write a program that checks the websites of several web comics and automatically downloads the images if the comic was updated since the program's last visit. Your operating system's scheduler (Scheduled Tasks on Windows, launchd on OS X, and cron on Linux) can run your Python program once a day. The Python program itself can download the comic and then copy it to your desktop so that it is easy to find. This will free you from having to check the website yourself to see whether it has updated. (A list of web comics is available at *http://nostarch.com/automatestuff/*.)

16

SENDING EMAIL AND TEXT MESSAGES

Checking and replying to email is a huge time sink. Of course, you can't just write a program to handle all your email for you, since each message requires its own response. But you can still automate plenty of email-related tasks once you know how to write programs that can send and receive email.

For example, maybe you have a spreadsheet full of customer records and want to send each customer a different form letter depending on their age and location details. Commercial software might not be able to do this for you; fortunately, you can write your own program to send these emails, saving yourself a lot of time copying and pasting form emails.

You can also write programs to send emails and SMS texts to notify you of things even while you're away from your computer. If you're automating a task that takes a couple of hours to do, you don't want to go back to your computer every few minutes to check on the program's status. Instead, the program can just text your phone when it's done—freeing you to focus on more important things while you're away from your computer.

SMTP

Much like HTTP is the protocol used by computers to send web pages across the Internet, *Simple Mail Transfer Protocol (SMTP)* is the protocol used for sending email. SMTP dictates how email messages should be formatted, encrypted, and relayed between mail servers, and all the other details that your computer handles after you click Send. You don't need to know these technical details, though, because Python's `smtplib` module simplifies them into a few functions.

SMTP just deals with sending emails to others. A different protocol, called IMAP, deals with retrieving emails sent to you and is described in "IMAP" on page 366.

Sending Email

You may be familiar with sending emails from Outlook or Thunderbird or through a website such as Gmail or Yahoo! Mail. Unfortunately, Python doesn't offer you a nice graphical user interface like those services. Instead, you call functions to perform each major step of SMTP, as shown in the following interactive shell example.

NOTE *Don't enter this example in IDLE; it won't work because* smtp.example.com, bob@example.com, MY_SECRET_PASSWORD, *and* alice@example.com *are just placeholders. This code is just an overview of the process of sending email with Python.*

```
>>> import smtplib
>>> smtpObj = smtplib.SMTP('smtp.example.com', 587)
>>> smtpObj.ehlo()
(250, b'mx.example.com at your service, [216.172.148.131]\nSIZE 35882577\
n8BITMIME\nSTARTTLS\nENHANCEDSTATUSCODES\nCHUNKING')
>>> smtpObj.starttls()
(220, b'2.0.0 Ready to start TLS')
>>> smtpObj.login('bob@example.com', 'MY_SECRET_PASSWORD')
(235, b'2.7.0 Accepted')
>>> smtpObj.sendmail('bob@example.com', 'alice@example.com', 'Subject: So
long.\nDear Alice, so long and thanks for all the fish. Sincerely, Bob')
{}
>>> smtpObj.quit()
(221, b'2.0.0 closing connection ko10sm23097611pbd.52 - gsmtp')
```

In the following sections, we'll go through each step, replacing the placeholders with your information to connect and log in to an SMTP server, send an email, and disconnect from the server.

Connecting to an SMTP Server

If you've ever set up Thunderbird, Outlook, or another program to connect to your email account, you may be familiar with configuring the SMTP server and port. These settings will be different for each email provider, but a web search for *<your provider> smtp settings* should turn up the server and port to use.

The domain name for the SMTP server will usually be the name of your email provider's domain name, with *smtp.* in front of it. For example, Gmail's SMTP server is at *smtp.gmail.com*. Table 16-1 lists some common email providers and their SMTP servers. (The port is an integer value and will almost always be 587, which is used by the command encryption standard, TLS.)

Table 16-1: Email Providers and Their SMTP Servers

Provider	SMTP server domain name
Gmail	*smtp.gmail.com*
Outlook.com/Hotmail.com	*smtp-mail.outlook.com*
Yahoo Mail	*smtp.mail.yahoo.com*
AT&T	*smpt.mail.att.net* (port 465)
Comcast	*smtp.comcast.net*
Verizon	*smtp.verizon.net* (port 465)

Once you have the domain name and port information for your email provider, create an `SMTP` object by calling `smptlib.SMTP()`, passing the domain name as a string argument, and passing the port as an integer argument. The `SMTP` object represents a connection to an SMTP mail server and has methods for sending emails. For example, the following call creates an `SMTP` object for connecting to Gmail:

```
>>> smtpObj = smtplib.SMTP('smtp.gmail.com', 587)
>>> type(smtpObj)
<class 'smtplib.SMTP'>
```

Entering `type(smtpObj)` shows you that there's an `SMTP` object stored in `smtpObj`. You'll need this `SMTP` object in order to call the methods that log you in and send emails. If the `smptlib.SMTP()` call is not successful, your SMTP server might not support TLS on port 587. In this case, you will need to create an `SMTP` object using `smtplib.SMTP_SSL()` and port 465 instead.

```
>>> smtpObj = smtplib.SMTP_SSL('smtp.gmail.com', 465)
```

NOTE *If you are not connected to the Internet, Python will raise a `socket.gaierror: [Errno 11004] getaddrinfo failed` or similar exception.*

For your programs, the differences between TLS and SSL aren't important. You only need to know which encryption standard your SMTP server uses so you know how to connect to it. In all of the interactive shell examples that follow, the `smtpObj` variable will contain an `SMTP` object returned by the `smtplib.SMTP()` or `smtplib.SMTP_SSL()` function.

Sending the SMTP "Hello" Message

Once you have the `SMTP` object, call its oddly named `ehlo()` method to "say hello" to the SMTP email server. This greeting is the first step in SMTP and is important for establishing a connection to the server. You don't need to know the specifics of these protocols. Just be sure to call the `ehlo()` method first thing after getting the `SMTP` object or else the later method calls will result in errors. The following is an example of an `ehlo()` call and its return value:

```
>>> smtpObj.ehlo()
(250, b'mx.google.com at your service, [216.172.148.131]\nSIZE 35882577\
n8BITMIME\nSTARTTLS\nENHANCEDSTATUSCODES\nCHUNKING')
```

If the first item in the returned tuple is the integer `250` (the code for "success" in SMTP), then the greeting succeeded.

Starting TLS Encryption

If you are connecting to port 587 on the SMTP server (that is, you're using TLS encryption), you'll need to call the `starttls()` method next. This required step enables encryption for your connection. If you are connecting to port 465 (using SSL), then encryption is already set up, and you should skip this step.

Here's an example of the `starttls()` method call:

```
>>> smtpObj.starttls()
(220, b'2.0.0 Ready to start TLS')
```

`starttls()` puts your SMTP connection in TLS mode. The `220` in the return value tells you that the server is ready.

Logging in to the SMTP Server

Once your encrypted connection to the SMTP server is set up, you can log in with your username (usually your email address) and email password by calling the `login()` method.

```
>>> smtpObj.login('my_email_address@gmail.com', 'MY_SECRET_PASSWORD')
(235, b'2.7.0 Accepted')
```

GMAIL'S APPLICATION-SPECIFIC PASSWORDS

Gmail has an additional security feature for Google accounts called *application-specific passwords*. If you receive an `Application-specific password required` error message when your program tries to log in, you will have to set up one of these passwords for your Python script. Check out the resources at *http://nostarch.com/automatestuff/* for detailed directions on how to set up an application-specific password for your Google account.

Pass a string of your email address as the first argument and a string of your password as the second argument. The `235` in the return value means authentication was successful. Python will raise an `smtplib.SMTPAuthenticationError` exception for incorrect passwords.

WARNING *Be careful about putting passwords in your source code. If anyone ever copies your program, they'll have access to your email account! It's a good idea to call* `input()` *and have the user type in the password. It may be inconvenient to have to enter a password each time you run your program, but this approach will prevent you from leaving your password in an unencrypted file on your computer where a hacker or laptop thief could easily get it.*

Sending an Email

Once you are logged in to your email provider's SMTP server, you can call the `sendmail()` method to actually send the email. The `sendmail()` method call looks like this:

```
>>> smtpObj.sendmail('my_email_address@gmail.com', 'recipient@example.com',
'Subject: So long.\nDear Alice, so long and thanks for all the fish. Sincerely,
Bob')
{}
```

The `sendmail()` method requires three arguments.

- Your email address as a string (for the email's "from" address)
- The recipient's email address as a string or a list of strings for multiple recipients (for the "to" address)
- The email body as a string

The start of the email body string *must* begin with `'Subject: \n'` for the subject line of the email. The `'\n'` newline character separates the subject line from the main body of the email.

The return value from `sendmail()` is a dictionary. There will be one key-value pair in the dictionary for each recipient for whom email delivery *failed*. An empty dictionary means all recipients were *successfully* sent the email.

Disconnecting from the SMTP Server

Be sure to call the `quit()` method when you are done sending emails. This will disconnect your program from the SMTP server.

```
>>> smtpObj.quit()
(221, b'2.0.0 closing connection ko10sm23097611pbd.52 - gsmtp')
```

The `221` in the return value means the session is ending.

To review all the steps for connecting and logging in to the server, sending email, and disconnection, see "Sending Email" on page 362.

IMAP

Just as SMTP is the protocol for sending email, the *Internet Message Access Protocol (IMAP)* specifies how to communicate with an email provider's server to retrieve emails sent to your email address. Python comes with an `imaplib` module, but in fact the third-party `imapclient` module is easier to use. This chapter provides an introduction to using IMAPClient; the full documentation is at *http://imapclient.readthedocs.org/*.

The `imapclient` module downloads emails from an IMAP server in a rather complicated format. Most likely, you'll want to convert them from this format into simple string values. The `pyzmail` module does the hard job of parsing these email messages for you. You can find the complete documentation for PyzMail at *http://www.magiksys.net/pyzmail/*.

Install `imapclient` and `pyzmail` from a Terminal window. Appendix A has steps on how to install third-party modules.

Retrieving and Deleting Emails with IMAP

Finding and retrieving an email in Python is a multistep process that requires both the `imapclient` and `pyzmail` third-party modules. Just to give you an overview, here's a full example of logging in to an IMAP server, searching for emails, fetching them, and then extracting the text of the email messages from them.

```
>>> import imapclient
>>> imapObj = imapclient.IMAPClient('imap.gmail.com', ssl=True)
>>> imapObj.login('my_email_address@gmail.com', 'MY_SECRET_PASSWORD')
'my_email_address@gmail.com Jane Doe authenticated (Success)'
>>> imapObj.select_folder('INBOX', readonly=True)
>>> UIDs = imapObj.search(['SINCE 05-Jul-2014'])
>>> UIDs
[40032, 40033, 40034, 40035, 40036, 40037, 40038, 40039, 40040, 40041]
>>> rawMessages = imapObj.fetch([40041], ['BODY[]', 'FLAGS'])
>>> import pyzmail
>>> message = pyzmail.PyzMessage.factory(rawMessages[40041]['BODY[]'])
>>> message.get_subject()
'Hello!'
```

```
>>> message.get_addresses('from')
[('Edward Snowden', 'esnowden@nsa.gov')]
>>> message.get_addresses('to')
[(Jane Doe', 'jdoe@example.com')]
>>> message.get_addresses('cc')
[]
>>> message.get_addresses('bcc')
[]
>>> message.text_part != None
True
>>> message.text_part.get_payload().decode(message.text_part.charset)
'Follow the money.\r\n\r\n-Ed\r\n'
>>> message.html_part != None
True
>>> message.html_part.get_payload().decode(message.html_part.charset)
'<div dir="ltr"><div>So long, and thanks for all the fish!<br><br></div>-
Al<br></div>\r\n'
>>> imapObj.logout()
```

You don't have to memorize these steps. After we go through each step in detail, you can come back to this overview to refresh your memory.

Connecting to an IMAP Server

Just like you needed an `SMTP` object to connect to an SMTP server and send email, you need an `IMAPClient` object to connect to an IMAP server and receive email. First you'll need the domain name of your email provider's IMAP server. This will be different from the SMTP server's domain name. Table 16-2 lists the IMAP servers for several popular email providers.

Table 16-2: Email Providers and Their IMAP Servers

Provider	IMAP server domain name
Gmail	*imap.gmail.com*
Outlook.com/Hotmail.com	*imap-mail.outlook.com*
Yahoo Mail	*imap.mail.yahoo.com*
AT&T	*imap.mail.att.net*
Comcast	*imap.comcast.net*
Verizon	*incoming.verizon.net*

Once you have the domain name of the IMAP server, call the `imapclient.IMAPClient()` function to create an `IMAPClient` object. Most email providers require SSL encryption, so pass the `ssl=True` keyword argument. Enter the following into the interactive shell (using your provider's domain name):

```
>>> import imapclient
>>> imapObj = imapclient.IMAPClient('imap.gmail.com', ssl=True)
```

In all of the interactive shell examples in the following sections, the `imapObj` variable will contain an `IMAPClient` object returned from the `imapclient.IMAPClient()` function. In this context, a *client* is the object that connects to the server.

Logging in to the IMAP Server

Once you have an `IMAPClient` object, call its `login()` method, passing in the username (this is usually your email address) and password as strings.

```
>>> imapObj.login('my_email_address@gmail.com', 'MY_SECRET_PASSWORD')
'my_email_address@gmail.com Jane Doe authenticated (Success)'
```

WARNING *Remember, never write a password directly into your code! Instead, design your program to accept the password returned from `input()`.*

If the IMAP server rejects this username/password combination, Python will raise an `imaplib.error` exception. For Gmail accounts, you may need to use an application-specific password; for more details, see "Gmail's Application-Specific Passwords" on page 365.

Searching for Email

Once you're logged on, actually retrieving an email that you're interested in is a two-step process. First, you must select a folder you want to search through. Then, you must call the `IMAPClient` object's `search()` method, passing in a string of IMAP search keywords.

Selecting a Folder

Almost every account has an `INBOX` folder by default, but you can also get a list of folders by calling the `IMAPClient` object's `list_folders()` method. This returns a list of tuples. Each tuple contains information about a single folder. Continue the interactive shell example by entering the following:

```
>>> import pprint
>>> pprint.pprint(imapObj.list_folders())
[((\\HasNoChildren',), '/', 'Drafts'),
 ((\\HasNoChildren',), '/', 'Filler'),
 ((\\HasNoChildren',), '/', 'INBOX'),
 ((\\HasNoChildren',), '/', 'Sent'),
--snip--
 ((\\HasNoChildren', \\Flagged'), '/', '[Gmail]/Starred'),
 ((\\HasNoChildren', \\Trash'), '/', '[Gmail]/Trash')]
```

This is what your output might look like if you have a Gmail account. (Gmail calls its folders *labels*, but they work the same way as folders.) The three values in each of the tuples—for example, `(('\\HasNoChildren',), '/', 'INBOX')`—are as follows:

- A tuple of the folder's flags. (Exactly what these flags represent is beyond the scope of this book, and you can safely ignore this field.)
- The delimiter used in the name string to separate parent folders and subfolders.
- The full name of the folder.

To select a folder to search through, pass the folder's name as a string into the `IMAPClient` object's `select_folder()` method.

```
>>> imapObj.select_folder('INBOX', readonly=True)
```

You can ignore `select_folder()`'s return value. If the selected folder does not exist, Python will raise an `imaplib.error` exception.

The `readonly=True` keyword argument prevents you from accidentally making changes or deletions to any of the emails in this folder during the subsequent method calls. Unless you *want* to delete emails, it's a good idea to always set `readonly` to `True`.

Performing the Search

With a folder selected, you can now search for emails with the `IMAPClient` object's `search()` method. The argument to `search()` is a list of strings, each formatted to the IMAP's search keys. Table 16-3 describes the various search keys.

Table 16-3: IMAP Search Keys

Search key	Meaning
`'ALL'`	Returns all messages in the folder. You may run in to `imaplib` size limits if you request all the messages in a large folder. See "Size Limits" on page 371.
`'BEFORE date'`, `'ON date'`, `'SINCE date'`	These three search keys return, respectively, messages that were received by the IMAP server before, on, or after the given *date*. The date must be formatted like `05-Jul-2015`. Also, while `'SINCE 05-Jul-2015'` will match messages on and after July 5, `'BEFORE 05-Jul-2015'` will match only messages before July 5 but not on July 5 itself.
`'SUBJECT string'`, `'BODY string'`, `'TEXT string'`	Returns messages where *string* is found in the subject, body, or either, respectively. If *string* has spaces in it, then enclose it with double quotes: `'TEXT "search with spaces"'`.

(continued)

Table 16-3 (continued)

Search key	Meaning
`'FROM string'`, `'TO string'`, `'CC string'`, `'BCC string'`	Returns all messages where *string* is found in the "from" emailaddress, "to" addresses, "cc" (carbon copy) addresses, or "bcc" (blind carbon copy) addresses, respectively. If there are multiple email addresses in *string*, then separate them with spaces and enclose them all with double quotes: `'CC "firstcc@example.com secondcc@example.com"'`.
`'SEEN'`, `'UNSEEN'`	Returns all messages with and without the *\Seen* flag, respectively. An email obtains the *\Seen* flag if it has been accessed with a `fetch()` method call (described later) or if it is clicked when you're checking your email in an email program or web browser. It's more common to say the email has been "read" rather than "seen," but they mean the same thing.
`'ANSWERED'`, `'UNANSWERED'`	Returns all messages with and without the *\Answered* flag, respectively. A message obtains the *\Answered* flag when it is replied to.
`'DELETED'`, `'UNDELETED'`	Returns all messages with and without the *\Deleted* flag, respectively. Email messages deleted with the `delete_messages()` method are given the *\Deleted* flag but are not permanently deleted until the `expunge()` method is called (see "Deleting Emails" on page 375). Note that some email providers, such as Gmail, automatically expunge emails.
`'DRAFT'`, `'UNDRAFT'`	Returns all messages with and without the *\Draft* flag, respectively. Draft messages are usually kept in a separate `Drafts` folder rather than in the `INBOX` folder.
`'FLAGGED'`, `'UNFLAGGED'`	Returns all messages with and without the *\Flagged* flag, respectively. This flag is usually used to mark email messages as "Important" or "Urgent."
`'LARGER N'`, `'SMALLER N'`	Returns all messages larger or smaller than *N* bytes, respectively.
`'NOT search-key'`	Returns the messages that *search-key* would *not* have returned.
`'OR search-key1 search-key2'`	Returns the messages that match *either* the first or second *search-key*.

Note that some IMAP servers may have slightly different implementations for how they handle their flags and search keys. It may require some experimentation in the interactive shell to see exactly how they behave.

You can pass multiple IMAP search key strings in the list argument to the `search()` method. The messages returned are the ones that match *all* the search keys. If you want to match *any* of the search keys, use the `OR` search key. For the `NOT` and `OR` search keys, one and two complete search keys follow the `NOT` and `OR`, respectively.

Here are some example `search()` method calls along with their meanings:

`imapObj.search(['ALL'])` Returns every message in the currently selected folder.

`imapObj.search(['ON 05-Jul-2015'])` Returns every message sent on July 5, 2015.

`imapObj.search(['SINCE 01-Jan-2015', 'BEFORE 01-Feb-2015', 'UNSEEN'])` Returns every message sent in January 2015 that is unread. (Note that this means *on and after* January 1 and up to *but not including* February 1.)

`imapObj.search(['SINCE 01-Jan-2015', 'FROM alice@example.com'])` Returns every message from *alice@example.com* sent since the start of 2015.

`imapObj.search(['SINCE 01-Jan-2015', 'NOT FROM alice@example.com'])` Returns every message sent from everyone except *alice@example.com* since the start of 2015.

`imapObj.search(['OR FROM alice@example.com FROM bob@example.com'])` Returns every message ever sent from *alice@example.com* or *bob@example.com*.

`imapObj.search(['FROM alice@example.com', 'FROM bob@example.com'])` Trick example! This search will never return any messages, because messages must match *all* search keywords. Since there can be only one "from" address, it is impossible for a message to be from both *alice@example.com* and *bob@example.com*.

The `search()` method doesn't return the emails themselves but rather unique IDs (UIDs) for the emails, as integer values. You can then pass these UIDs to the `fetch()` method to obtain the email content.

Continue the interactive shell example by entering the following:

```
>>> UIDs = imapObj.search(['SINCE 05-Jul-2015'])
>>> UIDs
[40032, 40033, 40034, 40035, 40036, 40037, 40038, 40039, 40040, 40041]
```

Here, the list of message IDs (for messages received July 5 onward) returned by `search()` is stored in `UIDs`. The list of UIDs returned on your computer will be different from the ones shown here; they are unique to a particular email account. When you later pass UIDs to other function calls, use the UID values you received, not the ones printed in this book's examples.

Size Limits

If your search matches a large number of email messages, Python might raise an exception that says `imaplib.error: got more than 10000 bytes`. When this happens, you will have to disconnect and reconnect to the IMAP server and try again.

This limit is in place to prevent your Python programs from eating up too much memory. Unfortunately, the default size limit is often too small. You can change this limit from 10,000 bytes to 10,000,000 bytes by running this code:

```
>>> import imaplib
>>> imaplib._MAXLINE = 10000000
```

This should prevent this error message from coming up again. You may want to make these two lines part of every IMAP program you write.

USING IMAPCLIENT'S GMAIL_SEARCH() METHOD

If you are logging in to the *imap.gmail.com* server to access a Gmail account, the `IMAPClient` object provides an extra search function that mimics the search bar at the top of the Gmail web page, as highlighted in Figure 16-1.

Figure 16-1: The search bar at the top of the Gmail web page

Instead of searching with IMAP search keys, you can use Gmail's more sophisticated search engine. Gmail does a good job of matching closely related words (for example, a search for *driving* will also match *drive* and *drove*) and sorting the search results by most significant matches. You can also use Gmail's advanced search operators (see *http://nostarch.com/automatestuff/* for more information). If you are logging in to a Gmail account, pass the search terms to the `gmail_search()` method instead of the `search()` method, like in the following interactive shell example:

```
>>> UIDs = imapObj.gmail_search('meaning of life')
>>> UIDs
[42]
```

Ah, yes—*there's* that email with the meaning of life! I was looking for that.

Fetching an Email and Marking It As Read

Once you have a list of UIDs, you can call the `IMAPClient` object's `fetch()` method to get the actual email content.

The list of UIDs will be `fetch()`'s first argument. The second argument should be the list `['BODY[]']`, which tells `fetch()` to download all the body content for the emails specified in your UID list.

Let's continue our interactive shell example.

```
>>> rawMessages = imapObj.fetch(UIDs, ['BODY[]'])
>>> import pprint
>>> pprint.pprint(rawMessages)
{40040: {'BODY[]': 'Delivered-To: my_email_address@gmail.com\r\n'
                   'Received: by 10.76.71.167 with SMTP id '
--snip--
                   '\r\n'
                   '------=_Part_6000970_707736290.1404819487066--\r\n',
         'SEQ': 5430}}
```

Import `pprint` and pass the return value from `fetch()`, stored in the variable `rawMessages`, to `pprint.pprint()` to "pretty print" it, and you'll see that this return value is a nested dictionary of messages with UIDs as the keys. Each message is stored as a dictionary with two keys: `'BODY[]'` and `'SEQ'`. The `'BODY[]'` key maps to the actual body of the email. The `'SEQ'` key is for a *sequence number*, which has a similar role to the UID. You can safely ignore it.

As you can see, the message content in the `'BODY[]'` key is pretty unintelligible. It's in a format called RFC 822, which is designed for IMAP servers to read. But you don't need to understand the RFC 822 format; later in this chapter, the `pyzmail` module will make sense of it for you.

When you selected a folder to search through, you called `select_folder()` with the `readonly=True` keyword argument. Doing this will prevent you from accidentally deleting an email—but it also means that emails will not get marked as read if you fetch them with the `fetch()` method. If you *do* want emails to be marked as read when you fetch them, you will need to pass `readonly=False` to `select_folder()`. If the selected folder is already in read-only mode, you can reselect the current folder with another call to `select_folder()`, this time with the `readonly=False` keyword argument:

```
>>> imapObj.select_folder('INBOX', readonly=False)
```

Getting Email Addresses from a Raw Message

The raw messages returned from the `fetch()` method still aren't very useful to people who just want to read their email. The `pyzmail` module parses these raw messages and returns them as `PyzMessage` objects, which make the subject, body, "To" field, "From" field, and other sections of the email easily accessible to your Python code.

Continue the interactive shell example with the following (using UIDs from your own email account, not the ones shown here):

```
>>> import pyzmail
>>> message = pyzmail.PyzMessage.factory(rawMessages[40041]['BODY[]'])
```

First, import `pyzmail`. Then, to create a `PyzMessage` object of an email, call the `pyzmail.PyzMessage.factory()` function and pass it the `'BODY[]'` section of the raw message. Store the result in `message`. Now `message` contains

a PyzMessage object, which has several methods that make it easy to get the email's subject line, as well as all sender and recipient addresses. The get_subject() method returns the subject as a simple string value. The get_addresses() method returns a list of addresses for the field you pass it. For example, the method calls might look like this:

```
>>> message.get_subject()
'Hello!'
>>> message.get_addresses('from')
[('Edward Snowden', 'esnowden@nsa.gov')]
>>> message.get_addresses('to')
[(Jane Doe', 'my_email_address@gmail.com')]
>>> message.get_addresses('cc')
[]
>>> message.get_addresses('bcc')
[]
```

Notice that the argument for get_addresses() is 'from', 'to', 'cc', or 'bcc'. The return value of get_addresses() is a list of tuples. Each tuple contains two strings: The first is the name associated with the email address, and the second is the email address itself. If there are no addresses in the requested field, get_addresses() returns a blank list. Here, the 'cc' carbon copy and 'bcc' blind carbon copy fields both contained no addresses and so returned empty lists.

Getting the Body from a Raw Message

Emails can be sent as plaintext, HTML, or both. Plaintext emails contain only text, while HTML emails can have colors, fonts, images, and other features that make the email message look like a small web page. If an email is only plaintext, its PyzMessage object will have its html_part attributes set to None. Likewise, if an email is only HTML, its PyzMessage object will have its text_part attribute set to None.

Otherwise, the text_part or html_part value will have a get_payload() method that returns the email's body as a value of the *bytes* data type. (The bytes data type is beyond the scope of this book.) But this *still* isn't a string value that we can use. Ugh! The last step is to call the decode() method on the bytes value returned by get_payload(). The decode() method takes one argument: the message's character encoding, stored in the text_part.charset or html_part.charset attribute. This, finally, will return the string of the email's body.

Continue the interactive shell example by entering the following:

```
❶ >>> message.text_part != None
  True
  >>> message.text_part.get_payload().decode(message.text_part.charset)
❷ 'So long, and thanks for all the fish!\r\n\r\n-Al\r\n'
❸ >>> message.html_part != None
  True
```

```
❹ >>> message.html_part.get_payload().decode(message.html_part.charset)
'<div dir="ltr"><div>So long, and thanks for all the fish!<br><br></div>-Al
<br></div>\r\n'
```

The email we're working with has both plaintext and HTML content, so the `PyzMessage` object stored in `message` has `text_part` and `html_part` attributes not equal to `None` ❶ ❸. Calling `get_payload()` on the message's `text_part` and then calling `decode()` on the bytes value returns a string of the text version of the email ❷. Using `get_payload()` and `decode()` with the message's `html_part` returns a string of the HTML version of the email ❹.

Deleting Emails

To delete emails, pass a list of message UIDs to the `IMAPClient` object's `delete_messages()` method. This marks the emails with the *\Deleted* flag. Calling the `expunge()` method will permanently delete all emails with the *\Deleted* flag in the currently selected folder. Consider the following interactive shell example:

```
❶ >>> imapObj.select_folder('INBOX', readonly=False)
❷ >>> UIDs = imapObj.search(['ON 09-Jul-2015'])
  >>> UIDs
  [40066]
  >>> imapObj.delete_messages(UIDs)
❸ {40066: ('\\Seen', '\\Deleted')}
  >>> imapObj.expunge()
  ('Success', [(5452, 'EXISTS')])
```

Here we select the inbox by calling `select_folder()` on the `IMAPClient` object and passing `'INBOX'` as the first argument; we also pass the keyword argument `readonly=False` so that we can delete emails ❶. We search the inbox for messages received on a specific date and store the returned message IDs in `UIDs` ❷. Calling `delete_message()` and passing it `UIDs` returns a dictionary; each key-value pair is a message ID and a tuple of the message's flags, which should now include *\Deleted* ❸. Calling `expunge()` then permanently deletes messages with the *\Deleted* flag and returns a success message if there were no problems expunging the emails. Note that some email providers, such as Gmail, automatically expunge emails deleted with `delete_messages()` instead of waiting for an expunge command from the IMAP client.

Disconnecting from the IMAP Server

When your program has finished retrieving or deleting emails, simply call the IMAPClient's `logout()` method to disconnect from the IMAP server.

```
>>> imapObj.logout()
```

If your program runs for several minutes or more, the IMAP server may *time out*, or automatically disconnect. In this case, the next method call your program makes on the `IMAPClient` object will raise an exception like the following:

```
imaplib.abort: socket error: [WinError 10054] An existing connection was
forcibly closed by the remote host
```

In this event, your program will have to call `imapclient.IMAPClient()` to connect again.

Whew! That's it. There were a lot of hoops to jump through, but you now have a way to get your Python programs to log in to an email account and fetch emails. You can always consult the overview in "Retrieving and Deleting Emails with IMAP" on page 366 whenever you need to remember all of the steps.

Project: Sending Member Dues Reminder Emails

Say you have been "volunteered" to track member dues for the Mandatory Volunteerism Club. This is a truly boring job, involving maintaining a spreadsheet of everyone who has paid each month and emailing reminders to those who haven't. Instead of going through the spreadsheet yourself and copying and pasting the same email to everyone who is behind on dues, let's—you guessed it—write a script that does this for you.

At a high level, here's what your program will do:

- Read data from an Excel spreadsheet.
- Find all members who have not paid dues for the latest month.
- Find their email addresses and send them personalized reminders.

This means your code will need to do the following:

- Open and read the cells of an Excel document with the `openpyxl` module. (See Chapter 12 for working with Excel files.)
- Create a dictionary of members who are behind on their dues.
- Log in to an SMTP server by calling `smtplib.SMTP()`, `ehlo()`, `starttls()`, and `login()`.
- For all members behind on their dues, send a personalized reminder email by calling the `sendmail()` method.

Open a new file editor window and save it as *sendDuesReminders.py*.

Step 1: Open the Excel File

Let's say the Excel spreadsheet you use to track membership dues payments looks like Figure 16-2 and is in a file named *duesRecords.xlsx*. You can download this file from *http://nostarch.com/automatestuff/*.

	A	B	C	D	E	F	G	H
1	Member	Email	Jan 2014	Feb 2014	Mar 2014	Apr 2014	May 2014	Jun 2014
2	Alice	alice@example.com	paid	paid	paid	paid	paid	
3	Bob	bob@example.com	paid	paid	paid	paid		
4	Carol	carol@example.com	paid	paid	paid	paid	paid	paid
5	David	david@example.com	paid	paid	paid	paid	paid	paid
6	Eve	eve@example.com	paid	paid	paid			
7	Fred	fred@example.com	paid	paid	paid	paid	paid	paid
8								
9								

Figure 16-2: The spreadsheet for tracking member dues payments

This spreadsheet has every member's name and email address. Each month has a column tracking members' payment statuses. The cell for each member is marked with the text *paid* once they have paid their dues.

The program will have to open *duesRecords.xlsx* and figure out the column for the latest month by calling the `get_highest_column()` method. (You can consult Chapter 12 for more information on accessing cells in Excel spreadsheet files with the `openpyxl` module.) Enter the following code into the file editor window:

```
#! python3
# sendDuesReminders.py - Sends emails based on payment status in spreadsheet.

import openpyxl, smtplib, sys

# Open the spreadsheet and get the latest dues status.
❶ wb = openpyxl.load_workbook('duesRecords.xlsx')
❷ sheet = wb.get_sheet_by_name('Sheet1')

❸ lastCol = sheet.get_highest_column()
❹ latestMonth = sheet.cell(row=1, column=lastCol).value

# TODO: Check each member's payment status.

# TODO: Log in to email account.

# TODO: Send out reminder emails.
```

After importing the `openpyxl`, `smtplib`, and `sys` modules, we open our *duesRecords.xlsx* file and store the resulting `Workbook` object in `wb` ❶. Then we get Sheet 1 and store the resulting `Worksheet` object in `sheet` ❷. Now that we have a `Worksheet` object, we can access rows, columns, and cells. We store the highest column in `lastCol` ❸, and we then use row number 1 and `lastCol` to access the cell that should hold the most recent month. We get the value in this cell and store it in `latestMonth` ❹.

Step 2: Find All Unpaid Members

Once you've determined the column number of the latest month (stored in `lastCol`), you can loop through all rows after the first row (which has the column headers) to see which members have the text *paid* in the cell for that month's dues. If the member hasn't paid, you can grab the member's name and email address from columns 1 and 2, respectively. This information will go into the `unpaidMembers` dictionary, which will track all members who haven't paid in the most recent month. Add the following code to *sendDuesReminder.py*.

```
#! python3
# sendDuesReminders.py - Sends emails based on payment status in spreadsheet.

--snip--

# Check each member's payment status.
unpaidMembers = {}
❶ for r in range(2, sheet.get_highest_row() + 1):
❷     payment = sheet.cell(row=r, column=lastCol).value
      if payment != 'paid':
❸         name = sheet.cell(row=r, column=1).value
❹         email = sheet.cell(row=r, column=2).value
❺         unpaidMembers[name] = email
```

This code sets up an empty dictionary `unpaidMembers` and then loops through all the rows after the first ❶. For each row, the value in the most recent column is stored in `payment` ❷. If `payment` is not equal to `'paid'`, then the value of the first column is stored in `name` ❸, the value of the second column is stored in `email` ❹, and `name` and `email` are added to `unpaidMembers` ❺.

Step 3: Send Customized Email Reminders

Once you have a list of all unpaid members, it's time to send them email reminders. Add the following code to your program, except with your real email address and provider information:

```
#! python3
# sendDuesReminders.py - Sends emails based on payment status in spreadsheet.

--snip--

# Log in to email account.
smtpObj = smtplib.SMTP('smtp.gmail.com', 587)
smtpObj.ehlo()
smtpObj.starttls()
smtpObj.login('my_email_address@gmail.com', sys.argv[1])
```

Create an SMTP object by calling smtplib.SMTP() and passing it the domain name and port for your provider. Call ehlo() and starttls(), and then call login() and pass it your email address and sys.argv[1], which will store your password string. You'll enter the password as a command line argument each time you run the program, to avoid saving your password in your source code.

Once your program has logged in to your email account, it should go through the unpaidMembers dictionary and send a personalized email to each member's email address. Add the following to *sendDuesReminders.py*:

```
#! python3
# sendDuesReminders.py - Sends emails based on payment status in spreadsheet.

--snip--

# Send out reminder emails.
for name, email in unpaidMembers.items():
❶    body = "Subject: %s dues unpaid.\nDear %s,\nRecords show that you have not
paid dues for %s. Please make this payment as soon as possible. Thank you!'" %
(latestMonth, name, latestMonth)
❷    print('Sending email to %s...' % email)
❸    sendmailStatus = smtpObj.sendmail('my_email_address@gmail.com', email, body)

❹    if sendmailStatus != {}:
         print('There was a problem sending email to %s: %s' % (email,
         sendmailStatus))
smtpObj.quit()
```

This code loops through the names and emails in unpaidMembers. For each member who hasn't paid, we customize a message with the latest month and the member's name, and store the message in body ❶. We print output saying that we're sending an email to this member's email address ❷. Then we call sendmail(), passing it the from address and the customized message ❸. We store the return value in sendmailStatus.

Remember that the sendmail() method will return a nonempty dictionary value if the SMTP server reported an error sending that particular email. The last part of the for loop at ❹ checks if the returned dictionary is nonempty, and if it is, prints the recipient's email address and the returned dictionary.

After the program is done sending all the emails, the quit() method is called to disconnect from the SMTP server.

When you run the program, the output will look something like this:

```
Sending email to alice@example.com...
Sending email to bob@example.com...
Sending email to eve@example.com...
```

The recipients will receive an email that looks like Figure 16-3.

Figure 16-3: An automatically sent email from sendDuesReminders.py

Sending Text Messages with Twilio

Most people are more likely to be near their phones than their computers, so text messages can be a more immediate and reliable way of sending notifications than email. Also, the short length of text messages makes it more likely that a person will get around to reading them.

In this section, you'll learn how to sign up for the free Twilio service and use its Python module to send text messages. Twilio is an *SMS gateway service*, which means it's a service that allows you to send text messages from your programs. Although you will be limited in how many texts you can send per month and the texts will be prefixed with the words *Sent from a Twilio trial account*, this trial service is probably adequate for your personal programs. The free trial is indefinite; you won't have to upgrade to a paid plan later.

Twilio isn't the only SMS gateway service. If you prefer not to use Twilio, you can find alternative services by searching online for *free sms gateway*, *python sms api*, or even *twilio alternatives*.

Before signing up for a Twilio account, install the `twilio` module. Appendix A has more details about installing third-party modules.

NOTE *This section is specific to the United States. Twilio does offer SMS texting services for countries outside of the United States, but those specifics aren't covered in this book. The* `twilio` *module and its functions, however, will work the same outside the United States. See* http://twilio.com/ *for more information.*

Signing Up for a Twilio Account

Go to *http://twilio.com/* and fill out the sign-up form. Once you've signed up for a new account, you'll need to verify a mobile phone number that you want to send texts to. (This verification is necessary to prevent people from using the service to spam random phone numbers with text messages.)

After receiving the text with the verification number, enter it into the Twilio website to prove that you own the mobile phone you are verifying. You will now be able to send texts to this phone number using the `twilio` module.

Twilio provides your trial account with a phone number to use as the sender of text messages. You will need two more pieces of information: your account SID and the auth (authentication) token. You can find this information on the Dashboard page when you are logged in to your Twilio account. These values act as your Twilio username and password when logging in from a Python program.

Sending Text Messages

Once you've installed the `twilio` module, signed up for a Twilio account, verified your phone number, registered a Twilio phone number, and obtained your account SID and auth token, you will finally be ready to send yourself text messages from your Python scripts.

Compared to all the registration steps, the actual Python code is fairly simple. With your computer connected to the Internet, enter the following into the interactive shell, replacing the `accountSID`, `authToken`, `myTwilioNumber`, and `myCellPhone` variable values with your real information:

```
❶ >>> from twilio.rest import TwilioRestClient
  >>> accountSID = 'ACxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
  >>> authToken  = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
❷ >>> twilioCli = TwilioRestClient(accountSID, authToken)
  >>> myTwilioNumber = '+14955551234'
  >>> myCellPhone = '+14955558888'
❸ >>> message = twilioCli.messages.create(body='Mr. Watson - Come here - I want
  to see you.', from_=myTwilioNumber, to=myCellPhone)
```

A few moments after typing the last line, you should receive a text message that reads *Sent from your Twilio trial account - Mr. Watson - Come here - I want to see you.*

Because of the way the `twilio` module is set up, you need to import it using `from twilio.rest import TwilioRestClient`, not just `import twilio` ❶. Store your account SID in `accountSID` and your auth token in `authToken` and then call `TwilioRestClient()` and pass it `accountSID` and `authToken`. The call to `TwilioRestClient()` returns a `TwilioRestClient` object ❷. This object has a `messages` attribute, which in turn has a `create()` method you can use to send text messages. This is the method that will instruct Twilio's servers to send your text message. After storing your Twilio number and cell phone number in `myTwilioNumber` and `myCellPhone`, respectively, call `create()` and pass it keyword arguments specifying the body of the text message, the sender's number (`myTwilioNumber`), and the recipient's number (`myCellPhone`) ❸.

The `Message` object returned from the `create()` method will have information about the text message that was sent. Continue the interactive shell example by entering the following:

```
>>> message.to
'+14955558888'
>>> message.from_
'+14955551234'
>>> message.body
'Mr. Watson - Come here - I want to see you.'
```

The `to`, `from_`, and `body` attributes should hold your cell phone number, Twilio number, and message, respectively. Note that the sending phone number is in the `from_` attribute—with an underscore at the end—not `from`. This is because `from` is a keyword in Python (you've seen it used in the `from`

modulename import * form of import statement, for example), so it cannot be used as an attribute name. Continue the interactive shell example with the following:

```
>>> message.status
'queued'
>>> message.date_created
datetime.datetime(2015, 7, 8, 1, 36, 18)
>>> message.date_sent == None
True
```

The status attribute should give you a string. The date_created and date_sent attributes should give you a datetime object if the message has been created and sent. It may seem odd that the status attribute is set to 'queued' and the date_sent attribute is set to None when you've already received the text message. This is because you captured the Message object in the message variable *before* the text was actually sent. You will need to refetch the Message object in order to see its most up-to-date status and date_sent. Every Twilio message has a unique string ID (SID) that can be used to fetch the latest update of the Message object. Continue the interactive shell example by entering the following:

```
>>> message.sid
'SM09520de7639ba3af137c6fcb7c5f4b51'
❶ >>> updatedMessage = twilioCli.messages.get(message.sid)
>>> updatedMessage.status
'delivered'
>>> updatedMessage.date_sent
datetime.datetime(2015, 7, 8, 1, 36, 18)
```

Entering message.sid show you this message's long SID. By passing this SID to the Twilio client's get() method ❶, you can retrieve a new Message object with the most up-to-date information. In this new Message object, the status and date_sent attributes are correct.

The status attribute will be set to one of the following string values: 'queued', 'sending', 'sent', 'delivered', 'undelivered', or 'failed'. These statuses are self-explanatory, but for more precise details, take a look at the resources at *http://nostarch.com/automatestuff/*.

RECEIVING TEXT MESSAGES WITH PYTHON

Unfortunately, receiving text messages with Twilio is a bit more complicated than sending them. Twilio requires that you have a website running its own web application. That's beyond the scope of this book, but you can find more details in the resources for this book (*http://nostarch.com/automatestuff/*).

Project: "Just Text Me" Module

The person you'll most often text from your programs is probably you. Texting is a great way to send yourself notifications when you're away from your computer. If you've automated a boring task with a program that takes a couple of hours to run, you could have it notify you with a text when it's finished. Or you may have a regularly scheduled program running that sometimes needs to contact you, such as a weather-checking program that texts you a reminder to pack an umbrella.

As a simple example, here's a small Python program with a `textmyself()` function that sends a message passed to it as a string argument. Open a new file editor window and enter the following code, replacing the account SID, auth token, and phone numbers with your own information. Save it as *textMyself.py*.

```
#! python3
# textMyself.py - Defines the textmyself() function that texts a message
# passed to it as a string.

# Preset values:
accountSID = 'ACxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
authToken  = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
myNumber = '+15559998888'
twilioNumber = '+15552225678'

from twilio.rest import TwilioRestClient

❶ def textmyself(message):
❷     twilioCli = TwilioRestClient(accountSID, authToken)
❸     twilioCli.messages.create(body=message, from_=twilioNumber, to=myNumber)
```

This program stores an account SID, auth token, sending number, and receiving number. It then defined `textmyself()` to take on argument ❶, make a `TwilioRestClient` object ❷, and call `create()` with the message you passed ❸.

If you want to make the `textmyself()` function available to your other programs, simply place the *textMyself.py* file in the same folder as the Python executable (*C:\Python34* on Windows, */usr/local/lib/python3.4* on OS X, and */usr/bin/python3* on Linux). Now you can use the function in your other programs. Whenever you want one of your programs to text you, just add the following:

```
import textmyself
textmyself.textmyself('The boring task is finished.')
```

You need to sign up for Twilio and write the texting code only once. After that, it's just two lines of code to send a text from any of your other programs.

Summary

We communicate with each other on the Internet and over cell phone networks in dozens of different ways, but email and texting predominate. Your programs can communicate through these channels, which gives them powerful new notification features. You can even write programs running on different computers that communicate with one another directly via email, with one program sending emails with SMTP and the other retrieving them with IMAP.

Python's `smtplib` provides functions for using the SMTP to send emails through your email provider's SMTP server. Likewise, the third-party `imapclient` and `pyzmail` modules let you access IMAP servers and retrieve emails sent to you. Although IMAP is a bit more involved than SMTP, it's also quite powerful and allows you to search for particular emails, download them, and parse them to extract the subject and body as string values.

Texting is a bit different from email, since, unlike email, more than just an Internet connection is needed to send SMS texts. Fortunately, services such as Twilio provide modules to allow you to send text messages from your programs. Once you go through an initial setup process, you'll be able to send texts with just a couple lines of code.

With these modules in your skill set, you'll be able to program the specific conditions under which your programs should send notifications or reminders. Now your programs will have reach far beyond the computer they're running on!

Practice Questions

1. What is the protocol for sending email? For checking and receiving email?
2. What four `smtplib` functions/methods must you call to log in to an SMTP server?
3. What two `imapclient` functions/methods must you call to log in to an IMAP server?
4. What kind of argument do you pass to `imapObj.search()`?
5. What do you do if your code gets an error message that says `got more than 10000 bytes`?
6. The `imapclient` module handles connecting to an IMAP server and finding emails. What is one module that handles reading the emails that `imapclient` collects?
7. What three pieces of information do you need from Twilio before you can send text messages?

Practice Projects

For practice, write programs that do the following.

Random Chore Assignment Emailer

Write a program that takes a list of people's email addresses and a list of chores that need to be done and randomly assigns chores to people. Email each person their assigned chores. If you're feeling ambitious, keep a record of each person's previously assigned chores so that you can make sure the program avoids assigning anyone the same chore they did last time. For another possible feature, schedule the program to run once a week automatically.

Here's a hint: If you pass a list to the `random.choice()` function, it will return a randomly selected item from the list. Part of your code could look like this:

```
chores = ['dishes', 'bathroom', 'vacuum', 'walk dog']
randomChore = random.choice(chores)
chores.remove(randomChore)    # this chore is now taken, so remove it
```

Umbrella Reminder

Chapter 11 showed you how to use the `requests` module to scrape data from *http://weather.gov/*. Write a program that runs just before you wake up in the morning and checks whether it's raining that day. If so, have the program text you a reminder to pack an umbrella before leaving the house.

Auto Unsubscriber

Write a program that scans through your email account, finds all the unsubscribe links in all your emails, and automatically opens them in a browser. This program will have to log in to your email provider's IMAP server and download all of your emails. You can use BeautifulSoup (covered in Chapter 11) to check for any instance where the word *unsubscribe* occurs within an HTML link tag.

Once you have a list of these URLs, you can use `webbrowser.open()` to automatically open all of these links in a browser.

You'll still have to manually go through and complete any additional steps to unsubscribe yourself from these lists. In most cases, this involves clicking a link to confirm.

But this script saves you from having to go through all of your emails looking for unsubscribe links. You can then pass this script along to your friends so they can run it on their email accounts. (Just make sure your email password isn't hardcoded in the source code!)

Controlling Your Computer Through Email

Write a program that checks an email account every 15 minutes for any instructions you email it and executes those instructions automatically. For example, BitTorrent is a peer-to-peer downloading system. Using free BitTorrent software such as qBittorrent, you can download large media files on your home computer. If you email the program a (completely legal, not at all piratical) BitTorrent link, the program will eventually check its email, find this message, extract the link, and then launch qBittorrent to start downloading the file. This way, you can have your home computer begin downloads while you're away, and the (completely legal, not at all piratical) download can be finished by the time you return home.

Chapter 15 covers how to launch programs on your computer using the `subprocess.Popen()` function. For example, the following call would launch the qBittorrent program, along with a torrent file:

```
qbProcess = subprocess.Popen(['C:\\Program Files (x86)\\qBittorrent\\
qbittorrent.exe', 'shakespeare_complete_works.torrent'])
```

Of course, you'll want the program to make sure the emails come from you. In particular, you might want to require that the emails contain a password, since it is fairly trivial for hackers to fake a "from" address in emails. The program should delete the emails it finds so that it doesn't repeat instructions every time it checks the email account. As an extra feature, have the program email or text you a confirmation every time it executes a command. Since you won't be sitting in front of the computer that is running the program, it's a good idea to use the logging functions (see Chapter 10) to write a text file log that you can check if errors come up.

qBittorrent (as well as other BitTorrent applications) has a feature where it can quit automatically after the download completes. Chapter 15 explains how you can determine when a launched application has quit with the `wait()` method for `Popen` objects. The `wait()` method call will block until qBittorrent has stopped, and then your program can email or text you a notification that the download has completed.

There are a lot of possible features you could add to this project. If you get stuck, you can download an example implementation of this program from *http://nostarch.com/automatestuff/*.

17

MANIPULATING IMAGES

If you have a digital camera or even if you just upload photos from your phone to Facebook, you probably cross paths with digital image files all the time. You may know how to use basic graphics software, such as Microsoft Paint or Paintbrush, or even more advanced applications such as Adobe Photoshop. But if you need to edit a massive number of images, editing them by hand can be a lengthy, boring job.

Enter Python. Pillow is a third-party Python module for interacting with image files. The module has several functions that make it easy to crop, resize, and edit the content of an image. With the power to manipulate images the same way you would with software such as Microsoft Paint or Adobe Photoshop, Python can automatically edit hundreds or thousands of images with ease.

Computer Image Fundamentals

In order to manipulate an image, you need to understand the basics of how computers deal with colors and coordinates in images and how you can work with colors and coordinates in Pillow. But before you continue, install the `pillow` module. See Appendix A for help installing third-party modules.

Colors and RGBA Values

Computer programs often represent a color in an image as an *RGBA value*. An RGBA value is a group of numbers that specify the amount of red, green, blue, and *alpha* (or transparency) in a color. Each of these component values is an integer from `0` (none at all) to `255` (the maximum). These RGBA values are assigned to individual *pixels*; a pixel is the smallest dot of a single color the computer screen can show (as you can imagine, there are millions of pixels on a screen). A pixel's RGB setting tells it precisely what shade of color it should display. Images also have an alpha value to create RGBA values. If an image is displayed on the screen over a background image or desktop wallpaper, the alpha value determines how much of the background you can "see through" the image's pixel.

In Pillow, RGBA values are represented by a tuple of four integer values. For example, the color red is represented by `(255, 0, 0, 255)`. This color has the maximum amount of red, no green or blue, and the maximum alpha value, meaning it is fully opaque. Green is represented by `(0, 255, 0, 255)`, and blue is `(0, 0, 255, 255)`. White, the combination of all colors, is `(255, 255, 255, 255)`, while black, which has no color at all, is `(0, 0, 0, 255)`.

If a color has an alpha value of `0`, it is invisible, and it doesn't really matter what the RGB values are. After all, invisible red looks the same as invisible black.

Pillow uses the standard color names that HTML uses. Table 17-1 lists a selection of standard color names and their values.

Table 17-1: Standard Color Names and Their RGBA Values

Name	RGBA value	Name	RGBA value
White	`(255, 255, 255, 255)`	Red	`(255, 0, 0, 255)`
Green	`(0, 128, 0, 255)`	Blue	`(0, 0, 255, 255)`
Gray	`(128, 128, 128, 255)`	Yellow	`(255, 255, 0, 255)`
Black	`(0, 0, 0, 255)`	Purple	`(128, 0, 128, 255)`

Pillow offers the `ImageColor.getcolor()` function so you don't have to memorize RGBA values for the colors you want to use. This function takes a color name string as its first argument, and the string `'RGBA'` as its second argument, and it returns an RGBA tuple.

CMYK AND RGB COLORING

In grade school you learned that mixing red, yellow, and blue paints can form other colors; for example, you can mix blue and yellow to make green paint. This is known as the *subtractive color model*, and it applies to dyes, inks, and pigments. This is why color printers have *CMYK* ink cartridges: the *Cyan* (blue), *Magenta* (red), *Yellow*, and *blacK* ink can be mixed together to form any color.

However, the physics of light uses what's called an *additive color model*. When combining light (such as the light given off by your computer screen), red, green, and blue light can be combined to form any other color. This is why *RGB* values represent color in computer programs.

To see how this function works, enter the following into the interactive shell:

```
❶ >>> from PIL import ImageColor
❷ >>> ImageColor.getcolor('red', 'RGBA')
(255, 0, 0, 255)
❸ >>> ImageColor.getcolor('RED', 'RGBA')
(255, 0, 0, 255)
>>> ImageColor.getcolor('Black', 'RGBA')
(0, 0, 0, 255)
>>> ImageColor.getcolor('chocolate', 'RGBA')
(210, 105, 30, 255)
>>> ImageColor.getcolor('CornflowerBlue', 'RGBA')
(100, 149, 237, 255)
```

First, you need to import the `ImageColor` module from PIL ❶ (not from Pillow; you'll see why in a moment). The color name string you pass to `ImageColor.getcolor()` is case insensitive, so passing `'red'` ❷ and passing `'RED'` ❸ give you the same RGBA tuple. You can also pass more unusual color names, like `'chocolate'` and `'Cornflower Blue'`.

Pillow supports a huge number of color names, from `'aliceblue'` to `'whitesmoke'`. You can find the full list of more than 100 standard color names in the resources at *http://nostarch.com/automatestuff/*.

Coordinates and Box Tuples

Image pixels are addressed with x- and y-coordinates, which respectively specify a pixel's horizontal and vertical location in an image. The *origin* is the pixel at the top-left corner of the image and is specified with the notation (0, 0). The first zero represents the x-coordinate, which starts at zero at the origin and increases going from left to right. The second zero represents the y-coordinate, which starts at zero at the origin and increases going

down the image. This bears repeating: y-coordinates increase going downward, which is the opposite of how you may remember y-coordinates being used in math class. Figure 17-1 demonstrates how this coordinate system works.

Many of Pillow's functions and methods take a *box tuple* argument. This means Pillow is expecting a tuple of four integer coordinates that represent a rectangular region in an image. The four integers are, in order, as follows:

- *Left*: The x-coordinate of the leftmost edge of the box.
- *Top*: The y-coordinate of the top edge of the box.
- *Right*: The x-coordinate of one pixel to the right of the rightmost edge of the box. This integer must be greater than the left integer.
- *Bottom*: The y-coordinate of one pixel lower than the bottom edge of the box. This integer must be greater than the top integer.

Note that the box includes the left and top coordinates and goes up to but does not include the right and bottom coordinates. For example, the box tuple `(3, 1, 9, 6)` represents all the pixels in the black box in Figure 17-2.

Figure 17-1: The x- and y-coordinates of a 27×26 image of some sort of ancient data storage device

Figure 17-2: The area represented by the box tuple `(3, 1, 9, 6)`

Manipulating Images with Pillow

Now that you know how colors and coordinates work in Pillow, let's use Pillow to manipulate an image. Figure 17-3 is the image that will be used for all the interactive shell examples in this chapter. You can download it from *http://nostarch.com/automatestuff/*.

Once you have the image file *Zophie.png* in your current working directory, you'll be ready to load the image of Zophie into Python, like so:

```
>>> from PIL import Image
>>> catIm = Image.open('zophie.png')
```

Figure 17-3: My cat Zophie. The camera adds 10 pounds (which is a lot for a cat).

To load the image, you import the `Image` module from Pillow and call `Image.open()`, passing it the image's filename. You can then store the loaded image in a variable like `CatIm`. The module name of Pillow is `PIL` to make it backward compatible with an older module called Python Imaging Library, which is why you must run `from PIL import Image` instead of `from Pillow import Image`. Because of the way Pillow's creators set up the `pillow` module, you must use the `from PIL import Image` form of `import` statement, rather than simply `import PIL`.

If the image file isn't in the current working directory, change the working directory to the folder that contains the image file by calling the `os.chdir()` function.

```
>>> import os
>>> os.chdir('C:\\folder_with_image_file')
```

The `Image.open()` function returns a value of the `Image` object data type, which is how Pillow represents an image as a Python value. You can load an `Image` object from an image file (of any format) by passing the `Image.open()` function a string of the filename. Any changes you make to the `Image` object can be saved to an image file (also of any format) with the `save()` method. All the rotations, resizing, cropping, drawing, and other image manipulations will be done through method calls on this `Image` object.

To shorten the examples in this chapter, I'll assume you've imported Pillow's `Image` module and that you have the Zophie image stored in a variable named `catIm`. Be sure that the *zophie.png* file is in the current working directory so that the `Image.open()` function can find it. Otherwise, you will also have to specify the full absolute path in the string argument to `Image.open()`.

Working with the Image Data Type

An `Image` object has several useful attributes that give you basic information about the image file it was loaded from: its width and height, the filename, and the graphics format (such as JPEG, GIF, or PNG).

For example, enter the following into the interactive shell:

```
>>> from PIL import Image
>>> catIm = Image.open('zophie.png')
>>> catIm.size
❶ (816, 1088)
❷ >>> width, height = catIm.size
❸ >>> width
816
❹ >>> height
1088
>>> catIm.filename
'zophie.png'
>>> catIm.format
'PNG'
>>> catIm.format_description
'Portable network graphics'
❺ >>> catIm.save('zophie.jpg')
```

After making an `Image` object from *Zophie.png* and storing the `Image` object in `catIm`, we can see that the object's `size` attribute contains a tuple of the image's width and height in pixels ❶. We can assign the values in the tuple to `width` and `height` variables ❷ in order to access with width ❸ and height ❹ individually. The `filename` attribute describes the original file's name. The `format` and `format_description` attributes are strings that describe the image format of the original file (with `format_description` being a bit more verbose).

Finally, calling the `save()` method and passing it `'zophie.jpg'` saves a new image with the filename *zophie.jpg* to your hard drive ❺. Pillow sees that the file extension is *.jpg* and automatically saves the image using the JPEG image format. Now you should have two images, *zophie.png* and *zophie.jpg*, on your hard drive. While these files are based on the same image, they are not identical because of their different formats.

Pillow also provides the `Image.new()` function, which returns an `Image` object—much like `Image.open()`, except the image represented by `Image.new()`'s object will be blank. The arguments to `Image.new()` are as follows:

- The string `'RGBA'`, which sets the color mode to RGBA. (There are other modes that this book doesn't go into.)
- The size, as a two-integer tuple of the new image's width and height.

- The background color that the image should start with, as a four-integer tuple of an RGBA value. You can use the return value of the `ImageColor.getcolor()` function for this argument. Alternatively, `Image.new()` also supports just passing the string of the standard color name.

  For example, enter the following into the interactive shell:

```
  >>> from PIL import Image
❶ >>> im = Image.new('RGBA', (100, 200), 'purple')
  >>> im.save('purpleImage.png')
❷ >>> im2 = Image.new('RGBA', (20, 20))
  >>> im2.save('transparentImage.png')
```

Here we create an `Image` object for an image that's 100 pixels wide and 200 pixels tall, with a purple background ❶. This image is then saved to the file *purpleImage.png*. We call `Image.new()` again to create another `Image` object, this time passing (20, 20) for the dimensions and nothing for the background color ❷. Invisible black, `(0, 0, 0, 0)`, is the default color used if no color argument is specified, so the second image has a transparent background; we save this 20×20 transparent square in *transparentImage.png*.

Cropping Images

Cropping an image means selecting a rectangular region inside an image and removing everything outside the rectangle. The `crop()` method on `Image` objects takes a box tuple and returns an `Image` object representing the cropped image. The cropping does not happen in place—that is, the original `Image` object is left untouched, and the `crop()` method returns a new `Image` object. Remeber that a boxed tuple—in this case, the cropped section—includes the left column and top row of pixels but only goes up to and does *not* include the right column and bottom row of pixels.

Enter the following into the interactive shell:

```
>>> croppedIm = catIm.crop((335, 345, 565, 560))
>>> croppedIm.save('cropped.png')
```

This makes a new `Image` object for the cropped image, stores the object in `croppedIm`, and then calls `save()` on `croppedIm` to save the cropped image in *cropped.png*. The new file *cropped.png* will be created from the original image, like in Figure 17-4.

Figure 17-4: The new image will be just the cropped section of the original image.

Copying and Pasting Images onto Other Images

The `copy()` method will return a new `Image` object with the same image as the `Image` object it was called on. This is useful if you need to make changes to an image but also want to keep an untouched version of the original. For example, enter the following into the interactive shell:

```
>>> catIm = Image.open('zophie.png')
>>> catCopyIm = catIm.copy()
```

The `catIm` and `catCopyIm` variables contain two separate `Image` objects, which both have the same image on them. Now that you have an `Image` object stored in `catCopyIm`, you can modify `catCopyIm` as you like and save it to a new filename, leaving *zophie.png* untouched. For example, let's try modifying `catCopyIm` with the `paste()` method.

The `paste()` method is called on an `Image` object and pastes another image on top of it. Let's continue the shell example by pasting a smaller image onto `catCopyIm`.

```
>>> faceIm = catIm.crop((335, 345, 565, 560))
>>> faceIm.size
(230, 215)
>>> catCopyIm.paste(faceIm, (0, 0))
>>> catCopyIm.paste(faceIm, (400, 500))
>>> catCopyIm.save('pasted.png')
```

First we pass `crop()` a box tuple for the rectangular area in *zophie.png* that contains Zophie's face. This creates an `Image` object representing a 230×215 crop, which we store in `faceIm`. Now we can paste `faceIm` onto `catCopyIm`. The `paste()` method takes two arguments: a "source" `Image` object and a tuple of the x- and y-coordinates where you want to paste the top-left corner of the source `Image` object onto the main `Image` object. Here we call `paste()` twice on `catCopyIm`, passing (0, 0) the first time and (400, 500) the second time. This pastes `faceIm` onto `catCopyIm` twice: once with the top-left corner of `faceIm` at (0, 0) on `catCopyIm`, and once with the top-left corner of `faceIm` at (400, 500). Finally, we save the modified `catCopyIm` to *pasted.png*. The *pasted.png* image looks like Figure 17-5.

Figure 17-5: Zophie the cat, with her face pasted twice

NOTE *Despite their names, the `copy()` and `paste()` methods in Pillow do not use your computer's clipboard.*

Note that the `paste()` method modifies its `Image` object *in place*; it does not return an `Image` object with the pasted image. If you want to call `paste()` but also keep an untouched version of the original image around, you'll need to first copy the image and then call `paste()` on that copy.

Say you want to tile Zophie's head across the entire image, as in Figure 17-6. You can achieve this effect with just a couple `for` loops. Continue the interactive shell example by entering the following:

```
>>> catImWidth, catImHeight = catIm.size
>>> faceImWidth, faceImHeight = faceIm.size
❶ >>> catCopyTwo = catIm.copy()
```

```
❷ >>> for left in range(0, catImWidth, faceImWidth):
❸         for top in range(0, catImHeight, faceImHeight):
              print(left, top)
              catCopyTwo.paste(faceIm, (left, top))
0 0
0 215
0 430
0 645
0 860
0 1075
230 0
230 215
--snip--
690 860
690 1075
>>> catCopyTwo.save('tiled.png')
```

Here we store the width of height of `catIm` in `catImWidth` and `catImHeight`. At ❶ we make a copy of `catIm` and store it in `catCopyTwo`. Now that we have a copy that we can paste onto, we start looping to paste `faceIm` onto `catCopyTwo`. The outer `for` loop's `left` variable starts at 0 and increases by `faceImWidth(230)` ❷. The inner `for` loop's `top` variable start at 0 and increases by `faceImHeight(215)` ❸. These nested `for` loops produce values for `left` and `top` to paste a grid of `faceIm` images over the `catCopyTwo` `Image` object, as in Figure 17-6. To see our nested loops working, we print `left` and `top`. After the pasting is complete, we save the modified `catCopyTwo` to *tiled.png*.

Figure 17-6: Nested for loops used with paste() to duplicate the cat's face (a dupli-cat, if you will).

PASTING TRANSPARENT PIXELS

Normally transparent pixels are pasted as white pixels. If the image you want to paste has transparent pixels, pass the `Image` object as the third argument so that a solid rectangle isn't pasted. This third argument is the "mask" `Image` object. A mask is an `Image` object where the alpha value is significant, but the red, green, and blue values are ignored. The mask tells the `paste()` function which pixels it should copy and which it should leave transparent. Advanced usage of masks is beyond this book, but if you want to paste an image that has transparent pixels, pass the `Image` object again as the third argument.

Resizing an Image

The `resize()` method is called on an `Image` object and returns a new `Image` object of the specified width and height. It accepts a two-integer tuple argument, representing the new width and height of the returned image. Enter the following into the interactive shell:

```
❶ >>> width, height = catIm.size
❷ >>> quartersizedIm = catIm.resize((int(width / 2), int(height / 2)))
  >>> quartersizedIm.save('quartersized.png')
❸ >>> svelteIm = catIm.resize((width, height + 300))
  >>> svelteIm.save('svelte.png')
```

Here we assign the two values in the `catIm.size` tuple to the variables `width` and `height` ❶. Using `width` and `height` instead of `catIm.size[0]` and `catIm.size[1]` makes the rest of the code more readable.

The first `resize()` call passes `int(width / 2)` for the new width and `int(height / 2)` for the new height ❷, so the `Image` object returned from `resize()` will be half the length and width of the original image, or one-quarter of the original image size overall. The `resize()` method accepts only integers in its tuple argument, which is why you needed to wrap both divisions by `2` in an `int()` call.

This resizing keeps the same proportions for the width and height. But the new width and height passed to `resize()` do not have to be proportional to the original image. The `svelteIm` variable contains an `Image` object that has the original width but a height that is 300 pixels taller ❸, giving Zophie a more slender look.

Note that the `resize()` method does not edit the `Image` object in place but instead returns a new `Image` object.

Rotating and Flipping Images

Images can be rotated with the `rotate()` method, which returns a new `Image` object of the rotated image and leaves the original `Image` object unchanged. The argument to `rotate()` is a single integer or float representing the number of degrees to rotate the image counterclockwise. Enter the following into the interactive shell:

```
>>> catIm.rotate(90).save('rotated90.png')
>>> catIm.rotate(180).save('rotated180.png')
>>> catIm.rotate(270).save('rotated270.png')
```

Note how you can *chain* method calls by calling `save()` directly on the `Image` object returned from `rotate()`. The first `rotate()` and `save()` call makes a new `Image` object representing the image rotated counterclockwise by 90 degrees and saves the rotated image to *rotated90.png*. The second and third calls do the same, but with 180 degress and 270 degress. The results look like Figure 17-7.

Figure 17-7: The original image (left) and the image rotated counterclockwise by 90, 180, and 270 degrees

Notice that the width and height of the image change when the image is rotated 90 or 270 degrees. If you rotate an image by some other amount, the original dimensions of the image are maintained. On Windows, a black background is used to fill in any gaps made by the rotation, like in Figure 17-8. On OS X, transparent pixels are used for the gaps instead.

The `rotate()` method has an optional `expand` keyword argument that can be set to `True` to enlarge the dimensions of the image to fit the entire rotated new image. For example, enter the following into the interactive shell:

```
>>> catIm.rotate(6).save('rotated6.png')
>>> catIm.rotate(6, expand=True).save('rotated6_expanded.png')
```

The first call rotates the image 6 degrees and saves it to *rotate6.png* (see the image on the left of Figure 17-8). The second call rotates the image 6 degrees with `expand` set to `True` and saves it to *rotate6_expanded.png* (see the image on the right of Figure 17-8).

Figure 17-8: The image rotated 6 degrees normally (left) and with expand=True (right)

You can also get a "mirror flip" of an image with the `transpose()` method. You must pass either `Image.FLIP_LEFT_RIGHT` or `Image.FLIP_TOP_BOTTOM` to the `transpose()` method. Enter the following into the interactive shell:

```
>>> catIm.transpose(Image.FLIP_LEFT_RIGHT).save('horizontal_flip.png')
>>> catIm.transpose(Image.FLIP_TOP_BOTTOM).save('vertical_flip.png')
```

Like `rotate()`, `transpose()` creates a new `Image` object. Here was pass `Image.FLIP_LEFT_RIGHT` to flip the image horizontally and then save the result to *horizontal_flip.png*. To flip the image vertically, we pass `Image.FLIP_TOP_BOTTOM` and save to *vertical_flip.png*. The results look like Figure 17-9.

Figure 17-9: The original image (left), horizontal flip (center), and vertical flip (right)

Changing Individual Pixels

The color of an individual pixel can be retrieved or set with the `getpixel()` and `putpixel()` methods. These methods both take a tuple representing the x- and y-coordinates of the pixel. The `putpixel()` method also takes an additional tuple argument for the color of the pixel. This color argument is a four-integer RGBA tuple or a three-integer RGB tuple. Enter the following into the interactive shell:

```
❶ >>> im = Image.new('RGBA', (100, 100))
❷ >>> im.getpixel((0, 0))
  (0, 0, 0, 0)
❸ >>> for x in range(100):
          for y in range(50):
❹             im.putpixel((x, y), (210, 210, 210))

  >>> from PIL import ImageColor
❺ >>> for x in range(100):
          for y in range(50, 100):
❻             im.putpixel((x, y), ImageColor.getcolor('darkgray', 'RGBA'))
  >>> im.getpixel((0, 0))
  (210, 210, 210, 255)
  >>> im.getpixel((0, 50))
  (169, 169, 169, 255)
  >>> im.save('putPixel.png')
```

At ❶ we make a new image that is a 100×100 transparent square. Calling `getpixel()` on some coordinates in this image returns `(0, 0, 0, 0)` because the image is transparent ❷. To color pixels in this image, we can use nested `for` loops to go through all the pixels in the top half of the image ❸ and color each pixel using `putpixel()` ❹. Here we pass `putpixel()` the RGB tuple `(210, 210, 210)`, a light gray.

Say we want to color the bottom half of the image dark gray but don't know the RGB tuple for dark gray. The `putpixel()` method doesn't accept a standard color name like `'darkgray'`, so you have to use `ImageColor.getcolor()` to get a color tuple from `'darkgray'`. Loop through the pixels in the bottom half of the image ❺ and pass `putpixel()` the return value of `ImageColor.getcolor()` ❻, and you should now have an image that is light gray in its top half and dark gray in the bottom half, as shown in Figure 17-10. You can call `getpixel()` on some coordinates to confirm that the color at any given pixel is what you expect. Finally, save the image to *putPixel.png*.

Figure 17-10: The putPixel.png *image*

Of course, drawing one pixel at a time onto an image isn't very convenient. If you need to draw shapes, use the `ImageDraw` functions explained later in this chapter.

Project: Adding a Logo

Say you have the boring job of resizing thousands of images and adding a small logo watermark to the corner of each. Doing this with a basic graphics program such as Paintbrush or Paint would take forever. A fancier graphics application such as Photoshop can do batch processing, but that software costs hundreds of dollars. Let's write a script to do it instead.

Figure 17-11: The logo to be added to the image.

Say that Figure 17-11 is the logo you want to add to the bottom-right corner of each image: a black cat icon with a white border, with the rest of the image transparent.

At a high level, here's what the program should do:

- Load the logo image.
- Loop over all *.png* and *.jpg* files in the working directory.
- Check whether the image is wider or taller than 300 pixels.
- If so, reduce the width or height (whichever is larger) to 300 pixels and scale down the other dimension proportionally.
- Paste the logo image into the corner.
- Save the altered images to another folder.

This means the code will need to do the following:

- Open the *catlogo.png* file as an `Image` object.
- Loop over the strings returned from `os.listdir('.')`.
- Get the width and height of the image from the `size` attribute.
- Calculate the new width and height of the resized image.
- Call the `resize()` method to resize the image.
- Call the `paste()` method to paste the logo.
- Call the `save()` method to save the changes, using the original filename.

Step 1: Open the Logo Image

For this project, open a new file editor window, enter the following code, and save it as *resizeAndAddLogo.py*:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.
```

```
import os
from PIL import Image

❶ SQUARE_FIT_SIZE = 300
❷ LOGO_FILENAME = 'catlogo.png'

❸ logoIm = Image.open(LOGO_FILENAME)
❹ logoWidth, logoHeight = logoIm.size

# TODO: Loop over all files in the working directory.

# TODO: Check if image needs to be resized.

# TODO: Calculate the new width and height to resize to.

# TODO: Resize the image.

# TODO: Add the logo.

# TODO: Save changes.
```

By setting up the `SQUARE_FIT_SIZE` ❶ and `LOGO_FILENAME` ❷ constants at the start of the program, we've made it easy to change the program later. Say the logo that you're adding isn't the cat icon, or say you're reducing the output images' largest dimension to something other than 300 pixels. With these constants at the start of the program, you can just open the code, change those values once, and you're done. (Or you can make it so that the values for these constants are taken from the command line arguments.) Without these constants, you'd instead have to search the code for all instances of `300` and `'catlogo.png'` and replace them with the values for your new project. In short, using constants makes your program more generalized.

The logo `Image` object is returned from `Image.open()` ❸. For readability, `logoWidth` and `logoHeight` are assigned to the values from `logoIm.size` ❹.

The rest of the program is a skeleton of `TODO` comments for now.

Step 2: Loop Over All Files and Open Images

Now you need to find every *.png* file and *.jpg* file in the current working directory. Note that you don't want to add the logo image to the logo image itself, so the program should skip any image with a filename that's the same as `LOGO_FILENAME`. Add the following to your code:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.

import os
from PIL import Image

--snip--
```

```
os.makedirs('withLogo', exist_ok=True)
# Loop over all files in the working directory.
❶ for filename in os.listdir('.'):
❷     if not (filename.endswith('.png') or filename.endswith('.jpg')) \
         or filename == LOGO_FILENAME:
❸         continue    # skip non-image files and the logo file itself

❹     im = Image.open(filename)
      width, height = im.size

--snip--
```

First, the `os.makedirs()` call creates a *withLogo* folder to store the finished images with logos, instead of overwriting the original image files. The `exist_ok=True` keyword argument will keep `os.makedirs()` from raising an exception if *withLogo* already exists. While looping through all the files in the working directory with `os.listdir('.')` ❶, the long `if` statement ❷ checks whether each filename doesn't end with *.png* or *.jpg*. If so—or if the file is the logo image itself—then the loop should skip it and use `continue` ❸ to go to the next file. If `filename` *does* end with `'.png'` or `'.jpg'` (and isn't the logo file), you can open it as an `Image` object ❹ and set `width` and `height`.

Step 3: Resize the Images

The program should resize the image only if the width or height is larger than `SQUARE_FIT_SIZE` (300 pixels, in this case), so put all of the resizing code inside an `if` statement that checks the `width` and `height` variables. Add the following code to your program:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.

import os
from PIL import Image

--snip--

    # Check if image needs to be resized.
    if width > SQUARE_FIT_SIZE and height > SQUARE_FIT_SIZE:
        # Calculate the new width and height to resize to.
        if width > height:
❶           height = int((SQUARE_FIT_SIZE / width) * height)
            width = SQUARE_FIT_SIZE
        else:
❷           width = int((SQUARE_FIT_SIZE / height) * width)
            height = SQUARE_FIT_SIZE

        # Resize the image.
        print('Resizing %s...' % (filename))
❸          im = im.resize((width, height))

--snip--
```

If the image does need to be resized, you need to find out whether it is a wide or tall image. If `width` is greater than `height`, then the height should be reduced by the same proportion that the width would be reduced ❶. This proportion is the `SQUARE_FIT_SIZE` value divided by the current width. The new `height` value is this proportion multiplied by the current `height` value. Since the division operator returns a float value and `resize()` requires the dimensions to be integers, remember to convert the result to an integer with the `int()` function. Finally, the new `width` value will simply be set to `SQUARE_FIT_SIZE`.

If the `height` is greater than or equal to the `width` (both cases are handled in the `else` clause), then the same calculation is done, except with the `height` and `width` variables swapped ❷.

Once `width` and `height` contain the new image dimensions, pass them to the `resize()` method and store the returned `Image` object in `im` ❸.

Step 4: Add the Logo and Save the Changes

Whether or not the image was resized, the logo should still be pasted to the bottom-right corner. Where exactly the logo should be pasted depends on both the size of the image and the size of the logo. Figure 17-12 shows how to calculate the pasting position. The left coordinate for where to paste the logo will be the image width minus the logo width; the top coordinate for where to paste the logo will be the image height minus the logo height.

Figure 17-12: The left and top coordinates for placing the logo in the bottom-right corner should be the image width/height minus the logo width/height.

After your code pastes the logo into the image, it should save the modified `Image` object. Add the following to your program:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.

import os
from PIL import Image
```

```
--snip--

    # Check if image needs to be resized.
    --snip--

    # Add the logo.
❶   print('Adding logo to %s...' % (filename))
❷   im.paste(logoIm, (width - logoWidth, height - logoHeight), logoIm)

    # Save changes.
❸   im.save(os.path.join('withLogo', filename))
```

The new code prints a message telling the user that the logo is being added ❶, pastes `logoIm` onto `im` at the calculated coordinates ❷, and saves the changes to a filename in the *withLogo* directory ❸. When you run this program with the *zophie.png* file as the only image in the working directory, the output will look like this:

```
Resizing zophie.png...
Adding logo to zophie.png...
```

The image *zophie.png* will be changed to a 225×300-pixel image that looks like Figure 17-13. Remember that the `paste()` method will not paste the transparency pixels if you do not pass the `logoIm` for the third argument as well. This program can automatically resize and "logo-ify" hundreds of images in just a couple minutes.

Figure 17-13: The image zophie.png *resized and the logo added (left). If you forget the third argument, the transparent pixels in the logo will be copied as solid white pixels (right).*

Ideas for Similar Programs

Being able to composite images or modify image sizes in a batch can be useful in many applications. You could write similar programs to do the following:

- Add text or a website URL to images.
- Add timestamps to images.
- Copy or move images into different folders based on their sizes.
- Add a mostly transparent watermark to an image to prevent others from copying it.

Drawing on Images

If you need to draw lines, rectangles, circles, or other simple shapes on an image, use Pillow's `ImageDraw` module. Enter the following into the interactive shell:

```
>>> from PIL import Image, ImageDraw
>>> im = Image.new('RGBA', (200, 200), 'white')
>>> draw = ImageDraw.Draw(im)
```

First, we import `Image` and `ImageDraw`. Then we create a new image, in this case, a 200×200 white image, and store the `Image` object in `im`. We pass the `Image` object to the `ImageDraw.Draw()` function to receive an `ImageDraw` object. This object has several methods for drawing shapes and text onto an `Image` object. Store the `ImageDraw` object in a variable like `draw` so you can use it easily in the following example.

Drawing Shapes

The following ImageDraw methods draw various kinds of shapes on the image. The `fill` and `outline` parameters for these methods are optional and will default to white if left unspecified.

Points

The `point(`*`xy, fill`*`)` method draws individual pixels. The *`xy`* argument represents a list of the points you want to draw. The list can be a list of x- and y-coordinate tuples, such as `[(x, y), (x, y), ...]`, or a list of x- and y-coordinates without tuples, such as `[x1, y1, x2, y2, ...]`. The *`fill`* argument is the color of the points and is either an RGBA tuple or a string of a color name, such as `'red'`. The *`fill`* argument is optional.

Lines

The `line(`*`xy, fill, width`*`)` method draws a line or series of lines. *`xy`* is either a list of tuples, such as `[(x, y), (x, y), ...]`, or a list of integers, such as `[x1, y1, x2, y2, ...]`. Each point is one of the connecting points on the

lines you're drawing. The optional *fill* argument is the color of the lines, as an RGBA tuple or color name. The optional *width* argument is the width of the lines and defaults to 1 if left unspecified.

Rectangles

The `rectangle(xy, fill, outline)` method draws a rectangle. The *xy* argument is a box tuple of the form (*left*, *top*, *right*, *bottom*). The *left* and *top* values specify the x- and y-coordinates of the upper-left corner of the rectangle, while *right* and *bottom* specify the lower-right corner. The optional *fill* argument is the color that will fill the inside of the rectangle. The optional *outline* argument is the color of the rectangle's outline.

Ellipses

The `ellipse(xy, fill, outline)` method draws an ellipse. If the width and height of the ellipse are identical, this method will draw a circle. The *xy* argument is a box tuple (*left*, *top*, *right*, *bottom*) that represents a box that precisely contains the ellipse. The optional *fill* argument is the color of the inside of the ellipse, and the optional *outline* argument is the color of the ellipse's outline.

Polygons

The `polygon(xy, fill, outline)` method draws an arbitrary polygon. The *xy* argument is a list of tuples, such as `[(x, y), (x, y), ...]`, or integers, such as `[x1, y1, x2, y2, ...]`, representing the connecting points of the polygon's sides. The last pair of coordinates will be automatically connected to the first pair. The optional *fill* argument is the color of the inside of the polygon, and the optional *outline* argument is the color of the polygon's outline.

Drawing Example

Enter the following into the interactive shell:

```
>>> from PIL import Image, ImageDraw
>>> im = Image.new('RGBA', (200, 200), 'white')
>>> draw = ImageDraw.Draw(im)
❶ >>> draw.line([(0, 0), (199, 0), (199, 199), (0, 199), (0, 0)], fill='black')
❷ >>> draw.rectangle((20, 30, 60, 60), fill='blue')
❸ >>> draw.ellipse((120, 30, 160, 60), fill='red')
❹ >>> draw.polygon(((57, 87), (79, 62), (94, 85), (120, 90), (103, 113)),
fill='brown')
❺ >>> for i in range(100, 200, 10):
        draw.line([(i, 0), (200, i - 100)], fill='green')

>>> im.save('drawing.png')
```

After making an `Image` object for a 200×200 white image, passing it to `ImageDraw.Draw()` to get an `ImageDraw` object, and storing the `ImageDraw` object in `draw`, you can call drawing methods on `draw`. Here we make a thin, black

outline at the edges of the image ❶, a blue rectangle with its top-left corner at (20, 30) and bottom-right corner at (60, 60) ❷, a red ellipse defined by a box from (120, 30) to (160, 60) ❸, a brown polygon with five points ❹, and a pattern of green lines drawn with a `for` loop ❺. The resulting *drawing.png* file will look like Figure 17-14.

Figure 17-14: The resulting drawing.png *image*

There are several other shape-drawing methods for `ImageDraw` objects. The full documentation is available at *http://pillow.readthedocs.org/en/latest/reference/ImageDraw.html*.

Drawing Text

The `ImageDraw` object also has a `text()` method for drawing text onto an image. The `text()` method takes four arguments: *`xy`*, *`text`*, *`fill`*, and *`font`*.

- The *`xy`* argument is a two-integer tuple specifying the upper-left corner of the text box.
- The *`text`* argument is the string of text you want to write.
- The optional *`fill`* argument is the color of the text.
- The optional *`font`* argument is an `ImageFont` object, used to set the typeface and size of the text. This is described in more detail in the next section.

Since it's often hard to know in advance what size a block of text will be in a given font, the `ImageDraw` module also offers a `textsize()` method. Its first argument is the string of text you want to measure, and its second argument is an optional `ImageFont` object. The `textsize()` method will then return a two-integer tuple of the width and height that the text in the given

font would be if it were written onto the image. You can use this width and height to help you calculate exactly where you want to put the text on your image.

The first three arguments for `text()` are straightforward. Before we use `text()` to draw text onto an image, let's look at the optional fourth argument, the `ImageFont` object.

Both `text()` and `textsize()` take an optional `ImageFont` object as their final arguments. To create one of these objects, first run the following:

```
>>> from PIL import ImageFont
```

Now that you've imported Pillow's `ImageFont` module, you can call the `ImageFont.truetype()` function, which takes two arguments. The first argument is a string for the font's *TrueType file*—this is the actual font file that lives on your hard drive. A TrueType file has the *.ttf* file extension and can usually be found in the following folders:

- On Windows: *C:\Windows\Fonts*
- On OS X: */Library/Fonts* and */System/Library/Fonts*
- On Linux: */usr/share/fonts/truetype*

You don't actually need to enter these paths as part of the TrueType file string because Python knows to automatically search for fonts in these directories. But Python will display an error if it is unable to find the font you specified.

The second argument to `ImageFont.truetype()` is an integer for the font size in *points* (rather than, say, pixels). Keep in mind that Pillow creates PNG images that are 72 pixels per inch by default, and a point is 1/72 of an inch.

Enter the following into the interactive shell, replacing `FONT_FOLDER` with the actual folder name your operating system uses:

```
  >>> from PIL import Image, ImageDraw, ImageFont
  >>> import os
❶ >>> im = Image.new('RGBA', (200, 200), 'white')
❷ >>> draw = ImageDraw.Draw(im)
❸ >>> draw.text((20, 150), 'Hello', fill='purple')
  >>> fontsFolder = 'FONT_FOLDER' # e.g. '/Library/Fonts'
❹ >>> arialFont = ImageFont.truetype(os.path.join(fontsFolder, 'arial.ttf'), 32)
❺ >>> draw.text((100, 150), 'Howdy', fill='gray', font=arialFont)
  >>> im.save('text.png')
```

After importing `Image`, `ImageDraw`, `ImageFont`, and `os`, we make an `Image` object for a new 200×200 white image ❶ and make an `ImageDraw` object from the `Image` object ❷. We use `text()` to draw *Hello* at (20, 150) in purple ❸. We didn't pass the optional fourth argument in this `text()` call, so the typeface and size of this text aren't customized.

To set a typeface and size, we first store the folder name (like */Library/Fonts*) in `fontsFolder`. Then we call `ImageFont.truetype()`, passing it the *.ttf* file for the font we want, followed by an integer font size ❹. Store the `Font` object you get from `ImageFont.truetype()` in a variable like `arialFont`, and then pass the variable to `text()` in the final keyword argument. The `text()` call at ❺ draws *Howdy* at (100, 150) in gray in 32-point Arial.

The resulting *text.png* file will look like Figure 17-15.

Figure 17-15: The resulting text.png *image*

Summary

Images consist of a collection of pixels, and each pixel has an RGBA value for its color and its addressable by x- and y-coordinates. Two common image formats are JPEG and PNG. The `pillow` module can handle both of these image formats and others.

When an image is loaded into an `Image` object, its width and height dimensions are stored as a two-integer tuple in the `size` attribute. Objects of the `Image` data type also have methods for common image manipulations: `crop()`, `copy()`, `paste()`, `resize()`, `rotate()`, and `transpose()`. To save the `Image` object to an image file, call the `save()` method.

If you want your program to draw shapes onto an image, use `ImageDraw` methods to draw points, lines, rectangles, ellipses, and polygons. The module also provides methods for drawing text in a typeface and font size of your choosing.

Although advanced (and expensive) applications such as Photoshop provide automatic batch processing features, you can use Python scripts to do many of the same modifications for free. In the previous chapters, you wrote Python programs to deal with plaintext files, spreadsheets, PDFs, and other formats. With the `pillow` module, you've extended your programming powers to processing images as well!

Practice Questions

1. What is an RGBA value?
2. How can you get the RGBA value of `'CornflowerBlue'` from the `Pillow` module?
3. What is a box tuple?
4. What function returns an `Image` object for, say, an image file named *zophie.png*?

5. How can you find out the width and height of an `Image` object's image?
6. What method would you call to get `Image` object for a 100×100 image, excluding the lower left quarter of it?
7. After making changes to an `Image` object, how could you save it as an image file?
8. What module contains Pillow's shape-drawing code?
9. `Image` objects do not have drawing methods. What kind of object does? How do you get this kind of object?

Practice Projects

For practice, write programs that do the following.

Extending and Fixing the Chapter Project Programs

The *resizeAndAddLogo.py* program in this chapter works with PNG and JPEG files, but Pillow supports many more formats than just these two. Extend *resizeAndAddLogo.py* to process GIF and BMP images as well.

Another small issue is that the program modifies PNG and JPEG files only if their file extensions are set in lowercase. For example, it will process *zophie.png* but not *zophie.PNG*. Change the code so that the file extension check is case insensitive.

Finally, the logo added to the bottom-right corner is meant to be just a small mark, but if the image is about the same size as the logo itself, the result will look like Figure 17-16. Modify *resizeAndAddLogo.py* so that the image must be at least twice the width and height of the logo image before the logo is pasted. Otherwise, it should skip adding the logo.

Figure 17-16: When the image isn't much larger than the logo, the results look ugly.

Identifying Photo Folders on the Hard Drive

I have a bad habit of transferring files from my digital camera to temporary folders somewhere on the hard drive and then forgetting about these folders. It would be nice to write a program that could scan the entire hard drive and find these leftover "photo folders."

Write a program that goes through every folder on your hard drive and finds potential photo folders. Of course, first you'll have to define what you consider a "photo folder" to be; let's say that it's any folder where more than half of the files are photos. And how do you define what files are photos?

First, a photo file must have the file extension *.png* or *.jpg*. Also, photos are large images; a photo file's width and height must both be larger than 500 pixels. This is a safe bet, since most digital camera photos are several thousand pixels in width and height.

As a hint, here's a rough skeleton of what this program might look like:

```
#! python3
# Import modules and write comments to describe this program.

for foldername, subfolders, filenames in os.walk('C:\\'):
    numPhotoFiles = 0
    numNonPhotoFiles = 0
    for filename in filenames:
        # Check if file extension isn't .png or .jpg.
        if TODO:
            numNonPhotoFiles += 1
            continue    # skip to next filename

        # Open image file using Pillow.

        # Check if width & height are larger than 500.
        if TODO:
            # Image is large enough to be considered a photo.
            numPhotoFiles += 1
        else:
            # Image is too small to be a photo.
            numNonPhotoFiles += 1

    # If more than half of files were photos,
    # print the absolute path of the folder.
    if TODO:
        print(TODO)
```

When the program runs, it should print the absolute path of any photo folders to the screen.

Custom Seating Cards

Chapter 13 included a practice project to create custom invitations from a list of guests in a plaintext file. As an additional project, use the `pillow` module to create images for custom seating cards for your guests. For each of the guests listed in the *guests.txt* file from the resources at *http://nostarch.com/automatestuff/*, generate an image file with the guest name and some flowery decoration. A public domain flower image is available in the resources at *http://nostarch.com/automatestuff/*.

To ensure that each seating card is the same size, add a black rectangle on the edges of the invitation image so that when the image is printed out, there will be a guideline for cutting. The PNG files that Pillow produces are set to 72 pixels per inch, so a 4×5-inch card would require a 288×360-pixel image.

18

CONTROLLING THE KEYBOARD AND MOUSE WITH GUI AUTOMATION

Knowing various Python modules for editing spreadsheets, downloading files, and launching programs is useful, but sometimes there just aren't any modules for the applications you need to work with. The ultimate tools for automating tasks on your computer are programs you write that directly control the keyboard and mouse. These programs can control other applications by sending them virtual keystrokes and mouse clicks, just as if you were sitting at your computer and interacting with the applications yourself. This technique is known as *graphical user interface automation*, or *GUI automation* for short. With GUI automation, your programs can do anything that a human user sitting at the computer can do, except spill coffee on the keyboard.

Think of GUI automation as programming a robotic arm. You can program the robotic arm to type at your keyboard and move your mouse for you. This technique is particularly useful for tasks that involve a lot of mindless clicking or filling out of forms.

The `pyautogui` module has functions for simulating mouse movements, button clicks, and scrolling the mouse wheel. This chapter covers only a subset of PyAutoGUI's features; you can find the full documentation at *http://pyautogui.readthedocs.org/*.

Installing the pyautogui Module

The `pyautogui` module can send virtual keypresses and mouse clicks to Windows, OS X, and Linux. Depending on which operating system you're using, you may have to install some other modules (called *dependencies*) before you can install PyAutoGUI.

- On Windows, there are no other modules to install.
- On OS X, run **`sudo pip3 install pyobjc-framework-Quartz`**, **`sudo pip3 install pyobjc-core`**, and then **`sudo pip3 install pyobjc`**.
- On Linux, run **`sudo pip3 install python3-xlib`**, **`sudo apt-get install scrot`**, **`sudo apt-get install python3-tk`**, and **`sudo apt-get install python3-dev`**. (Scrot is a screenshot program that PyAutoGUI uses.)

After these dependencies are installed, run **`pip install pyautogui`** (or **`pip3`** on OS X and Linux) to install PyAutoGUI.

Appendix A has complete information on installing third-party modules. To test whether PyAutoGUI has been installed correctly, run **`import pyautogui`** from the interactive shell and check for any error messages.

Staying on Track

Before you jump in to a GUI automation, you should know how to escape problems that may arise. Python can move your mouse and type keystrokes at an incredible speed. In fact, it might be too fast for other programs to keep up with. Also, if something goes wrong but your program keeps moving the mouse around, it will be hard to tell what exactly the program is doing or how to recover from the problem. Like the enchanted brooms from Disney's *The Sorcerer's Apprentice*, which kept filling—and then overfilling—Mickey's tub with water, your program could get out of control even though it's following your instructions perfectly. Stopping the program can be difficult if the mouse is moving around on its own, preventing you from clicking the IDLE window to close it. Fortunately, there are several ways to prevent or recover from GUI automation problems.

Shutting Down Everything by Logging Out

Perhaps the simplest way to stop an out-of-control GUI automation program is to log out, which will shut down all running programs. On Windows and Linux, the logout hotkey is CTRL-ALT-DEL. On OS X, it is ⌘-SHIFT-OPTION-Q. By logging out, you'll lose any unsaved work, but at least you won't have to wait for a full reboot of the computer.

Pauses and Fail-Safes

You can tell your script to wait after every function call, giving you a short window to take control of the mouse and keyboard if something goes wrong. To do this, set the `pyautogui.PAUSE` variable to the number of seconds you want it to pause. For example, after setting `pyautogui.PAUSE = 1.5`, every PyAutoGUI function call will wait one and a half seconds after performing its action. Non-PyAutoGUI instructions will not have this pause.

PyAutoGUI also has a fail-safe feature. Moving the mouse cursor to the upper-left corner of the screen will cause PyAutoGUI to raise the `pyautogui.FailSafeException` exception. Your program can either handle this exception with `try` and `except` statements or let the exception crash your program. Either way, the fail-safe feature will stop the program if you quickly move the mouse as far up and left as you can. You can disable this feature by setting `pyautogui.FAILSAFE = False`. Enter the following into the interactive shell:

```
>>> import pyautogui
>>> pyautogui.PAUSE = 1
>>> pyautogui.FAILSAFE = True
```

Here we import `pyautogui` and set `pyautogui.PAUSE` to `1` for a one-second pause after each function call. We set `pyautogui.FAILSAFE` to `True` to enable the fail-safe feature.

Controlling Mouse Movement

In this section, you'll learn how to move the mouse and track its position on the screen using PyAutoGUI, but first you need to understand how PyAutoGUI works with coordinates.

The mouse functions of PyAutoGUI use x- and y-coordinates. Figure 18-1 shows the coordinate system for the computer screen; it's similar to the coordinate system used for images, discussed in Chapter 17. The *origin*, where *x* and *y* are both zero, is at the upper-left corner of the screen. The x-coordinates increase going to the right, and the y-coordinates increase going down. All coordinates are positive integers; there are no negative coordinates.

Figure 18-1: The coordinates of a computer screen with 1920×1080 resolution

Your *resolution* is how many pixels wide and tall your screen is. If your screen's resolution is set to 1920×1080, then the coordinate for the upper-left corner will be (0, 0), and the coordinate for the bottom-right corner will be (1919, 1079).

The `pyautogui.size()` function returns a two-integer tuple of the screen's width and height in pixels. Enter the following into the interactive shell:

```
>>> import pyautogui
>>> pyautogui.size()
(1920, 1080)
>>> width, height = pyautogui.size()
```

`pyautogui.size()` returns `(1920, 1080)` on a computer with a 1920×1080 resolution; depending on your screen's resolution, your return value may be different. You can store the width and height from `pyautogui.size()` in variables like `width` and `height` for better readability in your programs.

Moving the Mouse

Now that you understand screen coordinates, let's move the mouse. The `pyautogui.moveTo()` function will instantly move the mouse cursor to a specified position on the screen. Integer values for the x- and y-coordinates make up the function's first and second arguments, respectively. An optional `duration` integer or float keyword argument specifies the number of seconds it should take to move the mouse to the destination. If you leave it out, the default is `0` for instantaneous movement. (All of the `duration` keyword arguments in PyAutoGUI functions are optional.) Enter the following into the interactive shell:

```
>>> import pyautogui
>>> for i in range(10):
      pyautogui.moveTo(100, 100, duration=0.25)
      pyautogui.moveTo(200, 100, duration=0.25)
      pyautogui.moveTo(200, 200, duration=0.25)
      pyautogui.moveTo(100, 200, duration=0.25)
```

This example moves the mouse cursor clockwise in a square pattern among the four coordinates provided a total of ten times. Each movement takes a quarter of a second, as specified by the `duration=0.25` keyword argument. If you hadn't passed a third argument to any of the `pyautogui.moveTo()` calls, the mouse cursor would have instantly teleported from point to point.

The `pyautogui.moveRel()` function moves the mouse cursor *relative* to its current position. The following example moves the mouse in the same square pattern, except it begins the square from wherever the mouse happens to be on the screen when the code starts running:

```
>>> import pyautogui
>>> for i in range(10):
      pyautogui.moveRel(100, 0, duration=0.25)
      pyautogui.moveRel(0, 100, duration=0.25)
```

```
pyautogui.moveRel(-100, 0, duration=0.25)
pyautogui.moveRel(0, -100, duration=0.25)
```

`pyautogui.moveRel()` also takes three arguments: how many pixels to move horizontally to the right, how many pixels to move vertically downward, and (optionally) how long it should take to complete the movement. A negative integer for the first or second argument will cause the mouse to move left or upward, respectively.

Getting the Mouse Position

You can determine the mouse's current position by calling the `pyautogui.position()` function, which will return a tuple of the mouse cursor's *x* and *y* positions at the time of the function call. Enter the following into the interactive shell, moving the mouse around after each call:

```
>>> pyautogui.position()
(311, 622)
>>> pyautogui.position()
(377, 481)
>>> pyautogui.position()
(1536, 637)
```

Of course, your return values will vary depending on where your mouse cursor is.

Project: "Where Is the Mouse Right Now?"

Being able to determine the mouse position is an important part of setting up your GUI automation scripts. But it's almost impossible to figure out the exact coordinates of a pixel just by looking at the screen. It would be handy to have a program that constantly displays the x- and y-coordinates of the mouse cursor as you move it around.

At a high level, here's what your program should do:

- Display the current x- and y-coordinates of the mouse cursor.
- Update these coordinates as the mouse moves around the screen.

This means your code will need to do the following:

- Call the `position()` function to fetch the current coordinates.
- Erase the previously printed coordinates by printing `\b` backspace characters to the screen.
- Handle the `KeyboardInterrupt` exception so the user can press CTRL-C to quit.

Open a new file editor window and save it as *mouseNow.py*.

Step 1: Import the Module

Start your program with the following:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
import pyautogui
print('Press Ctrl-C to quit.')
#TODO: Get and print the mouse coordinates.
```

The beginning of the program imports the `pyautogui` module and prints a reminder to the user that they have to press CTRL-C to quit.

Step 2: Set Up the Quit Code and Infinite Loop

You can use an infinite `while` loop to constantly print the current mouse coordinates from `mouse.position()`. As for the code that quits the program, you'll need to catch the `KeyboardInterrupt` exception, which is raised whenever the user presses CTRL-C. If you don't handle this exception, it will display an ugly traceback and error message to the user. Add the following to your program:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
import pyautogui
print('Press Ctrl-C to quit.')
try:
    while True:
        # TODO: Get and print the mouse coordinates.
❶ except KeyboardInterrupt:
❷     print('\nDone.')
```

To handle the exception, enclose the infinite `while` loop in a `try` statement. When the user presses CTRL-C, the program execution will move to the `except` clause ❶ and `Done.` will be printed in a new line ❷.

Step 3: Get and Print the Mouse Coordinates

The code inside the `while` loop should get the current mouse coordinates, format them to look nice, and print them. Add the following code to the inside of the `while` loop:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
import pyautogui
print('Press Ctrl-C to quit.')
--snip--
        # Get and print the mouse coordinates.
        x, y = pyautogui.position()
        positionStr = 'X: ' + str(x).rjust(4) + ' Y: ' + str(y).rjust(4)
--snip--
```

Using the multiple assignment trick, the `x` and `y` variables are given the values of the two integers returned in the tuple from `pyautogui.position()`. By passing `x` and `y` to the `str()` function, you can get string forms of the integer coordinates. The `rjust()` string method will right-justify them so that they take up the same amount of space, whether the coordinate has one, two, three, or four digits. Concatenating the right-justified string coordinates with `'X: '` and `' Y: '` labels gives us a neatly formatted string, which will be stored in `positionStr`.

At the end of your program, add the following code:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
--snip--
        print(positionStr, end='')
❶       print('\b' * len(positionStr), end='', flush=True)
```

This actually prints `positionStr` to the screen. The `end=''` keyword argument to `print()` prevents the default newline character from being added to the end of the printed line. It's possible to erase text you've already printed to the screen—but only for the most recent line of text. Once you print a newline character, you can't erase anything printed before it.

To erase text, print the `\b` backspace escape character. This special character erases a character at the end of the current line on the screen. The line at ❶ uses string replication to produce a string with as many `\b` characters as the length of the string stored in `positionStr`, which has the effect of erasing the `positionStr` string that was last printed.

For a technical reason beyond the scope of this book, always pass `flush=True` to `print()` calls that print `\b` backspace characters. Otherwise, the screen might not update the text as desired.

Since the `while` loop repeats so quickly, the user won't actually notice that you're deleting and reprinting the whole number on the screen. For example, if the x-coordinate is 563 and the mouse moves one pixel to the right, it will look like only the 3 in 563 is changed to a 4.

When you run the program, there will be only two lines printed. They should look like something like this:

```
Press Ctrl-C to quit.
X:  290 Y:  424
```

The first line displays the instruction to press CTRL-C to quit. The second line with the mouse coordinates will change as you move the mouse around the screen. Using this program, you'll be able to figure out the mouse coordinates for your GUI automation scripts.

Controlling Mouse Interaction

Now that you know how to move the mouse and figure out where it is on the screen, you're ready to start clicking, dragging, and scrolling.

Clicking the Mouse

To send a virtual mouse click to your computer, call the `pyautogui.click()` method. By default, this click uses the left mouse button and takes place wherever the mouse cursor is currently located. You can pass x- and y-coordinates of the click as optional first and second arguments if you want it to take place somewhere other than the mouse's current position.

If you want to specify which mouse button to use, include the `button` keyword argument, with a value of `'left'`, `'middle'`, or `'right'`. For example, `pyautogui.click(100, 150, button='left')` will click the left mouse button at the coordinates (100, 150), while `pyautogui.click(200, 250, button='right')` will perform a right-click at (200, 250).

Enter the following into the interactive shell:

```
>>> import pyautogui
>>> pyautogui.click(10, 5)
```

You should see the mouse pointer move to near the top-left corner of your screen and click once. A full "click" is defined as pushing a mouse button down and then releasing it back up without moving the cursor. You can also perform a click by calling `pyautogui.mouseDown()`, which only pushes the mouse button down, and `pyautogui.mouseUp()`, which only releases the button. These functions have the same arguments as `click()`, and in fact, the `click()` function is just a convenient wrapper around these two function calls.

As a further convenience, the `pyautogui.doubleClick()` function will perform two clicks with the left mouse button, while the `pyautogui.rightClick()` and `pyautogui.middleClick()` functions will perform a click with the right and middle mouse buttons, respectively.

Dragging the Mouse

Dragging means moving the mouse while holding down one of the mouse buttons. For example, you can move files between folders by dragging the folder icons, or you can move appointments around in a calendar app.

PyAutoGUI provides the `pyautogui.dragTo()` and `pyautogui.dragRel()` functions to drag the mouse cursor to a new location or a location relative to its current one. The arguments for `dragTo()` and `dragRel()` are the same as `moveTo()` and `moveRel()`: the x-coordinate/horizontal movement, the y-coordinate/vertical movement, and an optional duration of time. (OS X does not drag correctly when the mouse moves too quickly, so passing a `duration` keyword argument is recommended.)

To try these functions, open a graphics-drawing application such as Paint on Windows, Paintbrush on OS X, or GNU Paint on Linux. (If you don't have a drawing application, you can use the online one at *http://sumopaint.com/*.) I will use PyAutoGUI to draw in these applications.

With the mouse cursor over the drawing application's canvas and the Pencil or Brush tool selected, enter the following into a new file editor window and save it as *spiralDraw.py*:

```
import pyautogui, time
❶ time.sleep(5)
❷ pyautogui.click()    # click to put drawing program in focus
distance = 200
while distance > 0:
❸     pyautogui.dragRel(distance, 0, duration=0.2)   # move right
❹     distance = distance - 5
❺     pyautogui.dragRel(0, distance, duration=0.2)   # move down
❻     pyautogui.dragRel(-distance, 0, duration=0.2)  # move left
      distance = distance - 5
      pyautogui.dragRel(0, -distance, duration=0.2)  # move up
```

When you run this program, there will be a five-second delay ❶ for you to move the mouse cursor over the drawing program's window with the Pencil or Brush tool selected. Then *spiralDraw.py* will take control of the mouse and click to put the drawing program in focus ❷. A window is in *focus* when it has an active blinking cursor, and the actions you take—like typing or, in this case, dragging the mouse—will affect that window. Once the drawing program is in focus, *spiralDraw.py* draws a square spiral pattern like the one in Figure 18-2.

Figure 18-2: The results from the `pyautogui.dragRel()` example

The `distance` variable starts at `200`, so on the first iteration of the `while` loop, the first `dragRel()` call drags the cursor 200 pixels to the right, taking 0.2 seconds ❸. `distance` is then decreased to 195 ❹, and the second `dragRel()` call drags the cursor 195 pixels down ❺. The third `dragRel()` call drags the cursor –195 horizontally (195 to the left) ❻, `distance` is decreased to 190, and the last `dragRel()` call drags the cursor 190 pixels up. On each iteration, the mouse is dragged right, down, left, and up, and `distance` is slightly smaller than it was in the previous iteration. By looping over this code, you can move the mouse cursor to draw a square spiral.

You could draw this spiral by hand (or rather, by mouse), but you'd have to work slowly to be so precise. PyAutoGUI can do it in a few seconds!

NOTE *You could have your code draw the image using the `pillow` module's drawing functions—see Chapter 17 for more information. But using GUI automation allows you to make use of the advanced drawing tools that graphics programs can provide, such as gradients, different brushes, or the fill bucket.*

Scrolling the Mouse

The final PyAutoGUI mouse function is `scroll()`, which you pass an integer argument for how many units you want to scroll the mouse up or down. The size of a unit varies for each operating system and application, so you'll have to experiment to see exactly how far it scrolls in your particular situation. The scrolling takes place at the mouse cursor's current position. Passing a positive integer scrolls up, and passing a negative integer scrolls down. Run the following in IDLE's interactive shell while the mouse cursor is over the IDLE window:

```
>>> pyautogui.scroll(200)
```

You'll see IDLE briefly scroll upward—and then go back down. The downward scrolling happens because IDLE automatically scrolls down to the bottom after executing an instruction. Enter this code instead:

```
>>> import pyperclip
>>> numbers = ''
>>> for i in range(200):
      numbers = numbers + str(i) + '\n'

>>> pyperclip.copy(numbers)
```

This imports `pyperclip` and sets up an empty string, `numbers`. The code then loops through 200 numbers and adds each number to `numbers`, along with a newline. After `pyperclip.copy(numbers)`, the clipboard will be loaded with 200 lines of numbers. Open a new file editor window and paste the text into it. This will give you a large text window to try scrolling in. Enter the following code into the interactive shell:

```
>>> import time, pyautogui
>>> time.sleep(5); pyautogui.scroll(100)
```

On the second line, you enter two commands separated by a semicolon, which tells Python to run the commands as if they were on separate lines. The only difference is that the interactive shell won't prompt you for input between the two instructions. This is important for this example because we want to the call to `pyautogui.scroll()` to happen automatically after the wait. (Note that while putting two commands on one line can be useful in the interactive shell, you should still have each instruction on a separate line in your programs.)

After pressing ENTER to run the code, you will have five seconds to click the file editor window to put it in focus. Once the pause is over, the `pyautogui.scroll()` call will cause the file editor window to scroll up after the five-second delay.

Working with the Screen

Your GUI automation programs don't have to click and type blindly. PyAutoGUI has screenshot features that can create an image file based on the current contents of the screen. These functions can also return a Pillow `Image` object of the current screen's appearance. If you've been skipping around in this book, you'll want to read Chapter 17 and install the `pillow` module before continuing with this section.

On Linux computers, the `scrot` program needs to be installed to use the screenshot functions in PyAutoGUI. In a Terminal window, run **`sudo apt-get install scrot`** to install this program. If you're on Windows or OS X, skip this step and continue with the section.

Getting a Screenshot

To take screenshots in Python, call the `pyautogui.screenshot()` function. Enter the following into the interactive shell:

```
>>> import pyautogui
>>> im = pyautogui.screenshot()
```

The `im` variable will contain the `Image` object of the screenshot. You can now call methods on the `Image` object in the `im` variable, just like any other `Image` object. Enter the following into the interactive shell:

```
>>> im.getpixel((0, 0))
(176, 176, 175)
>>> im.getpixel((50, 200))
(130, 135, 144)
```

Pass `getpixel()` a tuple of coordinates, like (0, 0) or (50, 200), and it'll tell you the color of the pixel at those coordinates in your image. The return value from `getpixel()` is an RGB tuple of three integers for the amount of red, green, and blue in the pixel. (There is no fourth value for alpha, because screenshot images are fully opaque.) This is how your programs can "see" what is currently on the screen.

Analyzing the Screenshot

Say that one of the steps in your GUI automation program is to click a gray button. Before calling the `click()` method, you could take a screenshot and look at the pixel where the script is about to click. If it's not the same gray as the gray button, then your program knows something is wrong. Maybe the window moved unexpectedly, or maybe a pop-up dialog has blocked the button. At this point, instead of continuing—and possibly wreaking havoc by clicking the wrong thing—your program can "see" that it isn't clicking on the right thing and stop itself.

PyAutoGUI's `pixelMatchesColor()` function will return `True` if the pixel at the given x- and y-coordinates on the screen matches the given color. The first and second arguments are integers for the x- and y-coordinates, and the third argument is a tuple of three integers for the RGB color the screen pixel must match. Enter the following into the interactive shell:

```
  >>> import pyautogui
  >>> im = pyautogui.screenshot()
❶ >>> im.getpixel((50, 200))
  (130, 135, 144)
❷ >>> pyautogui.pixelMatchesColor(50, 200, (130, 135, 144))
  True
❸ >>> pyautogui.pixelMatchesColor(50, 200, (255, 135, 144))
  False
```

After taking a screenshot and using `getpixel()` to get an RGB tuple for the color of a pixel at specific coordinates ❶, pass the same coordinates and RGB tuple to `pixelMatchesColor()` ❷, which should return `True`. Then change a value in the RGB tuple and call `pixelMatchesColor()` again for the same coordinates ❸. This should return `false`. This method can be useful to call whenever your GUI automation programs are about to call `click()`. Note that the color at the given coordinates must *exactly* match. If it is even slightly different—for example, `(255, 255, 254)` instead of `(255, 255, 255)`—then `pixelMatchesColor()` will return `False`.

Project: Extending the mouseNow Program

You could extend the *mouseNow.py* project from earlier in this chapter so that it not only gives the x- and y-coordinates of the mouse cursor's current position but also gives the RGB color of the pixel under the cursor. Modify the code inside the `while` loop of *mouseNow.py* to look like this:

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
--snip--
        positionStr = 'X: ' + str(x).rjust(4) + ' Y: ' + str(y).rjust(4)
        pixelColor = pyautogui.screenshot().getpixel((x, y))
        positionStr += ' RGB: (' + str(pixelColor[0]).rjust(3)
        positionStr += ', ' + str(pixelColor[1]).rjust(3)
        positionStr += ', ' + str(pixelColor[2]).rjust(3) + ')'
```

```
        print(positionStr, end='')
--snip--
```

Now, when you run *mouseNow.py*, the output will include the RGB color value of the pixel under the mouse cursor.

```
Press Ctrl-C to quit.
X:  406 Y:   17 RGB: (161,  50,  50)
```

This information, along with the `pixelMatchesColor()` function, should make it easy to add pixel color checks to your GUI automation scripts.

Image Recognition

But what if you do not know beforehand where PyAutoGUI should click? You can use image recognition instead. Give PyAutoGUI an image of what you want to click and let it figure out the coordinates.

For example, if you have previously taken a screenshot to capture the image of a Submit button in *submit.png*, the `locateOnScreen()` function will return the coordinates where that image is found. To see how `locateOnScreen()` works, try taking a screenshot of a small area on your screen; then save the image and enter the following into the interactive shell, replacing `'submit.png'` with the filename of your screenshot:

```
>>> import pyautogui
>>> pyautogui.locateOnScreen('submit.png')
(643, 745, 70, 29)
```

The four-integer tuple that `locateOnScreen()` returns has the x-coordinate of the left edge, the y-coordinate of the top edge, the width, and the height for the first place on the screen the image was found. If you're trying this on your computer with your own screenshot, your return value will be different from the one shown here.

If the image cannot be found on the screen, `locateOnScreen()` will return `None`. Note that the image on the screen must match the provided image perfectly in order to be recognized. If the image is even a pixel off, `locateOnScreen()` will return `None`.

If the image can be found in several places on the screen, `locateAllOnScreen()` will return a `Generator` object, which can be passed to `list()` to return a list of four-integer tuples. There will be one four-integer tuple for each location where the image is found on the screen. Continue the interactive shell example by entering the following (and replacing `'submit.png'` with your own image filename):

```
>>> list(pyautogui.locateAllOnScreen('submit.png'))
[(643, 745, 70, 29), (1007, 801, 70, 29)]
```

Each of the four-integer tuples represents an area on the screen. If your image is only found in one area, then using `list()` and `locateAllOnScreen()` just returns a list containing one tuple.

Once you have the four-integer tuple for the area on the screen where your image was found, you can click the center of this area by passing the tuple to the `center()` function to return x- and y-coordinates of the area's center. Enter the following into the interactive shell, replacing the arguments with your own filename, four-integer tuple, and coordinate pair:

```
>>> pyautogui.locateOnScreen('submit.png')
(643, 745, 70, 29)
>>> pyautogui.center((643, 745, 70, 29))
(678, 759)
>>> pyautogui.click((678, 759))
```

Once you have center coordinates from `center()`, passing the coordinates to `click()` should click the center of the area on the screen that matches the image you passed to `locateOnScreen()`.

Controlling the Keyboard

PyAutoGUI also has functions for sending virtual keypresses to your computer, which enables you to fill out forms or enter text into applications.

Sending a String from the Keyboard

The `pyautogui.typewrite()` function sends virtual keypresses to the computer. What these keypresses do depends on what window and text field have focus. You may want to first send a mouse click to the text field you want in order to ensure that it has focus.

As a simple example, let's use Python to automatically type the words *Hello world!* into a file editor window. First, open a new file editor window and position it in the upper-left corner of your screen so that PyAutoGUI will click in the right place to bring it into focus. Next, enter the following into the interactive shell:

```
>>> pyautogui.click(100, 100); pyautogui.typewrite('Hello world!')
```

Notice how placing two commands on the same line, separated by a semicolon, keeps the interactive shell from prompting you for input between running the two instructions. This prevents you from accidentally bringing a new window into focus between the `click()` and `typewrite()` calls, which would mess up the example.

Python will first send a virtual mouse click to the coordinates (100, 100), which should click the file editor window and put it in focus. The `typewrite()` call will send the text *Hello world!* to the window, making it look like Figure 18-3. You now have code that can type for you!

Figure 18-3: Using PyAutogGUI to click the file editor window and type Hello world! *into it*

By default, the `typewrite()` function will type the full string instantly. However, you can pass an optional second argument to add a short pause between each character. This second argument is an integer or float value of the number of seconds to pause. For example, `pyautogui.typewrite('Hello world!', 0.25)` will wait a quarter-second after typing *H*, another quarter-second after *e*, and so on. This gradual typewriter effect may be useful for slower applications that can't process keystrokes fast enough to keep up with PyAutoGUI.

For characters such as *A* or *!*, PyAutoGUI will automatically simulate holding down the SHIFT key as well.

Key Names

Not all keys are easy to represent with single text characters. For example, how do you represent SHIFT or the left arrow key as a single character? In PyAutoGUI, these keyboard keys are represented by short string values instead: `'esc'` for the ESC key or `'enter'` for the ENTER key.

Instead of a single string argument, a list of these keyboard key strings can be passed to `typewrite()`. For example, the following call presses the A key, then the B key, then the left arrow key twice, and finally the X and Y keys:

```
>>> pyautogui.typewrite(['a', 'b', 'left', 'left', 'X', 'Y'])
```

Because pressing the left arrow key moves the keyboard cursor, this will output *XYab*. Table 18-1 lists the PyAutoGUI keyboard key strings that you can pass to `typewrite()` to simulate pressing any combination of keys.

You can also examine the `pyautogui.KEYBOARD_KEYS` list to see all possible keyboard key strings that PyAutoGUI will accept. The `'shift'` string refers to the left SHIFT key and is equivalent to `'shiftleft'`. The same applies for `'ctrl'`, `'alt'`, and `'win'` strings; they all refer to the left-side key.

Table 18-1: PyKeyboard Attributes

Keyboard key string	Meaning
'a', 'b', 'c', 'A', 'B', 'C', '1', '2', '3', '!', '@', '#', and so on	The keys for single characters
'enter' (or 'return' or '\n')	The ENTER key
'esc'	The ESC key
'shiftleft', 'shiftright'	The left and right SHIFT keys
'altleft', 'altright'	The left and right ALT keys
'ctrlleft', 'ctrlright'	The left and right CTRL keys
'tab' (or '\t')	The TAB key
'backspace', 'delete'	The BACKSPACE and DELETE keys
'pageup', 'pagedown'	The PAGE UP and PAGE DOWN keys
'home', 'end'	The HOME and END keys
'up', 'down', 'left', 'right'	The up, down, left, and right arrow keys
'f1', 'f2', 'f3', and so on	The F1 to F12 keys
'volumemute', 'volumedown', 'volumeup'	The mute, volume down, and volume up keys (some keyboards do not have these keys, but your operating system will still be able to understand these simulated keypresses)
'pause'	The PAUSE key
'capslock', 'numlock', 'scrolllock'	The CAPS LOCK, NUM LOCK, and SCROLL LOCK keys
'insert'	The INS or INSERT key
'printscreen'	The PRTSC or PRINT SCREEN key
'winleft', 'winright'	The left and right WIN keys (on Windows)
'command'	The Command (⌘) key (on OS X)
'option'	The OPTION key (on OS X)

Pressing and Releasing the Keyboard

Much like the mouseDown() and mouseUp() functions, pyautogui.keyDown() and pyautogui.keyUp() will send virtual keypresses and releases to the computer. They are passed a keyboard key string (see Table 18-1) for their argument. For convenience, PyAutoGUI provides the pyautogui.press() function, which calls both of these functions to simulate a complete keypress.

Run the following code, which will type a dollar sign character (obtained by holding the SHIFT key and pressing 4):

```
>>> pyautogui.keyDown('shift'); pyautogui.press('4'); pyautogui.keyUp('shift')
```

This line presses down SHIFT, presses (and releases) 4, and then releases SHIFT. If you need to type a string into a text field, the `typewrite()` function is more suitable. But for applications that take single-key commands, the `press()` function is the simpler approach.

Hotkey Combinations

A *hotkey* or *shortcut* is a combination of keypresses to invoke some application function. The common hotkey for copying a selection is CTRL-C (on Windows and Linux) or ⌘-C (on OS X). The user presses and holds the CTRL key, then presses the C key, and then releases the C and CTRL keys. To do this with PyAutoGUI's `keyDown()` and `keyUp()` functions, you would have to enter the following:

```
pyautogui.keyDown('ctrl')
pyautogui.keyDown('c')
pyautogui.keyUp('c')
pyautogui.keyUp('ctrl')
```

This is rather complicated. Instead, use the `pyautogui.hotkey()` function, which takes multiple keyboard key string arguments, presses them in order, and releases them in the reverse order. For the CTRL-C example, the code would simply be as follows:

```
pyautogui.hotkey('ctrl', 'c')
```

This function is especially useful for larger hotkey combinations. In Word, the CTRL-ALT-SHIFT-S hotkey combination displays the Style pane. Instead of making eight different function calls (four `keyDown()` calls and four `keyUp()` calls), you can just call `hotkey('ctrl', 'alt', 'shift', 's')`.

With a new IDLE file editor window in the upper-left corner of your screen, enter the following into the interactive shell (in OS X, replace `'alt'` with `'ctrl'`):

```
>>> import pyautogui, time
>>> def commentAfterDelay():
❶       pyautogui.click(100, 100)
❷       pyautogui.typewrite('In IDLE, Alt-3 comments out a line.')
        time.sleep(2)
❸       pyautogui.hotkey('alt', '3')

>>> commentAfterDelay()
```

This defines a function `commentAfterDelay()` that, when called, will click the file editor window to bring it into focus ❶, type *In IDLE, Atl-3 comments out a line* ❷, pause for 2 seconds, and then simulate pressing the ALT-3 hotkey (or CTRL-3 on OS X) ❸. This keyboard shortcut adds two # characters to the current line, commenting it out. (This is a useful trick to know when writing your own code in IDLE.)

Review of the PyAutoGUI Functions

Since this chapter covered many different functions, here is a quick summary reference:

`moveTo(x, y)` Moves the mouse cursor to the given *x* and *y* coordinates.

`moveRel(xOffset, yOffset)` Moves the mouse cursor relative to its current position.

`dragTo(x, y)` Moves the mouse cursor while the left button is held down.

`dragRel(xOffset, yOffset)` Moves the mouse cursor relative to its current position while the left button is held down.

`click(x, y, button)` Simulates a click (left button by default).

`rightClick()` Simulates a right-button click.

`middleClick()` Simulates a middle-button click.

`doubleClick()` Simulates a double left-button click.

`mouseDown(x, y, button)` Simulates pressing down the given button at the position *x*, *y*.

`mouseUp(x, y, button)` Simulates releasing the given button at the position *x*, *y*.

`scroll(units)` Simulates the scroll wheel. A positive argument scrolls up; a negative argument scrolls down.

`typewrite(message)` Types the characters in the given message string.

`typewrite([key1, key2, key3])` Types the given keyboard key strings.

`press(key)` Presses the given keyboard key string.

`keyDown(key)` Simulates pressing down the given keyboard key.

`keyUp(key)` Simulates releasing the given keyboard key.

`hotkey([key1, key2, key3])` Simulates pressing the given keyboard key strings down in order and then releasing them in reverse order.

`screenshot()` Returns a screenshot as an `Image` object. (See Chapter 17 for information on `Image` objects.)

Project: Automatic Form Filler

Of all the boring tasks, filling out forms is the most dreaded of chores. It's only fitting that now, in the final chapter project, you will slay it. Say you have a huge amount of data in a spreadsheet, and you have to tediously retype it into some other application's form interface—with no intern to do it for you. Although some applications will have an Import feature that will allow you to upload a spreadsheet with the information, sometimes it seems that there is no other way than mindlessly clicking and typing for hours on end. You've come this far in this book; you know that *of course* there's another way.

The form for this project is a Google Docs form that you can find at *http://nostarch.com/automatestuff*. It looks like Figure 18-4.

Figure 18-4: The form used for this project

At a high level, here's what your program should do:

- Click the first text field of the form.
- Move through the form, typing information into each field.
- Click the Submit button.
- Repeat the process with the next set of data.

This means your code will need to do the following:

- Call `pyautogui.click()` to click the form and Submit button.
- Call `pyautogui.typewrite()` to enter text into the fields.
- Handle the `KeyboardInterrupt` exception so the user can press CTRL-C to quit.

Open a new file editor window and save it as *formFiller.py*.

Step 1: Figure Out the Steps

Before writing code, you need to figure out the exact keystrokes and mouse clicks that will fill out the form once. The *mouseNow.py* script on page 417 can help you figure out specific mouse coordinates. You need to know only the coordinates of the first text field. After clicking the first field, you can just press TAB to move focus to the next field. This will save you from having to figure out the x- and y-coordinates to click for every field.

Here are the steps for entering data into the form:

1. Click the Name field. (Use *mouseNow.py* to determine the coordinates after maximizing the browser window. On OS X, you may need to click twice: once to put the browser in focus and again to click the Name field.)
2. Type a name and then press TAB.
3. Type a greatest fear and then press TAB.
4. Press the down arrow key the correct number of times to select the wizard power source: once for *wand*, twice for *amulet*, three times for *crystal ball*, and four times for *money*. Then press TAB. (Note that on OS X, you will have to press the down arrow key one more time for each option. For some browsers, you may need to press the ENTER key as well.)
5. Press the right arrow key to select the answer to the RoboCop question. Press it once for *2*, twice for *3*, three times for *4*, or four times for *5*; or just press the spacebar to select *1* (which is highlighted by default). Then press TAB.
6. Type an additional comment and then press TAB.
7. Press the ENTER key to "click" the Submit button.
8. After submitting the form, the browser will take you to a page where you will need to click a link to return to the form page.

Note that if you run this program again later, you may have to update the mouse click coordinates, since the browser window might have changed position. To work around this, always make sure the browser window is maximized before finding the coordinates of the first form field. Also, different browsers on different operating systems might work slightly differently from the steps given here, so check that these keystroke combinations work for your computer before running your program.

Step 2: Set Up Coordinates

Load the example form you downloaded (Figure 18-4) in a browser and maximize your browser window. Open a new Terminal or command line window to run the *mouseNow.py* script, and then mouse over the Name field to figure out its the x- and y-coordinates. These numbers will be assigned to the `nameField` variable in your program. Also, find out the x- and y-coordinates and RGB tuple value of the blue Submit button. These values will be assigned to the `submitButton` and `submitButtonColor` variables, respectively.

Next, fill in some dummy data for the form and click **Submit**. You need to see what the next page looks like so that you can use *mouseNow.py* to find the coordinates of the *Submit another response* link on this new page.

Make your source code look like the following, being sure to replace all the values in italics with the coordinates you determined from your own tests:

```
#! python3
# formFiller.py - Automatically fills in the form.

import pyautogui, time

# Set these to the correct coordinates for your computer.
nameField = (648, 319)
submitButton = (651, 817)
submitButtonColor = (75, 141, 249)
submitAnotherLink = (760, 224)

# TODO: Give the user a chance to kill the script.

# TODO: Wait until the form page has loaded.

# TODO: Fill out the Name Field.

# TODO: Fill out the Greatest Fear(s) field.

# TODO: Fill out the Source of Wizard Powers field.

# TODO: Fill out the RoboCop field.

# TODO: Fill out the Additional Comments field.

# TODO: Click Submit.

# TODO: Wait until form page has loaded.

# TODO: Click the Submit another response link.
```

Now you need the data you actually want to enter into this form. In the real world, this data might come from a spreadsheet, a plaintext file, or a website, and it would require additional code to load into the program. But for this project, you'll just hardcode all this data in a variable. Add the following to your program:

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip--

formData = [{'name': 'Alice', 'fear': 'eavesdroppers', 'source': 'wand',
            'robocop': 4, 'comments': 'Tell Bob I said hi.'},
            {'name': 'Bob', 'fear': 'bees', 'source': 'amulet', 'robocop': 4,
            'comments': 'n/a'},
```

```
            {'name': 'Carol', 'fear': 'puppets', 'source': 'crystal ball',
            'robocop': 1, 'comments': 'Please take the puppets out of the
            break room.'},
            {'name': 'Alex Murphy', 'fear': 'ED-209', 'source': 'money',
            'robocop': 5, 'comments': 'Protect the innocent. Serve the public
            trust. Uphold the law.'},
           ]

--snip--
```

The `formData` list contains four dictionaries for four different names. Each dictionary has names of text fields as keys and responses as values. The last bit of setup is to set PyAutoGUI's `PAUSE` variable to wait half a second after each function call. Add the following to your program after the `formData` assignment statement:

```
pyautogui.PAUSE = 0.5
```

Step 3: Start Typing Data

A `for` loop will iterate over each of the dictionaries in the `formData` list, passing the values in the dictionary to the PyAutoGUI functions that will virtually type in the text fields.

Add the following code to your program:

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip--

for person in formData:
    # Give the user a chance to kill the script.
    print('>>> 5 SECOND PAUSE TO LET USER PRESS CTRL-C <<<')
❶   time.sleep(5)

    # Wait until the form page has loaded.
❷   while not pyautogui.pixelMatchesColor(submitButton[0], submitButton[1],
    submitButtonColor):
        time.sleep(0.5)

--snip--
```

As a small safety feature, the script has a five-second pause ❶ that gives the user a chance to hit CTRL-C (or move the mouse cursor to the upper-left corner of the screen to raise the `FailSafeException` exception) to shut the program down in case it's doing something unexpected. Then the program waits until the Submit button's color is visible ❷, letting the program know that the form page has loaded. Remember that you figured out the

coordinate and color information in step 2 and stored it in the `submitButton` and `submitButtonColor` variables. To use `pixelMatchesColor()`, you pass the coordinates `submitButton[0]` and `submitButton[1]`, and the color `submitButtonColor`.

After the code that waits until the Submit button's color is visible, add the following:

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip--

❶     print('Entering %s info...' % (person['name']))
❷     pyautogui.click(nameField[0], nameField[1])

      # Fill out the Name field.
❸     pyautogui.typewrite(person['name'] + '\t')

      # Fill out the Greatest Fear(s) field.
❹     pyautogui.typewrite(person['fear'] + '\t')

--snip--
```

We add an occasional `print()` call to display the program's status in its Terminal window to let the user know what's going on ❶.

Since the program knows that the form is loaded, it's time to call `click()` to click the Name field ❷ and `typewrite()` to enter the string in `person['name']` ❸. The `'\t'` character is added to the end of the string passed to `typewrite()` to simulate pressing TAB, which moves the keyboard focus to the next field, Greatest Fear(s). Another call to `typewrite()` will type the string in `person['fear']` into this field and then tab to the next field in the form ❹.

Step 4: Handle Select Lists and Radio Buttons

The drop-down menu for the "wizard powers" question and the radio buttons for the RoboCop field are trickier to handle than the text fields. To click these options with the mouse, you would have to figure out the x- and y-coordinates of each possible option. It's easier to use the keyboard arrow keys to make a selection instead.

Add the following to your program:

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip--

      # Fill out the Source of Wizard Powers field.
❶     if person['source'] == 'wand':
❷         pyautogui.typewrite(['down', '\t'])
```

```
    elif person['source'] == 'amulet':
        pyautogui.typewrite(['down', 'down', '\t'])
    elif person['source'] == 'crystal ball':
        pyautogui.typewrite(['down', 'down', 'down', '\t'])
    elif person['source'] == 'money':
        pyautogui.typewrite(['down', 'down', 'down', 'down', '\t'])

    # Fill out the RoboCop field.
❸   if person['robocop'] == 1:
❹       pyautogui.typewrite([' ', '\t'])
    elif person['robocop'] == 2:
        pyautogui.typewrite(['right', '\t'])
    elif person['robocop'] == 3:
        pyautogui.typewrite(['right', 'right', '\t'])
    elif person['robocop'] == 4:
        pyautogui.typewrite(['right', 'right', 'right', '\t'])
    elif person['robocop'] == 5:
        pyautogui.typewrite(['right', 'right', 'right', 'right', '\t'])

--snip--
```

Once the drop-down menu has focus (remember that you wrote code to simulate pressing TAB after filling out the Greatest Fear(s) field), pressing the down arrow key will move to the next item in the selection list. Depending on the value in `person['source']`, your program should send a number of down arrow keypresses before tabbing to the next field. If the value at the `'source'` key in this user's dictionary is `'wand'` ❶, we simulate pressing the down arrow key once (to select *Wand*) and pressing TAB ❷. If the value at the `'source'` key is `'amulet'`, we simulate pressing the down arrow key twice and pressing TAB, and so on for the other possible answers.

The radio buttons for the RoboCop question can be selected with the right arrow keys—or, if you want to select the first choice ❸, by just pressing the spacebar ❹.

Step 5: Submit the Form and Wait

You can fill out the Additional Comments field with the `typewrite()` function by passing `person['comments']` as an argument. You can type an additional `'\t'` to move the keyboard focus to the next field or the Submit button. Once the Submit button is in focus, calling `pyautogui.press('enter')` will simulate pressing the ENTER key and submit the form. After submitting the form, your program will wait five seconds for the next page to load.

Once the new page has loaded, it will have a *Submit another response* link that will direct the browser to a new, empty form page. You stored the coordinates of this link as a tuple in `submitAnotherLink` in step 2, so pass these coordinates to `pyautogui.click()` to click this link.

With the new form ready to go, the script's outer `for` loop can continue to the next iteration and enter the next person's information into the form.

Complete your program by adding the following code:

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip--

    # Fill out the Additional Comments field.
    pyautogui.typewrite(person['comments'] + '\t')

    # Click Submit.
    pyautogui.press('enter')

    # Wait until form page has loaded.
    print('Clicked Submit.')
    time.sleep(5)

    # Click the Submit another response link.
    pyautogui.click(submitAnotherLink[0], submitAnotherLink[1])
```

Once the main `for` loop has finished, the program will have plugged in the information for each person. In this example, there are only four people to enter. But if you had *4,000* people, then writing a program to do this would save you a lot of time and typing!

Summary

GUI automation with the `pyautogui` module allows you to interact with applications on your computer by controlling the mouse and keyboard. While this approach is flexible enough to do anything that a human user can do, the downside is that these programs are fairly blind to what they are clicking or typing. When writing GUI automation programs, try to ensure that they will crash quickly if they're given bad instructions. Crashing is annoying, but it's much better than the program continuing in error.

You can move the mouse cursor around the screen and simulate mouse clicks, keystrokes, and keyboard shortcuts with PyAutoGUI. The `pyautogui` module can also check the colors on the screen, which can provide your GUI automation program with enough of an idea of the screen contents to know whether it has gotten offtrack. You can even give PyAutoGUI a screenshot and let it figure out the coordinates of the area you want to click.

You can combine all of these PyAutoGUI features to automate any mindlessly repetitive task on your computer. In fact, it can be downright hypnotic to watch the mouse cursor move on its own and see text appear on the screen automatically. Why not spend the time you saved by sitting back and watching your program do all your work for you? There's a certain satisfaction that comes from seeing how your cleverness has saved you from the boring stuff.

Practice Questions

1. How can you trigger PyAutoGUI's fail safe to stop a program?
2. What function returns the current `resolution()`?
3. What function returns the coordinates for the mouse cursor's current position?
4. What is the difference between `pyautogui.moveTo()` and `pyautogui.moveRel()`?
5. What functions can be used to drag the mouse?
6. What function call will type out the characters of `"Hello world!"`?
7. How can you do keypresses for special keys such as the keyboard's left arrow key?
8. How can you save the current contents of the screen to an image file named *screenshot.png*?
9. What code would set a two second pause after every PyAutoGUI function call?

Practice Projects

For practice, write programs that do the following.

Looking Busy

Many instant messaging programs determine whether you are idle, or away from your computer, by detecting a lack of mouse movement over some period of time—say, ten minutes. Maybe you'd like to sneak away from your desk for a while but don't want others to see your instant messenger status go into idle mode. Write a script to nudge your mouse cursor slightly every ten seconds. The nudge should be small enough so that it won't get in the way if you do happen to need to use your computer while the script is running.

Instant Messenger Bot

Google Talk, Skype, Yahoo Messenger, AIM, and other instant messaging applications often use proprietary protocols that make it difficult for others to write Python modules that can interact with these programs. But even these proprietary protocols can't stop you from writing a GUI automation tool.

The Google Talk application has a search bar that lets you enter a username on your friend list and open a messaging window when you press ENTER. The keyboard focus automatically moves to the new window. Other instant messenger applications have similar ways to open new message windows. Write a program that will automatically send out a notification message to a select group of people on your friend list. Your program may have to deal with exceptional cases, such as friends being offline, the chat window appearing at different coordinates on the screen, or confirmation

boxes that interrupt your messaging. Your program will have to take screenshots to guide its GUI interaction and adopt ways of detecting when its virtual keystrokes aren't being sent.

NOTE *You may want to set up some fake test accounts so that you don't accidentally spam your real friends while writing this program.*

Game-Playing Bot Tutorial

There is a great tutorial titled "How to Build a Python Bot That Can Play Web Games" that you can find at *http://nostarch.com/automatestuff/*. This tutorial explains how to create a GUI automation program in Python that plays a Flash game called Sushi Go Round. The game involves clicking the correct ingredient buttons to fill customers' sushi orders. The faster you fill orders without mistakes, the more points you get. This is a perfectly suited task for a GUI automation program—and a way to cheat to a high score! The tutorial covers many of the same topics that this chapter covers but also includes descriptions of PyAutoGUI's basic image recognition features.

A

INSTALLING THIRD-PARTY MODULES

Beyond the standard library of modules packaged with Python, other developers have written their own modules to extend Python's capabilities even further. The primary way to install third-party modules is to use Python's pip tool. This tool securely downloads and installs Python modules onto your computer from *https://pypi.python.org/*, the website of the Python Software Foundation. PyPI, or the Python Package Index, is a sort of free app store for Python modules.

The pip Tool

The executable file for the pip tool is called *pip* on Windows and *pip3* on OS X and Linux. On Windows, you can find pip at *C:\Python34\Scripts\pip.exe*. On OS X, it is in */Library/Frameworks/Python.framework/Versions/3.4/bin/pip3*. On Linux, it is in */usr/bin/pip3*.

While pip comes automatically installed with Python 3.4 on Windows and OS X, you must install it separately on Linux. To install pip3 on Ubuntu or Debian Linux, open a new Terminal window and enter **`sudo apt-get install python3-pip`**. To install pip3 on Fedora Linux, enter **`sudo yum install python3-pip`** into a Terminal window. You will need to enter the administrator password for your computer in order to install this software.

Installing Third-Party Modules

The pip tool is meant to be run from the command line: You pass it the command `install` followed by the name of the module you want to install. For example, on Windows you would enter `pip install` *`ModuleName`*, where *`ModuleName`* is the name of the module. On OS X and Linux, you'll have to run pip3 with the `sudo` prefix to grant administrative privileges to install the module. You would need to type `sudo pip3 install` *`ModuleName`*.

If you already have the module installed but would like to upgrade it to the latest version available on PyPI, run `pip install -U` *`ModuleName`* (or `pip3 install -U` *`ModuleName`* on OS X and Linux).

After installing the module, you can test that it installed successfully by running `import` *`ModuleName`* in the interactive shell. If no error messages are displayed, you can assume the module was installed successfully.

You can install all of the modules covered in this book by running the commands listed next. (Remember to replace `pip` with `pip3` if you're on OS X or Linux.)

- `pip install send2trash`
- `pip install requests`
- `pip install beautifulsoup4`
- `pip install selenium`
- `pip install openpyxl`
- `pip install PyPDF2`
- `pip install python-docx` (install `python-docx`, not `docx`)
- `pip install imapclient`
- `pip install pyzmail`
- `pip install twilio`
- `pip install pillow`
- `pip install pyobjc-core` (on OS X only)
- `pip install pyobjc` (on OS X only)
- `pip install python3-xlib` (on Linux only)
- `pip install pyautogui`

NOTE *For OS X users: The `pyobjc` module can take 20 minutes or longer to install, so don't be alarmed if it takes a while. You should also install the `pyobjc-core` module first, which will reduce the overall installation time.*

B

RUNNING PROGRAMS

If you have a program open in IDLE's file editor, running it is a simple matter of pressing F5 or selecting the Run ▸Run Module menu item. This is an easy way to run programs while writing them, but opening IDLE to run your finished programs can be a burden. There are more convenient ways to execute Python scripts.

Shebang Line

The first line of all your Python programs should be a *shebang* line, which tells your computer that you want Python to execute this program. The shebang line begins with `#!`, but the rest depends on your operating system.

- On Windows, the shebang line is `#! python3`.
- On OS X, the shebang line is `#! /usr/bin/env python3`.
- On Linux, the shebang line is `#! /usr/bin/python3`.

You will be able to run Python scripts from IDLE without the shebang line, but the line is needed to run them from the command line.

Running Python Programs on Windows

On Windows, the Python 3.4 interpreter is located at *C:\Python34\python.exe*. Alternatively, the convenient *py.exe* program will read the shebang line at the top of the *.py* file's source code and run the appropriate version of Python for that script. The *py.exe* program will make sure to run the Python program with the correct version of Python if multiple versions are installed on your computer.

To make it convenient to run your Python program, create a *.bat batch file* for running the Python program with *py.exe*. To make a batch file, make a new text file containing a single line like the following:

```
@py.exe C:\path\to\your\pythonScript.py %*
```

Replace this path with the absolute path to your own program, and save this file with a *.bat* file extension (for example, *pythonScript.bat*). This batch file will keep you from having to type the full absolute path for the Python program every time you want to run it. I recommend you place all your batch and *.py* files in a single folder, such as *C:\MyPythonScripts* or *C:\Users\YourName\PythonScripts*.

The *C:\MyPythonScripts* folder should be added to the system path on Windows so that you can run the batch files in it from the Run dialog. To do this, modify the `PATH` environment variable. Click the **Start** button and type **Edit environment variables for your account**. This option should autocomplete after you've begun to type it. The Environment Variables window that appears will look like Figure B-1.

From System variables, select the Path variable and click **Edit**. In the Value text field, append a semicolon, type **C:\MyPythonScripts**, and then click **OK**. Now you can run any Python script in the *C:\MyPythonScripts* folder by simply pressing WIN-R and entering the script's name. Running `pythonScript`, for instance, will run *pythonScript.bat*, which in turn will save you from having to run the whole command `py.exe C:\MyPythonScripts\pythonScript.py` from the Run dialog.

Figure B-1: The Environment Variables window on Windows

Running Python Programs on OS X and Linux

On OS X, selecting Applications ▸Utilities ▸Terminal will bring up a *Terminal* window. A Terminal window is a way to enter commands on your computer using only text, rather than clicking through a graphic interface. To bring up the Terminal window on Ubuntu Linux, press the WIN (or SUPER) key to bring up Dash and type in **Terminal**.

The Terminal window will begin in the home folder of your user account. If my username is *asweigart*, the home folder will be */Users/asweigart* on OS X and */home/asweigart* on Linux. The tilde (~) character is a shortcut for your home folder, so you can enter `cd ~` to change to your home folder. You can also use the `cd` command to change the current working directory to any other directory. On both OS X and Linux, the `pwd` command will print the current working directory.

To run your Python programs, save your *.py* file to your home folder. Then, change the *.py* file's permissions to make it executable by running `chmod +x pythonScript.py`. File permissions are beyond the scope of this book, but you will need to run this command on your Python file if you want to run the program from the Terminal window. Once you do so, you will be able to run your script whenever you want by opening a Terminal window and entering `./pythonScript.py`. The shebang line at the top of the script will tell the operating system where to locate the Python interpreter.

Running Python Programs with Assertions Disabled

You can disable the `assert` statements in your Python programs for a slight performance improvement. When running Python from the terminal, include the `-O` switch after `python` or `python3` and before the name of the *.py* file. This will run an optimized version of your program that skips the assertion checks.

C

ANSWERS TO THE PRACTICE QUESTIONS

This appendix contains the answers to the practice problems at the end of each chapter. I highly recommend that you take the time to work through these problems. Programming is more than memorizing syntax and a list of function names. As when learning a foreign language, the more practice you put into it, the more you will get out of it. There are many websites with practice programming problems as well. You can find a list of these at *http://nostarch.com/automatestuff/*.

Chapter 1

1. The operators are `+`, `-`, `*`, and `/`. The values are `'hello'`, `-88.8`, and `5`.
2. The string is `'spam'`; the variable is `spam`. Strings always start and end with quotes.
3. The three data types introduced in this chapter are integers, floating-point numbers, and strings.
4. An expression is a combination of values and operators. All expressions evaluate (that is, reduce) to a single value.
5. An expression evaluates to a single value. A statement does not.
6. The `bacon` variable is set to `20`. The `bacon + 1` expression does not reassign the value in `bacon` (that would need an assignment statement: `bacon = bacon + 1`).
7. Both expressions evaluate to the string `'spamspamspam'`.
8. Variable names cannot begin with a number.
9. The `int()`, `float()`, and `str()` functions will evaluate to the integer, floating-point number, and string versions of the value passed to them.
10. The expression causes an error because `99` is an integer, and only strings can be concatenated to other strings with the `+` operator. The correct way is `I have eaten ' + str(99) + ' burritos.'`.

Chapter 2

1. `True` and `False`, using capital *T* and *F*, with the rest of the word in lowercase
2. `and`, `or`, and `not`
3. `True and True` is `True`.
   `True and False` is `False`.
   `False and True` is `False`.
   `False and False` is `False`.
   `True or True` is `True`.
   `True or False` is `True`.
   `False or True` is `True`.
   `False or False` is `False`.
   `not True` is `False`.
   `not False` is `True`.
4. `False`
   `False`
   `True`
   `False`
   `False`
   `True`

5. ==, !=, <, >, <=, and >=.
6. == is the equal to operator that compares two values and evaluates to a Boolean, while = is the assignment operator that stores a value in a variable.
7. A condition is an expression used in a flow control statement that evaluates to a Boolean value.
8. The three blocks are everything inside the `if` statement and the lines `print('bacon')` and `print('ham')`.

```
print('eggs')
if spam > 5:
    print('bacon')
else:
    print('ham')
print('spam')
```

9. The code:

```
if spam == 1:
    print('Hello')
elif spam == 2:
    print('Howdy')
else:
    print('Greetings!')
```

10. Press CTRL-C to stop a program stuck in an infinite loop.
11. The `break` statement will move the execution outside and just after a loop. The `continue` statement will move the execution to the start of the loop.
12. They all do the same thing. The `range(10)` call ranges from `0` up to (but not including) `10`, `range(0, 10)` explicitly tells the loop to start at `0`, and `range(0, 10, 1)` explicitly tells the loop to increase the variable by `1` on each iteration.
13. The code:

```
for i in range(1, 11):
    print(i)
```

and:

```
i = 1
while i <= 10:
    print(i)
    i = i + 1
```

14. This function can be called with `spam.bacon()`.

Chapter 3

1. Functions reduce the need for duplicate code. This makes programs shorter, easier to read, and easier to update.
2. The code in a function executes when the function is called, not when the function is defined.
3. The `def` statement defines (that is, creates) a function.
4. A function consists of the `def` statement and the code in its `def` clause. A function call is what moves the program execution into the function, and the function call evaluates to the function's return value.
5. There is one global scope, and a local scope is created whenever a function is called.
6. When a function returns, the local scope is destroyed, and all the variables in it are forgotten.
7. A return value is the value that a function call evaluates to. Like any value, a return value can be used as part of an expression.
8. If there is no return statement for a function, its return value is `None`.
9. A `global` statement will force a variable in a function to refer to the global variable.
10. The data type of `None` is `NoneType`.
11. That `import` statement imports a module named `areallyourpetsnamederic`. (This isn't a real Python module, by the way.)
12. This function can be called with `spam.bacon()`.
13. Place the line of code that might cause an error in a `try` clause.
14. The code that could potentially cause an error goes in the `try` clause. The code that executes if an error happens goes in the `except` clause.

Chapter 4

1. The empty list value, which is a list value that contains no items. This is similar to how `''` is the empty string value.
2. `spam[2] = 'hello'` (Notice that the third value in a list is at index `2` because the first index is `0`.)
3. `'d'` (Note that `'3' * 2` is the string `'33'`, which is passed to `int()` before being divided by `11`. This eventually evaluates to `3`. Expressions can be used wherever values are used.)
4. `'d'` (Negative indexes count from the end.)
5. `['a', 'b']`
6. `1`
7. `[3.14, 'cat', 11, 'cat', True, 99]`
8. `[3.14, 11, 'cat', True]`

9. The operator for list concatenation is `+`, while the operator for replication is `*`. (This is the same as for strings.)
10. While `append()` will add values only to the end of a list, `insert()` can add them anywhere in the list.
11. The `del` statement and the `remove()` list method are two ways to remove values from a list.
12. Both lists and strings can be passed to `len()`, have indexes and slices, be used in `for` loops, be concatenated or replicated, and be used with the `in` and `not in` operators.
13. Lists are mutable; they can have values added, removed, or changed. Tuples are immutable; they cannot be changed at all. Also, tuples are written using parentheses, (and), while lists use the square brackets, [and].
14. `(42,)` (The trailing comma is mandatory.)
15. The `tuple()` and `list()` functions, respectively
16. They contain references to list values.
17. The `copy.copy()` function will do a shallow copy of a list, while the `copy.deepcopy()` function will do a deep copy of a list. That is, only `copy.deepcopy()` will duplicate any lists inside the list.

Chapter 5

1. Two curly brackets: {}
2. `{'foo': 42}`
3. The items stored in a dictionary are unordered, while the items in a list are ordered.
4. You get a `KeyError` error.
5. There is no difference. The `in` operator checks whether a value exists as a key in the dictionary.
6. `'cat' in spam` checks whether there is a `'cat'` key in the dictionary, while `'cat' in spam.values()` checks whether there is a value `'cat'` for one of the keys in `spam`.
7. `spam.setdefault('color', 'black')`
8. `pprint.pprint()`

Chapter 6

1. Escape characters represent characters in string values that would otherwise be difficult or impossible to type into code.
2. `\n` is a newline; `\t` is a tab.
3. The `\\` escape character will represent a backslash character.

4. The single quote in `Howl's` is fine because you've used double quotes to mark the beginning and end of the string.
5. Multiline strings allow you to use newlines in strings without the `\n` escape character.
6. The expressions evaluate to the following:
   - `'e'`
   - `'Hello'`
   - `'Hello'`
   - `'lo world!`
7. The expressions evaluate to the following:
   - `'HELLO'`
   - `True`
   - `'hello'`
8. The expressions evaluate to the following:
   - `['Remember,', 'remember,', 'the', 'fifth', 'of', 'November.']`
   - `'There-can-be-only-one.'`
9. The `rjust()`, `ljust()`, and `center()` string methods, respectively
10. The `lstrip()` and `rstrip()` methods remove whitespace from the left and right ends of a string, respectively.

Chapter 7

1. The `re.compile()` function returns `Regex` objects.
2. Raw strings are used so that backslashes do not have to be escaped.
3. The `search()` method returns `Match` objects.
4. The `group()` method returns strings of the matched text.
5. Group `0` is the entire match, group `1` covers the first set of parentheses, and group `2` covers the second set of parentheses.
6. Periods and parentheses can be escaped with a backslash: `\.`, `\(`, and `\)`.
7. If the regex has no groups, a list of strings is returned. If the regex has groups, a list of tuples of strings is returned.
8. The `|` character signifies matching "either, or" between two groups.
9. The `?` character can either mean " match zero or one of the preceding group" or be used to signify nongreedy matching.
10. The `+` matches one or more. The `*` matches zero or more.
11. The `{3}` matches exactly three instances of the preceding group. The `{3,5}` matches between three and five instances.
12. The `\d`, `\w`, and `\s` shorthand character classes match a single digit, word, or space character, respectively.
13. The `\D`, `\W`, and `\S` shorthand character classes match a single character that is not a digit, word, or space character, respectively.

14. Passing `re.I` or `re.IGNORECASE` as the second argument to `re.compile()` will make the matching case insensitive.
15. The `.` character normally matches any character except the newline character. If `re.DOTALL` is passed as the second argument to `re.compile()`, then the dot will also match newline characters.
16. The `.*` performs a greedy match, and the `.*?` performs a nongreedy match.
17. Either `[0-9a-z]` or `[a-z0-9]`
18. `'X drummers, X pipers, five rings, X hens'`
19. The `re.VERBOSE` argument allows you to add whitespace and comments to the string passed to `re.compile()`.
20. `re.compile(r'^\d{1,3}(,{3})*$')` will create this regex, but other regex strings can produce a similar regular expression.
21. `re.compile(r'[A-Z][a-z]*\sNakamoto')`
22. `re.compile(r'(Alice|Bob|Carol)\s(eats|pets|throws)\s(apples|cats|baseballs)\.', re.IGNORECASE)`

Chapter 8

1. Relative paths are relative to the current working directory.
2. Absolute paths start with the root folder, such as / or *C:*.
3. The `os.getcwd()` function returns the current working directory. The `os.chdir()` function *changes* the current working directory.
4. The `.` folder is the current folder, and `..` is the parent folder.
5. *C:\bacon\eggs* is the dir name, while *spam.txt* is the base name.
6. The string `'r'` for read mode, `'w'` for write mode, and `'a'` for append mode
7. An existing file opened in write mode is erased and completely overwritten.
8. The `read()` method returns the file's entire contents as a single string value. The `readlines()` method returns a list of strings, where each string is a line from the file's contents.
9. A shelf value resembles a dictionary value; it has keys and values, along with `keys()` and `values()` methods that work similarly to the dictionary methods of the same names.

Chapter 9

1. The `shutil.copy()` function will copy a single file, while `shutil.copytree()` will copy an entire folder, along with all its contents.
2. The `shutil.move()` function is used for renaming files, as well as moving them.

3. The `send2trash` functions will move a file or folder to the recycle bin, while `shutil` functions will permanently delete files and folders.
4. The `zipfile.ZipFile()` function is equivalent to the `open()` function; the first argument is the filename, and the second argument is the mode to open the ZIP file in (read, write, or append).

Chapter 10

1. `assert(spam >= 10, 'The spam variable is less than 10.')`
2. `assert(eggs.lower() != bacon.lower(), 'The eggs and bacon variables are the same!')` or `assert(eggs.upper() != bacon.upper(), 'The eggs and bacon variables are the same!')`
3. `assert(False, 'This assertion always triggers.')`
4. To be able to call `logging.debug()`, you must have these two lines at the start of your program:

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s -  %(message)s')
```

5. To be able to send logging messages to a file named *programLog.txt* with `logging.debug()`, you must have these two lines at the start of your program:

```
import logging
>>> logging.basicConfig(filename='programLog.txt', level=logging.DEBUG,
format=' %(asctime)s -  %(levelname)s -  %(message)s')
```

6. DEBUG, INFO, WARNING, ERROR, and CRITICAL
7. `logging.disable(logging.CRITICAL)`
8. You can disable logging messages without removing the logging function calls. You can selectively disable lower-level logging messages. You can create logging messages. Logging messages provides a timestamp.
9. The Step button will move the debugger into a function call. The Over button will quickly execute the function call without stepping into it. The Out button will quickly execute the rest of the code until it steps out of the function it currently is in.
10. After you click Go, the debugger will stop when it has reached the end of the program or a line with a breakpoint.
11. A breakpoint is a setting on a line of code that causes the debugger to pause when the program execution reaches the line.
12. To set a breakpoint in IDLE, right-click the line and select **Set Breakpoint** from the context menu.

Chapter 11

1. The `webbrowser` module has an `open()` method that will launch a web browser to a specific URL, and that's it. The `requests` module can download files and pages from the Web. The `BeautifulSoup` module parses HTML. Finally, the `selenium` module can launch and control a browser.
2. The `requests.get()` function returns a `Response` object, which has a `text` attribute that contains the downloaded content as a string.
3. The `raise_for_status()` method raises an exception if the download had problems and does nothing if the download succeeded.
4. The `status_code` attribute of the `Response` object contains the HTTP status code.
5. After opening the new file on your computer in `'wb'` "write binary" mode, use a `for` loop that iterates over the `Response` object's `iter_content()` method to write out chunks to the file. Here's an example:

```
saveFile = open('filename.html', 'wb')
for chunk in res.iter_content(100000):
    saveFile.write(chunk)
```

6. F12 brings up the developer tools in Chrome. Pressing CTRL-SHIFT-C (on Windows and Linux) or ⌘-OPTION-C (on OS X) brings up the developer tools in Firefox.
7. Right-click the element in the page, and select **Inspect Element** from the menu.
8. `'#main'`
9. `'.highlight'`
10. `'div div'`
11. `'button[value="favorite"]'`
12. `spam.getText()`
13. `linkElem.attrs`
14. The `selenium` module is imported with `from selenium import webdriver`.
15. The `find_element_*` methods return the first matching element as a `WebElement` object. The `find_elements_*` methods return a list of all matching elements as `WebElement` objects.
16. The `click()` and `send_keys()` methods simulate mouse clicks and keyboard keys, respectively.
17. Calling the `submit()` method on any element within a form submits the form.
18. The `forward()`, `back()`, and `refresh()` `WebDriver` object methods simulate these browser buttons.

Chapter 12

1. The `openpyxl.load_workbook()` function returns a `Workbook` object.
2. The `get_sheet_names()` method returns a `Worksheet` object.
3. Call `wb.get_sheet_by_name('Sheet1')`.
4. Call `wb.get_active_sheet()`.
5. `sheet['C5'].value` or `sheet.cell(row=5, column=3).value`
6. `sheet['C5'] = 'Hello'` or `sheet.cell(row=5, column=3).value = 'Hello'`
7. `cell.row` and `cell.column`
8. They return the highest column and row with values in the sheet, respectively, as integer values.
9. `openpyxl.cell.column_index_from_string('M')`
10. `openpyxl.cell.get_column_letter(14)`
11. `sheet['A1':'F1']`
12. `wb.save('example.xlsx')`
13. A formula is set the same way as any value. Set the cell's `value` attribute to a string of the formula text. Remember that formulas begin with the = sign.
14. When calling `load_workbook()`, pass `True` for the `data_only` keyword argument.
15. `sheet.row_dimensions[5].height = 100`
16. `sheet.column_dimensions['C'].hidden = True`
17. OpenPyXL 2.0.5 does not load freeze panes, print titles, images, or charts.
18. Freeze panes are rows and columns that will always appear on the screen. They are useful for headers.
19. `openpyxl.charts.Reference()`, `openpyxl.charts.Series()`, `openpyxl.charts.BarChart()`, `chartObj.append(seriesObj)`, and `add_chart()`

Chapter 13

1. A `File` object returned from `open()`
2. Read-binary (`'rb'`) for `PdfFileReader()` and write-binary (`'wb'`) for `PdfFileWriter()`
3. Calling `getPage(4)` will return a `Page` object for page 5, since page 0 is the first page.
4. The `numPages` variable stores an integer of the number of pages in the `PdfFileReader` object.
5. Call `decrypt('swordfish')`.
6. The `rotateClockwise()` and `rotateCounterClockwise()` methods. The degrees to rotate is passed as an integer argument.

7. `docx.Document('demo.docx')`
8. A document contains multiple paragraphs. A paragraph begins on a new line and contains multiple runs. Runs are contiguous groups of characters within a paragraph.
9. Use `doc.paragraphs`.
10. A `Run` object has these variables (*not* a Paragraph).
11. `True` always makes the `Run` object bolded and `False` makes it always not bolded, no matter what the style's bold setting is. `None` will make the `Run` object just use the style's bold setting.
12. Call the `docx.Document()` function.
13. `doc.add_paragraph('Hello there!')`
14. The integers `0`, `1`, `2`, `3`, and `4`

Chapter 14

1. In Excel, spreadsheets can have values of data types other than strings; cells can have different fonts, sizes, or color settings; cells can have varying widths and heights; adjacent cells can be merged; and you can embed images and charts.
2. You pass a `File` object, obtained from a call to `open()`.
3. `File` objects need to be opened in read-binary (`'rb'`) for `Reader` objects and write-binary (`'wb'`) for `Writer` objects.
4. The `writerow()` method
5. The `delimiter` argument changes the string used to separate cells in a row. The `lineterminator` argument changes the string used to separate rows.
6. `json.loads()`
7. `json.dumps()`

Chapter 15

1. A reference moment that many date and time programs use. The moment is January 1st, 1970, UTC.
2. `time.time()`
3. `time.sleep(5)`
4. It returns the closest integer to the argument passed. For example, `round(2.4)` returns `2`.
5. A `datetime` object represents a specific moment in time. A `timedelta` object represents a duration of time.
6. `threadObj = threading.Thread(target=spam)`
7. `threadObj.start()`

8. Make sure that code running in one thread does not read or write the same variables as code running in another thread.
9. `subprocess.Popen('c:\\Windows\\System32\\calc.exe')`

Chapter 16

1. SMTP and IMAP, respectively
2. `smtplib.SMTP()`, `smtpObj.ehlo()`, `smptObj.starttls()`, and `smtpObj.login()`
3. `imapclient.IMAPClient()` and `imapObj.login()`
4. A list of strings of IMAP keywords, such as `'BEFORE <date>'`, `'FROM <string>'`, or `'SEEN'`
5. Assign the variable `imaplib._MAXLINE` a large integer value, such as `10000000`.
6. The `pyzmail` module reads downloaded emails.
7. You will need the Twilio account SID number, the authentication token number, and your Twilio phone number.

Chapter 17

1. An RGBA value is a tuple of 4 integers, each ranging from 0 to 255. The four integers correspond to the amount of red, green, blue, and alpha (transparency) in the color.
2. A function call to `ImageColor.getcolor('CornflowerBlue', 'RGBA')` will return `(100, 149, 237, 255)`, the RGBA value for that color.
3. A box tuple is a tuple value of four integers: the left edge x-coordinate, the top edge y-coordinate, the width, and the height, respectively.
4. `Image.open('zophie.png')`
5. `imageObj.size` is a tuple of two integers, the width and the height.
6. `imageObj.crop((0, 50, 50, 50))`. Notice that you are passing a box tuple to `crop()`, not four separate integer arguments.
7. Call the `imageObj.save('new_filename.png')` method of the `Image` object.
8. The `ImageDraw` module contains code to draw on images.
9. `ImageDraw` objects have shape-drawing methods such as `point()`, `line()`, or `rectangle()`. They are returned by passing the `Image` object to the `ImageDraw.Draw()` function.

Chapter 18

1. Move the mouse to the top-left corner of the screen, that is, the (0, 0) coordinates.
2. `pyautogui.size()` returns a tuple with two integers for the width and height of the screen.

3. `pyautogui.position()` returns a tuple with two integers for the x- and y-coordinates of the mouse cursor.
4. The `moveTo()` function moves the mouse to absolute coordinates on the screen, while the `moveRel()` function moves the mouse relative to the mouse's current position.
5. `pyautogui.dragTo()` and `pyautogui.dragRel()`
6. `pyautogui.typewrite('Hello world!')`
7. Either pass a list of keyboard key strings to `pyautogui.typewrite()` (such as `'left'`) or pass a single keyboard key string to `pyautogui.press()`.
8. `pyautogui.screenshot('screenshot.png')`
9. `pyautogui.PAUSE = 2`

INDEX

Symbols

= (assignment) operator, 18, 34
\ (backslash), 124, 151, 162, 174–175
    line continuation character, 93
^ (caret symbol), 162
    matching beginning of string, 159–160
    negative character classes, 159
: (colon), 38, 45, 54, 82, 127
{} (curly brackets), 105, 162
    greedy vs. nongreedy matching, 156–157
    matching specific repetitions with, 156
$ (dollar sign), 159–160, 162
. (dot character), 160–162
    using in paths, 175–176
    wildcard matches, 160–162
" (double quotes), 124
** (exponent) operator, 15
== (equal to) operator, 33, 34
/ (forward slash), 174–175
    division operator, 15, 88
> (greater than) operator, 33
>= (greater than or equal to) operator, 33
(hash character), 126
// (integer division/floored quotient) operator, 15
< (less than) operator, 33
<= (less than or equal to) operator, 33
% (modulus/remainder) operator, 15, 88
* (multiplication) operator, 15, 83, 88
!= (not equal to) operator, 33
() (parentheses), 96–97, 152–153
| (pipe character), 153–154, 164–165
+ (plus sign), 155–156, 162
    addition operator, 15, 17, 83, 88
? (question mark), 154–155, 162
' (single quote), 124
[] (square brackets), 80, 162
* (star), 162
    using with wildcard character, 161
    zero or more matches with, 155
- (subtraction) operator, 15, 88
''' (triple quotes), 125, 164
_ (underscore), 20

A

%A directive, 344
%a directive, 344
absolute paths, 175–179
abspath() function, 177
addition (+) operator, 15, 17, 83, 88
additive color model, 389
add_heading() method, 314
addPage() method, 299
add_paragraph() method, 313–314
add_picture() method, 315
add_run() method, 313–314
algebraic chess notation, 112–113
all_caps attribute, 311
ALL search key, 369
alpha, defined, 388
and operator, 35
ANSWERED search key, 370
API (application programming interface), 327–328
append() method, 89–90
application-specific passwords, 365
args keyword, 349
arguments, function, 23, 63
    keyword arguments, 65–66
    passing to processes, 354
    passing to threads, 348–349
assertions, 219–221
    disabling, 445
assignment (=) operator, 18, 34
AT&T mail, 363, 367
attributes, HTML, 241, 248
augmented assignment operators, 88–89

B

\b backspace escape character, 419
%B directive, 344
%b directive, 344
back() method, 261
backslash (\), 124, 151, 162, 174–175
BarChart() function, 290
basename() function, 178
BCC search key, 370
Beautiful Soup, 245. *See also* bs4 module
BeautifulSoup objects, 245–246
BEFORE search key, 369
binary files, 180–181, 184–185
binary operators, 35–37
bitwise or operator, 164–165
blank strings, 17
blocking execution, 337
blocks of code, 37–38
BODY search key, 369
bold attribute, 311
Boolean data type
    binary operators, 35–36
    flow control and, 32–33
    in operator, 87
    not in operator, 87
    "truthy" and "falsey" values, 53
    using binary and comparison operators together, 36–37
box tuples, 390
breakpoints, debugging using, 229–231
break statements
    overview, 49–50
    using in for loop, 55
browser, opening using webbrowser module, 234–236
bs4 module
    creating object from HTML, 245–246
    finding element with select() method, 246–247
    getting attribute, 248
    overview, 245
built-in functions, 57
bulleted list, creating in Wiki markup, 139–141
    copying and pasting clipboard, 139–140
    joining modified lines, 141
    overview, 139
    separating lines of text, 140

C

calling functions, 23
call stack, defined, 217
camelcase, 21
caret symbol (^), 162
    matching beginning of string, 159–160
    negative character classes, 159
Cascading Style Sheets (CSS)
    matching with selenium module, 258
    selectors, 246–247
case sensitivity, 21, 163
CC search key, 370
Cell objects, 268–269
cells, in Excel spreadsheets, 266
    accessing Cell object by its name, 268–269
    merging and unmerging, 286–287
    writing values to, 278–279
center() method, 133–134, 426
chaining method calls, 398
character classes, 158–159, 162
character styles, 310
charts, Excel, 288–290
chdir() function, 175
Chrome, developer tools in, 242–243
clear() method, 258
click() function, 420, 430, 431
clicking mouse, 420
click() method, 259
clipboard, using string from, 236
CMYK color model, 389
colon (:), 38, 45, 54, 82, 127
color values
    CMYK vs. RGB color models, 389
    RGBA values, 388–389
column_index_from_string() function, 270
columns, in Excel spreadsheets
    setting height and width of, 285–286
    slicing Worksheet objects to get Cell objects in, 270–272
Comcast mail, 363, 367
comma-delimited items, 80
command line arguments, 235
commentAfterDelay() function, 429
comments
    multiline, 126
    overview, 23

comparison operators
overview, 33–35
using binary operators with, 36–37
compile() function, 151, 152, 164–165
compressed files
backing up folder into, 209–212
creating ZIP files, 205–206
extracting ZIP files, 205
overview, 203–204
reading ZIP files, 204
computer screen
coordinates of, 415
resolution of, 416
concatenation
of lists, 83
string, 17–18
concurrency issues, 349
conditions, defined, 37
continue statements
overview, 50–53
using in for loop, 55
Coordinated Universal Time (UTC), 336
coordinates
of computer screen, 415
of an image, 389–390
copy() function, 100–101, 135, 198, 394
copytree() function, 198–199
countdown project, 357–358
counting down, 357
overview, 357
playing sound file, 357–358
cProfile.run() function, 337
crashes, program, 14
create_sheet() method, 278
CRITICAL level, 224
cron, 354
cropping images, 393–394
CSS (Cascading Style Sheets)
matching with selenium module, 258
selectors, 246–247
CSV files
defined, 319
delimeter for, 324
format overview, 320
line terminator for, 324
Reader objects, 321
reading data in loop, 322
removing header from, 324–327
looping through CSV files, 325
overview, 324–325
reading in CSV file, 325–326
writing out CSV file, 326–327
Writer objects, 322–323
curly brackets ({}), 105, 162
greedy vs. nongreedy matching, 156–157
matching specific repetitions with, 156
current working directory, 175

D

\D character class, 158
\d character class, 158
%d directive, 344
data structures
algebraic chess notation, 112–113
tic-tac-toe board, 113–117
data types
Booleans, 32
defined, 16
dictionaries, 105–106
floating-point numbers, 17
integers, 17
list() function, 97
lists, 80
mutable vs. immutable, 94–96
None value, 65
strings, 17
tuple() function, 97
tuples, 96–97
datetime module
arithmetic using, 343
converting objects to strings, 344–345
converting strings to objects, 345
fromtimestamp() function, 341
now() function, 341
overview, 341–342, 346
pausing program until time, 344
timedelta data type, 342–343
total_seconds() method, 342
datetime objects, 341–342
converting to strings, 344–345
converting from strings to, 345
debug() function, 222
debugging
assertions, 219–221
defined, 4
getting traceback as string, 217–218
in IDLE
overview, 225–227
stepping through program, 227–229
using breakpoints, 229–231

debugging (*continued*)
logging
disabling, 224–225
to file, 225
levels of, 223–224
logging module, 221–223
print() function and, 223
raising exceptions, 216–217
DEBUG level, 223
decimal numbers. *See* floating-point numbers
decode() method, 374–375
decryption, of PDF files, 297–298
deduplicating code, 62
deepcopy() function, 100–101
def statements, 62
with parameters, 63
DELETED search key, 370
delete_messages() method, 375
deleting files/folders
permanently, 200–201
using send2trash module, 201–202
del statements, 84
dictionaries
copy() function, 100–101
deepcopy() function, 100–101
get() method, 109
in operator, 109
items() method, 107–108
keys() method, 107–108
lists vs., 106–107
nesting, 117–119
not in operator, 109
overview, 105–106
setdefault() method, 110–111
values() method, 107–108
directories
absolute vs. relative paths, 175–176
backslash vs. forward slash, 174–175
copying, 198–199
creating, 176
current working directory, 175
defined, 173–174
deleting permanently, 200–201
deleting using send2trash module, 201–202
moving, 199–200
os.path module
absolute paths in, 177–179
file sizes, 179–180
folder contents, 179–180
overview, 177
path validity, 180
relative paths in, 177–179
renaming, 199–200
walking, 202–203
dirname() function, 178
disable() function, 224
division (/) operator, 15, 88
Document objects, 307–308
dollar sign ($), 159–160, 162
dot character (.), 160–162
using in paths, 175–176
wildcard matches, 160–162
dot-star character (.*), 161
doubleClick() function, 420, 430
double quotes ("), 124
double_strike attribute, 311
downloading
files from web, 239–240
web pages, 237–238
XKCD comics, 251–256, 350–352
DRAFT search key, 370
dragging mouse, 420–422
dragRel() function, 420, 422, 430
dragTo() function, 420, 430
drawing on images
ellipses, 407
example program, 407–408
ImageDraw module, 406
lines, 406–407
points, 406
polygons, 407
rectangles, 407
text, 408–410
dumps() function, 329
duration keyword arguments, 416

E

ehlo() method, 364, 379
elements, HTML, 240
elif statements, 40–45
ellipse() method, 407
else statements, 39–40
email addresses, extracting, 165–169
creating regex, 166–167
finding matches on clipboard, 167–168
joining matches into a string, 168
overview, 165–166
emails
deleting, 375
disconnecting from server, 375–376
fetching

folders, 368–369
getting message content, 372–373
logging into server, 368
overview, 366–367
raw messages, 373–375
gmail_search() method, 372
IMAP, 366
marking message as read, 372–373
searching, 368–371
sending
connecting to SMTP server, 363–364
disconnecting from server, 366
logging into server, 364–365
overview, 362
reminder, 376–380
sending "hello" message, 364
sending message, 365
TLS encryption, 364
SMTP, 362
emboss attribute, 311
encryption, of PDF files, 302–303
endswith() method, 131
epoch timestamps, 336, 341, 346
equal to (==) operator, 33, 34
ERROR level, 224
errors
crashes and, 14
help for, 8–9
escape characters, 124–125
evaluation, defined, 14
Excel spreadsheets
application support, 265–266
charts in, 288–290
column width, 285–286
converting between column letters and numbers, 270
creating documents, 277
creating worksheets, 278
deleting worksheets, 278
font styles, 282–284
formulas in, 284–285
freezing panes, 287–288
getting cell values, 268–269
getting rows and columns, 270–272
getting worksheet names, 268
merging and unmerging cells, 286–287
opening documents, 267
openpyxl module, 266
overview, 266–267
reading files
overview, 272–273
populating data structure, 274–275
reading data, 273–274
writing results to file, 275–276
and reminder emails project, 376–380
row height, 285–286
saving workbooks, 277
updating, 279–281
overview, 279–280
setup, 280
workbooks vs., 266
writing values to cells, 278–279
Exception objects, 217
exceptions
assertions and, 219–221
getting traceback as string, 217–218
handling, 72–74
raising, 216–217
execution, program
defined, 31
overview, 38
pausing until specific time, 344
terminating program with sys.exit(), 58
exists() function, 180
exit codes, 353–354
expand keyword, 398
exponent (**) operator, 15
expressions
conditions and, 37
in interactive shell, 14–16
expunge() method, 375
extensions, file, 173
extractall() method, 205
extracting ZIP files, 205
extract() method, 205

F

FailSafeException exception, 434
"falsey" values, 53
fetch() method, 371, 372–373
file editor, 21
file management
absolute vs. relative paths, 175–176
backslash vs. forward slash, 174–175
compressed files
backing up to, 209–212
creating ZIP files, 205–206

file management (*continued*)
compressed files (*continued*)
extracting ZIP files, 205
overview, 203–204
reading ZIP files, 204
creating directories, 176
current working directory, 175
multiclipboard project, 191–193
opening files, 181–182
os.path module
absolute paths in, 177–179
file sizes, 179–180
folder contents, 179–180
overview, 177
path validity, 180
relative paths in, 177–179
overview, 173–174
paths, 173–174
plaintext vs. binary files, 180–181
reading files, 182–183
renaming files, date styles, 206–209
saving variables with pformat() function, 185–186
send2trash module, 201–202
shelve module, 184–185
shutil module
copying files/folders, 198–199
deleting files/folders, 200–201
moving files/folders, 199–200
renaming files/folders, 199–200
walking directory trees, 202–203
writing files, 183–184
filenames, defined, 173
File objects, 182
findall() method, 157–158
find_element_by_* methods, 257–258
find_elements_by_* methods, 257–258
Firefox, developer tools in, 243
FLAGGED search key, 370
flipping images, 398–399
float() function, 25–28
floating-point numbers
integer equivalence, 27
overview, 17
rounding, 338
flow control
binary operators, 35–36
blocks of code, 37–38
Boolean values and, 32–33
break statements, 49–50
comparison operators, 33–35
conditions, 37
continue statements, 50–53
elif statements, 40–45
else statements, 39–40
if statements, 38–39
overview, 31–32
using binary and comparison operators together, 36–37
while loops, 45–49
folders
absolute vs. relative paths, 175–176
backing up to ZIP file, 209–212
creating new ZIP file, 211
figuring out ZIP filename, 210–211
walking directory tree, 211–212
backslash vs. forward slash, 174–175
copying, 198–199
creating, 176
current working directory, 175
defined, 173–174
deleting permanently, 200–201
deleting using send2trash module, 201–202
moving, 199–200
os.path module
absolute paths in, 177–179
file sizes, 179–180
folder contents, 179–180
overview, 177
path validity, 180
relative paths in, 177–179
renaming, 199–200
walking directory trees, 202–203
Font objects, 282–283
font styles, in Excel spreadsheets, 282–284
for loops
overview, 53–56
using dictionary items in, 108
using lists with, 86
format attribute, 392
format_description attribute, 392
formData list, 434
form filler project, 430–437
overview, 430–431
radio buttons, 435–436
select lists, 435–436
setting up coordinates, 432–434
steps in process, 431
submitting form, 436–437
typing data, 434–435

formulas, in Excel spreadsheets, 284–285
forward() method, 261
forward slash (/), 174–175
FROM search key, 370
fromtimestamp() function, 341, 346
functions. *See also names of individual functions*
  arguments, 23, 63
  as "black box", 72
  built-in, 57
  def statements, 63
  exception handling, 72–74
  keyword arguments, 65–66
  None value and, 65
  overview, 61–62
  parameters, 63
  return values, 63–65

G

get_active_sheet() method, 268
get_addresses() method, 374
get_attribute() method, 258
getcolor() function, 388–389, 393
get_column_letter() function, 270
getcwd() function, 175
get() function
  overview, 109
  requests module, 237
get_highest_column() method, 269, 377
get_highest_row() method, 269
get_payload() method, 374–375
getpixel() function, 400, 423, 424
get_sheet_by_name() method, 268
get_sheet_names() method, 268
getsize() function, 179
get_subject() method, 374
getText() function, 308–309
GIF format, 392
global scope, 70–71
Gmail, 363, 365, 367
gmail_search() method, 372
Google Maps, 234–236
graphical user interface automation. *See* GUI (graphical user interface) automation
greater than (>) operator, 33
greater than or equal to (>=) operator, 33
greedy matching
  dot-star for, 161
  in regular expressions, 156–157
group() method, 151, 152–153
groups, regular expression matching
  greedy, 156–157
  nongreedy, 157
  one or more, 155–156
  optional, 154–155
  specific reptitions, 156
  zero or more, 155
  using parentheses, 152–153
  using pipe character in, 153–154
Guess the Number program, 74–76
GUI (graphical user interface) automation. *See also* form filler project
  controlling keyboard, 426–429
    hotkey combinations, 429
    key names, 427–428
    pressing and releasing, 428–429
    sending string from keyboard, 426–427
  controlling mouse, 415–417, 419–423
    clicking mouse, 420
    dragging mouse, 420–422
    scrolling mouse, 422–423
  determining mouse position, 417–419
  image recognition, 425–426
  installing pyautogui module, 414
  logging out of program, 414
  overview, 413–414
  screenshots, 423–424
  stopping program, 414–415

H

%H directive, 344
hash character (#), 126
headings, Word document, 314–315
help
  asking online, 9–10
  for error messages, 8–9
hotkey combinations, 429
hotkey() function, 429, 430
Hotmail.com, 363, 367
HTML (Hypertext Markup Language)
  browser developer tools and, 242–243
  finding elements, 244
  learning resources, 240
  overview, 240–241
  viewing page source, 241–242

I

%I directive, 344
id attribute, 241
IDLE (interactive development environment)
  creating programs, 21–22
  debugging in
    overview, 225–227
    stepping through program, 227–229
    using breakpoints, 229–231
  expressions in, 14–16
  overview, 8
  running scripts outside of, 136
  starting, 7–8
if statements
  overview, 38–39
  using in while loop, 46–47
imageDraw module, 406
imageDraw objects, 406–408
ImageFont objects, 408–410
Image objects, 391–399
images
  adding logo to, 401–405
  attributes for, 392–393
  box tuples, 390
  color values in, 388–389
  coordinates in, 389–390
  copying and pasting in, 394–396
  cropping, 393–394
  drawing on
    example program, 407–408
    ellipses, 407
    ImageDraw module, 406
    lines, 406–407
    points, 406
    polygons, 407
    rectangles, 407
    text, 408–410
  flipping, 398–399
  opening with Pillow, 390–391
  pixel manipulation, 400
  recognition of, 425–426
  resizing, 397
  RGBA values, 388–389
  rotating, 398–399
  transparent pixels, 397
IMAP (Internet Message Access Protocol)
  defined, 366
  deleting messages, 375
  disconnecting from server, 375–376
  fetching messages, 372–375
  folders, 368–369
  logging into server, 368
  searching messages, 368–371
imapclient module, 366
IMAPClient objects, 367–368
immutable data types, 94–96
importing modules
  overview, 57–58
  pyautogui module, 417
imprint attribute, 311
im variable, 423
indentation, 93
indexes
  for dictionaries. *See* keys, dictionary
  for lists
    changing values using, 83
    getting value using, 80–81
    negative, 82
    removing values from list using, 84
  for strings, 126–127
IndexError, 106
index() method, 89
infinite loops, 49, 51, 418
INFO level, 223
in operator
  using with dictionaries, 109
  using with lists, 87
  using with strings, 127
input() function
  overview, 23–24, 89–90
  using for sensitive information, 365
installing
  openpyxl module, 266
  pyautogui module, 414
  Python, 6–7
  selenium module, 256
  third-party modules, 441–442
int, 17. See also integers
integer division/floored quotient (//) operator, 15
integers
  floating-point equivalence, 27
  overview, 17
interactive development environment. *See* IDLE (interactive development environment)
interactive shell. *See* IDLE
Internet Explorer, developer tools in, 242–243

Internet Message Access Protocol. *See* IMAP (Internet Message Access Protocol)
interpreter, Python, 7
`int()` function, 25–28
`isabs()` function, 177
`isalnum()` method, 129–131
`isalpha()` method, 129–130
`isdecimal()` method, 129–131
`isdir()` function, 180
`is_displayed()` method, 258
`is_enabled()` method, 258
`isfile()` function, 180
`islower()` method, 128–129
`is_selected()` method, 258
`isspace()` method, 130
`istitle()` method, 130
`isupper()` method, 128–129
`italic` attribute, 311
`items()` method, 107–108
`iter_content()` method, 239–240

J

`%j` directive, 344
`join()` method, 131–132, 174–175, 177, 352
JPEG format, 392
JSON files
  APIs for, 327–328
  defined, 319–320
  format overview, 327–328
  reading, 328–329
  and weather data project, 329–332
  writing, 329
justifying text, 133–134

K

keyboard
  controlling, with PyAutoGUI
    hotkey combinations, 429
    pressing and releasing keys, 428–429
    sending string from keyboard, 426–427
  key names, 427–428
`KeyboardInterrupt` exception, 340, 417, 418
`keyDown()` function, 428, 429, 430
keys, dictionary, 105
`keys()` method, 107–108
`keyUp()` function, 428, 429, 430
keyword arguments, 65–66

L

`LARGER` search key, 370
launchd, 354–355
launching programs
  and countdown project, 357–358
  opening files with default applications, 355–356
  opening websites, 355
  overview, 352–354
  passing command line arguments to processes, 354
  `poll()` method, 353
  running Python scripts, 355
  scheduling, 354–355
  `sleep()` function, 355
  `wait()` method, 354
`len()` function, 307–308
  finding number of values in list, 83
  overview, 24–25
less than (`<`) operator, 33
less than or equal to (`<=`) operator, 33
LibreOffice, 265, 306
line breaks, Word document, 315
`LineChart()` function, 290
line continuation character (`\`), 93
`line()` method, 406–407
linked styles, 310
Linux
  backslash vs. forward slash, 174–175
  cron, 354
  installing Python, 7
  installing third-party modules, 442
  launching processes from Python, 353
  logging out of automation program, 414
  opening files with default applications, 355
  pip tool on, 441–442
  Python support, 4
  running Python programs on, 445
  starting IDLE, 8
  Unix philosophy, 356
`listdir()` function, 179
`list_folders()` method, 368–369
`list()` function, 321, 426

lists
append() method, 89–90
augmented assignment operators, 88–89
changing values using index, 83
concatenation of, 83
copy() function, 100–101
deepcopy() function, 100–101
dictionaries vs., 106–107
finding number of values using len(), 83
getting sublists with slices, 82–83
getting value using index, 80–81
index() method, 89
in operator, 87
insert() method, 89–90
list() function, 97
Magic 8 Ball example program using, 92–93
multiple assignment trick, 87–88
mutable vs. immutable data types, 94–96
negative indexes, 82
nesting, 117–119
not in operator, 87
overview, 80
remove() method, 90–91
removing values from, 84
replication of, 83
sort() method, 91–92
storing variables as, 84–85
using with for loops, 86
ljust() method, 133–134
load_workbook() function, 267
loads() function, 328–329, 331
local scope, 67–70
locateAllOnScreen() function, 426
locateOnScreen() function, 425
location attribute, 258
logging
disabling, 224–225
to file, 225
levels of, 223–224
print() function and, 223
logging module, 221–223
logging out, of automation program, 414
login() method, 364, 368, 379
logo, adding to an image, 401–406
looping over files, 402–403
opening logo image, 401–402
overview, 404
resizing image, 403–404
logout() method, 375–376
LogRecord objects, 221
loops
break statements, 49–50
continue statements, 50–53
for loop, 53–56
range() function for, 56–57
reading data from CSV file, 322
using lists with, 86
while loop, 45–49
lower() method, 128–129
lstrip() method, 134–135

M

%M directive, 344
%m directive, 344
Mac OS X. *See* OS X
Magic 8 Ball example program, 92–93
makedirs() function, 176, 403
maps, open when location is copied, 234–236
figuring out URL, 234–235
handling clipboard content, 236
handling command line argument, 235–236
launching browser, 236
overview, 234
Match objects, 151
math
operators for, 15
programming and, 4
mergePage() method, 302
Message objects, 381–382
methods
chaining calls, 398
defined, 89
dictionary
get() method, 109
items() method, 107–108
keys() method, 107–108
setdefault() method, 110–111
values() method, 107–108
list
append() method, 89–90
index() method, 89
insert() method, 89–90
remove() method, 90–91
sort() method, 91–92
string
center() method, 133–134
copy() method, 135
endswith() method, 131

isalnum() method, 129–131
isalpha() method, 129–130
isdecimal() method, 129–131
islower() method, 128–129
isspace() method, 130
istitle() method, 130
isupper() method, 128–129
join() method, 131–132
ljust() method, 133–134
lower() method, 128–129
lstrip() method, 134–135
paste() method, 135
rjust() method, 133–134
rstrip() method, 134–135
split() method, 131–133
startswith() method, 131
strip() method, 134–135
upper() method, 128–129
Microsoft Windows. *See* Windows OS
middleClick() function, 420, 430
modules
importing, 57–58
third-party, installing, 442
modulus/remainder (%) operator, 15, 88
Monty Python, 4
mouse
controlling, 415–417, 419–423
clicking mouse, 420
dragging mouse, 420–422
scrolling mouse, 422–423
determining position of, 417–419
locating, 417–419
getting coordinates, 418–419
handling KeyboardInterrupt exception, 418
importing pyautogui module, 418
infinite loop, 418
overview, 417
and pixels, identifying colors of, 424–425
mouseDown() function, 420, 430
mouse.position() function, 418
mouseUp() function, 430
move() function, 199–200
moveRel() function, 416, 417, 420, 430
moveTo() function, 416, 420, 430
moving files/folders, 199–200
multiclipboard project, 191–193
listing keywords, 193
loading keyword content, 193
overview, 191
saving clipboard content, 192
setting up shelf file, 192
multiline comments, 126
multiline strings, 125–126
multiple assignment trick, 87–88
multiplication (*) operator, 15, 83, 88
multithreading
concurrency issues, 349
downloading multiple images, , 350–352
creating and starting threads, 351–352
using downloadXkcd() function, 350–351
waiting for threads to end, 352
join() method, 352
overview, 347–348
passing arguments to threads, 348–349
start() method, 348, 349
Thread() function, 347–348
mutable data types, 94–96

N

NameError, 84
namelist() method, 204
negative character classes, 159
negative indexes, 82
nested lists and dictionaries, 117–119
newline keyword argument, 322
None value, 65
nongreedy matching
dot, star, and question mark for, 161
in regular expressions, 157
not equal to (!=) operator, 33
not in operator
using with dictionaries, 109
using with lists, 87
using with strings, 127
not operator, 36
NOT search key, 370
now() function, 341, 346

O

ON search key, 369
open() function, 181–182, 234, 355–356, 391
opening files, 181–182
OpenOffice, 265, 306
open program, 355

openpyxl module, installing, 266
operators
    augmented assignment, 88–89
    binary, 35–36
    comparison, 33–35
    defined, 14
    math, 15
    using binary and comparison operators together, 36–37
order of operations, 15
or operator, 36
OR search key, 370
OS X
    backslash vs. forward slash, 174–175
    installing Python, 7
    installing third-party modules, 442
    launchd, 354
    launching processes from Python, 356
    logging out of automation program, 414
    opening files with default applications, 355–356
    pip tool on, 441–442
    Python support, 4
    running Python programs on, 445
    starting IDLE, 8
    Unix philosophy, 356
outline attribute, 311
Outlook.com, 363, 367

P

%p directive, 344
page breaks, Word document, 315
Page objects, 297
Paragraph objects, 307
paragraphs, Word document, 309–310
parameters, function, 63
parentheses (), 96–97, 152–153
parsing, defined, 245
passing arguments, 23
passing references, 100
passwords
    application-specific, 365
    managing project, 136–138
        command-line arguments, 137
        copying password, 137–138
        data structures, 136–137
        overview, 136
pastebin.com, 10
paste() method, 135, 394, 395
paths
    absolute vs. relative, 175–176
    backslash vs. forward slash, 174–175
    current working directory, 175
    overview, 173–174
    os.path module
        absolute paths in, 177–179
        file sizes, 179–180
        folder contents, 179–180
        overview, 177
        path validity, 180
        relative paths in, 177–179
PAUSE variable, 415, 434
PdfFileReader objects, 297–298
PDF files
    combining pages from multiple files, 303–306
        adding pages, 303
        finding PDF files, 304
        opening PDFs, 304–305
        overview, 303
        saving results, 305–306
    creating, 298–299
    decrypting, 297–298
    encrypting, 302–303
    extracting text from, 296–297
    format overview, 295–296
    pages in
        copying, 299–300
        overlaying, 301–302
        rotating, 300–301
PdfFileWriter objects, 298–299
pformat() function
    overview, 111–112
    saving variables in text files using, 185–186
phone numbers, extracting, 165–169
        creating regex, 166
        finding matches on clipboard, 167–168
        joining matches into a string, 168
        overview, 165–166
Pillow
    copying and pasting in images, 394–396
    cropping images, 393–394
    drawing on images
        ellipses, 407
        example program, 407–408
        ImageDraw module, 406
        lines, 406–407

points, 406
polygons, 407
rectangles, 407
text, 408–410
flipping images, 398–399
image attributes, 392–393
module, 388
opening images, 390–391
pixel manipulation, 400
resizing images, 397
rotating images, 398–399
transparent pixels, 397
pipe character (|), 153–154, 164–165
pip tool, 441–442
`pixelMatchesColor()` function, 424, 435
pixels, 388, 400
plaintext files, 180–181
plus sign (+), 155–156, 162
PNG format, 392
`point()` method, 406
`poll()` method, 353
`polygon()` method, 407
`Popen()` function, 352–353
opening files with default applications, 355–356
passing command line arguments to, 354
`position()` function, 417, 419
`pprint()` function, 111–112
precedence of math operators, 15
`press()` function, 429, 430, 436
`print()` function, 435
logging and, 223
overview, 23
passing multiple arguments to, 66
using variables with, 24
processes
and countdown project, 357–358
defined, 352
opening files with default applications, 355–356
opening websites, 355
passing command line arguments to, 354
`poll()` method, 353
`Popen()` function, 352–353
`wait()` method, 354
profiling code, 336–337
programming
blocks of code, 37–38
comments, 23
creativity needed for, 5
deduplicating code, 62
defined, 3
exception handling, 72–74
execution, program, 38
functions as "black boxes", 72
global scope, 70–71
indentation, 93
local scope, 67–70
math and, 4
Python, 4
terminating program with `sys.exit()`, 58
projects
Adding Bullets to Wiki Markup, 139–141
Adding a Logo, 401–406
Automatic Form Filler, 430–437
Backing Up a Folder into a ZIP File, 209–212
Combining Select Pages from Many PDFs, 303–306
Downloading All XKCD Comics, 251–256
Extending the mouseNow Program, 424–425
Fetching Current Weather Data, 329–332
Generating Random Quiz Files, 186–191
"I'm Feeling Lucky" Google Search, 248–251
"Just Text Me" Module, 383
mapIt.py with the `webbrowser` Module, 234–236
Multiclipboard, 191–193
Multithreaded XKCD Downloader, 350–352
Password Locker, 136–138
Phone Number and Email Address Extractor, 165–169
Reading Data from a Spreadsheet, 272–276
Removing the Header from CSV Files, 324–327
Renaming Files with American-Style Dates to European-Style Dates, 206–209
Sending Member Dues Reminder Emails, 376–379
Simple Countdown Program, 357–358
Super Stopwatch, 338–340

projects (*continued*)
Updating a Spreadsheet, 279–281
"Where Is the Mouse Right Now?", 417–419
putpixel() method, 400
pyautogui.click() function, 431
pyautogui.click() method, 420
pyautogui.doubleClick() function, 420
pyautogui.dragTo() function, 420
pyautogui.FailSafeException exception, 415
pyautogui.hotkey() function, 429
pyautogui.keyDown() function, 428
pyautogui.keyUp() function, 428
pyautogui.middleClick() function, 420
pyautogui module
form filler project, 430–437
controlling keyboard, 426–429
hotkey combinations, 429
key names, 427–428
pressing and releasing keys, 428–429
sending string from keyboard, 426–427
controlling mouse, 415–417, 419–423
clicking mouse, 420
dragging mouse, 420–422
scrolling mouse, 422–423
documentation for, 414
fail-safe feature, 415
functions, 430
image recognition, 425–426
importing, 417
installing, 414
pausing function calls, 415
screenshots, 423–424
pyautogui.mouseDown() function, 420
pyautogui.moveRel() function, 416, 417
pyautogui.moveTo() function, 416
pyautogui.PAUSE variable, 415
pyautogui.position() function, 419
pyautogui.press() function, 436
pyautogui.rightClick() function, 420
pyautogui.screenshot() function, 423
pyautogui.size() function, 416
pyautogui.typewrite() function, 426, 427, 431
py.exe program, 444–445
pyobjc module, 442
PyPDF2 module
combining pages from multiple PDFs, 303–306
creating PDFs, 298–299
decrypting PDFs, 297–298
encrypting PDFs, 302–303
extracting text from PDFs, 296–297
format overview, 295–296
pages in PDFs
copying, 299–300
overlaying, 301–302
rotating, 300–301
pyperclip module, 135
Python
data types, 16–17
downloading, 6
example program, 21–22
help, 8–9
installing, 6–7
interactive shell, 8
interpreter, defined, 7
math and, 4
overview, 4
programming overview, 3
starting IDLE, 7–8
python-docx module, 306
pyzmail module, 366, 373–375
PyzMessage objects, 373–375

Q

question mark (?), 154–155, 162
quit() method, 261, 366, 379
quiz generator, 186–190
creating quiz file, 188
creating answer options, 189
overview, 186
shuffling question order, 188
storing quiz data in dictionary, 187
writing content to files, 189–191

R

radio buttons, 435–436
raise_for_status() method, 238
raise keyword, 216–217
range() function, 56–57
raw strings, 125, 151
Reader objects, 321–322
reading files, 182–183, 204
readlines() method, 182

read() method, 182
rectangle() method, 407
Reddit, 9
Reference objects, 289
references
overview, 97–99
passing, 100
refresh() method, 261
Regex objects
creating, 150
matching, 151
regular expressions
beginning of string matches, 159–160
case sensitivity, 163
character classes, 158–159
creating Regex objects, 150–151
defined, 147–148
end of string matches, 159–160
extracting phone numbers and emails addresses, 165–169
findall() method, 157–158
finding text without, 148–150
greedy matching, 156–157
grouping
matching specific repetitions, 156
one or more matches, 155–156
optional matching, 154–155
using parentheses, 152–153
using pipe character in, 153–154
zero or more matches, 155
HTML and, 243
matching with, 151–152
multiple arguments for compile() function, 164–165
nongreedy matching, 157
patterns for, 150
spreading over multiple lines, 164
substituting strings using, 163–164
symbol reference, 162
wildcard character, 160–162
relative paths, 175–179
relpath() function, 177, 178
remainder/modulus (%) operator, 15, 88
remove() method, 90–91
remove_sheet() method, 278
renaming files/folders, 199–200
date styles, 206–209
creating regex for dates, 206
identifying dates in filenames, 207–208
overview, 206
renaming files, 209
replication
of lists, 83
string, 18
requests module
downloading files, 239–240
downloading pages, 237–238
resolution of computer screen, 416
Response objects, 237–238
return values, function, 63–65
reverse keyword, 91
RGBA values, 388–389
RGB color model, 389
rightClick() function, 420, 430
rjust() method, 133–134, 419
rmdir() function, 200
rmtree() function, 200
rotateClockwise() method, 300
rotateCounterClockwise() method, 300
rotating images, 398–399
rounding numbers, 338
rows, in Excel spreadsheets
setting height and width of, 285–286
slicing Worksheet objects to get Cell objects in, 270–272
rstrip() method, 134–135
rtl attribute, 311
Run objects, 310, 311–312
running programs
on Linux, 445
on OS X, 445
overview, 443
on Windows, 444–445
shebang line, 443–444

S

\S character class, 158
\s character class, 158
%S directive, 344
Safari, developer tools in, 243
save() method, 391
scope
global, 70–71
local, 67–70
screenshot() function, 423, 430
screenshots
analyzing, 424
getting, 423
scripts
running from Python program, 355
running outside of IDLE, 136

scroll() function, 422, 423, 430
scrolling mouse, 422–423
searching
email, 368–371
the Web, 248–251
finding results, 249–250
getting command line arguments, 249
opening web browser for results, 250–251
overview, 248
requesting search page, 249
search() method, 151
SEEN search key, 370
see program, 355
select_folder() method, 369
select lists, 435–436
select() method, bs4 module, 246–247
selectors, CSS, 246–247, 258
selenium module
clicking buttons, 261
finding elements, 257–259
following links, 259
installing, 256
sending special keystrokes, 260–261
submitting forms, 259–260
using Firefox with, 256–257
send2trash module, 201–202
sending reminder emails, 376–379
finding unpaid members, 378
opening Excel file, 376–377
overview, 376
sending emails, 378–379
send_keys() method, 259–260
sendmail() method, 365, 379
sequence numbers, 373
sequences, 86
setdefault() method, 110–111
shadow attribute, 311
shebang line, 443–444
shelve module, 184–185
Short Message Service (SMS)
sending messages, 381–382
Twilio service, 380
shutil module
deleting files/folders, 200–201
moving files/folders, 199–200
renaming files/folders, 199–200
SID (string ID), 382
Simple Mail Transfer Protocol. *See* SMTP (Simple Mail Transfer Protocol)
SINCE search key, 369
single quote ('), 124
single-threaded programs, 347
size() function, 416
sleep() function, 337–338, 344, 346, 355
slices
getting sublists with, 82–83
for strings, 126–127
small_caps attribute, 311
SMALLER search key, 370
SMS (Short Message Service)
sending messages, 381–382
Twilio service, 380
SMTP (Simple Mail Transfer Protocol)
connecting to server, 363–364
defined, 362
disconnecting from server, 366
logging into server, 364–365
sending "hello" message, 364
sending message, 365
TLS encryption, 364
SMTP objects, 363–364
sort() method, 91–92
sound files, playing, 357–358
source code, defined, 3
split() method, 131–133, 178, 320
spreadsheets. *See* Excel spreadsheets
square brackets [], 80
Stack Overflow, 9
standard library, 57
star (*), 161, 162
using with wildcard character, 161
zero or more matches with, 155
start() method, 348, 349, 351
start program, 355
startswith() method, 131
starttls() method, 364, 379
step argument, 56
stopwatch project, 338–340
overview, 338–339
set up, 339
tracking lap times, 339–340
strftime() function, 344–345, 346
str() function, 25–28, 97, 419
strike attribute, 311
string ID (SID), 382
strings
center() method, 133–134
concatenation, 17–18
converting datetime objects to, 344–345
converting to datetime objects, 345

copying and pasting, 135
double quotes for, 124
endswith() method, 131
escape characters, 124–125
extracting PDF text as, 296–297
getting traceback as, 217–218
indexes for, 126–127
in operator, 127
isalnum() method, 129–131
isalpha() method, 129–130
isdecimal() method, 129–131
islower() method, 128–129
isspace() method, 130
istitle() method, 130
isupper() method, 128–129
join() method, 131–132
literals, 124
ljust() method, 133–134
lower() method, 128–129
lstrip() method, 134–135
multiline, 125–126
mutable vs. immutable data types, 94–96
not in operator, 127
overview, 17
raw, 125
replication of, 18
rjust() method, 133–134
rstrip() method, 134–135
slicing, 126–127
split() method, 131–133
startswith() method, 131
strip() method, 134–135
substituting using regular expressions, 163–164
upper() method, 128–129
strip() method, 134–135
strptime() function, 345, 346
strs, 17, *See also* strings
Style objects, 282–283
SUBJECT search key, 369
sublists, getting with slices, 82–83
sub() method, 163–164
submitButtonColor variable, 432, 435
submitButton variable, 432
submit() method, 260
subprocess module, 335, 352–354
subtraction (-) operator, 15, 88
subtractive color model, 389
Sudoku puzzles, 4
sys.exit() function, 58

T

tag_name attribute, 258
Tag objects, 246–247
tags, HTML, 240
Task Scheduler, 354
termination, program, 22, 58
text attribute, 308, 311
text messaging
automatic notifications, 383
sending messages, 381–382
Twilio service, 380
text() method, 408–410
TEXT search key, 369
textsize() method, 409
third-party modules, installing, 441–442
Thread() function, 347–348, 351
threading module, 335, 347
Thread objects, 347–348
threads
concurrency issues, 349
join() method, 352
multithreading, 347–348
image downloader, 350–352
passing arguments to, 348–349
processes vs., 352
tic-tac-toe board, 113–117
timedelta data type, 342–343, 346
timedelta objects, 342–343
time module
overview, 346
sleep() function, 337–338, 344
stopwatch project, 338–340
time() function, 336–337
TLS encryption, 364
top-level domains, 167
TO search key, 370
total_seconds() method, 342, 346
traceback, getting from error, 217–218
transparency, 388, 397
transpose() method, 399
triple quotes ('''), 125, 164
truetype() function, 409
truth tables, 35–36
“truthy” values, 53
tuple data type
overview, 96–97
tuple() function, 97
twilio module, 380
TwilioRestClient objects, 381

Twilio service
automatic text messages, 383
overview, 380
sending text messages, 381–382
TypeError, 81, 94
typewrite() function, 426, 427, 430, 431, 435, 436

U

Ubuntu, 7
cron, 354–355
launching processes from Python, 353
opening files with default applications, 355
Unix philosophy, 356
UNANSWERED search key, 370
UNDELETED search key, 370
underline attribute, 311
underscore (_), 20
UNDRAFT search key, 370
UNFLAGGED search key, 370
Unicode encodings, 239
Unix epoch, 336, 341, 346
Unix philosophy, 356
unlink() function, 200
UNSEEN search key, 370
upper() method, 128–129
UTC (Coordinated Universal Time), 336

V

ValueError, 88, 345
values, defined, 14, 150
values() method, 107–108
variables. *See also* lists
assignment statements, 18–19
defined, 18
global, 70–71
initializing, 19
local, 67–70
naming, 20–21
None value and, 65
overwriting, 19–20
references, 97–99
saving with shelve module, 184–185
storing as list, 84–85
Verizon mail, 363, 367
volumes, defined, 174

W

\W character class, 158
\w character class, 158
%w directive, 344
walk() function, 202–203, 354
WARNING level, 223
weather data, fetching, 329–332
downloading JSON data, 330–331
getting location, 330
loading JSON data, 331–332
overview, 329
webbrowser module
open() function, 355
opening browser using, 234–236
WebDriver objects, 257
WebElement objects, 257–258
web scraping
bs4 module
creating object from HTML, 245–246
finding element with select() method, 246–247
getting attribute, 248
overview, 245
downloading
files, 239–240
images, 251–256
pages, 237–238
and Google maps project, 234–236
and Google search project, 248–251
HTML
browser developer tools and, 242–243
finding elements, 244
learning resources, 240
overview, 240–241
viewing page source, 241–242
overview, 233–234
requests module, 237–238
selenium module
clicking buttons, 261
finding elements, 257–259
following links, 259
installing, 256
sending special keystrokes, 260–261
submitting forms, 259–260
using Firefox with, 256–257
websites, opening from script, 355

while loops
getting and printing mouse coordinates using, 418
infinite, 418
overview, 45–49
whitespace, removing, 134–135
wildcard character (.), 160–162
Windows OS
backslash vs. forward slash, 174–175
installing Python, 6–7
installing third-party modules, 442
launching processes from Python, 353
logging out of automation program, 414
opening files with default applications, 355–356
pip tool on, 441–442
Python support, 4
running Python programs on, 444–445
starting IDLE, 7
Task Scheduler, 354
Word documents
adding headings, 314–315
creating documents with nondefault styles, 310–311
format overview, 306–307
getting text from, 308–309
line/page breaks, 315
pictures in, 315–316
python-docx module, 306
reading, 307–308
Run object attributes, 311–312
styling paragraphs, 309–310
writing to file, 312–314
Workbook objects, 267
workbooks, Excel, 266
creating worksheets, 278
deleting worksheets, 278
opening, 267
saving, 277
Worksheet objects, 268
write() method, 183–184
Writer objects, 322–323
writerow() method, 323

X

XKCD comics
downloading project, 251–256
designing program, 252–253
downloading web page, 253–254
overview, 251–252
saving image, 255–256
multithreaded downloading project, 350–352
creating and starting threads, 351–352
using downloadXkcd() function, 350–351
waiting for threads to end, 352

Y

%Y directive, 344
%y directive, 344
Yahoo! Mail, 363, 367

Z

zipfile module
creating ZIP files, 205–206
extracting ZIP files, 205
and folders, 209–212
overview, 203–204
reading ZIP files, 204
ZipFile objects, 204–205
ZipInfo objects, 204

RESOURCES

Visit *http://nostarch.com/automatestuff/* for resources, errata, and more information.

More no-nonsense books from **NO STARCH PRESS**

PYTHON PLAYGROUND
Geeky Weekend Projects for the Curious Programmer
by MAHESH VENKITACHALAM
MAY 2015, 304 PP., $29.95
ISBN 978-1-59327-604-1

PYTHON CRASH COURSE
A Hands-On, Project-Based Introduction to Programming
by ERIC MATTHES
JULY 2015, 624 PP., $34.95
ISBN 978-1-59327-603-4

THE LINUX COMMAND LINE
A Complete Introduction
by WILLIAM E. SHOTTS, JR.
JANUARY 2012, 480 PP., $39.95
ISBN 978-1-59327-389-7

JAVASCRIPT FOR KIDS
A Playful Introduction to Programming
by NICK MORGAN
DECEMBER 2014, 336 PP., $34.95
ISBN 978-1-59327-408-5
full color

STATISTICS DONE WRONG
The Woefully Complete Guide
by ALEX REINHART
MARCH 2015, 176 PP., $24.95
ISBN 978-1-59327-620-1

DATA VISUALIZATION WITH JAVASCRIPT
by STEPHEN A. THOMAS
MARCH 2015, 384 PP., $39.95
ISBN 978-1-59327-605-8
full color

PHONE:
800.420.7240 OR
415.863.9900

EMAIL:
SALES@NOSTARCH.COM

WEB:
WWW.NOSTARCH.COM

LEARN PYTHON.
GET STUFF DONE.

If you've ever spent hours renaming files or updating hundreds of spreadsheet cells, you know how tedious tasks like these can be. But what if you could have your computer do them for you?

In *Automate the Boring Stuff with Python*, you'll learn how to use Python to write programs that do in minutes what would take you hours to do by hand—no prior programming experience required. Once you've mastered the basics of programming, you'll create Python programs that effortlessly perform useful and impressive feats of automation to:

- Search for text in a file or across multiple files
- Create, update, move, and rename files and folders
- Search the Web and download online content
- Update and format data in Excel spreadsheets of any size
- Split, merge, watermark, and encrypt PDFs
- Send reminder emails and text notifications
- Fill out online forms

Step-by-step instructions walk you through each program, and practice projects at the end of each chapter challenge you to improve those programs and use your newfound skills to automate similar tasks.

Don't spend your time doing work a well-trained monkey could do. Even if you've never written a line of code, you can make your computer do the grunt work. Learn how in *Automate the Boring Stuff with Python*.

ABOUT THE AUTHOR

Al Sweigart is a software developer and teaches programming to kids and adults. He has written several Python books for beginners, including *Hacking Secret Ciphers with Python*, *Invent Your Own Computer Games with Python*, and *Making Games with Python & Pygame*.

COVERS PYTHON 3

no starch press

THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com

"I LIE FLAT."

This book uses a durable binding that won't snap shut.

SUSTAINABLE FORESTRY INITIATIVE Certified Sourcing
www.sfiprogram.org
SFI-00000

ISBN: 978-1-59327-599-0

52995

9 781593 275990

$29.95 ($34.95 CDN)

6 89145 75994 5

SHELVE IN:
PROGRAMMING LANGUAGES/
PYTHON

2ND EDITION

PYTHON CRASH COURSE

A HANDS-ON, PROJECT-BASED INTRODUCTION TO PROGRAMMING

ERIC MATTHES

no starch press

PRAISE FOR *PYTHON CRASH COURSE*

"It has been interesting to see No Starch Press producing future classics that should be alongside the more traditional programming books. *Python Crash Course* is one of those books."
—Greg Laden, ScienceBlogs

"Deals with some rather complex projects and lays them out in a consistent, logical, and pleasant manner that draws the reader into the subject."
—Full Circle Magazine

"Well presented with good explanations of the code snippets. The book works with you, one small step at a time, building more complex code, explaining what's going on all the way."
—FlickThrough Reviews

"Learning Python with *Python Crash Course* was an extremely positive experience! A great choice if you're new to Python."
—Mikke Goes Coding

"Does what it says on the tin, and does it really well. . . . Presents a large number of useful exercises as well as three challenging and entertaining projects."
—RealPython.com

"A fast-paced but comprehensive introduction to programming with Python, *Python Crash Course* is another superb book to add to your library and help you finally master Python."
—TutorialEdge.net

"A brilliant option for complete beginners without any coding experience. If you're looking for a solid, uncomplicated intro to this very deep language, I have to recommend this book."
—WhatPixel.com

"Contains literally everything you need to know about Python and even more."
—FireBearStudio.com

PYTHON CRASH COURSE

2ND EDITION

A Hands-On, Project-Based Introduction to Programming

by Eric Matthes

no starch press

San Francisco

PYTHON CRASH COURSE, 2ND EDITION. Copyright © 2019 by Eric Matthes.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-10: 1-59327-928-0
ISBN-13: 978-1-59327-928-8

Publisher: William Pollock
Production Editor: Riley Hoffman
Cover Illustration: Josh Ellingson
Cover and Interior Design: Octopod Studios
Developmental Editor: Liz Chadwick
Technical Reviewer: Kenneth Love
Copyeditor: Anne Marie Walker
Compositors: Riley Hoffman and Happenstance Type-O-Rama
Proofreader: James Fraleigh

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900; info@nostarch.com
www.nostarch.com

The Library of Congress has catalogued the first edition as follows:

Matthes, Eric, 1972-
Python crash course : a hands-on, project-based introduction to programming / by Eric Matthes.
pages cm
Includes index.
Summary: "A project-based introduction to programming in Python, with exercises. Covers general programming concepts, Python fundamentals, and problem solving. Includes three projects - how to create a simple video game, use data visualization techniques to make graphs and charts, and build an interactive web application"-- Provided by publisher.
ISBN 978-1-59327-603-4 -- ISBN 1-59327-603-6
1. Python (Computer program language) I. Title.
QA76.73.P98M38 2015
005.13'3--dc23
2015018135

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

About the Author

Eric Matthes is a high school science and math teacher living in Alaska, where he teaches an introductory Python course. He has been writing programs since he was five years old. Eric currently focuses on writing software that addresses inefficiencies in education and brings the benefits of open source software to the field of education. In his spare time he enjoys climbing mountains and spending time with his family.

About the Technical Reviewer

Kenneth Love has been a Python programmer, teacher, and conference organizer for many years. He has spoken and taught at many conferences, been a Python and Django freelancer, and is currently a software engineer for O'Reilly Media. Kenneth is co-creator of the django-braces package, which provides several handy mixins for Django's class-based views. You can keep up with him on Twitter at @kennethlove.

For my father, who always made time to
answer my questions about programming,
and for Ever, who is just beginning to ask me
his questions

BRIEF CONTENTS

Preface to the Second Edition xxvii

Acknowledgments xxxi

Introduction xxxiii

PART I: BASICS 1

Chapter 1: Getting Started 3

Chapter 2: Variables and Simple Data Types 15

Chapter 3: Introducing Lists 33

Chapter 4: Working with Lists 49

Chapter 5: if Statements 71

Chapter 6: Dictionaries 91

Chapter 7: User Input and while Loops 113

Chapter 8: Functions 129

Chapter 9: Classes 157

Chapter 10: Files and Exceptions 183

Chapter 11: Testing Your Code 209

PART II: PROJECTS 223

Project 1: Alien Invasion

Chapter 12: A Ship that Fires Bullets 227

Chapter 13: Aliens! 255

Chapter 14: Scoring 279

Project 2: Data Visualization

Chapter 15: Generating Data 305

Chapter 16: Downloading Data. 333

Chapter 17: Working with APIs 359

Project 3: Web Applications

Chapter 18: Getting Started with Django 379

Chapter 19: User Accounts 409

Chapter 20: Styling and Deploying an App 437

Afterword 465

Appendix A: Installation and Troubleshooting 467

Appendix B: Text Editors and IDEs 473

Appendix C: Getting Help 479

Appendix D: Using Git for Version Control 485

Index 495

CONTENTS IN DETAIL

PREFACE TO THE SECOND EDITION **xxvii**

ACKNOWLEDGMENTS **xxxi**

INTRODUCTION **xxxiii**

Who Is This Book For? . . . xxxiv
What Can You Expect to Learn? . . . xxxiv
Online Resources . . . xxxv
Why Python? . . . xxxvi

PART I: BASICS **1**

1
GETTING STARTED **3**

Setting Up Your Programming Environment . . . 3
Python Versions . . . 4
Running Snippets of Python Code . . . 4
About the Sublime Text Editor . . . 4
Python on Different Operating Systems . . . 5
Python on Windows . . . 5
Python on macOS . . . 7
Python on Linux . . . 8
Running a Hello World Program . . . 9
Configuring Sublime Text to Use the Correct Python Version . . . 9
Running hello_world.py . . . 10
Troubleshooting . . . 11
Running Python Programs from a Terminal . . . 12
On Windows . . . 12
On macOS and Linux . . . 12
Exercise 1-1: python.org . . . *13*
Exercise 1-2: Hello World Typos . . . *13*
Exercise 1-3: Infinite Skills . . . *13*
Summary . . . 13

2
VARIABLES AND SIMPLE DATA TYPES **15**

What Really Happens When You Run hello_world.py . . . 15
Variables . . . 16
Naming and Using Variables . . . 17
Avoiding Name Errors When Using Variables . . . 17
Variables Are Labels . . . 18
Exercise 2-1: Simple Message . . . *19*
Exercise 2-2: Simple Messages . . . *19*

Strings 19
Changing Case in a String with Methods 20
Using Variables in Strings 21
Adding Whitespace to Strings with Tabs or Newlines 22
Stripping Whitespace 22
Avoiding Syntax Errors with Strings 24
Exercise 2-3: Personal Message *25*
Exercise 2-4: Name Cases *25*
Exercise 2-5: Famous Quote *25*
Exercise 2-6: Famous Quote 2 . . . . . *25*
Exercise 2-7: Stripping Names *25*
Numbers 25
Integers 26
Floats 26
Integers and Floats 27
Underscores in Numbers 28
Multiple Assignment 28
Constants 28
Exercise 2-8: Number Eight *29*
Exercise 2-9: Favorite Number *29*
Comments 29
How Do You Write Comments? 29
What Kind of Comments Should You Write? 29
Exercise 2-10: Adding Comments *30*
The Zen of Python 30
Exercise 2-11: Zen of Python *31*
Summary 32

3
INTRODUCING LISTS **33**

What Is a List? 33
Accessing Elements in a List 34
Index Positions Start at 0, Not 1 35
Using Individual Values from a List 35
Exercise 3-1: Names *36*
Exercise 3-2: Greetings *36*
Exercise 3-3: Your Own List *36*
Changing, Adding, and Removing Elements 36
Modifying Elements in a List 36
Adding Elements to a List 37
Removing Elements from a List 38
Exercise 3-4: Guest List *42*
Exercise 3-5: Changing Guest List *42*
Exercise 3-6: More Guests *42*
Exercise 3-7: Shrinking Guest List *43*
Organizing a List 43
Sorting a List Permanently with the sort() Method 43
Sorting a List Temporarily with the sorted() Function 44
Printing a List in Reverse Order 45
Finding the Length of a List 45
Exercise 3-8: Seeing the World. *46*
Exercise 3-9: Dinner Guests *46*
Exercise 3-10: Every Function *46*

Avoiding Index Errors When Working with Lists . . . 46
Exercise 3-11: Intentional Error . . . *48*
Summary . . . 48

4
WORKING WITH LISTS **49**

Looping Through an Entire List . . . 49
A Closer Look at Looping . . . 50
Doing More Work Within a for Loop . . . 51
Doing Something After a for Loop . . . 52
Avoiding Indentation Errors . . . 53
Forgetting to Indent . . . 53
Forgetting to Indent Additional Lines . . . 54
Indenting Unnecessarily . . . 55
Indenting Unnecessarily After the Loop . . . 55
Forgetting the Colon . . . 56
Exercise 4-1: Pizzas . . . *56*
Exercise 4-2: Animals . . . *56*
Making Numerical Lists . . . 57
Using the range() Function . . . 57
Using range() to Make a List of Numbers . . . 58
Simple Statistics with a List of Numbers . . . 59
List Comprehensions . . . 59
Exercise 4-3: Counting to Twenty . . . *60*
Exercise 4-4: One Million . . . *60*
Exercise 4-5: Summing a Million . . . *60*
Exercise 4-6: Odd Numbers . . . *60*
Exercise 4-7: Threes . . . *60*
Exercise 4-8: Cubes . . . *60*
Exercise 4-9: Cube Comprehension . . . *60*
Working with Part of a List . . . 61
Slicing a List . . . 61
Looping Through a Slice . . . 62
Copying a List . . . 63
Exercise 4-10: Slices . . . *65*
Exercise 4-11: My Pizzas, Your Pizzas . . . *65*
Exercise 4-12: More Loops . . . *65*
Tuples . . . 65
Defining a Tuple . . . 66
Looping Through All Values in a Tuple . . . 67
Writing over a Tuple . . . 67
Exercise 4-13: Buffet . . . *68*
Styling Your Code . . . 68
The Style Guide . . . 68
Indentation . . . 69
Line Length . . . 69
Blank Lines . . . 69
Other Style Guidelines . . . 70
Exercise 4-14: PEP 8 . . . *70*
Exercise 4-15: Code Review . . . *70*
Summary . . . 70

5
IF STATEMENTS 71

A Simple Example . . . 72
Conditional Tests . . . 72
Checking for Equality . . . 72
Ignoring Case When Checking for Equality . . . 73
Checking for Inequality . . . 74
Numerical Comparisons . . . 74
Checking Multiple Conditions . . . 75
Checking Whether a Value Is in a List . . . 76
Checking Whether a Value Is Not in a List . . . 77
Boolean Expressions . . . 77
Exercise 5-1: Conditional Tests . . . *78*
Exercise 5-2: More Conditional Tests . . . *78*
if Statements . . . 78
Simple if Statements . . . 78
if-else Statements . . . 79
The if-elif-else Chain . . . 80
Using Multiple elif Blocks . . . 82
Omitting the else Block . . . 82
Testing Multiple Conditions . . . 83
Exercise 5-3: Alien Colors #1 . . . *84*
Exercise 5-4: Alien Colors #2 . . . *84*
Exercise 5-5: Alien Colors #3 . . . *85*
Exercise 5-6: Stages of Life . . . *85*
Exercise 5-7: Favorite Fruit . . . *85*
Using if Statements with Lists . . . 85
Checking for Special Items . . . 86
Checking That a List Is Not Empty . . . 87
Using Multiple Lists . . . 88
Exercise 5-8: Hello Admin . . . *89*
Exercise 5-9: No Users . . . *89*
Exercise 5-10: Checking Usernames . . . *89*
Exercise 5-11: Ordinal Numbers . . . *89*
Styling Your if Statements . . . 90
Exercise 5-12: Styling if statements . . . *90*
Exercise 5-13: Your Ideas . . . *90*
Summary . . . 90

6
DICTIONARIES 91

A Simple Dictionary . . . 92
Working with Dictionaries . . . 92
Accessing Values in a Dictionary . . . 93
Adding New Key-Value Pairs . . . 93
Starting with an Empty Dictionary . . . 94
Modifying Values in a Dictionary . . . 95
Removing Key-Value Pairs . . . 96
A Dictionary of Similar Objects . . . 97
Using get() to Access Values . . . 98

Exercise 6-1: Person *99*
Exercise 6-2: Favorite Numbers *99*
Exercise 6-3: Glossary *99*
Looping Through a Dictionary 99
Looping Through All Key-Value Pairs 99
Looping Through All the Keys in a Dictionary 101
Looping Through a Dictionary's Keys in a Particular Order 103
Looping Through All Values in a Dictionary 104
Exercise 6-4: Glossary 2 . . . . . *105*
Exercise 6-5: Rivers *105*
Exercise 6-6: Polling *105*
Nesting 106
A List of Dictionaries 106
A List in a Dictionary 108
A Dictionary in a Dictionary 110
Exercise 6-7: People *112*
Exercise 6-8: Pets *112*
Exercise 6-9: Favorite Places *112*
Exercise 6-10: Favorite Numbers *112*
Exercise 6-11: Cities *112*
Exercise 6-12: Extensions *112*
Summary 112

7
USER INPUT AND WHILE LOOPS **113**

How the input() Function Works 114
Writing Clear Prompts 114
Using int() to Accept Numerical Input 115
The Modulo Operator 116
Exercise 7-1: Rental Car *117*
Exercise 7-2: Restaurant Seating *117*
Exercise 7-3: Multiples of Ten *117*
Introducing while Loops 118
The while Loop in Action 118
Letting the User Choose When to Quit 118
Using a Flag 120
Using break to Exit a Loop 121
Using continue in a Loop 122
Avoiding Infinite Loops 122
Exercise 7-4: Pizza Toppings *123*
Exercise 7-5: Movie Tickets *123*
Exercise 7-6: Three Exits *124*
Exercise 7-7: Infinity *124*
Using a while Loop with Lists and Dictionaries 124
Moving Items from One List to Another 124
Removing All Instances of Specific Values from a List 125
Filling a Dictionary with User Input 126
Exercise 7-8: Deli *127*
Exercise 7-9: No Pastrami *127*
Exercise 7-10: Dream Vacation *127*
Summary 127

8
FUNCTIONS 129

Defining a Function . . . 130
Passing Information to a Function . . . 130
Arguments and Parameters . . . 131
Exercise 8-1: Message . . . 131
Exercise 8-2: Favorite Book . . . 131
Passing Arguments . . . 131
Positional Arguments . . . 132
Keyword Arguments . . . 133
Default Values . . . 134
Equivalent Function Calls . . . 135
Avoiding Argument Errors . . . 136
Exercise 8-3: T-Shirt . . . 137
Exercise 8-4: Large Shirts . . . 137
Exercise 8-5: Cities . . . 137
Return Values . . . 137
Returning a Simple Value . . . 138
Making an Argument Optional . . . 138
Returning a Dictionary . . . 140
Using a Function with a while Loop . . . 141
Exercise 8-6: City Names . . . 142
Exercise 8-7: Album . . . 142
Exercise 8-8: User Albums . . . 142
Passing a List . . . 143
Modifying a List in a Function . . . 143
Preventing a Function from Modifying a List . . . 145
Exercise 8-9: Messages . . . 146
Exercise 8-10: Sending Messages . . . 146
Exercise 8-11: Archived Messages . . . 146
Passing an Arbitrary Number of Arguments . . . 147
Mixing Positional and Arbitrary Arguments . . . 148
Using Arbitrary Keyword Arguments . . . 148
Exercise 8-12: Sandwiches . . . 150
Exercise 8-13: User Profile . . . 150
Exercise 8-14: Cars . . . 150
Storing Your Functions in Modules . . . 150
Importing an Entire Module . . . 150
Importing Specific Functions . . . 152
Using as to Give a Function an Alias . . . 152
Using as to Give a Module an Alias . . . 153
Importing All Functions in a Module . . . 153
Styling Functions . . . 154
Exercise 8-15: Printing Models . . . 155
Exercise 8-16: Imports . . . 155
Exercise 8-17: Styling Functions . . . 155
Summary . . . 155

9
CLASSES 157

Creating and Using a Class . . . 158
- Creating the Dog Class . . . 158
- Making an Instance from a Class . . . 160
- *Exercise 9-1: Restaurant* . . . *162*
- *Exercise 9-2: Three Restaurants* . . . *162*
- *Exercise 9-3: Users* . . . *162*

Working with Classes and Instances . . . 162
- The Car Class . . . 162
- Setting a Default Value for an Attribute . . . 163
- Modifying Attribute Values . . . 164
- *Exercise 9-4: Number Served* . . . *167*
- *Exercise 9-5: Login Attempts* . . . *167*

Inheritance . . . 167
- The __init__() Method for a Child Class . . . 167
- Defining Attributes and Methods for the Child Class . . . 169
- Overriding Methods from the Parent Class . . . 170
- Instances as Attributes . . . 170
- Modeling Real-World Objects . . . 173
- *Exercise 9-6: Ice Cream Stand* . . . *173*
- *Exercise 9-7: Admin* . . . *173*
- *Exercise 9-8: Privileges* . . . *173*
- *Exercise 9-9: Battery Upgrade* . . . *174*

Importing Classes . . . 174
- Importing a Single Class . . . 174
- Storing Multiple Classes in a Module . . . 175
- Importing Multiple Classes from a Module . . . 177
- Importing an Entire Module . . . 177
- Importing All Classes from a Module . . . 177
- Importing a Module into a Module . . . 178
- Using Aliases . . . 179
- Finding Your Own Workflow . . . 179
- *Exercise 9-10: Imported Restaurant* . . . *180*
- *Exercise 9-11: Imported Admin* . . . *180*
- *Exercise 9-12: Multiple Modules* . . . *180*

The Python Standard Library . . . 180
- *Exercise 9-13: Dice* . . . *181*
- *Exercise 9-14: Lottery* . . . *181*
- *Exercise 9-15: Lottery Analysis* . . . *181*
- *Exercise 9-16: Python Module of the Week* . . . *181*

Styling Classes . . . 181

Summary . . . 182

10
FILES AND EXCEPTIONS 183

Reading from a File . . . 184
- Reading an Entire File . . . 184
- File Paths . . . 185
- Reading Line by Line . . . 187
- Making a List of Lines from a File . . . 188
- Working with a File's Contents . . . 188

Large Files: One Million Digits . . . 189
Is Your Birthday Contained in Pi? . . . 190
Exercise 10-1: Learning Python . . . *191*
Exercise 10-2: Learning C . . . *191*
Writing to a File. . . . 191
Writing to an Empty File . . . 191
Writing Multiple Lines . . . 192
Appending to a File. . . . 193
Exercise 10-3: Guest . . . *193*
Exercise 10-4: Guest Book . . . *193*
Exercise 10-5: Programming Poll. . . . *193*
Exceptions. . . . 194
Handling the ZeroDivisionError Exception . . . 194
Using try-except Blocks . . . 194
Using Exceptions to Prevent Crashes . . . 195
The else Block. . . . 196
Handling the FileNotFoundError Exception . . . 197
Analyzing Text . . . 198
Working with Multiple Files . . . 199
Failing Silently . . . 200
Deciding Which Errors to Report. . . . 201
Exercise 10-6: Addition . . . *201*
Exercise 10-7: Addition Calculator . . . *202*
Exercise 10-8: Cats and Dogs . . . *202*
Exercise 10-9: Silent Cats and Dogs . . . *202*
Exercise 10-10: Common Words . . . *202*
Storing Data . . . 202
Using json.dump() and json.load() . . . 203
Saving and Reading User-Generated Data . . . 204
Refactoring. . . . 206
Exercise 10-11: Favorite Number . . . *208*
Exercise 10-12: Favorite Number Remembered . . . *208*
Exercise 10-13: Verify User . . . *208*
Summary . . . 208

11
TESTING YOUR CODE **209**

Testing a Function . . . 210
Unit Tests and Test Cases . . . 211
A Passing Test . . . 211
A Failing Test . . . 212
Responding to a Failed Test . . . 213
Adding New Tests. . . . 214
Exercise 11-1: City, Country . . . *215*
Exercise 11-2: Population. . . . *216*
Testing a Class . . . 216
A Variety of Assert Methods . . . 216
A Class to Test . . . 217
Testing the AnonymousSurvey Class . . . 218
The setUp() Method . . . 220
Exercise 11-3: Employee . . . *221*
Summary . . . 222

PART II: PROJECTS 223

PROJECT 1: ALIEN INVASION

12
A SHIP THAT FIRES BULLETS 227

Planning Your Project . . . 228
Installing Pygame . . . 228
Starting the Game Project . . . 229
    Creating a Pygame Window and Responding to User Input . . . 229
    Setting the Background Color . . . 230
    Creating a Settings Class . . . 231
Adding the Ship Image . . . 232
    Creating the Ship Class . . . 233
    Drawing the Ship to the Screen . . . 235
Refactoring: The _check_events() and _update_screen() Methods . . . 236
    The _check_events() Method . . . 236
    The _update_screen() Method . . . 237
    Exercise 12-1: Blue Sky . . . 238
    Exercise 12-2: Game Character . . . 238
Piloting the Ship . . . 238
    Responding to a Keypress . . . 238
    Allowing Continuous Movement . . . 239
    Moving Both Left and Right . . . 240
    Adjusting the Ship's Speed . . . 241
    Limiting the Ship's Range . . . 243
    Refactoring _check_events() . . . 243
    Pressing Q to Quit . . . 244
    Running the Game in Fullscreen Mode . . . 244
A Quick Recap . . . 245
    alien_invasion.py . . . 245
    settings.py . . . 246
    ship.py . . . 246
    Exercise 12-3: Pygame Documentation . . . 246
    Exercise 12-4: Rocket . . . 246
    Exercise 12-5: Keys . . . 246
Shooting Bullets . . . 246
    Adding the Bullet Settings . . . 247
    Creating the Bullet Class . . . 247
    Storing Bullets in a Group . . . 248
    Firing Bullets . . . 249
    Deleting Old Bullets . . . 250
    Limiting the Number of Bullets . . . 251
    Creating the _update_bullets() Method . . . 252
    Exercise 12-6: Sideways Shooter . . . 253
Summary . . . 253

13
ALIENS! 255

Reviewing the Project . . . 256
Creating the First Alien . . . 256
Creating the Alien Class . . . 257
Creating an Instance of the Alien . . . 258
Building the Alien Fleet . . . 259
Determining How Many Aliens Fit in a Row . . . 260
Creating a Row of Aliens . . . 260
Refactoring _create_fleet() . . . 262
Adding Rows . . . 262
Exercise 13-1: Stars. . . . 264
Exercise 13-2: Better Stars . . . 264
Making the Fleet Move . . . 265
Moving the Aliens Right . . . 265
Creating Settings for Fleet Direction . . . 266
Checking Whether an Alien Has Hit the Edge . . . 266
Dropping the Fleet and Changing Direction . . . 267
Exercise 13-3: Raindrops . . . 268
Exercise 13-4: Steady Rain. . . . 268
Shooting Aliens . . . 268
Detecting Bullet Collisions. . . . 268
Making Larger Bullets for Testing . . . 270
Repopulating the Fleet . . . 270
Speeding Up the Bullets . . . 271
Refactoring _update_bullets() . . . 271
Exercise 13-5: Sideways Shooter Part 2 . . . 272
Ending the Game . . . 272
Detecting Alien and Ship Collisions . . . 272
Responding to Alien and Ship Collisions . . . 273
Aliens that Reach the Bottom of the Screen . . . 276
Game Over! . . . 276
Identifying When Parts of the Game Should Run . . . 277
Exercise 13-6: Game Over. . . . 278
Summary . . . 278

14
SCORING 279

Adding the Play Button . . . 280
Creating a Button Class . . . 280
Drawing the Button to the Screen . . . 281
Starting the Game . . . 283
Resetting the Game . . . 283
Deactivating the Play Button . . . 284
Hiding the Mouse Cursor . . . 284
Exercise 14-1: Press P to Play . . . 285
Exercise 14-2: Target Practice . . . 285
Leveling Up . . . 285
Modifying the Speed Settings . . . 285
Resetting the Speed . . . 287

Exercise 14-3: Challenging Target Practice *288*
Exercise 14-4: Difficulty Levels *288*
Scoring 288
Displaying the Score 288
Making a Scoreboard 289
Updating the Score as Aliens Are Shot Down 291
Resetting the Score 291
Making Sure to Score All Hits 292
Increasing Point Values 292
Rounding the Score 293
High Scores 294
Displaying the Level 296
Displaying the Number of Ships 298
Exercise 14-5: All-Time High Score *301*
Exercise 14-6: Refactoring *301*
Exercise 14-7: Expanding the Game *302*
Exercise 14-8: Sideways Shooter, Final Version *302*
Summary 302

PROJECT 2: DATA VISUALIZATION

15
GENERATING DATA **305**

Installing Matplotlib 306
Plotting a Simple Line Graph 306
Changing the Label Type and Line Thickness 307
Correcting the Plot 309
Using Built-in Styles 310
Plotting and Styling Individual Points with scatter() 310
Plotting a Series of Points with scatter() 312
Calculating Data Automatically 312
Defining Custom Colors 314
Using a Colormap 314
Saving Your Plots Automatically 315
Exercise 15-1: Cubes *315*
Exercise 15-2: Colored Cubes *315*
Random Walks 315
Creating the RandomWalk() Class 316
Choosing Directions 316
Plotting the Random Walk 317
Generating Multiple Random Walks 318
Styling the Walk 319
Exercise 15-3: Molecular Motion *323*
Exercise 15-4: Modified Random Walks *323*
Exercise 15-5: Refactoring *323*
Rolling Dice with Plotly 323
Installing Plotly 324
Creating the Die Class 324
Rolling the Die 325

Analyzing the Results . . . 325
Making a Histogram . . . 326
Rolling Two Dice . . . 328
Rolling Dice of Different Sizes . . . 329
Exercise 15-6: Two D8s . . . *331*
Exercise 15-7: Three Dice . . . *331*
Exercise 15-8: Multiplication . . . *331*
Exercise 15-9: Die Comprehensions . . . *331*
Exercise 15-10: Practicing with Both Libraries . . . *331*
Summary . . . 331

16
DOWNLOADING DATA **333**

The CSV File Format . . . 334
Parsing the CSV File Headers . . . 334
Printing the Headers and Their Positions . . . 335
Extracting and Reading Data . . . 336
Plotting Data in a Temperature Chart . . . 336
The datetime Module . . . 337
Plotting Dates . . . 338
Plotting a Longer Timeframe . . . 340
Plotting a Second Data Series . . . 340
Shading an Area in the Chart . . . 342
Error Checking . . . 343
Downloading Your Own Data . . . 345
Exercise 16-1: Sitka Rainfall . . . *346*
Exercise 16-2: Sitka–Death Valley Comparison . . . *346*
Exercise 16-3: San Francisco . . . *346*
Exercise 16-4: Automatic Indexes . . . *347*
Exercise 16-5: Explore . . . *347*
Mapping Global Data Sets: JSON Format . . . 347
Downloading Earthquake Data . . . 347
Examining JSON Data . . . 347
Making a List of All Earthquakes . . . 350
Extracting Magnitudes . . . 350
Extracting Location Data . . . 351
Building a World Map . . . 351
A Different Way of Specifying Chart Data . . . 353
Customizing Marker Size . . . 353
Customizing Marker Colors . . . 354
Other Colorscales . . . 356
Adding Hover Text . . . 356
Exercise 16-6: Refactoring . . . *357*
Exercise 16-7: Automated Title . . . *357*
Exercise 16-8: Recent Earthquakes . . . *358*
Exercise 16-9: World Fires . . . *358*
Summary . . . 358

17
WORKING WITH APIs 359

Using a Web API . . . 359
Git and GitHub . . . 360
Requesting Data Using an API Call . . . 360
Installing Requests . . . 361
Processing an API Response . . . 361
Working with the Response Dictionary . . . 362
Summarizing the Top Repositories . . . 364
Monitoring API Rate Limits . . . 365
Visualizing Repositories Using Plotly . . . 366
Refining Plotly Charts . . . 368
Adding Custom Tooltips . . . 369
Adding Clickable Links to Our Graph . . . 370
More About Plotly and the GitHub API . . . 371
The Hacker News API . . . 372
Exercise 17-1: Other Languages . . . 375
Exercise 17-2: Active Discussions . . . 375
Exercise 17-3: Testing python_repos.py . . . 375
Exercise 17-4: Further Exploration . . . 375
Summary . . . 375

PROJECT 3: WEB APPLICATIONS

18
GETTING STARTED WITH DJANGO 379

Setting Up a Project . . . 380
Writing a Spec . . . 380
Creating a Virtual Environment . . . 380
Activating the Virtual Environment . . . 381
Installing Django . . . 381
Creating a Project in Django . . . 382
Creating the Database . . . 382
Viewing the Project . . . 383
Exercise 18-1: New Projects . . . 384
Starting an App . . . 384
Defining Models . . . 385
Activating Models . . . 386
The Django Admin Site . . . 387
Defining the Entry Model . . . 390
Migrating the Entry Model . . . 391
Registering Entry with the Admin Site . . . 391
The Django Shell . . . 392
Exercise 18-2: Short Entries . . . 394
Exercise 18-3: The Django API . . . 394
Exercise 18-4: Pizzeria . . . 394
Making Pages: The Learning Log Home Page . . . 394
Mapping a URL . . . 395
Writing a View . . . 396

Writing a Template . . . 397
Exercise 18-5: Meal Planner . . . *398*
Exercise 18-6: Pizzeria Home Page . . . *398*
Building Additional Pages . . . 398
Template Inheritance . . . 398
The Topics Page . . . 400
Individual Topic Pages . . . 403
Exercise 18-7: Template Documentation . . . *406*
Exercise 18-8: Pizzeria Pages . . . *406*
Summary . . . 407

19
USER ACCOUNTS **409**

Allowing Users to Enter Data . . . 410
Adding New Topics . . . 410
Adding New Entries . . . 414
Editing Entries . . . 418
Exercise 19-1: Blog . . . *421*
Setting Up User Accounts . . . 421
The users App . . . 421
The Login Page . . . 422
Logging Out . . . 424
The Registration Page . . . 426
Exercise 19-2: Blog Accounts . . . *428*
Allowing Users to Own Their Data . . . 428
Restricting Access with @login_required . . . 429
Connecting Data to Certain Users . . . 430
Restricting Topics Access to Appropriate Users . . . 433
Protecting a User's Topics . . . 434
Protecting the edit_entry Page . . . 434
Associating New Topics with the Current User . . . 435
Exercise 19-3: Refactoring . . . *436*
Exercise 19-4: Protecting new_entry . . . *436*
Exercise 19-5: Protected Blog . . . *436*
Summary . . . 436

20
STYLING AND DEPLOYING AN APP **437**

Styling Learning Log . . . 438
The django-bootstrap4 App . . . 438
Using Bootstrap to Style Learning Log . . . 438
Modifying base.html . . . 439
Styling the Home Page Using a Jumbotron . . . 443
Styling the Login Page . . . 444
Styling the Topics Page . . . 445
Styling the Entries on the Topic Page . . . 446
Exercise 20-1: Other Forms . . . *447*
Exercise 20-2: Stylish Blog . . . *447*
Deploying Learning Log . . . 448
Making a Heroku Account . . . 448
Installing the Heroku CLI . . . 448

Installing Required Packages . . . 448
Creating a requirements.txt File . . . 448
Specifying the Python Runtime . . . 449
Modifying settings.py for Heroku . . . 450
Making a Procfile to Start Processes . . . 450
Using Git to Track the Project's Files . . . 450
Pushing to Heroku . . . 452
Setting Up the Database on Heroku . . . 454
Refining the Heroku Deployment . . . 454
Securing the Live Project . . . 456
Committing and Pushing Changes . . . 457
Setting Environment Variables on Heroku . . . 458
Creating Custom Error Pages . . . 458
Ongoing Development . . . 461
The SECRET_KEY Setting . . . 461
Deleting a Project on Heroku . . . 461
Exercise 20-3: Live Blog . . . 462
Exercise 20-4: More 404s . . . 462
Exercise 20-5: Extended Learning Log . . . 462
Summary . . . 463

AFTERWORD 465

A
INSTALLATION AND TROUBLESHOOTING 467

Python on Windows . . . 467
Finding the Python Interpreter . . . 467
Adding Python to Your Path Variable . . . 468
Reinstalling Python . . . 469
Python on macOS . . . 469
Installing Homebrew . . . 469
Installing Python . . . 470
Python on Linux . . . 470
Python Keywords and Built-in Functions . . . 471
Python Keywords . . . 471
Python Built-in Functions . . . 471

B
TEXT EDITORS AND IDEs 473

Customizing Sublime Text Settings . . . 474
Converting Tabs to Spaces . . . 474
Setting the Line Length Indicator . . . 474
Indenting and Unindenting Code Blocks . . . 474
Commenting Out Blocks of Code . . . 475
Saving Your Configuration . . . 475
Further Customizations . . . 475
Other Text Editors and IDEs . . . 475
IDLE . . . 475
Geany . . . 476
Emacs and Vim . . . 476

Atom 476
Visual Studio Code 476
PyCharm 476
Jupyter Notebooks 477

C
GETTING HELP **479**

First Steps 479
Try It Again 480
Take a Break 480
Refer to This Book's Resources 480
Searching Online 481
Stack Overflow 481
The Official Python Documentation 481
Official Library Documentation 482
r/learnpython 482
Blog Posts 482
Internet Relay Chat 482
Making an IRC Account 482
Channels to Join 483
IRC Culture 483
Slack 483
Discord 484

D
USING GIT FOR VERSION CONTROL **485**

Installing Git 486
Installing Git on Windows 486
Installing Git on macOS 486
Installing Git on Linux 486
Configuring Git 486
Making a Project 486
Ignoring Files 487
Initializing a Repository 487
Checking the Status 487
Adding Files to the Repository 488
Making a Commit 488
Checking the Log 489
The Second Commit 489
Reverting a Change 490
Checking Out Previous Commits 491
Deleting the Repository 493

INDEX **495**

PREFACE TO THE SECOND EDITION

The response to the first edition of *Python Crash Course* has been overwhelmingly positive. More than 500,000 copies are in print, including translations in eight languages. I've received letters and emails from readers as young as 10, as well as from retirees who want to learn to program in their free time. *Python Crash Course* is being used in middle schools and high schools, and also in college classes. Students who are assigned more advanced textbooks are using *Python Crash Course* as a companion text for their classes and finding it a worthwhile supplement. People are using it to enhance their skills on the job and to start working on their own side projects. In short, people are using the book for the full range of purposes I had hoped they would.

The opportunity to write a second edition of *Python Crash Course* has been thoroughly enjoyable. Although Python is a mature language, it continues to evolve as every language does. My goal in revising the book was to make it leaner and simpler. There is no longer any reason to learn Python 2, so this edition focuses on Python 3 only. Many Python packages have become easier to install, so setup and installation instructions are easier. I've added a few topics that I've realized readers would benefit from, and I've updated some sections to reflect new, simpler ways of doing things in Python. I've also clarified some sections where certain details of the

language were not presented as accurately as they could have been. All the projects have been completely updated using popular, well-maintained libraries that you can confidently use to build your own projects.

The following is a summary of specific changes that have been made in the second edition:

- In Chapter 1, the instructions for installing Python have been simplified for users of all major operating systems. I now recommend the text editor Sublime Text, which is popular among beginner and professional programmers and works well on all operating systems.
- Chapter 2 includes a more accurate description of how variables are implemented in Python. Variables are described as *labels* for values, which leads to a better understanding of how variables behave in Python. The book now uses f-strings, introduced in Python 3.6. This is a much simpler way to use variable values in strings. The use of underscores to represent large numbers, such as `1_000_000`, was also introduced in Python 3.6 and is included in this edition. Multiple assignment of variables was previously introduced in one of the projects, and that description has been generalized and moved to Chapter 2 for the benefit of all readers. Finally, a clear convention for representing constant values in Python is included in this chapter.
- In Chapter 6, I introduce the `get()` method for retrieving values from a dictionary, which can return a default value if a key does not exist.
- The Alien Invasion project (Chapters 12–14) is now entirely class-based. The game itself is a class, rather than a series of functions. This greatly simplifies the overall structure of the game, vastly reducing the number of function calls and parameters required. Readers familiar with the first edition will appreciate the simplicity this new class-based approach provides. Pygame can now be installed in one line on all systems, and readers are given the option of running the game in fullscreen mode or in a windowed mode.
- In the data visualization projects, the installation instructions for Matplotlib are simpler for all operating systems. The visualizations featuring Matplotlib use the `subplots()` function, which will be easier to build upon as you learn to create more complex visualizations. The Rolling Dice project in Chapter 15 uses Plotly, a well-maintained visualization library that features a clean syntax and beautiful, fully customizable output.
- In Chapter 16, the weather project is based on data from NOAA, which should be more stable over the next few years than the site used in the first edition. The mapping project focuses on global earthquake activity; by the end of this project you'll have a stunning visualization showing Earth's tectonic plate boundaries through a focus on the locations of all earthquakes over a given time period. You'll learn to plot any data set involving geographic points.
- Chapter 17 uses Plotly to visualize Python-related activity in open source projects on GitHub.

- The Learning Log project (Chapters 18–20) is built using the latest version of Django and styled using the latest version of Bootstrap. The process of deploying the project to Heroku has been simplified using the `django-heroku` package, and uses environment variables rather than modifying the *settings.py* files. This is a simpler approach and is more consistent with how professional programmers deploy modern Django projects.
- Appendix A has been fully updated to recommend current best practices in installing Python. Appendix B includes detailed instructions for setting up Sublime Text and brief descriptions of most of the major text editors and IDEs in current use. Appendix C directs readers to newer, more popular online resources for getting help, and Appendix D continues to offer a mini crash course in using Git for version control.
- The index has been thoroughly updated to allow you to use *Python Crash Course* as a reference for all of your future Python projects.

Thank you for reading *Python Crash Course*! If you have any feedback or questions, please feel free to get in touch.

ACKNOWLEDGMENTS

This book would not have been possible without the wonderful and extremely professional staff at No Starch Press. Bill Pollock invited me to write an introductory book, and I deeply appreciate that original offer. Tyler Ortman helped shape my thinking in the early stages of drafting. Liz Chadwick's and Leslie Shen's initial feedback on each chapter was invaluable, and Anne Marie Walker helped to clarify many parts of the book. Riley Hoffman answered every question I had about the process of assembling a complete book and patiently turned my work into a beautiful finished product.

I'd like to thank Kenneth Love, the technical reviewer for *Python Crash Course*. I met Kenneth at PyCon one year, and his enthusiasm for the language and the Python community has been a constant source of professional inspiration ever since. Kenneth went beyond simple fact-checking and reviewed the book with the goal of helping beginning programmers develop a solid understanding of the Python language and programming in general. That said, any inaccuracies that remain are completely my own.

I'd like to thank my father for introducing me to programming at a young age and for not being afraid that I'd break his equipment. I'd like to thank my wife, Erin, for supporting and encouraging me through the writing of this book, and I'd like to thank my son, Ever, whose curiosity inspires me every single day.

INTRODUCTION

Every programmer has a story about how they learned to write their first program. I started programming as a child when my father was working for Digital Equipment Corporation, one of the pioneering companies of the modern computing era. I wrote my first program on a kit computer that my dad had assembled in our basement. The computer consisted of nothing more than a bare motherboard connected to a keyboard without a case, and its monitor was a bare cathode ray tube. My initial program was a simple number guessing game, which looked something like this:

```
I'm thinking of a number! Try to guess the number I'm thinking of: 25
Too low! Guess again: 50
Too high! Guess again: 42
That's it! Would you like to play again? (yes/no) no
Thanks for playing!
```

I'll always remember how satisfied I felt watching my family play a game that I created and that worked as I intended it to.

That early experience had a lasting impact. There is real satisfaction in building something with a purpose, something that solves a problem. The software I write now meets a more significant need than my childhood efforts, but the sense of satisfaction I get from creating a program that works is still largely the same.

Who Is This Book For?

The goal of this book is to bring you up to speed with Python as quickly as possible so you can build programs that work—games, data visualizations, and web applications—while developing a foundation in programming that will serve you well for the rest of your life. *Python Crash Course* is written for people of any age who have never before programmed in Python or have never programmed at all. This book is for those who want to learn the basics of programming quickly so they can focus on interesting projects, and those who like to test their understanding of new concepts by solving meaningful problems. *Python Crash Course* is also perfect for middle school and high school teachers who want to offer their students a project-based introduction to programming. If you're taking a college class and want a friendlier introduction to Python than the text you've been assigned, this book could make your class easier as well.

What Can You Expect to Learn?

The purpose of this book is to make you a good programmer in general and a good Python programmer in particular. You'll learn efficiently and adopt good habits as I provide you with a solid foundation in general programming concepts. After working your way through *Python Crash Course*, you should be ready to move on to more advanced Python techniques, and your next programming language will be even easier to grasp.

In the first part of this book, you'll learn basic programming concepts you need to know to write Python programs. These concepts are the same as those you'd learn when starting out in almost any programming language. You'll learn about different kinds of data and the ways you can store data in lists and dictionaries within your programs. You'll learn to build collections of data and work through those collections in efficient ways. You'll learn to use `while` loops and `if` statements to test for certain conditions so you can run specific sections of code while those conditions are true and run other sections when they're not—a technique that greatly helps you automate processes.

You'll learn to accept input from users to make your programs interactive and to keep your programs running as long as the user is active. You'll explore how to write functions to make parts of your program reusable, so you only have to write blocks of code that perform certain actions once and then use that code as many times as you like. You'll then extend

this concept to more complicated behavior with classes, making fairly simple programs respond to a variety of situations. You'll learn to write programs that handle common errors gracefully. After working through each of these basic concepts, you'll write a few short programs that solve some well-defined problems. Finally, you'll take your first step toward intermediate programming by learning how to write tests for your code so you can develop your programs further without worrying about introducing bugs. All the information in Part I will prepare you for taking on larger, more complex projects.

In Part II, you'll apply what you learned in Part I to three projects. You can do any or all of these projects in whichever order works best for you. In the first project (Chapters 12–14), you'll create a *Space Invaders*–style shooting game called *Alien Invasion*, which consists of levels of increasing difficulty. After you've completed this project, you should be well on your way to being able to develop your own 2D games.

The second project (Chapters 15–17) introduces you to data visualization. Data scientists aim to make sense of the vast amount of information available to them through a variety of visualization techniques. You'll work with data sets that you generate through code, data sets that you download from online sources, and data sets your programs download automatically. After you've completed this project, you'll be able to write programs that sift through large data sets and make visual representations of that stored information.

In the third project (Chapters 18–20), you'll build a small web application called Learning Log. This project allows you to keep a journal of ideas and concepts you've learned about a specific topic. You'll be able to keep separate logs for different topics and allow others to create an account and start their own journals. You'll also learn how to deploy your project so anyone can access it online from anywhere.

Online Resources

You can find all the supplementary resources for the book online at *https://nostarch.com/pythoncrashcourse2e/* or *http://ehmatthes.github.io/pcc_2e/*. These resources include:

Setup instructions These instructions are identical to what's in the book but include active links you can click for all the different pieces. If you're having any setup issues, refer to this resource.

Updates Python, like all languages, is constantly evolving. I maintain a thorough set of updates, so if anything isn't working, check here to see whether instructions have changed.

Solutions to exercises You should spend significant time on your own attempting the exercises in the "Try It Yourself" sections. But if you're stuck and can't make any progress, solutions to most of the exercises are online.

Cheat sheets A full set of downloadable cheat sheets for a quick reference to major concepts is also online.

Why Python?

Every year I consider whether to continue using Python or whether to move on to a different language—perhaps one that's newer to the programming world. But I continue to focus on Python for many reasons. Python is an incredibly efficient language: your programs will do more in fewer lines of code than many other languages would require. Python's syntax will also help you write "clean" code. Your code will be easy to read, easy to debug, and easy to extend and build upon compared to other languages.

People use Python for many purposes: to make games, build web applications, solve business problems, and develop internal tools at all kinds of interesting companies. Python is also used heavily in scientific fields for academic research and applied work.

One of the most important reasons I continue to use Python is because of the Python community, which includes an incredibly diverse and welcoming group of people. Community is essential to programmers because programming isn't a solitary pursuit. Most of us, even the most experienced programmers, need to ask advice from others who have already solved similar problems. Having a well-connected and supportive community is critical in helping you solve problems, and the Python community is fully supportive of people like you who are learning Python as your first programming language.

Python is a great language to learn, so let's get started!

PART I

BASICS

Part I of this book teaches you the basic concepts you'll need to write Python programs. Many of these concepts are common to all programming languages, so they'll be useful throughout your life as a programmer.

In **Chapter 1** you'll install Python on your computer and run your first program, which prints the message *Hello world!* to the screen.

In **Chapter 2** you'll learn to store information in variables and work with text and numerical values.

Chapters 3 and **4** introduce lists. Lists can store as much information as you want in one variable, allowing you to work with that data efficiently. You'll be able to work with hundreds, thousands, and even millions of values in just a few lines of code.

In **Chapter 5** you'll use `if` statements to write code that responds one way if certain conditions are true, and responds in a different way if those conditions are not true.

Chapter 6 shows you how to use Python's dictionaries, which let you make connections between different pieces of information. Like lists, dictionaries can contain as much information as you need to store.

In **Chapter 7** you'll learn how to accept input from users to make your programs interactive. You'll also learn about `while` loops, which run blocks of code repeatedly as long as certain conditions remain true.

In **Chapter 8** you'll write functions, which are named blocks of code that perform a specific task and can be run whenever you need them.

Chapter 9 introduces classes, which allow you to model real-world objects, such as dogs, cats, people, cars, rockets, and much more, so your code can represent anything real or abstract.

Chapter 10 shows you how to work with files and handle errors so your programs won't crash unexpectedly. You'll store data before your program closes, and read the data back in when the program runs again. You'll learn about Python's exceptions, which allow you to anticipate errors, and make your programs handle those errors gracefully.

In **Chapter 11** you'll learn to write tests for your code to check that your programs work the way you intend them to. As a result, you'll be able to expand your programs without worrying about introducing new bugs. Testing your code is one of the first skills that will help you transition from beginner to intermediate programmer.

1

GETTING STARTED

In this chapter, you'll run your first Python program, *hello_world.py*. First, you'll need to check whether a recent version of Python is installed on your computer; if it isn't, you'll install it. You'll also install a text editor to work with your Python programs. Text editors recognize Python code and highlight sections as you write, making it easy to understand your code's structure.

Setting Up Your Programming Environment

Python differs slightly on different operating systems, so you'll need to keep a few considerations in mind. In the following sections, we'll make sure Python is set up correctly on your system.

Python Versions

Every programming language evolves as new ideas and technologies emerge, and the developers of Python have continually made the language more versatile and powerful. As of this writing, the latest version is Python 3.7, but everything in this book should run on Python 3.6 or later. In this section, we'll find out if Python is already installed on your system and whether you need to install a newer version. Appendix A contains a comprehensive guide to installing the latest version of Python on each major operating system as well.

Some old Python projects still use Python 2, but you should use Python 3. If Python 2 is installed on your system, it's probably there to support some older programs that your system needs. We'll leave this installation as is, and make sure you have a more recent version to work with.

Running Snippets of Python Code

You can run Python's interpreter in a terminal window, allowing you to try bits of Python code without having to save and run an entire program.

Throughout this book, you'll see code snippets that look like this:

```
❶ >>> print("Hello Python interpreter!")
Hello Python interpreter!
```

The >>> prompt indicates that you should be using the terminal window, and the bold text is the code you should type in and then execute by pressing ENTER. Most of the examples in the book are small, self-contained programs that you'll run from your text editor rather than the terminal, because you'll write most of your code in the text editor. But sometimes basic concepts will be shown in a series of snippets run through a Python terminal session to demonstrate particular concepts more efficiently. When you see three angle brackets in a code listing ❶, you're looking at code and output from a terminal session. We'll try coding in the interpreter on your system in a moment.

We'll also use a text editor to create a simple program called *Hello World!* that has become a staple of learning to program. There's a long-held tradition in the programming world that printing a `Hello world!` message to the screen as your first program in a new language will bring you good luck. Such a simple program serves a very real purpose. If it runs correctly on your system, any Python program you write should work as well.

About the Sublime Text Editor

Sublime Text is a simple text editor that can be installed on all modern operating systems. Sublime Text lets you run almost all of your programs directly from the editor instead of through a terminal. Your code runs in a terminal session embedded in the Sublime Text window, which makes it easy to see the output.

Sublime Text is a beginner-friendly editor, but many professional programmers use it as well. If you become comfortable using it while learning Python, you can continue using it as you progress to larger and more complicated projects. Sublime Text has a very liberal licensing policy: you can use the editor free of charge as long as you want, but the developers request that you purchase a license if you like it and want to keep using it.

Appendix B provides information on other text editors. If you're curious about the other options, you might want to skim that appendix at this point. If you want to begin programming quickly, you can use Sublime Text to start and consider other editors once you've gained some experience as a programmer. In this chapter, I'll walk you through installing Sublime Text on your operating system.

Python on Different Operating Systems

Python is a cross-platform programming language, which means it runs on all the major operating systems. Any Python program you write should run on any modern computer that has Python installed. However, the methods for setting up Python on different operating systems vary slightly.

In this section, you'll learn how to set up Python on your system. You'll first check whether a recent version of Python is installed on your system and install it if it's not. Then you'll install Sublime Text. These are the only two steps that are different for each operating system.

In the sections that follow, you'll run the *Hello World!* program and troubleshoot anything that didn't work. I'll walk you through this process for each operating system, so you'll have a beginner-friendly Python programming environment.

Python on Windows

Windows doesn't always come with Python, so you'll probably need to install it, and then install Sublime Text.

Installing Python

First, check whether Python is installed on your system. Open a command window by entering `command` into the Start menu or by holding down the SHIFT key while right-clicking on your desktop and selecting **Open command window here** from the menu. In the terminal window, enter `python` in lowercase. If you get a Python prompt (>>>) in response, Python is installed on your system. If you see an error message telling you that `python` is not a recognized command, Python isn't installed.

In that case, or if you see a version of Python earlier than Python 3.6, you need to download a Python installer for Windows. Go to *https://python.org/* and hover over the **Downloads** link. You should see a button for downloading the latest version of Python. Click the button, which should automatically start downloading the correct installer for your system. After

you've downloaded the file, run the installer. Make sure you select the option **Add Python to PATH**, which will make it easier to configure your system correctly. Figure 1-1 shows this option selected.

Figure 1-1: Make sure you select the checkbox labeled Add Python to PATH.

Running Python in a Terminal Session

Open a command window and enter **`python`** in lowercase. You should see a Python prompt (`>>>`), which means Windows has found the version of Python you just installed.

```
C:\> python
Python 3.7.2 (v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

NOTE *If you don't see this output or something similar, see the more detailed setup instructions in Appendix A.*

Enter the following line in your Python session, and make sure you see the output `Hello Python interpreter!`

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

Any time you want to run a snippet of Python code, open a command window and start a Python terminal session. To close the terminal session, press CTRL-Z and then press ENTER, or enter the command `exit()`.

Installing Sublime Text

You can download an installer for Sublime Text at *https://sublimetext.com/*. Click the download link and look for a Windows installer. After downloading the installer, run the installer and accept all of its defaults.

Python on macOS

Python is already installed on most macOS systems, but it's most likely an outdated version that you won't want to learn on. In this section, you'll install the latest version of Python, and then you'll install Sublime Text and make sure it's configured correctly.

Checking Whether Python 3 Is Installed

Open a terminal window by going to **Applications ▸ Utilities ▸ Terminal**. You can also press ⌘-spacebar, type **`terminal`**, and then press ENTER. To see which version of Python is installed, enter **`python`** with a lowercase *p*—this also starts the Python interpreter within the terminal, allowing you to enter Python commands. You should see output telling you which Python version is installed on your system and a >>> prompt where you can start entering Python commands, like this:

```
$ python
Python 2.7.15 (default, Aug 17 2018, 22:39:05)
[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>>
```

This output indicates that Python 2.7.15 is currently the default version installed on this computer. Once you've seen this output, press CTRL-D or enter **`exit()`** to leave the Python prompt and return to a terminal prompt.

To check whether you have Python 3 installed, enter the command **`python3`**. You'll probably get an error message, meaning you don't have any versions of Python 3 installed. If the output shows you have Python 3.6 or a later version installed, you can skip ahead to "Running Python in a Terminal Session" on page 8. If Python 3 isn't installed by default, you'll need to install it manually. Note that whenever you see the `python` command in this book, you need to use the `python3` command instead to make sure you're using Python 3, not Python 2; they differ significantly enough that you'll run into trouble trying to run the code in this book using Python 2.

If you see any version earlier than Python 3.6, follow the instructions in the next section to install the latest version.

Installing the Latest Version of Python

You can find a Python installer for your system at *https://python.org/*. Hover over the **Download** link, and you should see a button for downloading the latest Python version. Click the button, which should automatically start

downloading the correct installer for your system. After the file downloads, run the installer.

When you're finished, enter the following at a terminal prompt:

```
$ python3 --version
Python 3.7.2
```

You should see output similar to this, in which case, you're ready to try out Python. Whenever you see the command **python**, make sure you use **python3**.

Running Python in a Terminal Session

You can now try running snippets of Python code by opening a terminal and typing **python3**. Enter the following line in the terminal session:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

Your message should print directly in the current terminal window. Remember that you can close the Python interpreter by pressing CTRL-D or by entering the command `exit()`.

Installing Sublime Text

To install the Sublime Text editor, you need to download the installer at *https://sublimetext.com/*. Click the **Download** link and look for an installer for macOS. After the installer downloads, open it and then drag the Sublime Text icon into your *Applications* folder.

Python on Linux

Linux systems are designed for programming, so Python is already installed on most Linux computers. The people who write and maintain Linux expect you to do your own programming at some point and encourage you to do so. For this reason, there's very little to install and only a few settings to change to start programming.

Checking Your Version of Python

Open a terminal window by running the Terminal application on your system (in Ubuntu, you can press CTRL-ALT-T). To find out which version of Python is installed, enter **python3** with a lowercase *p*. When Python is installed, this command starts the Python interpreter. You should see output indicating which version of Python is installed and a >>> prompt where you can start entering Python commands, like this:

```
$ python3
Python 3.7.2 (default, Dec 27 2018, 04:01:51)
[GCC 7.3.0] on linux
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This output indicates that Python 3.7.2 is currently the default version of Python installed on this computer. When you've seen this output, press CTRL-D or enter **exit()** to leave the Python prompt and return to a terminal prompt. Whenever you see the **python** command in this book, enter **python3** instead.

You'll need Python 3.6 or later to run the code in this book. If the Python version installed on your system is earlier than Python 3.6, refer to Appendix A to install the latest version.

Running Python in a Terminal Session

You can try running snippets of Python code by opening a terminal and entering **python3**, as you did when checking your version. Do this again, and when you have Python running, enter the following line in the terminal session:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

The message should print directly in the current terminal window. Remember that you can close the Python interpreter by pressing CTRL-D or by entering the command `exit()`.

Installing Sublime Text

On Linux, you can install Sublime Text from the Ubuntu Software Center. Click the Ubuntu Software icon in your menu, and search for **Sublime Text**. Click to install it, and then launch it.

Running a Hello World Program

With a recent version of Python and Sublime Text installed, you're almost ready to run your first Python program written in a text editor. But before doing so, you need to make sure Sublime Text is set up to use the correct version of Python on your system. Then you'll write the *Hello World!* program and run it.

Configuring Sublime Text to Use the Correct Python Version

If the `python` command on your system runs Python 3, you won't need to configure anything and can skip to the next section. If you use the `python3` command, you'll need to configure Sublime Text to use the correct Python version when it runs your programs.

Click the Sublime Text icon to launch it, or search for Sublime Text in your system's search bar and then launch it. Go to **Tools ▸ Build System ▸ New Build System**, which will open a new configuration file for you. Delete what you see and enter the following:

Python3 .sublime-build

```
{
    "cmd": ["python3", "-u", "$file"],
}
```

This code tells Sublime Text to use your system's `python3` command when running your Python program files. Save the file as *Python3.sublime-build* in the default directory that Sublime Text opens when you choose Save.

Running hello_world.py

Before you write your first program, make a folder called *python_work* somewhere on your system for your projects. It's best to use lowercase letters and underscores for spaces in file and folder names, because Python uses these naming conventions.

Open Sublime Text, and save an empty Python file (**File ▸ Save As**) called *hello_world.py* in your *python_work* folder. The extension *.py* tells Sublime Text that the code in your file is written in Python, which tells it how to run the program and highlight the text in a helpful way.

After you've saved your file, enter the following line in the text editor:

hello_world.py

```
print("Hello Python world!")
```

If the `python` command works on your system, you can run your program by selecting **Tools ▸ Build** in the menu or by pressing CTRL-B (⌘-B on macOS). If you had to configure Sublime Text in the previous section, select **Tools ▸ Build System** and then select **Python 3**. From now on you'll be able to select **Tools ▸ Build** or just press CTRL-B (or ⌘-B) to run your programs.

A terminal screen should appear at the bottom of the Sublime Text window, showing the following output:

```
Hello Python world!
[Finished in 0.1s]
```

If you don't see this output, something might have gone wrong in the program. Check every character on the line you entered. Did you accidentally capitalize `print`? Did you forget one or both of the quotation marks or parentheses? Programming languages expect very specific syntax, and if you don't provide that, you'll get errors. If you can't get the program to run, see the suggestions in the next section.

Troubleshooting

If you can't get *hello_world.py* to run, here are a few remedies you can try that are also good general solutions for any programming problem:

- When a program contains a significant error, Python displays a *traceback*, which is an error report. Python looks through the file and tries to identify the problem. Check the traceback; it might give you a clue as to what issue is preventing the program from running.
- Step away from your computer, take a short break, and then try again. Remember that syntax is very important in programming, so even a missing colon, a mismatched quotation mark, or mismatched parentheses can prevent a program from running properly. Reread the relevant parts of this chapter, look over your code, and try to find the mistake.
- Start over again. You probably don't need to uninstall any software, but it might make sense to delete your *hello_world.py* file and re-create it from scratch.
- Ask someone else to follow the steps in this chapter, on your computer or a different one, and watch what they do carefully. You might have missed one small step that someone else happens to catch.
- Find someone who knows Python and ask them to help you get set up. If you ask around, you might find that you unexpectedly know someone who uses Python.
- The setup instructions in this chapter are also available through the book's companion website at *https://nostarch.com/pythoncrashcourse2e/*. The online version of these instructions might work better for you because you can simply cut and paste code.
- Ask for help online. Appendix C provides a number of resources, such as forums and live chat sites, where you can ask for solutions from people who've already worked through the issue you're currently facing.

Never worry that you're bothering experienced programmers. Every programmer has been stuck at some point, and most programmers are happy to help you set up your system correctly. As long as you can state clearly what you're trying to do, what you've already tried, and the results you're getting, there's a good chance someone will be able to help you. As mentioned in the Introduction, the Python community is very friendly and welcoming to beginners.

Python should run well on any modern computer. Early setup issues can be frustrating, but they're well worth sorting out. Once you get *hello_world.py* running, you can start to learn Python, and your programming work will become more interesting and satisfying.

Running Python Programs from a Terminal

Most of the programs you write in your text editor you'll run directly from the editor. But sometimes it's useful to run programs from a terminal instead. For example, you might want to run an existing program without opening it for editing.

You can do this on any system with Python installed if you know how to access the directory where the program file is stored. To try this, make sure you've saved the *hello_world.py* file in the *python_work* folder on your desktop.

On Windows

You can use the terminal command `cd`, for *change directory*, to navigate through your filesystem in a command window. The command `dir`, for *directory*, shows you all the files that exist in the current directory.

Open a new terminal window and enter the following commands to run *hello_world.py*:

```
❶ C:\> cd Desktop\python_work
❷ C:\Desktop\python_work> dir
  hello_world.py
❸ C:\Desktop\python_work> python hello_world.py
  Hello Python world!
```

At ❶ you use the `cd` command to navigate to the *python_work* folder, which is in the *Desktop* folder. Next, you use the `dir` command to make sure *hello_world.py* is in this folder ❷. Then you run the file using the command `python hello_world.py` ❸.

Most of your programs will run fine directly from your editor. But as your work becomes more complex, you'll want to run some of your programs from a terminal.

On macOS and Linux

Running a Python program from a terminal session is the same on Linux and macOS. You can use the terminal command `cd`, for *change directory*, to navigate through your filesystem in a terminal session. The command `ls`, for *list*, shows you all the nonhidden files that exist in the current directory.

Open a new terminal window and enter the following commands to run *hello_world.py*:

```
❶ ~$ cd Desktop/python_work/
❷ ~/Desktop/python_work$ ls
  hello_world.py
❸ ~/Desktop/python_work$ python hello_world.py
  Hello Python world!
```

At ❶ you use the `cd` command to navigate to the *python_work* folder, which is in the *Desktop* folder. Next, you use the `ls` command to make sure *hello_world.py* is in this folder ❷. Then you run the file using the command `python hello_world.py` ❸.

It's that simple. You just use the `python` (or `python3`) command to run Python programs.

TRY IT YOURSELF

The exercises in this chapter are exploratory in nature. Starting in Chapter 2, the challenges you'll solve will be based on what you've learned.

1-1. python.org: Explore the Python home page (*https://python.org/*) to find topics that interest you. As you become familiar with Python, different parts of the site will be more useful to you.

1-2. Hello World Typos: Open the *hello_world.py* file you just created. Make a typo somewhere in the line and run the program again. Can you make a typo that generates an error? Can you make sense of the error message? Can you make a typo that doesn't generate an error? Why do you think it didn't make an error?

1-3. Infinite Skills: If you had infinite programming skills, what would you build? You're about to learn how to program. If you have an end goal in mind, you'll have an immediate use for your new skills; now is a great time to draft descriptions of what you want to create. It's a good habit to keep an "ideas" notebook that you can refer to whenever you want to start a new project. Take a few minutes now to describe three programs you want to create.

Summary

In this chapter, you learned a bit about Python in general, and you installed Python on your system if it wasn't already there. You also installed a text editor to make it easier to write Python code. You ran snippets of Python code in a terminal session, and you ran your first program, *hello_world.py*. You probably learned a bit about troubleshooting as well.

In the next chapter, you'll learn about the different kinds of data you can work with in your Python programs, and you'll use variables as well.

2

VARIABLES AND SIMPLE DATA TYPES

In this chapter you'll learn about the different kinds of data you can work with in your Python programs. You'll also learn how to use variables to represent data in your programs.

What Really Happens When You Run hello_world.py

Let's take a closer look at what Python does when you run *hello_world.py*. As it turns out, Python does a fair amount of work, even when it runs a simple program:

hello_world.py

```
print("Hello Python world!")
```

When you run this code, you should see this output:

```
Hello Python world!
```

When you run the file *hello_world.py*, the ending *.py* indicates that the file is a Python program. Your editor then runs the file through the *Python interpreter*, which reads through the program and determines what each word in the program means. For example, when the interpreter sees the word `print` followed by parentheses, it prints to the screen whatever is inside the parentheses.

As you write your programs, your editor highlights different parts of your program in different ways. For example, it recognizes that `print()` is the name of a function and displays that word in one color. It recognizes that `"Hello Python world!"` is not Python code and displays that phrase in a different color. This feature is called *syntax highlighting* and is quite useful as you start to write your own programs.

Variables

Let's try using a variable in *hello_world.py*. Add a new line at the beginning of the file, and modify the second line:

hello_world.py

```
message = "Hello Python world!"
print(message)
```

Run this program to see what happens. You should see the same output you saw previously:

```
Hello Python world!
```

We've added a *variable* named `message`. Every variable is connected to a *value*, which is the information associated with that variable. In this case the value is the `"Hello Python world!"` text.

Adding a variable makes a little more work for the Python interpreter. When it processes the first line, it associates the variable `message` with the `"Hello Python world!"` text. When it reaches the second line, it prints the value associated with `message` to the screen.

Let's expand on this program by modifying *hello_world.py* to print a second message. Add a blank line to *hello_world.py*, and then add two new lines of code:

```
message = "Hello Python world!"
print(message)

message = "Hello Python Crash Course world!"
print(message)
```

Now when you run *hello_world.py*, you should see two lines of output:

```
Hello Python world!
Hello Python Crash Course world!
```

You can change the value of a variable in your program at any time, and Python will always keep track of its current value.

Naming and Using Variables

When you're using variables in Python, you need to adhere to a few rules and guidelines. Breaking some of these rules will cause errors; other guidelines just help you write code that's easier to read and understand. Be sure to keep the following variable rules in mind:

- Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable *message_1* but not *1_message*.
- Spaces are not allowed in variable names, but underscores can be used to separate words in variable names. For example, *greeting_message* works, but *greeting message* will cause errors.
- Avoid using Python keywords and function names as variable names; that is, do not use words that Python has reserved for a particular programmatic purpose, such as the word `print`. (See "Python Keywords and Built-in Functions" on page 471.)
- Variable names should be short but descriptive. For example, *name* is better than *n*, *student_name* is better than *s_n*, and *name_length* is better than *length_of_persons_name*.
- Be careful when using the lowercase letter *l* and the uppercase letter *O* because they could be confused with the numbers *1* and *0*.

It can take some practice to learn how to create good variable names, especially as your programs become more interesting and complicated. As you write more programs and start to read through other people's code, you'll get better at coming up with meaningful names.

NOTE *The Python variables you're using at this point should be lowercase. You won't get errors if you use uppercase letters, but uppercase letters in variable names have special meanings that we'll discuss in later chapters.*

Avoiding Name Errors When Using Variables

Every programmer makes mistakes, and most make mistakes every day. Although good programmers might create errors, they also know how to respond to those errors efficiently. Let's look at an error you're likely to make early on and learn how to fix it.

We'll write some code that generates an error on purpose. Enter the following code, including the misspelled word *mesage* shown in bold:

```
message = "Hello Python Crash Course reader!"
print(mesage)
```

When an error occurs in your program, the Python interpreter does its best to help you figure out where the problem is. The interpreter provides a traceback when a program cannot run successfully. A *traceback* is a record of where the interpreter ran into trouble when trying to execute your code. Here's an example of the traceback that Python provides after you've accidentally misspelled a variable's name:

```
Traceback (most recent call last):
❶  File "hello_world.py", line 2, in <module>
❷    print(mesage)
❸ NameError: name 'mesage' is not defined
```

The output at ❶ reports that an error occurs in line 2 of the file *hello_world.py*. The interpreter shows this line ❷ to help us spot the error quickly and tells us what kind of error it found ❸. In this case it found a *name error* and reports that the variable being printed, `mesage`, has not been defined. Python can't identify the variable name provided. A name error usually means we either forgot to set a variable's value before using it, or we made a spelling mistake when entering the variable's name.

Of course, in this example we omitted the letter *s* in the variable name `message` in the second line. The Python interpreter doesn't spellcheck your code, but it does ensure that variable names are spelled consistently. For example, watch what happens when we spell *message* incorrectly in another place in the code as well:

```
mesage = "Hello Python Crash Course reader!"
print(mesage)
```

In this case, the program runs successfully!

```
Hello Python Crash Course reader!
```

Programming languages are strict, but they disregard good and bad spelling. As a result, you don't need to consider English spelling and grammar rules when you're trying to create variable names and writing code.

Many programming errors are simple, single-character typos in one line of a program. If you're spending a long time searching for one of these errors, know that you're in good company. Many experienced and talented programmers spend hours hunting down these kinds of tiny errors. Try to laugh about it and move on, knowing it will happen frequently throughout your programming life.

Variables Are Labels

Variables are often described as boxes you can store values in. This idea can be helpful the first few times you use a variable, but it isn't an accurate way to describe how variables are represented internally in Python. It's much better to think of variables as labels that you can assign to values. You can also say that a variable references a certain value.

This distinction probably won't matter much in your initial programs, but it's worth learning earlier rather than later. At some point, you'll see unexpected behavior from a variable, and an accurate understanding of how variables work will help you identify what's happening in your code.

NOTE *The best way to understand new programming concepts is to try using them in your programs. If you get stuck while working on an exercise in this book, try doing something else for a while. If you're still stuck, review the relevant part of that chapter. If you still need help, see the suggestions in Appendix C.*

TRY IT YOURSELF

Write a separate program to accomplish each of these exercises. Save each program with a filename that follows standard Python conventions, using lowercase letters and underscores, such as *simple_message.py* and *simple_messages.py*.

2-1. Simple Message: Assign a message to a variable, and then print that message.

2-2. Simple Messages: Assign a message to a variable, and print that message. Then change the value of the variable to a new message, and print the new message.

Strings

Because most programs define and gather some sort of data, and then do something useful with it, it helps to classify different types of data. The first data type we'll look at is the string. Strings are quite simple at first glance, but you can use them in many different ways.

A *string* is a series of characters. Anything inside quotes is considered a string in Python, and you can use single or double quotes around your strings like this:

```
"This is a string."
'This is also a string.'
```

This flexibility allows you to use quotes and apostrophes within your strings:

```
'I told my friend, "Python is my favorite language!"'
"The language 'Python' is named after Monty Python, not the snake."
"One of Python's strengths is its diverse and supportive community."
```

Let's explore some of the ways you can use strings.

Changing Case in a String with Methods

One of the simplest tasks you can do with strings is change the case of the words in a string. Look at the following code, and try to determine what's happening:

name.py

```
name = "ada lovelace"
print(name.title())
```

Save this file as *name.py*, and then run it. You should see this output:

```
Ada Lovelace
```

In this example, the variable `name` refers to the lowercase string `"ada lovelace"`. The method `title()` appears after the variable in the `print()` call. A *method* is an action that Python can perform on a piece of data. The dot (`.`) after `name` in `name.title()` tells Python to make the `title()` method act on the variable `name`. Every method is followed by a set of parentheses, because methods often need additional information to do their work. That information is provided inside the parentheses. The `title()` function doesn't need any additional information, so its parentheses are empty.

The `title()` method changes each word to title case, where each word begins with a capital letter. This is useful because you'll often want to think of a name as a piece of information. For example, you might want your program to recognize the input values `Ada`, `ADA`, and `ada` as the same name, and display all of them as `Ada`.

Several other useful methods are available for dealing with case as well. For example, you can change a string to all uppercase or all lowercase letters like this:

```
name = "Ada Lovelace"
print(name.upper())
print(name.lower())
```

This will display the following:

```
ADA LOVELACE
ada lovelace
```

The `lower()` method is particularly useful for storing data. Many times you won't want to trust the capitalization that your users provide, so you'll convert strings to lowercase before storing them. Then when you want to display the information, you'll use the case that makes the most sense for each string.

Using Variables in Strings

In some situations, you'll want to use a variable's value inside a string. For example, you might want two variables to represent a first name and a last name respectively, and then want to combine those values to display someone's full name:

full_name.py

```
first_name = "ada"
last_name = "lovelace"
❶ full_name = f"{first_name} {last_name}"
print(full_name)
```

To insert a variable's value into a string, place the letter `f` immediately before the opening quotation mark ❶. Put braces around the name or names of any variable you want to use inside the string. Python will replace each variable with its value when the string is displayed.

These strings are called *f-strings*. The *f* is for *format*, because Python formats the string by replacing the name of any variable in braces with its value. The output from the previous code is:

```
ada lovelace
```

You can do a lot with f-strings. For example, you can use f-strings to compose complete messages using the information associated with a variable, as shown here:

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
❶ print(f"Hello, {full_name.title()}!")
```

The full name is used in a sentence that greets the user ❶, and the `title()` method changes the name to title case. This code returns a simple but nicely formatted greeting:

```
Hello, Ada Lovelace!
```

You can also use f-strings to compose a message, and then assign the entire message to a variable:

```
first_name = "ada"
last_name = "lovelace"
full_name = f"{first_name} {last_name}"
❶ message = f"Hello, {full_name.title()}!"
❷ print(message)
```

This code displays the message `Hello, Ada Lovelace!` as well, but by assigning the message to a variable ❶ we make the final `print()` call much simpler ❷.

NOTE *F-strings were first introduced in Python 3.6. If you're using Python 3.5 or earlier, you'll need to use the `format()` method rather than this `f` syntax. To use `format()`, list the variables you want to use in the string inside the parentheses following `format`. Each variable is referred to by a set of braces; the braces will be filled by the values listed in parentheses in the order provided:*

```
full_name = "{} {}".format(first_name, last_name)
```

Adding Whitespace to Strings with Tabs or Newlines

In programming, *whitespace* refers to any nonprinting character, such as spaces, tabs, and end-of-line symbols. You can use whitespace to organize your output so it's easier for users to read.

To add a tab to your text, use the character combination `\t` as shown at ❶:

```
>>> print("Python")
Python
❶ >>> print("\tPython")
    Python
```

To add a newline in a string, use the character combination `\n`:

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

You can also combine tabs and newlines in a single string. The string `"\n\t"` tells Python to move to a new line, and start the next line with a tab. The following example shows how you can use a one-line string to generate four lines of output:

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
Languages:
    Python
    C
    JavaScript
```

Newlines and tabs will be very useful in the next two chapters when you start to produce many lines of output from just a few lines of code.

Stripping Whitespace

Extra whitespace can be confusing in your programs. To programmers `'python'` and `'python '` look pretty much the same. But to a program, they are two different strings. Python detects the extra space in `'python '` and considers it significant unless you tell it otherwise.

It's important to think about whitespace, because often you'll want to compare two strings to determine whether they are the same. For example, one important instance might involve checking people's usernames when they log in to a website. Extra whitespace can be confusing in much simpler situations as well. Fortunately, Python makes it easy to eliminate extraneous whitespace from data that people enter.

Python can look for extra whitespace on the right and left sides of a string. To ensure that no whitespace exists at the right end of a string, use the `rstrip()` method.

```
❶ >>> favorite_language = 'python '
❷ >>> favorite_language
  'python '
❸ >>> favorite_language.rstrip()
  'python'
❹ >>> favorite_language
  'python '
```

The value associated with `favorite_language` at ❶ contains extra whitespace at the end of the string. When you ask Python for this value in a terminal session, you can see the space at the end of the value ❷. When the `rstrip()` method acts on the variable `favorite_language` at ❸, this extra space is removed. However, it is only removed temporarily. If you ask for the value of `favorite_language` again, you can see that the string looks the same as when it was entered, including the extra whitespace ❹.

To remove the whitespace from the string permanently, you have to associate the stripped value with the variable name:

```
  >>> favorite_language = 'python '
❶ >>> favorite_language = favorite_language.rstrip()
  >>> favorite_language
  'python'
```

To remove the whitespace from the string, you strip the whitespace from the right side of the string and then associate this new value with the original variable, as shown at ❶. Changing a variable's value is done often in programming. This is how a variable's value can be updated as a program is executed or in response to user input.

You can also strip whitespace from the left side of a string using the `lstrip()` method, or from both sides at once using `strip()`:

```
❶ >>> favorite_language = ' python '
❷ >>> favorite_language.rstrip()
  ' python'
❸ >>> favorite_language.lstrip()
  'python '
❹ >>> favorite_language.strip()
  'python'
```

In this example, we start with a value that has whitespace at the beginning and the end ❶. We then remove the extra space from the right side at ❷, from the left side at ❸, and from both sides at ❹. Experimenting with these stripping functions can help you become familiar with manipulating strings. In the real world, these stripping functions are used most often to clean up user input before it's stored in a program.

Avoiding Syntax Errors with Strings

One kind of error that you might see with some regularity is a syntax error. A *syntax error* occurs when Python doesn't recognize a section of your program as valid Python code. For example, if you use an apostrophe within single quotes, you'll produce an error. This happens because Python interprets everything between the first single quote and the apostrophe as a string. It then tries to interpret the rest of the text as Python code, which causes errors.

Here's how to use single and double quotes correctly. Save this program as *apostrophe.py* and then run it:

apostrophe.py

```
message = "One of Python's strengths is its diverse community."
print(message)
```

The apostrophe appears inside a set of double quotes, so the Python interpreter has no trouble reading the string correctly:

```
One of Python's strengths is its diverse community.
```

However, if you use single quotes, Python can't identify where the string should end:

```
message = 'One of Python's strengths is its diverse community.'
print(message)
```

You'll see the following output:

```
  File "apostrophe.py", line 1
    message = 'One of Python's strengths is its diverse community.'
                              ^❶
SyntaxError: invalid syntax
```

In the output you can see that the error occurs at ❶ right after the second single quote. This syntax error indicates that the interpreter doesn't recognize something in the code as valid Python code. Errors can come from a variety of sources, and I'll point out some common ones as they arise. You might see syntax errors often as you learn to write proper Python code. Syntax errors are also the least specific kind of error, so they can be difficult and frustrating to identify and correct. If you get stuck on a particularly stubborn error, see the suggestions in Appendix C.

NOTE *Your editor's syntax highlighting feature should help you spot some syntax errors quickly as you write your programs. If you see Python code highlighted as if it's English or English highlighted as if it's Python code, you probably have a mismatched quotation mark somewhere in your file.*

TRY IT YOURSELF

Save each of the following exercises as a separate file with a name like *name_cases.py*. If you get stuck, take a break or see the suggestions in Appendix C.

2-3. Personal Message: Use a variable to represent a person's name, and print a message to that person. Your message should be simple, such as, "Hello Eric, would you like to learn some Python today?"

2-4. Name Cases: Use a variable to represent a person's name, and then print that person's name in lowercase, uppercase, and title case.

2-5. Famous Quote: Find a quote from a famous person you admire. Print the quote and the name of its author. Your output should look something like the following, including the quotation marks:

> *Albert Einstein once said, "A person who never made a mistake never tried anything new."*

2-6. Famous Quote 2: Repeat Exercise 2-5, but this time, represent the famous person's name using a variable called `famous_person`. Then compose your message and represent it with a new variable called `message`. Print your message.

2-7. Stripping Names: Use a variable to represent a person's name, and include some whitespace characters at the beginning and end of the name. Make sure you use each character combination, `"\t"` and `"\n"`, at least once.

Print the name once, so the whitespace around the name is displayed. Then print the name using each of the three stripping functions, `lstrip()`, `rstrip()`, and `strip()`.

Numbers

Numbers are used quite often in programming to keep score in games, represent data in visualizations, store information in web applications, and so on. Python treats numbers in several different ways, depending on how they're being used. Let's first look at how Python manages integers, because they're the simplest to work with.

Integers

You can add (+), subtract (-), multiply (*), and divide (/) integers in Python.

```
>>> 2 + 3
5
>>> 3 - 2
1
>>> 2 * 3
6
>>> 3 / 2
1.5
```

In a terminal session, Python simply returns the result of the operation. Python uses two multiplication symbols to represent exponents:

```
>>> 3 ** 2
9
>>> 3 ** 3
27
>>> 10 ** 6
1000000
```

Python supports the order of operations too, so you can use multiple operations in one expression. You can also use parentheses to modify the order of operations so Python can evaluate your expression in the order you specify. For example:

```
>>> 2 + 3*4
14
>>> (2 + 3) * 4
20
```

The spacing in these examples has no effect on how Python evaluates the expressions; it simply helps you more quickly spot the operations that have priority when you're reading through the code.

Floats

Python calls any number with a decimal point a *float*. This term is used in most programming languages, and it refers to the fact that a decimal point can appear at any position in a number. Every programming language must be carefully designed to properly manage decimal numbers so numbers behave appropriately no matter where the decimal point appears.

For the most part, you can use decimals without worrying about how they behave. Simply enter the numbers you want to use, and Python will most likely do what you expect:

```
>>> 0.1 + 0.1
0.2
>>> 0.2 + 0.2
0.4
>>> 2 * 0.1
0.2
>>> 2 * 0.2
0.4
```

But be aware that you can sometimes get an arbitrary number of decimal places in your answer:

```
>>> 0.2 + 0.1
0.30000000000000004
>>> 3 * 0.1
0.30000000000000004
```

This happens in all languages and is of little concern. Python tries to find a way to represent the result as precisely as possible, which is sometimes difficult given how computers have to represent numbers internally. Just ignore the extra decimal places for now; you'll learn ways to deal with the extra places when you need to in the projects in Part II.

Integers and Floats

When you divide any two numbers, even if they are integers that result in a whole number, you'll always get a float:

```
>>> 4/2
2.0
```

If you mix an integer and a float in any other operation, you'll get a float as well:

```
>>> 1 + 2.0
3.0
>>> 2 * 3.0
6.0
>>> 3.0 ** 2
9.0
```

Python defaults to a float in any operation that uses a float, even if the output is a whole number.

Underscores in Numbers

When you're writing long numbers, you can group digits using underscores to make large numbers more readable:

```
>>> universe_age = 14_000_000_000
```

When you print a number that was defined using underscores, Python prints only the digits:

```
>>> print(universe_age)
14000000000
```

Python ignores the underscores when storing these kinds of values. Even if you don't group the digits in threes, the value will still be unaffected. To Python, `1000` is the same as `1_000`, which is the same as `10_00`. This feature works for integers and floats, but it's only available in Python 3.6 and later.

Multiple Assignment

You can assign values to more than one variable using just a single line. This can help shorten your programs and make them easier to read; you'll use this technique most often when initializing a set of numbers.

For example, here's how you can initialize the variables `x`, `y`, and `z` to zero:

```
>>> x, y, z = 0, 0, 0
```

You need to separate the variable names with commas, and do the same with the values, and Python will assign each value to its respectively positioned variable. As long as the number of values matches the number of variables, Python will match them up correctly.

Constants

A *constant* is like a variable whose value stays the same throughout the life of a program. Python doesn't have built-in constant types, but Python programmers use all capital letters to indicate a variable should be treated as a constant and never be changed:

```
MAX_CONNECTIONS = 5000
```

When you want to treat a variable as a constant in your code, make the name of the variable all capital letters.

TRY IT YOURSELF

2-8. Number Eight: Write addition, subtraction, multiplication, and division operations that each result in the number 8. Be sure to enclose your operations in `print()` calls to see the results. You should create four lines that look like this:

```
print(5+3)
```

Your output should simply be four lines with the number 8 appearing once on each line.

2-9. Favorite Number: Use a variable to represent your favorite number. Then, using that variable, create a message that reveals your favorite number. Print that message.

Comments

Comments are an extremely useful feature in most programming languages. Everything you've written in your programs so far is Python code. As your programs become longer and more complicated, you should add notes within your programs that describe your overall approach to the problem you're solving. A *comment* allows you to write notes in English within your programs.

How Do You Write Comments?

In Python, the hash mark (#) indicates a comment. Anything following a hash mark in your code is ignored by the Python interpreter. For example:

comment.py

```
# Say hello to everyone.
print("Hello Python people!")
```

Python ignores the first line and executes the second line.

```
Hello Python people!
```

What Kind of Comments Should You Write?

The main reason to write comments is to explain what your code is supposed to do and how you are making it work. When you're in the middle of working on a project, you understand how all of the pieces fit together. But when you return to a project after some time away, you'll likely have forgotten

some of the details. You can always study your code for a while and figure out how segments were supposed to work, but writing good comments can save you time by summarizing your overall approach in clear English.

If you want to become a professional programmer or collaborate with other programmers, you should write meaningful comments. Today, most software is written collaboratively, whether by a group of employees at one company or a group of people working together on an open source project. Skilled programmers expect to see comments in code, so it's best to start adding descriptive comments to your programs now. Writing clear, concise comments in your code is one of the most beneficial habits you can form as a new programmer.

When you're determining whether to write a comment, ask yourself if you had to consider several approaches before coming up with a reasonable way to make something work; if so, write a comment about your solution. It's much easier to delete extra comments later on than it is to go back and write comments for a sparsely commented program. From now on, I'll use comments in examples throughout this book to help explain sections of code.

TRY IT YOURSELF

2-10. Adding Comments: Choose two of the programs you've written, and add at least one comment to each. If you don't have anything specific to write because your programs are too simple at this point, just add your name and the current date at the top of each program file. Then write one sentence describing what the program does.

The Zen of Python

Experienced Python programmers will encourage you to avoid complexity and aim for simplicity whenever possible. The Python community's philosophy is contained in "The Zen of Python" by Tim Peters. You can access this brief set of principles for writing good Python code by entering `import this` into your interpreter. I won't reproduce the entire "Zen of Python" here, but I'll share a few lines to help you understand why they should be important to you as a beginning Python programmer.

```
>>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
```

Python programmers embrace the notion that code can be beautiful and elegant. In programming, people solve problems. Programmers have always respected well-designed, efficient, and even beautiful solutions to

problems. As you learn more about Python and use it to write more code, someone might look over your shoulder one day and say, "Wow, that's some beautiful code!"

```
Simple is better than complex.
```

If you have a choice between a simple and a complex solution, and both work, use the simple solution. Your code will be easier to maintain, and it will be easier for you and others to build on that code later on.

```
Complex is better than complicated.
```

Real life is messy, and sometimes a simple solution to a problem is unattainable. In that case, use the simplest solution that works.

```
Readability counts.
```

Even when your code is complex, aim to make it readable. When you're working on a project that involves complex coding, focus on writing informative comments for that code.

```
There should be one-- and preferably only one --obvious way to do it.
```

If two Python programmers are asked to solve the same problem, they should come up with fairly compatible solutions. This is not to say there's no room for creativity in programming. On the contrary! But much of programming consists of using small, common approaches to simple situations within a larger, more creative project. The nuts and bolts of your programs should make sense to other Python programmers.

```
Now is better than never.
```

You could spend the rest of your life learning all the intricacies of Python and of programming in general, but then you'd never complete any projects. Don't try to write perfect code; write code that works, and then decide whether to improve your code for that project or move on to something new.

As you continue to the next chapter and start digging into more involved topics, try to keep this philosophy of simplicity and clarity in mind. Experienced programmers will respect your code more and will be happy to give you feedback and collaborate with you on interesting projects.

TRY IT YOURSELF

2-11. Zen of Python: Enter `import this` into a Python terminal session and skim through the additional principles.

Summary

In this chapter you learned to work with variables. You learned to use descriptive variable names and how to resolve name errors and syntax errors when they arise. You learned what strings are and how to display strings using lowercase, uppercase, and title case. You started using whitespace to organize output neatly, and you learned to strip unneeded whitespace from different parts of a string. You started working with integers and floats, and learned some of the ways you can work with numerical data. You also learned to write explanatory comments to make your code easier for you and others to read. Finally, you read about the philosophy of keeping your code as simple as possible, whenever possible.

In Chapter 3 you'll learn to store collections of information in data structures called *lists*. You'll learn to work through a list, manipulating any information in that list.

3

INTRODUCING LISTS

In this chapter and the next you'll learn what lists are and how to start working with the elements in a list. Lists allow you to store sets of information in one place, whether you have just a few items or millions of items. Lists are one of Python's most powerful features readily accessible to new programmers, and they tie together many important concepts in programming.

What Is a List?

A *list* is a collection of items in a particular order. You can make a list that includes the letters of the alphabet, the digits from 0–9, or the names of all the people in your family. You can put anything you want into a list, and

the items in your list don't have to be related in any particular way. Because a list usually contains more than one element, it's a good idea to make the name of your list plural, such as `letters`, `digits`, or `names`.

In Python, square brackets (`[]`) indicate a list, and individual elements in the list are separated by commas. Here's a simple example of a list that contains a few kinds of bicycles:

bicycles.py

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)
```

If you ask Python to print a list, Python returns its representation of the list, including the square brackets:

```
['trek', 'cannondale', 'redline', 'specialized']
```

Because this isn't the output you want your users to see, let's learn how to access the individual items in a list.

Accessing Elements in a List

Lists are ordered collections, so you can access any element in a list by telling Python the position, or *index*, of the item desired. To access an element in a list, write the name of the list followed by the index of the item enclosed in square brackets.

For example, let's pull out the first bicycle in the list `bicycles`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
❶ print(bicycles[0])
```

The syntax for this is shown at ❶. When we ask for a single item from a list, Python returns just that element without square brackets:

```
trek
```

This is the result you want your users to see—clean, neatly formatted output.

You can also use the string methods from Chapter 2 on any element in this list. For example, you can format the element `'trek'` more neatly by using the `title()` method:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0].title())
```

This example produces the same output as the preceding example except `'Trek'` is capitalized.

Index Positions Start at 0, Not 1

Python considers the first item in a list to be at position 0, not position 1. This is true of most programming languages, and the reason has to do with how the list operations are implemented at a lower level. If you're receiving unexpected results, determine whether you are making a simple off-by-one error.

The second item in a list has an index of 1. Using this counting system, you can get any element you want from a list by subtracting one from its position in the list. For instance, to access the fourth item in a list, you request the item at index 3.

The following asks for the bicycles at index `1` and index `3`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[1])
print(bicycles[3])
```

This code returns the second and fourth bicycles in the list:

```
cannondale
specialized
```

Python has a special syntax for accessing the last element in a list. By asking for the item at index `-1`, Python always returns the last item in the list:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[-1])
```

This code returns the value `'specialized'`. This syntax is quite useful, because you'll often want to access the last items in a list without knowing exactly how long the list is. This convention extends to other negative index values as well. The index `-2` returns the second item from the end of the list, the index `-3` returns the third item from the end, and so forth.

Using Individual Values from a List

You can use individual values from a list just as you would any other variable. For example, you can use f-strings to create a message based on a value from a list.

Let's try pulling the first bicycle from the list and composing a message using that value.

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
❶ message = f"My first bicycle was a {bicycles[0].title()}."

print(message)
```

At ❶, we build a sentence using the value at `bicycles[0]` and assign it to the variable `message`. The output is a simple sentence about the first bicycle in the list:

```
My first bicycle was a Trek.
```

TRY IT YOURSELF

Try these short programs to get some firsthand experience with Python's lists. You might want to create a new folder for each chapter's exercises to keep them organized.

3-1. Names: Store the names of a few of your friends in a list called `names`. Print each person's name by accessing each element in the list, one at a time.

3-2. Greetings: Start with the list you used in Exercise 3-1, but instead of just printing each person's name, print a message to them. The text of each message should be the same, but each message should be personalized with the person's name.

3-3. Your Own List: Think of your favorite mode of transportation, such as a motorcycle or a car, and make a list that stores several examples. Use your list to print a series of statements about these items, such as "I would like to own a Honda motorcycle."

Changing, Adding, and Removing Elements

Most lists you create will be dynamic, meaning you'll build a list and then add and remove elements from it as your program runs its course. For example, you might create a game in which a player has to shoot aliens out of the sky. You could store the initial set of aliens in a list and then remove an alien from the list each time one is shot down. Each time a new alien appears on the screen, you add it to the list. Your list of aliens will increase and decrease in length throughout the course of the game.

Modifying Elements in a List

The syntax for modifying an element is similar to the syntax for accessing an element in a list. To change an element, use the name of the list followed by the index of the element you want to change, and then provide the new value you want that item to have.

For example, let's say we have a list of motorcycles, and the first item in the list is `'honda'`. How would we change the value of this first item?

motorcycles.py

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)

❷ motorcycles[0] = 'ducati'
  print(motorcycles)
```

The code at ❶ defines the original list, with `'honda'` as the first element. The code at ❷ changes the value of the first item to `'ducati'`. The output shows that the first item has indeed been changed, and the rest of the list stays the same:

```
['honda', 'yamaha', 'suzuki']
['ducati', 'yamaha', 'suzuki']
```

You can change the value of any item in a list, not just the first item.

Adding Elements to a List

You might want to add a new element to a list for many reasons. For example, you might want to make new aliens appear in a game, add new data to a visualization, or add new registered users to a website you've built. Python provides several ways to add new data to existing lists.

Appending Elements to the End of a List

The simplest way to add a new element to a list is to *append* the item to the list. When you append an item to a list, the new element is added to the end of the list. Using the same list we had in the previous example, we'll add the new element `'ducati'` to the end of the list:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

❶ motorcycles.append('ducati')
  print(motorcycles)
```

The `append()` method at ❶ adds `'ducati'` to the end of the list without affecting any of the other elements in the list:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha', 'suzuki', 'ducati']
```

The `append()` method makes it easy to build lists dynamically. For example, you can start with an empty list and then add items to the list using a series of `append()` calls. Using an empty list, let's add the elements `'honda'`, `'yamaha'`, and `'suzuki'` to the list:

```
motorcycles = []

motorcycles.append('honda')
motorcycles.append('yamaha')
motorcycles.append('suzuki')

print(motorcycles)
```

The resulting list looks exactly the same as the lists in the previous examples:

```
['honda', 'yamaha', 'suzuki']
```

Building lists this way is very common, because you often won't know the data your users want to store in a program until after the program is running. To put your users in control, start by defining an empty list that will hold the users' values. Then append each new value provided to the list you just created.

Inserting Elements into a List

You can add a new element at any position in your list by using the `insert()` method. You do this by specifying the index of the new element and the value of the new item.

```
motorcycles = ['honda', 'yamaha', 'suzuki']

❶ motorcycles.insert(0, 'ducati')
  print(motorcycles)
```

In this example, the code at ❶ inserts the value `'ducati'` at the beginning of the list. The `insert()` method opens a space at position `0` and stores the value `'ducati'` at that location. This operation shifts every other value in the list one position to the right:

```
['ducati', 'honda', 'yamaha', 'suzuki']
```

Removing Elements from a List

Often, you'll want to remove an item or a set of items from a list. For example, when a player shoots down an alien from the sky, you'll most likely want to remove it from the list of active aliens. Or when a user

decides to cancel their account on a web application you created, you'll want to remove that user from the list of active users. You can remove an item according to its position in the list or according to its value.

Removing an Item Using the del Statement

If you know the position of the item you want to remove from a list, you can use the `del` statement.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

❶ del motorcycles[0]
print(motorcycles)
```

The code at ❶ uses `del` to remove the first item, `'honda'`, from the list of motorcycles:

```
['honda', 'yamaha', 'suzuki']
['yamaha', 'suzuki']
```

You can remove an item from any position in a list using the `del` statement if you know its index. For example, here's how to remove the second item, `'yamaha'`, in the list:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[1]
print(motorcycles)
```

The second motorcycle is deleted from the list:

```
['honda', 'yamaha', 'suzuki']
['honda', 'suzuki']
```

In both examples, you can no longer access the value that was removed from the list after the `del` statement is used.

Removing an Item Using the pop() Method

Sometimes you'll want to use the value of an item after you remove it from a list. For example, you might want to get the *x* and *y* position of an alien that was just shot down, so you can draw an explosion at that position. In a web application, you might want to remove a user from a list of active members and then add that user to a list of inactive members.

The `pop()` method removes the last item in a list, but it lets you work with that item after removing it. The term *pop* comes from thinking of a list as a stack of items and popping one item off the top of the stack. In this analogy, the top of a stack corresponds to the end of a list.

Let's pop a motorcycle from the list of motorcycles:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki']
  print(motorcycles)

❷ popped_motorcycle = motorcycles.pop()
❸ print(motorcycles)
❹ print(popped_motorcycle)
```

We start by defining and printing the list `motorcycles` at ❶. At ❷ we pop a value from the list and store that value in the variable `popped_motorcycle`. We print the list at ❸ to show that a value has been removed from the list. Then we print the popped value at ❹ to prove that we still have access to the value that was removed.

The output shows that the value `'suzuki'` was removed from the end of the list and is now assigned to the variable `popped_motorcycle`:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha']
suzuki
```

How might this `pop()` method be useful? Imagine that the motorcycles in the list are stored in chronological order according to when we owned them. If this is the case, we can use the `pop()` method to print a statement about the last motorcycle we bought:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

last_owned = motorcycles.pop()
print(f"The last motorcycle I owned was a {last_owned.title()}.")
```

The output is a simple sentence about the most recent motorcycle we owned:

```
The last motorcycle I owned was a Suzuki.
```

Popping Items from any Position in a List

You can use `pop()` to remove an item from any position in a list by including the index of the item you want to remove in parentheses.

```
motorcycles = ['honda', 'yamaha', 'suzuki']

❶ first_owned = motorcycles.pop(0)
❷ print(f"The first motorcycle I owned was a {first_owned.title()}.")
```

We start by popping the first motorcycle in the list at ❶, and then we print a message about that motorcycle at ❷. The output is a simple sentence describing the first motorcycle I ever owned:

```
The first motorcycle I owned was a Honda.
```

Remember that each time you use `pop()`, the item you work with is no longer stored in the list.

If you're unsure whether to use the `del` statement or the `pop()` method, here's a simple way to decide: when you want to delete an item from a list and not use that item in any way, use the `del` statement; if you want to use an item as you remove it, use the `pop()` method.

Removing an Item by Value

Sometimes you won't know the position of the value you want to remove from a list. If you only know the value of the item you want to remove, you can use the `remove()` method.

For example, let's say we want to remove the value `'ducati'` from the list of motorcycles.

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)

❶ motorcycles.remove('ducati')
  print(motorcycles)
```

The code at ❶ tells Python to figure out where `'ducati'` appears in the list and remove that element:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

You can also use the `remove()` method to work with a value that's being removed from a list. Let's remove the value `'ducati'` and print a reason for removing it from the list:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
  print(motorcycles)

❷ too_expensive = 'ducati'
❸ motorcycles.remove(too_expensive)
  print(motorcycles)
❹ print(f"\nA {too_expensive.title()} is too expensive for me.")
```

After defining the list at ❶, we assign the value `'ducati'` to a variable called `too_expensive` ❷. We then use this variable to tell Python which value to remove from the list at ❸. At ❹ the value `'ducati'` has been removed

from the list but is still accessible through the variable `too_expensive`, allowing us to print a statement about why we removed `'ducati'` from the list of motorcycles:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']

A Ducati is too expensive for me.
```

NOTE *The `remove()` method deletes only the first occurrence of the value you specify. If there's a possibility the value appears more than once in the list, you'll need to use a loop to make sure all occurrences of the value are removed. You'll learn how to do this in Chapter 7.*

TRY IT YOURSELF

The following exercises are a bit more complex than those in Chapter 2, but they give you an opportunity to use lists in all of the ways described.

3-4. Guest List: If you could invite anyone, living or deceased, to dinner, who would you invite? Make a list that includes at least three people you'd like to invite to dinner. Then use your list to print a message to each person, inviting them to dinner.

3-5. Changing Guest List: You just heard that one of your guests can't make the dinner, so you need to send out a new set of invitations. You'll have to think of someone else to invite.

- Start with your program from Exercise 3-4. Add a `print()` call at the end of your program stating the name of the guest who can't make it.
- Modify your list, replacing the name of the guest who can't make it with the name of the new person you are inviting.
- Print a second set of invitation messages, one for each person who is still in your list.

3-6. More Guests: You just found a bigger dinner table, so now more space is available. Think of three more guests to invite to dinner.

- Start with your program from Exercise 3-4 or Exercise 3-5. Add a `print()` call to the end of your program informing people that you found a bigger dinner table.
- Use `insert()` to add one new guest to the beginning of your list.
- Use `insert()` to add one new guest to the middle of your list.
- Use `append()` to add one new guest to the end of your list.
- Print a new set of invitation messages, one for each person in your list.

3-7. Shrinking Guest List: You just found out that your new dinner table won't arrive in time for the dinner, and you have space for only two guests.

- Start with your program from Exercise 3-6. Add a new line that prints a message saying that you can invite only two people for dinner.
- Use `pop()` to remove guests from your list one at a time until only two names remain in your list. Each time you pop a name from your list, print a message to that person letting them know you're sorry you can't invite them to dinner.
- Print a message to each of the two people still on your list, letting them know they're still invited.
- Use `del` to remove the last two names from your list, so you have an empty list. Print your list to make sure you actually have an empty list at the end of your program.

Organizing a List

Often, your lists will be created in an unpredictable order, because you can't always control the order in which your users provide their data. Although this is unavoidable in most circumstances, you'll frequently want to present your information in a particular order. Sometimes you'll want to preserve the original order of your list, and other times you'll want to change the original order. Python provides a number of different ways to organize your lists, depending on the situation.

Sorting a List Permanently with the sort() Method

Python's `sort()` method makes it relatively easy to sort a list. Imagine we have a list of cars and want to change the order of the list to store them alphabetically. To keep the task simple, let's assume that all the values in the list are lowercase.

cars.py

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
❶ cars.sort()
print(cars)
```

The `sort()` method, shown at ❶, changes the order of the list permanently. The cars are now in alphabetical order, and we can never revert to the original order:

```
['audi', 'bmw', 'subaru', 'toyota']
```

You can also sort this list in reverse alphabetical order by passing the argument `reverse=True` to the `sort()` method. The following example sorts the list of cars in reverse alphabetical order:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort(reverse=True)
print(cars)
```

Again, the order of the list is permanently changed:

```
['toyota', 'subaru', 'bmw', 'audi']
```

Sorting a List Temporarily with the sorted() Function

To maintain the original order of a list but present it in a sorted order, you can use the `sorted()` function. The `sorted()` function lets you display your list in a particular order but doesn't affect the actual order of the list.

Let's try this function on the list of cars.

```
cars = ['bmw', 'audi', 'toyota', 'subaru']

❶ print("Here is the original list:")
  print(cars)

❷ print("\nHere is the sorted list:")
  print(sorted(cars))

❸ print("\nHere is the original list again:")
  print(cars)
```

We first print the list in its original order at ❶ and then in alphabetical order at ❷. After the list is displayed in the new order, we show that the list is still stored in its original order at ❸.

```
Here is the original list:
['bmw', 'audi', 'toyota', 'subaru']

Here is the sorted list:
['audi', 'bmw', 'subaru', 'toyota']

❹ Here is the original list again:
['bmw', 'audi', 'toyota', 'subaru']
```

Notice that the list still exists in its original order at ❹ after the `sorted()` function has been used. The `sorted()` function can also accept a `reverse=True` argument if you want to display a list in reverse alphabetical order.

NOTE *Sorting a list alphabetically is a bit more complicated when all the values are not in lowercase. There are several ways to interpret capital letters when determining a sort order, and specifying the exact order can be more complex than we want to deal with at this time. However, most approaches to sorting will build directly on what you learned in this section.*

Printing a List in Reverse Order

To reverse the original order of a list, you can use the `reverse()` method. If we originally stored the list of cars in chronological order according to when we owned them, we could easily rearrange the list into reverse chronological order:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(cars)

cars.reverse()
print(cars)
```

Notice that `reverse()` doesn't sort backward alphabetically; it simply reverses the order of the list:

```
['bmw', 'audi', 'toyota', 'subaru']
['subaru', 'toyota', 'audi', 'bmw']
```

The `reverse()` method changes the order of a list permanently, but you can revert to the original order anytime by applying `reverse()` to the same list a second time.

Finding the Length of a List

You can quickly find the length of a list by using the `len()` function. The list in this example has four items, so its length is `4`:

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> len(cars)
4
```

You'll find `len()` useful when you need to identify the number of aliens that still need to be shot down in a game, determine the amount of data you have to manage in a visualization, or figure out the number of registered users on a website, among other tasks.

NOTE *Python counts the items in a list starting with one, so you shouldn't run into any off-by-one errors when determining the length of a list.*

TRY IT YOURSELF

3-8. Seeing the World: Think of at least five places in the world you'd like to visit.

- Store the locations in a list. Make sure the list is not in alphabetical order.
- Print your list in its original order. Don't worry about printing the list neatly, just print it as a raw Python list.
- Use `sorted()` to print your list in alphabetical order without modifying the actual list.
- Show that your list is still in its original order by printing it.
- Use `sorted()` to print your list in reverse alphabetical order without changing the order of the original list.
- Show that your list is still in its original order by printing it again.
- Use `reverse()` to change the order of your list. Print the list to show that its order has changed.
- Use `reverse()` to change the order of your list again. Print the list to show it's back to its original order.
- Use `sort()` to change your list so it's stored in alphabetical order. Print the list to show that its order has been changed.
- Use `sort()` to change your list so it's stored in reverse alphabetical order. Print the list to show that its order has changed.

3-9. Dinner Guests: Working with one of the programs from Exercises 3-4 through 3-7 (page 42), use `len()` to print a message indicating the number of people you are inviting to dinner.

3-10. Every Function: Think of something you could store in a list. For example, you could make a list of mountains, rivers, countries, cities, languages, or anything else you'd like. Write a program that creates a list containing these items and then uses each function introduced in this chapter at least once.

Avoiding Index Errors When Working with Lists

One type of error is common to see when you're working with lists for the first time. Let's say you have a list with three items, and you ask for the fourth item:

motorcycles.py

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[3])
```

This example results in an *index error*:

```
Traceback (most recent call last):
  File "motorcycles.py", line 2, in <module>
    print(motorcycles[3])
IndexError: list index out of range
```

Python attempts to give you the item at index 3. But when it searches the list, no item in `motorcycles` has an index of 3. Because of the off-by-one nature of indexing in lists, this error is typical. People think the third item is item number 3, because they start counting at 1. But in Python the third item is number 2, because it starts indexing at 0.

An index error means Python can't find an item at the index you requested. If an index error occurs in your program, try adjusting the index you're asking for by one. Then run the program again to see if the results are correct.

Keep in mind that whenever you want to access the last item in a list you use the index `-1`. This will always work, even if your list has changed size since the last time you accessed it:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[-1])
```

The index `-1` always returns the last item in a list, in this case the value `'suzuki'`:

```
'suzuki'
```

The only time this approach will cause an error is when you request the last item from an empty list:

```
motorcycles = []
print(motorcycles[-1])
```

No items are in `motorcycles`, so Python returns another index error:

```
Traceback (most recent call last):
  File "motorcyles.py", line 3, in <module>
    print(motorcycles[-1])
IndexError: list index out of range
```

NOTE *If an index error occurs and you can't figure out how to resolve it, try printing your list or just printing the length of your list. Your list might look much different than you thought it did, especially if it has been managed dynamically by your program. Seeing the actual list, or the exact number of items in your list, can help you sort out such logical errors.*

TRY IT YOURSELF

3-11. Intentional Error: If you haven't received an index error in one of your programs yet, try to make one happen. Change an index in one of your programs to produce an index error. Make sure you correct the error before closing the program.

Summary

In this chapter you learned what lists are and how to work with the individual items in a list. You learned how to define a list and how to add and remove elements. You learned to sort lists permanently and temporarily for display purposes. You also learned how to find the length of a list and how to avoid index errors when you're working with lists.

In Chapter 4 you'll learn how to work with items in a list more efficiently. By looping through each item in a list using just a few lines of code you'll be able to work efficiently, even when your list contains thousands or millions of items.

4

WORKING WITH LISTS

In Chapter 3 you learned how to make a simple list, and you learned to work with the individual elements in a list. In this chapter you'll learn how to *loop* through an entire list using just a few lines of code regardless of how long the list is. Looping allows you to take the same action, or set of actions, with every item in a list. As a result, you'll be able to work efficiently with lists of any length, including those with thousands or even millions of items.

Looping Through an Entire List

You'll often want to run through all entries in a list, performing the same task with each item. For example, in a game you might want to move every element on the screen by the same amount, or in a list of numbers you might want to perform the same statistical operation on every element. Or perhaps you'll want to display each headline from a list of articles on a website. When you want to do the same action with every item in a list, you can use Python's `for` loop.

Let's say we have a list of magicians' names, and we want to print out each name in the list. We could do this by retrieving each name from the list individually, but this approach could cause several problems. For one, it would be repetitive to do this with a long list of names. Also, we'd have to change our code each time the list's length changed. A `for` loop avoids both of these issues by letting Python manage these issues internally.

Let's use a `for` loop to print out each name in a list of magicians:

magicians.py

```
❶ magicians = ['alice', 'david', 'carolina']
❷ for magician in magicians:
❸     print(magician)
```

We begin by defining a list at ❶, just as we did in Chapter 3. At ❷, we define a `for` loop. This line tells Python to pull a name from the list `magicians`, and associate it with the variable `magician`. At ❸ we tell Python to print the name that's just been assigned to `magician`. Python then repeats lines ❷ and ❸, once for each name in the list. It might help to read this code as "For every magician in the list of magicians, print the magician's name." The output is a simple printout of each name in the list:

```
alice
david
carolina
```

A Closer Look at Looping

The concept of looping is important because it's one of the most common ways a computer automates repetitive tasks. For example, in a simple loop like we used in *magicians.py*, Python initially reads the first line of the loop:

```
for magician in magicians:
```

This line tells Python to retrieve the first value from the list `magicians` and associate it with the variable `magician`. This first value is `'alice'`. Python then reads the next line:

```
    print(magician)
```

Python prints the current value of `magician`, which is still `'alice'`. Because the list contains more values, Python returns to the first line of the loop:

```
for magician in magicians:
```

Python retrieves the next name in the list, `'david'`, and associates that value with the variable `magician`. Python then executes the line:

```
    print(magician)
```

Python prints the current value of `magician` again, which is now `'david'`. Python repeats the entire loop once more with the last value in the list, `'carolina'`. Because no more values are in the list, Python moves on to the next line in the program. In this case nothing comes after the `for` loop, so the program simply ends.

When you're using loops for the first time, keep in mind that the set of steps is repeated once for each item in the list, no matter how many items are in the list. If you have a million items in your list, Python repeats these steps a million times—and usually very quickly.

Also keep in mind when writing your own `for` loops that you can choose any name you want for the temporary variable that will be associated with each value in the list. However, it's helpful to choose a meaningful name that represents a single item from the list. For example, here's a good way to start a `for` loop for a list of cats, a list of dogs, and a general list of items:

```
for cat in cats:
for dog in dogs:
for item in list_of_items:
```

These naming conventions can help you follow the action being done on each item within a `for` loop. Using singular and plural names can help you identify whether a section of code is working with a single element from the list or the entire list.

Doing More Work Within a for Loop

You can do just about anything with each item in a `for` loop. Let's build on the previous example by printing a message to each magician, telling them that they performed a great trick:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
❶    print(f"{magician.title()}, that was a great trick!")
```

The only difference in this code is at ❶ where we compose a message to each magician, starting with that magician's name. The first time through the loop the value of magician is `'alice'`, so Python starts the first message with the name `'Alice'`. The second time through the message will begin with `'David'`, and the third time through the message will begin with `'Carolina'`.

The output shows a personalized message for each magician in the list:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
```

You can also write as many lines of code as you like in the `for` loop. Every indented line following the line `for magician in magicians` is considered *inside the loop*, and each indented line is executed once for each

value in the list. Therefore, you can do as much work as you like with each value in the list.

Let's add a second line to our message, telling each magician that we're looking forward to their next trick:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
❶   print(f"I can't wait to see your next trick, {magician.title()}.\n")
```

Because we have indented both calls to `print()`, each line will be executed once for every magician in the list. The newline (`"\n"`) in the second `print()` call ❶ inserts a blank line after each pass through the loop. This creates a set of messages that are neatly grouped for each person in the list:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

You can use as many lines as you like in your `for` loops. In practice you'll often find it useful to do a number of different operations with each item in a list when you use a `for` loop.

Doing Something After a for Loop

What happens once a `for` loop has finished executing? Usually, you'll want to summarize a block of output or move on to other work that your program must accomplish.

Any lines of code after the `for` loop that are not indented are executed once without repetition. Let's write a thank you to the group of magicians as a whole, thanking them for putting on an excellent show. To display this group message after all of the individual messages have been printed, we place the thank you message after the `for` loop without indentation:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\n")

❶ print("Thank you, everyone. That was a great magic show!")
```

The first two calls to `print()` are repeated once for each magician in the list, as you saw earlier. However, because the line at ❶ is not indented, it's printed only once:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

David, that was a great trick!
I can't wait to see your next trick, David.

Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you, everyone. That was a great magic show!
```

When you're processing data using a `for` loop, you'll find that this is a good way to summarize an operation that was performed on an entire data set. For example, you might use a `for` loop to initialize a game by running through a list of characters and displaying each character on the screen. You might then write some additional code after this loop that displays a *Play Now* button after all the characters have been drawn to the screen.

Avoiding Indentation Errors

Python uses indentation to determine how a line, or group of lines, is related to the rest of the program. In the previous examples, the lines that printed messages to individual magicians were part of the `for` loop because they were indented. Python's use of indentation makes code very easy to read. Basically, it uses whitespace to force you to write neatly formatted code with a clear visual structure. In longer Python programs, you'll notice blocks of code indented at a few different levels. These indentation levels help you gain a general sense of the overall program's organization.

As you begin to write code that relies on proper indentation, you'll need to watch for a few common *indentation errors*. For example, people sometimes indent lines of code that don't need to be indented or forget to indent lines that need to be indented. Seeing examples of these errors now will help you avoid them in the future and correct them when they do appear in your own programs.

Let's examine some of the more common indentation errors.

Forgetting to Indent

Always indent the line after the `for` statement in a loop. If you forget, Python will remind you:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
❶ print(magician)
```

The call to `print()` ❶ should be indented, but it's not. When Python expects an indented block and doesn't find one, it lets you know which line it had a problem with.

```
  File "magicians.py", line 3
    print(magician)
        ^
IndentationError: expected an indented block
```

You can usually resolve this kind of indentation error by indenting the line or lines immediately after the `for` statement.

Forgetting to Indent Additional Lines

Sometimes your loop will run without any errors but won't produce the expected result. This can happen when you're trying to do several tasks in a loop and you forget to indent some of its lines.

For example, this is what happens when we forget to indent the second line in the loop that tells each magician we're looking forward to their next trick:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
❶ print(f"I can't wait to see your next trick, {magician.title()}.\n")
```

The call to `print()` ❶ is supposed to be indented, but because Python finds at least one indented line after the `for` statement, it doesn't report an error. As a result, the first `print()` call is executed once for each name in the list because it is indented. The second `print()` call is not indented, so it is executed only once after the loop has finished running. Because the final value associated with `magician` is `'carolina'`, she is the only one who receives the "looking forward to the next trick" message:

```
Alice, that was a great trick!
David, that was a great trick!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.
```

This is a *logical error*. The syntax is valid Python code, but the code does not produce the desired result because a problem occurs in its logic. If you expect to see a certain action repeated once for each item in a list and it's executed only once, determine whether you need to simply indent a line or a group of lines.

Indenting Unnecessarily

If you accidentally indent a line that doesn't need to be indented, Python informs you about the unexpected indent:

hello_world.py

```
message = "Hello Python world!"
❶    print(message)
```

We don't need to indent the `print()` call ❶, because it isn't part of a loop; hence, Python reports that error:

```
  File "hello_world.py", line 2
    print(message)
    ^
IndentationError: unexpected indent
```

You can avoid unexpected indentation errors by indenting only when you have a specific reason to do so. In the programs you're writing at this point, the only lines you should indent are the actions you want to repeat for each item in a `for` loop.

Indenting Unnecessarily After the Loop

If you accidentally indent code that should run after a loop has finished, that code will be repeated once for each item in the list. Sometimes this prompts Python to report an error, but often this will result in a logical error.

For example, let's see what happens when we accidentally indent the line that thanked the magicians as a group for putting on a good show:

magicians.py

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
    print(f"{magician.title()}, that was a great trick!")
    print(f"I can't wait to see your next trick, {magician.title()}.\n")

❶    print("Thank you everyone, that was a great magic show!")
```

Because the line at ❶ is indented, it's printed once for each person in the list, as shown here:

```
Alice, that was a great trick!
I can't wait to see your next trick, Alice.

Thank you everyone, that was a great magic show!
David, that was a great trick!
I can't wait to see your next trick, David.

Thank you everyone, that was a great magic show!
Carolina, that was a great trick!
I can't wait to see your next trick, Carolina.

Thank you everyone, that was a great magic show!
```

This is another logical error, similar to the one in "Forgetting to Indent Additional Lines" on page 54. Because Python doesn't know what you're trying to accomplish with your code, it will run all code that is written in valid syntax. If an action is repeated many times when it should be executed only once, you probably need to unindent the code for that action.

Forgetting the Colon

The colon at the end of a `for` statement tells Python to interpret the next line as the start of a loop.

```
magicians = ['alice', 'david', 'carolina']
❶ for magician in magicians
    print(magician)
```

If you accidentally forget the colon, as shown at ❶, you'll get a syntax error because Python doesn't know what you're trying to do. Although this is an easy error to fix, it's not always an easy error to find. You'd be surprised by the amount of time programmers spend hunting down single-character errors like this. Such errors are difficult to find because we often just see what we expect to see.

TRY IT YOURSELF

4-1. Pizzas: Think of at least three kinds of your favorite pizza. Store these pizza names in a list, and then use a `for` loop to print the name of each pizza.

- Modify your `for` loop to print a sentence using the name of the pizza instead of printing just the name of the pizza. For each pizza you should have one line of output containing a simple statement like *I like pepperoni pizza.*
- Add a line at the end of your program, outside the `for` loop, that states how much you like pizza. The output should consist of three or more lines about the kinds of pizza you like and then an additional sentence, such as *I really love pizza!*

4-2. Animals: Think of at least three different animals that have a common characteristic. Store the names of these animals in a list, and then use a `for` loop to print out the name of each animal.

- Modify your program to print a statement about each animal, such as *A dog would make a great pet.*
- Add a line at the end of your program stating what these animals have in common. You could print a sentence such as *Any of these animals would make a great pet!*

Making Numerical Lists

Many reasons exist to store a set of numbers. For example, you'll need to keep track of the positions of each character in a game, and you might want to keep track of a player's high scores as well. In data visualizations, you'll almost always work with sets of numbers, such as temperatures, distances, population sizes, or latitude and longitude values, among other types of numerical sets.

Lists are ideal for storing sets of numbers, and Python provides a variety of tools to help you work efficiently with lists of numbers. Once you understand how to use these tools effectively, your code will work well even when your lists contain millions of items.

Using the range() Function

Python's `range()` function makes it easy to generate a series of numbers. For example, you can use the `range()` function to print a series of numbers like this:

first_numbers.py

```
for value in range(1, 5):
    print(value)
```

Although this code looks like it should print the numbers from 1 to 5, it doesn't print the number 5:

```
1
2
3
4
```

In this example, `range()` prints only the numbers 1 through 4. This is another result of the off-by-one behavior you'll see often in programming languages. The `range()` function causes Python to start counting at the first value you give it, and it stops when it reaches the second value you provide. Because it stops at that second value, the output never contains the end value, which would have been 5 in this case.

To print the numbers from 1 to 5, you would use `range(1, 6)`:

```
for value in range(1, 6):
    print(value)
```

This time the output starts at 1 and ends at 5:

```
1
2
3
4
5
```

If your output is different than what you expect when you're using `range()`, try adjusting your end value by 1.

You can also pass `range()` only one argument, and it will start the sequence of numbers at 0. For example, `range(6)` would return the numbers from 0 through 5.

Using range() to Make a List of Numbers

If you want to make a list of numbers, you can convert the results of `range()` directly into a list using the `list()` function. When you wrap `list()` around a call to the `range()` function, the output will be a list of numbers.

In the example in the previous section, we simply printed out a series of numbers. We can use `list()` to convert that same set of numbers into a list:

```
numbers = list(range(1, 6))
print(numbers)
```

And this is the result:

```
[1, 2, 3, 4, 5]
```

We can also use the `range()` function to tell Python to skip numbers in a given range. If you pass a third argument to `range()`, Python uses that value as a step size when generating numbers.

For example, here's how to list the even numbers between 1 and 10:

even_numbers.py

```
even_numbers = list(range(2, 11, 2))
print(even_numbers)
```

In this example, the `range()` function starts with the value 2 and then adds 2 to that value. It adds 2 repeatedly until it reaches or passes the end value, 11, and produces this result:

```
[2, 4, 6, 8, 10]
```

You can create almost any set of numbers you want to using the `range()` function. For example, consider how you might make a list of the first 10 square numbers (that is, the square of each integer from 1 through 10). In Python, two asterisks (**) represent exponents. Here's how you might put the first 10 square numbers into a list:

squares.py

```
❶ squares = []
❷ for value in range(1, 11):
❸     square = value ** 2
❹     squares.append(square)

❺ print(squares)
```

We start with an empty list called `squares` ❶. At ❷, we tell Python to loop through each value from 1 to 10 using the `range()` function. Inside the loop,

the current value is raised to the second power and assigned to the variable `square` ❸. At ❹, each new value of `square` is appended to the list `squares`. Finally, when the loop has finished running, the list of squares is printed ❺:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

To write this code more concisely, omit the temporary variable `square` and append each new value directly to the list:

```
squares = []
for value in range(1,11):
❶    squares.append(value**2)

print(squares)
```

The code at ❶ does the same work as the lines at ❸ and ❹ in *squares.py*. Each value in the loop is raised to the second power and then immediately appended to the list of squares.

You can use either of these two approaches when you're making more complex lists. Sometimes using a temporary variable makes your code easier to read; other times it makes the code unnecessarily long. Focus first on writing code that you understand clearly, which does what you want it to do. Then look for more efficient approaches as you review your code.

Simple Statistics with a List of Numbers

A few Python functions are helpful when working with lists of numbers. For example, you can easily find the minimum, maximum, and sum of a list of numbers:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>> min(digits)
0
>>> max(digits)
9
>>> sum(digits)
45
```

NOTE *The examples in this section use short lists of numbers in order to fit easily on the page. They would work just as well if your list contained a million or more numbers.*

List Comprehensions

The approach described earlier for generating the list `squares` consisted of using three or four lines of code. A *list comprehension* allows you to generate this same list in just one line of code. A list comprehension combines the `for` loop and the creation of new elements into one line, and automatically appends each new element. List comprehensions are not always presented to beginners, but I have included them here because you'll most likely see them as soon as you start looking at other people's code.

The following example builds the same list of square numbers you saw earlier but uses a list comprehension:

squares.py

```
squares = [value**2 for value in range(1, 11)]
print(squares)
```

To use this syntax, begin with a descriptive name for the list, such as `squares`. Next, open a set of square brackets and define the expression for the values you want to store in the new list. In this example the expression is `value**2`, which raises the value to the second power. Then, write a `for` loop to generate the numbers you want to feed into the expression, and close the square brackets. The `for` loop in this example is `for value in range(1, 11)`, which feeds the values 1 through 10 into the expression `value**2`. Notice that no colon is used at the end of the `for` statement.

The result is the same list of square numbers you saw earlier:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

It takes practice to write your own list comprehensions, but you'll find them worthwhile once you become comfortable creating ordinary lists. When you're writing three or four lines of code to generate lists and it begins to feel repetitive, consider writing your own list comprehensions.

TRY IT YOURSELF

4-3. Counting to Twenty: Use a `for` loop to print the numbers from 1 to 20, inclusive.

4-4. One Million: Make a list of the numbers from one to one million, and then use a `for` loop to print the numbers. (If the output is taking too long, stop it by pressing CTRL-C or by closing the output window.)

4-5. Summing a Million: Make a list of the numbers from one to one million, and then use `min()` and `max()` to make sure your list actually starts at one and ends at one million. Also, use the `sum()` function to see how quickly Python can add a million numbers.

4-6. Odd Numbers: Use the third argument of the `range()` function to make a list of the odd numbers from 1 to 20. Use a `for` loop to print each number.

4-7. Threes: Make a list of the multiples of 3 from 3 to 30. Use a `for` loop to print the numbers in your list.

4-8. Cubes: A number raised to the third power is called a *cube*. For example, the cube of 2 is written as `2**3` in Python. Make a list of the first 10 cubes (that is, the cube of each integer from 1 through 10), and use a `for` loop to print out the value of each cube.

4-9. Cube Comprehension: Use a list comprehension to generate a list of the first 10 cubes.

Working with Part of a List

In Chapter 3 you learned how to access single elements in a list, and in this chapter you've been learning how to work through all the elements in a list. You can also work with a specific group of items in a list, which Python calls a *slice*.

Slicing a List

To make a slice, you specify the index of the first and last elements you want to work with. As with the `range()` function, Python stops one item before the second index you specify. To output the first three elements in a list, you would request indices `0` through `3`, which would return elements `0`, `1`, and `2`.

The following example involves a list of players on a team:

players.py

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
❶ print(players[0:3])
```

The code at ❶ prints a slice of this list, which includes just the first three players. The output retains the structure of the list and includes the first three players in the list:

```
['charles', 'martina', 'michael']
```

You can generate any subset of a list. For example, if you want the second, third, and fourth items in a list, you would start the slice at index `1` and end at index `4`:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[1:4])
```

This time the slice starts with `'martina'` and ends with `'florence'`:

```
['martina', 'michael', 'florence']
```

If you omit the first index in a slice, Python automatically starts your slice at the beginning of the list:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[:4])
```

Without a starting index, Python starts at the beginning of the list:

```
['charles', 'martina', 'michael', 'florence']
```

A similar syntax works if you want a slice that includes the end of a list. For example, if you want all items from the third item through the last item, you can start with index 2 and omit the second index:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[2:])
```

Python returns all items from the third item through the end of the list:

```
['michael', 'florence', 'eli']
```

This syntax allows you to output all of the elements from any point in your list to the end regardless of the length of the list. Recall that a negative index returns an element a certain distance from the end of a list; therefore, you can output any slice from the end of a list. For example, if we want to output the last three players on the roster, we can use the slice `players[-3:]`:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']
print(players[-3:])
```

This prints the names of the last three players and would continue to work as the list of players changes in size.

NOTE *You can include a third value in the brackets indicating a slice. If a third value is included, this tells Python how many items to skip between items in the specified range.*

Looping Through a Slice

You can use a slice in a `for` loop if you want to loop through a subset of the elements in a list. In the next example we loop through the first three players and print their names as part of a simple roster:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

print("Here are the first three players on my team:")
❶ for player in players[:3]:
    print(player.title())
```

Instead of looping through the entire list of players at ❶, Python loops through only the first three names:

```
Here are the first three players on my team:
Charles
Martina
Michael
```

Slices are very useful in a number of situations. For instance, when you're creating a game, you could add a player's final score to a list every time that

player finishes playing. You could then get a player's top three scores by sorting the list in decreasing order and taking a slice that includes just the first three scores. When you're working with data, you can use slices to process your data in chunks of a specific size. Or, when you're building a web application, you could use slices to display information in a series of pages with an appropriate amount of information on each page.

Copying a List

Often, you'll want to start with an existing list and make an entirely new list based on the first one. Let's explore how copying a list works and examine one situation in which copying a list is useful.

To copy a list, you can make a slice that includes the entire original list by omitting the first index and the second index (`[:]`). This tells Python to make a slice that starts at the first item and ends with the last item, producing a copy of the entire list.

For example, imagine we have a list of our favorite foods and want to make a separate list of foods that a friend likes. This friend likes everything in our list so far, so we can create their list by copying ours:

foods.py

```
❶ my_foods = ['pizza', 'falafel', 'carrot cake']
❷ friend_foods = my_foods[:]

print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

At ❶ we make a list of the foods we like called `my_foods`. At ❷ we make a new list called `friend_foods`. We make a copy of `my_foods` by asking for a slice of `my_foods` without specifying any indices and store the copy in `friend_foods`. When we print each list, we see that they both contain the same foods:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake']

My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake']
```

To prove that we actually have two separate lists, we'll add a new food to each list and show that each list keeps track of the appropriate person's favorite foods:

```
  my_foods = ['pizza', 'falafel', 'carrot cake']
❶ friend_foods = my_foods[:]

❷ my_foods.append('cannoli')
❸ friend_foods.append('ice cream')
```

```
print("My favorite foods are:")
print(my_foods)

print("\nMy friend's favorite foods are:")
print(friend_foods)
```

At ❶ we copy the original items in `my_foods` to the new list `friend_foods`, as we did in the previous example. Next, we add a new food to each list: at ❷ we add `'cannoli'` to `my_foods`, and at ❸ we add `'ice cream'` to `friend_foods`. We then print the two lists to see whether each of these foods is in the appropriate list.

```
  My favorite foods are:
❹ ['pizza', 'falafel', 'carrot cake', 'cannoli']

  My friend's favorite foods are:
❺ ['pizza', 'falafel', 'carrot cake', 'ice cream']
```

The output at ❹ shows that `'cannoli'` now appears in our list of favorite foods but `'ice cream'` doesn't. At ❺ we can see that `'ice cream'` now appears in our friend's list but `'cannoli'` doesn't. If we had simply set `friend_foods` equal to `my_foods`, we would not produce two separate lists. For example, here's what happens when you try to copy a list without using a slice:

```
  my_foods = ['pizza', 'falafel', 'carrot cake']

  # This doesn't work:
❶ friend_foods = my_foods

  my_foods.append('cannoli')
  friend_foods.append('ice cream')

  print("My favorite foods are:")
  print(my_foods)

  print("\nMy friend's favorite foods are:")
  print(friend_foods)
```

Instead of storing a copy of `my_foods` in `friend_foods` at ❶, we set `friend_foods` equal to `my_foods`. This syntax actually tells Python to associate the new variable `friend_foods` with the list that is already associated with `my_foods`, so now both variables point to the same list. As a result, when we add `'cannoli'` to `my_foods`, it will also appear in `friend_foods`. Likewise `'ice cream'` will appear in both lists, even though it appears to be added only to `friend_foods`.

The output shows that both lists are the same now, which is not what we wanted:

```
My favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

```
My friend's favorite foods are:
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

NOTE *Don't worry about the details in this example for now. Basically, if you're trying to work with a copy of a list and you see unexpected behavior, make sure you are copying the list using a slice, as we did in the first example.*

TRY IT YOURSELF

4-10. Slices: Using one of the programs you wrote in this chapter, add several lines to the end of the program that do the following:

- Print the message *The first three items in the list are:*. Then use a slice to print the first three items from that program's list.
- Print the message *Three items from the middle of the list are:*. Use a slice to print three items from the middle of the list.
- Print the message *The last three items in the list are:*. Use a slice to print the last three items in the list.

4-11. My Pizzas, Your Pizzas: Start with your program from Exercise 4-1 (page 56). Make a copy of the list of pizzas, and call it `friend_pizzas`. Then, do the following:

- Add a new pizza to the original list.
- Add a different pizza to the list `friend_pizzas`.
- Prove that you have two separate lists. Print the message *My favorite pizzas are:*, and then use a `for` loop to print the first list. Print the message *My friend's favorite pizzas are:*, and then use a `for` loop to print the second list. Make sure each new pizza is stored in the appropriate list.

4-12. More Loops: All versions of *foods.py* in this section have avoided using `for` loops when printing to save space. Choose a version of *foods.py*, and write two `for` loops to print each list of foods.

Tuples

Lists work well for storing collections of items that can change throughout the life of a program. The ability to modify lists is particularly important when you're working with a list of users on a website or a list of characters in a game. However, sometimes you'll want to create a list of items that cannot change. Tuples allow you to do just that. Python refers to values that cannot change as *immutable*, and an immutable list is called a *tuple*.

Defining a Tuple

A tuple looks just like a list except you use parentheses instead of square brackets. Once you define a tuple, you can access individual elements by using each item's index, just as you would for a list.

For example, if we have a rectangle that should always be a certain size, we can ensure that its size doesn't change by putting the dimensions into a tuple:

dimensions.py

```
❶ dimensions = (200, 50)
❷ print(dimensions[0])
  print(dimensions[1])
```

We define the tuple `dimensions` at ❶, using parentheses instead of square brackets. At ❷ we print each element in the tuple individually, using the same syntax we've been using to access elements in a list:

```
200
50
```

Let's see what happens if we try to change one of the items in the tuple `dimensions`:

```
  dimensions = (200, 50)
❶ dimensions[0] = 250
```

The code at ❶ tries to change the value of the first dimension, but Python returns a type error. Basically, because we're trying to alter a tuple, which can't be done to that type of object, Python tells us we can't assign a new value to an item in a tuple:

```
Traceback (most recent call last):
  File "dimensions.py", line 2, in <module>
    dimensions[0] = 250
TypeError: 'tuple' object does not support item assignment
```

This is beneficial because we want Python to raise an error when a line of code tries to change the dimensions of the rectangle.

NOTE *Tuples are technically defined by the presence of a comma; the parentheses make them look neater and more readable. If you want to define a tuple with one element, you need to include a trailing comma:*

```
my_t = (3,)
```

It doesn't often make sense to build a tuple with one element, but this can happen when tuples are generated automatically.

Looping Through All Values in a Tuple

You can loop over all the values in a tuple using a `for` loop, just as you did with a list:

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)
```

Python returns all the elements in the tuple, just as it would for a list:

```
200
50
```

Writing over a Tuple

Although you can't modify a tuple, you can assign a new value to a variable that represents a tuple. So if we wanted to change our dimensions, we could redefine the entire tuple:

```
❶ dimensions = (200, 50)
  print("Original dimensions:")
  for dimension in dimensions:
      print(dimension)

❷ dimensions = (400, 100)
❸ print("\nModified dimensions:")
  for dimension in dimensions:
      print(dimension)
```

The lines starting at ❶ define the original tuple and print the initial dimensions. At ❷, we associate a new tuple with the variable `dimensions`. We then print the new dimensions at ❸. Python doesn't raise any errors this time, because reassigning a variable is valid:

```
Original dimensions:
200
50

Modified dimensions:
400
100
```

When compared with lists, tuples are simple data structures. Use them when you want to store a set of values that should not be changed throughout the life of a program.

TRY IT YOURSELF

4-13. Buffet: A buffet-style restaurant offers only five basic foods. Think of five simple foods, and store them in a tuple.

- Use a `for` loop to print each food the restaurant offers.
- Try to modify one of the items, and make sure that Python rejects the change.
- The restaurant changes its menu, replacing two of the items with different foods. Add a line that rewrites the tuple, and then use a `for` loop to print each of the items on the revised menu.

Styling Your Code

Now that you're writing longer programs, ideas about how to style your code are worthwhile to know. Take the time to make your code as easy as possible to read. Writing easy-to-read code helps you keep track of what your programs are doing and helps others understand your code as well.

Python programmers have agreed on a number of styling conventions to ensure that everyone's code is structured in roughly the same way. Once you've learned to write clean Python code, you should be able to understand the overall structure of anyone else's Python code, as long as they follow the same guidelines. If you're hoping to become a professional programmer at some point, you should begin following these guidelines as soon as possible to develop good habits.

The Style Guide

When someone wants to make a change to the Python language, they write a *Python Enhancement Proposal (PEP)*. One of the oldest PEPs is *PEP 8*, which instructs Python programmers on how to style their code. PEP 8 is fairly lengthy, but much of it relates to more complex coding structures than what you've seen so far.

The Python style guide was written with the understanding that code is read more often than it is written. You'll write your code once and then start reading it as you begin debugging. When you add features to a program, you'll spend more time reading your code. When you share your code with other programmers, they'll read your code as well.

Given the choice between writing code that's easier to write or code that's easier to read, Python programmers will almost always encourage you to write code that's easier to read. The following guidelines will help you write clear code from the start.

Indentation

PEP 8 recommends that you use four spaces per indentation level. Using four spaces improves readability while leaving room for multiple levels of indentation on each line.

In a word processing document, people often use tabs rather than spaces to indent. This works well for word processing documents, but the Python interpreter gets confused when tabs are mixed with spaces. Every text editor provides a setting that lets you use the TAB key but then converts each tab to a set number of spaces. You should definitely use your TAB key, but also make sure your editor is set to insert spaces rather than tabs into your document.

Mixing tabs and spaces in your file can cause problems that are very difficult to diagnose. If you think you have a mix of tabs and spaces, you can convert all tabs in a file to spaces in most editors.

Line Length

Many Python programmers recommend that each line should be less than 80 characters. Historically, this guideline developed because most computers could fit only 79 characters on a single line in a terminal window. Currently, people can fit much longer lines on their screens, but other reasons exist to adhere to the 79-character standard line length. Professional programmers often have several files open on the same screen, and using the standard line length allows them to see entire lines in two or three files that are open side by side onscreen. PEP 8 also recommends that you limit all of your comments to 72 characters per line, because some of the tools that generate automatic documentation for larger projects add formatting characters at the beginning of each commented line.

The PEP 8 guidelines for line length are not set in stone, and some teams prefer a 99-character limit. Don't worry too much about line length in your code as you're learning, but be aware that people who are working collaboratively almost always follow the PEP 8 guidelines. Most editors allow you to set up a visual cue, usually a vertical line on your screen, that shows you where these limits are.

NOTE *Appendix B shows you how to configure your text editor so it always inserts four spaces each time you press the TAB key and shows a vertical guideline to help you follow the 79-character limit.*

Blank Lines

To group parts of your program visually, use blank lines. You should use blank lines to organize your files, but don't do so excessively. By following the examples provided in this book, you should strike the right balance. For example, if you have five lines of code that build a list, and then another three lines that do something with that list, it's appropriate to place a blank line between the two sections. However, you should not place three or four blank lines between the two sections.

Blank lines won't affect how your code runs, but they will affect the readability of your code. The Python interpreter uses horizontal indentation to interpret the meaning of your code, but it disregards vertical spacing.

Other Style Guidelines

PEP 8 has many additional styling recommendations, but most of the guidelines refer to more complex programs than what you're writing at this point. As you learn more complex Python structures, I'll share the relevant parts of the PEP 8 guidelines.

TRY IT YOURSELF

4-14. PEP 8: Look through the original PEP 8 style guide at *https://python.org/dev/peps/pep-0008/*. You won't use much of it now, but it might be interesting to skim through it.

4-15. Code Review: Choose three of the programs you've written in this chapter and modify each one to comply with PEP 8:

- Use four spaces for each indentation level. Set your text editor to insert four spaces every time you press TAB, if you haven't already done so (see Appendix B for instructions on how to do this).
- Use less than 80 characters on each line, and set your editor to show a vertical guideline at the 80th character position.
- Don't use blank lines excessively in your program files.

Summary

In this chapter you learned how to work efficiently with the elements in a list. You learned how to work through a list using a `for` loop, how Python uses indentation to structure a program, and how to avoid some common indentation errors. You learned to make simple numerical lists, as well as a few operations you can perform on numerical lists. You learned how to slice a list to work with a subset of items and how to copy lists properly using a slice. You also learned about tuples, which provide a degree of protection to a set of values that shouldn't change, and how to style your increasingly complex code to make it easy to read.

In Chapter 5, you'll learn to respond appropriately to different conditions by using `if` statements. You'll learn to string together relatively complex sets of conditional tests to respond appropriately to exactly the kind of situation or information you're looking for. You'll also learn to use `if` statements while looping through a list to take specific actions with selected elements from a list.

5

IF STATEMENTS

Programming often involves examining a set of conditions and deciding which action to take based on those conditions. Python's `if` statement allows you to examine the current state of a program and respond appropriately to that state.

In this chapter you'll learn to write conditional tests, which allow you to check any condition of interest. You'll learn to write simple `if` statements, and you'll learn how to create a more complex series of `if` statements to identify when the exact conditions you want are present. You'll then apply this concept to lists, so you'll be able to write a `for` loop that handles most items in a list one way but handles certain items with specific values in a different way.

A Simple Example

The following short example shows how `if` tests let you respond to special situations correctly. Imagine you have a list of cars and you want to print out the name of each car. Car names are proper names, so the names of most cars should be printed in title case. However, the value `'bmw'` should be printed in all uppercase. The following code loops through a list of car names and looks for the value `'bmw'`. Whenever the value is `'bmw'`, it's printed in uppercase instead of title case:

cars.py

```
cars = ['audi', 'bmw', 'subaru', 'toyota']

for car in cars:
❶    if car == 'bmw':
        print(car.upper())
    else:
        print(car.title())
```

The loop in this example first checks if the current value of `car` is `'bmw'` ❶. If it is, the value is printed in uppercase. If the value of `car` is anything other than `'bmw'`, it's printed in title case:

```
Audi
BMW
Subaru
Toyota
```

This example combines a number of the concepts you'll learn about in this chapter. Let's begin by looking at the kinds of tests you can use to examine the conditions in your program.

Conditional Tests

At the heart of every `if` statement is an expression that can be evaluated as `True` or `False` and is called a *conditional test*. Python uses the values `True` and `False` to decide whether the code in an `if` statement should be executed. If a conditional test evaluates to `True`, Python executes the code following the `if` statement. If the test evaluates to `False`, Python ignores the code following the `if` statement.

Checking for Equality

Most conditional tests compare the current value of a variable to a specific value of interest. The simplest conditional test checks whether the value of a variable is equal to the value of interest:

```
❶ >>> car = 'bmw'
❷ >>> car == 'bmw'
True
```

The line at ❶ sets the value of car to 'bmw' using a single equal sign, as you've seen many times already. The line at ❷ checks whether the value of car is 'bmw' using a double equal sign (==). This *equality operator* returns True if the values on the left and right side of the operator match, and False if they don't match. The values in this example match, so Python returns True.

When the value of car is anything other than 'bmw', this test returns False:

```
❶ >>> car = 'audi'
❷ >>> car == 'bmw'
False
```

A single equal sign is really a statement; you might read the code at ❶ as "Set the value of car equal to 'audi'." On the other hand, a double equal sign, like the one at ❷, asks a question: "Is the value of car equal to 'bmw'?" Most programming languages use equal signs in this way.

Ignoring Case When Checking for Equality

Testing for equality is case sensitive in Python. For example, two values with different capitalization are not considered equal:

```
>>> car = 'Audi'
>>> car == 'audi'
False
```

If case matters, this behavior is advantageous. But if case doesn't matter and instead you just want to test the value of a variable, you can convert the variable's value to lowercase before doing the comparison:

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

This test would return True no matter how the value 'Audi' is formatted because the test is now case insensitive. The lower() function doesn't change the value that was originally stored in car, so you can do this kind of comparison without affecting the original variable:

```
❶ >>> car = 'Audi'
❷ >>> car.lower() == 'audi'
True
❸ >>> car
'Audi'
```

At ❶ we assign the capitalized string 'Audi' to the variable car. At ❷ we convert the value of car to lowercase and compare the lowercase value

to the string 'audi'. The two strings match, so Python returns True. At ❸ we can see that the value stored in car has not been affected by the lower() method.

Websites enforce certain rules for the data that users enter in a manner similar to this. For example, a site might use a conditional test like this to ensure that every user has a truly unique username, not just a variation on the capitalization of another person's username. When someone submits a new username, that new username is converted to lowercase and compared to the lowercase versions of all existing usernames. During this check, a username like 'John' will be rejected if any variation of 'john' is already in use.

Checking for Inequality

When you want to determine whether two values are not equal, you can combine an exclamation point and an equal sign (!=). The exclamation point represents *not*, as it does in many programming languages.

Let's use another if statement to examine how to use the inequality operator. We'll store a requested pizza topping in a variable and then print a message if the person did not order anchovies:

toppings.py

```
requested_topping = 'mushrooms'

❶ if requested_topping != 'anchovies':
    print("Hold the anchovies!")
```

The line at ❶ compares the value of requested_topping to the value 'anchovies'. If these two values do not match, Python returns True and executes the code following the if statement. If the two values match, Python returns False and does not run the code following the if statement.

Because the value of requested_topping is not 'anchovies', the print() function is executed:

```
Hold the anchovies!
```

Most of the conditional expressions you write will test for equality, but sometimes you'll find it more efficient to test for inequality.

Numerical Comparisons

Testing numerical values is pretty straightforward. For example, the following code checks whether a person is 18 years old:

```
>>> age = 18
>>> age == 18
True
```

You can also test to see if two numbers are not equal. For example, the following code prints a message if the given answer is not correct:

magic_number.py

```
answer = 17

❶ if answer != 42:
    print("That is not the correct answer. Please try again!")
```

The conditional test at ❶ passes, because the value of `answer` (`17`) is not equal to `42`. Because the test passes, the indented code block is executed:

```
That is not the correct answer. Please try again!
```

You can include various mathematical comparisons in your conditional statements as well, such as less than, less than or equal to, greater than, and greater than or equal to:

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Each mathematical comparison can be used as part of an `if` statement, which can help you detect the exact conditions of interest.

Checking Multiple Conditions

You may want to check multiple conditions at the same time. For example, sometimes you might need two conditions to be `True` to take an action. Other times you might be satisfied with just one condition being `True`. The keywords `and` and `or` can help you in these situations.

Using and to Check Multiple Conditions

To check whether two conditions are both `True` simultaneously, use the keyword `and` to combine the two conditional tests; if each test passes, the overall expression evaluates to `True`. If either test fails or if both tests fail, the expression evaluates to `False`.

For example, you can check whether two people are both over 21 using the following test:

```
❶ >>> age_0 = 22
  >>> age_1 = 18
❷ >>> age_0 >= 21 and age_1 >= 21
  False
```

```
❸ >>> age_1 = 22
  >>> age_0 >= 21 and age_1 >= 21
  True
```

At ❶ we define two ages, `age_0` and `age_1`. At ❷ we check whether both ages are 21 or older. The test on the left passes, but the test on the right fails, so the overall conditional expression evaluates to `False`. At ❸ we change `age_1` to 22. The value of `age_1` is now greater than 21, so both individual tests pass, causing the overall conditional expression to evaluate as `True`.

To improve readability, you can use parentheses around the individual tests, but they are not required. If you use parentheses, your test would look like this:

```
(age_0 >= 21) and (age_1 >= 21)
```

Using or to Check Multiple Conditions

The keyword `or` allows you to check multiple conditions as well, but it passes when either or both of the individual tests pass. An `or` expression fails only when both individual tests fail.

Let's consider two ages again, but this time we'll look for only one person to be over 21:

```
❶ >>> age_0 = 22
  >>> age_1 = 18
❷ >>> age_0 >= 21 or age_1 >= 21
  True
❸ >>> age_0 = 18
  >>> age_0 >= 21 or age_1 >= 21
  False
```

We start with two age variables again at ❶. Because the test for `age_0` at ❷ passes, the overall expression evaluates to `True`. We then lower `age_0` to 18. In the test at ❸, both tests now fail and the overall expression evaluates to `False`.

Checking Whether a Value Is in a List

Sometimes it's important to check whether a list contains a certain value before taking an action. For example, you might want to check whether a new username already exists in a list of current usernames before completing someone's registration on a website. In a mapping project, you might want to check whether a submitted location already exists in a list of known locations.

To find out whether a particular value is already in a list, use the keyword `in`. Let's consider some code you might write for a pizzeria. We'll make a list of toppings a customer has requested for a pizza and then check whether certain toppings are in the list.

```
>>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
❶ >>> 'mushrooms' in requested_toppings
True
❷ >>> 'pepperoni' in requested_toppings
False
```

At ❶ and ❷, the keyword `in` tells Python to check for the existence of `'mushrooms'` and `'pepperoni'` in the list `requested_toppings`. This technique is quite powerful because you can create a list of essential values, and then easily check whether the value you're testing matches one of the values in the list.

Checking Whether a Value Is Not in a List

Other times, it's important to know if a value does not appear in a list. You can use the keyword `not` in this situation. For example, consider a list of users who are banned from commenting in a forum. You can check whether a user has been banned before allowing that person to submit a comment:

banned_users.py

```
banned_users = ['andrew', 'carolina', 'david']
user = 'marie'

❶ if user not in banned_users:
    print(f"{user.title()}, you can post a response if you wish.")
```

The line at ❶ reads quite clearly. If the value of `user` is not in the list `banned_users`, Python returns `True` and executes the indented line.

The user `'marie'` is not in the list `banned_users`, so she sees a message inviting her to post a response:

```
Marie, you can post a response if you wish.
```

Boolean Expressions

As you learn more about programming, you'll hear the term *Boolean expression* at some point. A Boolean expression is just another name for a conditional test. A *Boolean value* is either `True` or `False`, just like the value of a conditional expression after it has been evaluated.

Boolean values are often used to keep track of certain conditions, such as whether a game is running or whether a user can edit certain content on a website:

```
game_active = True
can_edit = False
```

Boolean values provide an efficient way to track the state of a program or a particular condition that is important in your program.

TRY IT YOURSELF

5-1. Conditional Tests: Write a series of conditional tests. Print a statement describing each test and your prediction for the results of each test. Your code should look something like this:

```
car = 'subaru'
print("Is car == 'subaru'? I predict True.")
print(car == 'subaru')

print("\nIs car == 'audi'? I predict False.")
print(car == 'audi')
```

- Look closely at your results, and make sure you understand why each line evaluates to `True` or `False`.
- Create at least ten tests. Have at least five tests evaluate to `True` and another five tests evaluate to `False`.

5-2. More Conditional Tests: You don't have to limit the number of tests you create to ten. If you want to try more comparisons, write more tests and add them to *conditional_tests.py*. Have at least one `True` and one `False` result for each of the following:

- Tests for equality and inequality with strings
- Tests using the `lower()` method
- Numerical tests involving equality and inequality, greater than and less than, greater than or equal to, and less than or equal to
- Tests using the `and` keyword and the `or` keyword
- Test whether an item is in a list
- Test whether an item is not in a list

if Statements

When you understand conditional tests, you can start writing `if` statements. Several different kinds of `if` statements exist, and your choice of which to use depends on the number of conditions you need to test. You saw several examples of `if` statements in the discussion about conditional tests, but now let's dig deeper into the topic.

Simple if Statements

The simplest kind of `if` statement has one test and one action:

```
if conditional_test:
    do something
```

You can put any conditional test in the first line and just about any action in the indented block following the test. If the conditional test evaluates to `True`, Python executes the code following the `if` statement. If the test evaluates to `False`, Python ignores the code following the `if` statement.

Let's say we have a variable representing a person's age, and we want to know if that person is old enough to vote. The following code tests whether the person can vote:

voting.py

```
age = 19
❶ if age >= 18:
❷     print("You are old enough to vote!")
```

At ❶ Python checks to see whether the value of `age` is greater than or equal to 18. It is, so Python executes the indented `print()` call at ❷:

```
You are old enough to vote!
```

Indentation plays the same role in `if` statements as it did in `for` loops. All indented lines after an `if` statement will be executed if the test passes, and the entire block of indented lines will be ignored if the test does not pass.

You can have as many lines of code as you want in the block following the `if` statement. Let's add another line of output if the person is old enough to vote, asking if the individual has registered to vote yet:

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
```

The conditional test passes, and both `print()` calls are indented, so both lines are printed:

```
You are old enough to vote!
Have you registered to vote yet?
```

If the value of `age` is less than 18, this program would produce no output.

if-else Statements

Often, you'll want to take one action when a conditional test passes and a different action in all other cases. Python's `if-else` syntax makes this possible. An `if-else` block is similar to a simple `if` statement, but the `else` statement allows you to define an action or set of actions that are executed when the conditional test fails.

We'll display the same message we had previously if the person is old enough to vote, but this time we'll add a message for anyone who is not old enough to vote:

```
age = 17
❶ if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
❷ else:
    print("Sorry, you are too young to vote.")
    print("Please register to vote as soon as you turn 18!")
```

If the conditional test at ❶ passes, the first block of indented `print()` calls is executed. If the test evaluates to `False`, the `else` block at ❷ is executed. Because `age` is less than 18 this time, the conditional test fails and the code in the `else` block is executed:

```
Sorry, you are too young to vote.
Please register to vote as soon as you turn 18!
```

This code works because it has only two possible situations to evaluate: a person is either old enough to vote or not old enough to vote. The `if-else` structure works well in situations in which you want Python to always execute one of two possible actions. In a simple `if-else` chain like this, one of the two actions will always be executed.

The if-elif-else Chain

Often, you'll need to test more than two possible situations, and to evaluate these you can use Python's `if-elif-else` syntax. Python executes only one block in an `if-elif-else` chain. It runs each conditional test in order until one passes. When a test passes, the code following that test is executed and Python skips the rest of the tests.

Many real-world situations involve more than two possible conditions. For example, consider an amusement park that charges different rates for different age groups:

- Admission for anyone under age 4 is free.
- Admission for anyone between the ages of 4 and 18 is $25.
- Admission for anyone age 18 or older is $40.

How can we use an `if` statement to determine a person's admission rate? The following code tests for the age group of a person and then prints an admission price message:

amusement_park.py

```
age = 12

❶ if age < 4:
    print("Your admission cost is $0.")
```

```
❷ elif age < 18:
      print("Your admission cost is $25.")
❸ else:
      print("Your admission cost is $40.")
```

The `if` test at ❶ tests whether a person is under 4 years old. If the test passes, an appropriate message is printed and Python skips the rest of the tests. The `elif` line at ❷ is really another `if` test, which runs only if the previous test failed. At this point in the chain, we know the person is at least 4 years old because the first test failed. If the person is under 18, an appropriate message is printed and Python skips the `else` block. If both the `if` and `elif` tests fail, Python runs the code in the `else` block at ❸.

In this example the test at ❶ evaluates to `False`, so its code block is not executed. However, the second test evaluates to `True` (12 is less than 18) so its code is executed. The output is one sentence, informing the user of the admission cost:

```
Your admission cost is $25.
```

Any age greater than 17 would cause the first two tests to fail. In these situations, the `else` block would be executed and the admission price would be $40.

Rather than printing the admission price within the `if-elif-else` block, it would be more concise to set just the price inside the `if-elif-else` chain and then have a simple `print()` call that runs after the chain has been evaluated:

```
age = 12

if age < 4:
❶     price = 0
elif age < 18:
❷     price = 25
else:
❸     price = 40

❹ print(f"Your admission cost is ${price}.")
```

The lines at ❶, ❷, and ❸ set the value of `price` according to the person's age, as in the previous example. After the price is set by the `if-elif-else` chain, a separate unindented `print()` call ❹ uses this value to display a message reporting the person's admission price.

This code produces the same output as the previous example, but the purpose of the `if-elif-else` chain is narrower. Instead of determining a price and displaying a message, it simply determines the admission price. In addition to being more efficient, this revised code is easier to modify than the original approach. To change the text of the output message, you would need to change only one `print()` call rather than three separate `print()` calls.

Using Multiple elif Blocks

You can use as many `elif` blocks in your code as you like. For example, if the amusement park were to implement a discount for seniors, you could add one more conditional test to the code to determine whether someone qualified for the senior discount. Let's say that anyone 65 or older pays half the regular admission, or $20:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
❶ elif age < 65:
    price = 40
❷ else:
    price = 20

print(f"Your admission cost is ${price}.")
```

Most of this code is unchanged. The second `elif` block at ❶ now checks to make sure a person is less than age 65 before assigning them the full admission rate of $40. Notice that the value assigned in the `else` block at ❷ needs to be changed to $20, because the only ages that make it to this block are people 65 or older.

Omitting the else Block

Python does not require an `else` block at the end of an `if-elif` chain. Sometimes an `else` block is useful; sometimes it is clearer to use an additional `elif` statement that catches the specific condition of interest:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
❶ elif age >= 65:
    price = 20

print(f"Your admission cost is ${price}.")
```

The extra `elif` block at ❶ assigns a price of $20 when the person is 65 or older, which is a bit clearer than the general `else` block. With this change, every block of code must pass a specific test in order to be executed.

The `else` block is a catchall statement. It matches any condition that wasn't matched by a specific `if` or `elif` test, and that can sometimes include invalid or even malicious data. If you have a specific final condition you are testing for, consider using a final `elif` block and omit the `else` block. As a result, you'll gain extra confidence that your code will run only under the correct conditions.

Testing Multiple Conditions

The `if-elif-else` chain is powerful, but it's only appropriate to use when you just need one test to pass. As soon as Python finds one test that passes, it skips the rest of the tests. This behavior is beneficial, because it's efficient and allows you to test for one specific condition.

However, sometimes it's important to check all of the conditions of interest. In this case, you should use a series of simple `if` statements with no `elif` or `else` blocks. This technique makes sense when more than one condition could be `True`, and you want to act on every condition that is `True`.

Let's reconsider the pizzeria example. If someone requests a two-topping pizza, you'll need to be sure to include both toppings on their pizza:

toppings.py

```
❶ requested_toppings = ['mushrooms', 'extra cheese']

❷ if 'mushrooms' in requested_toppings:
      print("Adding mushrooms.")
❸ if 'pepperoni' in requested_toppings:
      print("Adding pepperoni.")
❹ if 'extra cheese' in requested_toppings:
      print("Adding extra cheese.")

  print("\nFinished making your pizza!")
```

We start at ❶ with a list containing the requested toppings. The `if` statement at ❷ checks to see whether the person requested mushrooms on their pizza. If so, a message is printed confirming that topping. The test for pepperoni at ❸ is another simple `if` statement, not an `elif` or `else` statement, so this test is run regardless of whether the previous test passed or not. The code at ❹ checks whether extra cheese was requested regardless of the results from the first two tests. These three independent tests are executed every time this program is run.

Because every condition in this example is evaluated, both mushrooms and extra cheese are added to the pizza:

```
Adding mushrooms.
Adding extra cheese.

Finished making your pizza!
```

This code would not work properly if we used an `if-elif-else` block, because the code would stop running after only one test passes. Here's what that would look like:

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
elif 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
elif 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

The test for `'mushrooms'` is the first test to pass, so mushrooms are added to the pizza. However, the values `'extra cheese'` and `'pepperoni'` are never checked, because Python doesn't run any tests beyond the first test that passes in an `if-elif-else` chain. The customer's first topping will be added, but all of their other toppings will be missed:

```
Adding mushrooms.

Finished making your pizza!
```

In summary, if you want only one block of code to run, use an `if-elif-else` chain. If more than one block of code needs to run, use a series of independent `if` statements.

TRY IT YOURSELF

5-3. Alien Colors #1: Imagine an alien was just shot down in a game. Create a variable called `alien_color` and assign it a value of `'green'`, `'yellow'`, or `'red'`.

- Write an `if` statement to test whether the alien's color is green. If it is, print a message that the player just earned 5 points.
- Write one version of this program that passes the `if` test and another that fails. (The version that fails will have no output.)

5-4. Alien Colors #2: Choose a color for an alien as you did in Exercise 5-3, and write an `if-else` chain.

- If the alien's color is green, print a statement that the player just earned 5 points for shooting the alien.
- If the alien's color isn't green, print a statement that the player just earned 10 points.
- Write one version of this program that runs the `if` block and another that runs the `else` block.

5-5. Alien Colors #3: Turn your `if-else` chain from Exercise 5-4 into an `if-elif-else` chain.

- If the alien is green, print a message that the player earned 5 points.
- If the alien is yellow, print a message that the player earned 10 points.
- If the alien is red, print a message that the player earned 15 points.
- Write three versions of this program, making sure each message is printed for the appropriate color alien.

5-6. Stages of Life: Write an `if-elif-else` chain that determines a person's stage of life. Set a value for the variable `age`, and then:

- If the person is less than 2 years old, print a message that the person is a baby.
- If the person is at least 2 years old but less than 4, print a message that the person is a toddler.
- If the person is at least 4 years old but less than 13, print a message that the person is a kid.
- If the person is at least 13 years old but less than 20, print a message that the person is a teenager.
- If the person is at least 20 years old but less than 65, print a message that the person is an adult.
- If the person is age 65 or older, print a message that the person is an elder.

5-7. Favorite Fruit: Make a list of your favorite fruits, and then write a series of independent `if` statements that check for certain fruits in your list.

- Make a list of your three favorite fruits and call it `favorite_fruits`.
- Write five `if` statements. Each should check whether a certain kind of fruit is in your list. If the fruit is in your list, the `if` block should print a statement, such as *You really like bananas!*

Using if Statements with Lists

You can do some interesting work when you combine lists and `if` statements. You can watch for special values that need to be treated differently than other values in the list. You can manage changing conditions efficiently, such as the availability of certain items in a restaurant throughout a shift. You can also begin to prove that your code works as you expect it to in all possible situations.

Checking for Special Items

This chapter began with a simple example that showed how to handle a special value like `'bmw'`, which needed to be printed in a different format than other values in the list. Now that you have a basic understanding of conditional tests and `if` statements, let's take a closer look at how you can watch for special values in a list and handle those values appropriately.

Let's continue with the pizzeria example. The pizzeria displays a message whenever a topping is added to your pizza, as it's being made. The code for this action can be written very efficiently by making a list of toppings the customer has requested and using a loop to announce each topping as it's added to the pizza:

toppings.py

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")
```

The output is straightforward because this code is just a simple `for` loop:

```
Adding mushrooms.
Adding green peppers.
Adding extra cheese.

Finished making your pizza!
```

But what if the pizzeria runs out of green peppers? An `if` statement inside the `for` loop can handle this situation appropriately:

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
❶   if requested_topping == 'green peppers':
        print("Sorry, we are out of green peppers right now.")
❷   else:
        print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")
```

This time we check each requested item before adding it to the pizza. The code at ❶ checks to see if the person requested green peppers. If so, we display a message informing them why they can't have green peppers. The `else` block at ❷ ensures that all other toppings will be added to the pizza.

The output shows that each requested topping is handled appropriately.

```
Adding mushrooms.
Sorry, we are out of green peppers right now.
Adding extra cheese.

Finished making your pizza!
```

Checking That a List Is Not Empty

We've made a simple assumption about every list we've worked with so far; we've assumed that each list has at least one item in it. Soon we'll let users provide the information that's stored in a list, so we won't be able to assume that a list has any items in it each time a loop is run. In this situation, it's useful to check whether a list is empty before running a `for` loop.

As an example, let's check whether the list of requested toppings is empty before building the pizza. If the list is empty, we'll prompt the user and make sure they want a plain pizza. If the list is not empty, we'll build the pizza just as we did in the previous examples:

```
❶ requested_toppings = []

❷ if requested_toppings:
      for requested_topping in requested_toppings:
          print(f"Adding {requested_topping}.")
      print("\nFinished making your pizza!")
❸ else:
      print("Are you sure you want a plain pizza?")
```

This time we start out with an empty list of requested toppings at ❶. Instead of jumping right into a `for` loop, we do a quick check at ❷. When the name of a list is used in an `if` statement, Python returns `True` if the list contains at least one item; an empty list evaluates to `False`. If `requested_toppings` passes the conditional test, we run the same `for` loop we used in the previous example. If the conditional test fails, we print a message asking the customer if they really want a plain pizza with no toppings ❸.

The list is empty in this case, so the output asks if the user really wants a plain pizza:

```
Are you sure you want a plain pizza?
```

If the list is not empty, the output will show each requested topping being added to the pizza.

Using Multiple Lists

People will ask for just about anything, especially when it comes to pizza toppings. What if a customer actually wants french fries on their pizza? You can use lists and `if` statements to make sure your input makes sense before you act on it.

Let's watch out for unusual topping requests before we build a pizza. The following example defines two lists. The first is a list of available toppings at the pizzeria, and the second is the list of toppings that the user has requested. This time, each item in `requested_toppings` is checked against the list of available toppings before it's added to the pizza:

```
❶ available_toppings = ['mushrooms', 'olives', 'green peppers',
                        'pepperoni', 'pineapple', 'extra cheese']

❷ requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

❸ for requested_topping in requested_toppings:
❹     if requested_topping in available_toppings:
          print(f"Adding {requested_topping}.")
❺     else:
          print(f"Sorry, we don't have {requested_topping}.")

  print("\nFinished making your pizza!")
```

At ❶ we define a list of available toppings at this pizzeria. Note that this could be a tuple if the pizzeria has a stable selection of toppings. At ❷, we make a list of toppings that a customer has requested. Note the unusual request, `'french fries'`. At ❸ we loop through the list of requested toppings. Inside the loop, we first check to see if each requested topping is actually in the list of available toppings ❹. If it is, we add that topping to the pizza. If the requested topping is not in the list of available toppings, the `else` block will run ❺. The `else` block prints a message telling the user which toppings are unavailable.

This code syntax produces clean, informative output:

```
Adding mushrooms.
Sorry, we don't have french fries.
Adding extra cheese.

Finished making your pizza!
```

In just a few lines of code, we've managed a real-world situation pretty effectively!

TRY IT YOURSELF

5-8. Hello Admin: Make a list of five or more usernames, including the name `'admin'`. Imagine you are writing code that will print a greeting to each user after they log in to a website. Loop through the list, and print a greeting to each user:

- If the username is `'admin'`, print a special greeting, such as *Hello admin, would you like to see a status report?*
- Otherwise, print a generic greeting, such as *Hello Jaden, thank you for logging in again.*

5-9. No Users: Add an `if` test to *hello_admin.py* to make sure the list of users is not empty.

- If the list is empty, print the message *We need to find some users!*
- Remove all of the usernames from your list, and make sure the correct message is printed.

5-10. Checking Usernames: Do the following to create a program that simulates how websites ensure that everyone has a unique username.

- Make a list of five or more usernames called `current_users`.
- Make another list of five usernames called `new_users`. Make sure one or two of the new usernames are also in the `current_users` list.
- Loop through the `new_users` list to see if each new username has already been used. If it has, print a message that the person will need to enter a new username. If a username has not been used, print a message saying that the username is available.
- Make sure your comparison is case insensitive. If `'John'` has been used, `'JOHN'` should not be accepted. (To do this, you'll need to make a copy of `current_users` containing the lowercase versions of all existing users.)

5-11. Ordinal Numbers: Ordinal numbers indicate their position in a list, such as *1st* or *2nd*. Most ordinal numbers end in *th*, except 1, 2, and 3.

- Store the numbers 1 through 9 in a list.
- Loop through the list.
- Use an `if-elif-else` chain inside the loop to print the proper ordinal ending for each number. Your output should read `"1st 2nd 3rd 4th 5th 6th 7th 8th 9th"`, and each result should be on a separate line.

Styling Your if Statements

In every example in this chapter, you've seen good styling habits. The only recommendation PEP 8 provides for styling conditional tests is to use a single space around comparison operators, such as ==, >=, <=. For example:

```
if age < 4:
```

is better than:

```
if age<4:
```

Such spacing does not affect the way Python interprets your code; it just makes your code easier for you and others to read.

TRY IT YOURSELF

5-12. Styling `if` statements: Review the programs you wrote in this chapter, and make sure you styled your conditional tests appropriately.

5-13. Your Ideas: At this point, you're a more capable programmer than you were when you started this book. Now that you have a better sense of how real-world situations are modeled in programs, you might be thinking of some problems you could solve with your own programs. Record any new ideas you have about problems you might want to solve as your programming skills continue to improve. Consider games you might want to write, data sets you might want to explore, and web applications you'd like to create.

Summary

In this chapter you learned how to write conditional tests, which always evaluate to `True` or `False`. You learned to write simple `if` statements, `if-else` chains, and `if-elif-else` chains. You began using these structures to identify particular conditions you needed to test and to know when those conditions have been met in your programs. You learned to handle certain items in a list differently than all other items while continuing to utilize the efficiency of a `for` loop. You also revisited Python's style recommendations to ensure that your increasingly complex programs are still relatively easy to read and understand.

In Chapter 6 you'll learn about Python's dictionaries. A dictionary is similar to a list, but it allows you to connect pieces of information. You'll learn to build dictionaries, loop through them, and use them in combination with lists and `if` statements. Learning about dictionaries will enable you to model an even wider variety of real-world situations.

6

DICTIONARIES

In this chapter you'll learn how to use Python's dictionaries, which allow you to connect pieces of related information. You'll learn how to access the information once it's in a dictionary and how to modify that information. Because dictionaries can store an almost limitless amount of information, I'll show you how to loop through the data in a dictionary. Additionally, you'll learn to nest dictionaries inside lists, lists inside dictionaries, and even dictionaries inside other dictionaries.

Understanding dictionaries allows you to model a variety of real-world objects more accurately. You'll be able to create a dictionary representing a person and then store as much information as you want about that person. You can store their name, age, location, profession, and any other aspect of a person you can describe. You'll be able to store any two kinds of

information that can be matched up, such as a list of words and their meanings, a list of people's names and their favorite numbers, a list of mountains and their elevations, and so forth.

A Simple Dictionary

Consider a game featuring aliens that can have different colors and point values. This simple dictionary stores information about a particular alien:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

The dictionary `alien_0` stores the alien's color and point value. The last two lines access and display that information, as shown here:

```
green
5
```

As with most new programming concepts, using dictionaries takes practice. Once you've worked with dictionaries for a bit you'll soon see how effectively they can model real-world situations.

Working with Dictionaries

A *dictionary* in Python is a collection of *key-value pairs*. Each *key* is connected to a value, and you can use a key to access the value associated with that key. A key's value can be a number, a string, a list, or even another dictionary. In fact, you can use any object that you can create in Python as a value in a dictionary.

In Python, a dictionary is wrapped in braces, {}, with a series of key-value pairs inside the braces, as shown in the earlier example:

```
alien_0 = {'color': 'green', 'points': 5}
```

A *key-value pair* is a set of values associated with each other. When you provide a key, Python returns the value associated with that key. Every key is connected to its value by a colon, and individual key-value pairs are separated by commas. You can store as many key-value pairs as you want in a dictionary.

The simplest dictionary has exactly one key-value pair, as shown in this modified version of the `alien_0` dictionary:

```
alien_0 = {'color': 'green'}
```

This dictionary stores one piece of information about `alien_0`, namely the alien's color. The string `'color'` is a key in this dictionary, and its associated value is `'green'`.

Accessing Values in a Dictionary

To get the value associated with a key, give the name of the dictionary and then place the key inside a set of square brackets, as shown here:

alien.py

```
alien_0 = {'color': 'green'}
print(alien_0['color'])
```

This returns the value associated with the key `'color'` from the dictionary `alien_0`:

```
green
```

You can have an unlimited number of key-value pairs in a dictionary. For example, here's the original `alien_0` dictionary with two key-value pairs:

```
alien_0 = {'color': 'green', 'points': 5}
```

Now you can access either the color or the point value of `alien_0`. If a player shoots down this alien, you can look up how many points they should earn using code like this:

```
alien_0 = {'color': 'green', 'points': 5}

❶ new_points = alien_0['points']
❷ print(f"You just earned {new_points} points!")
```

Once the dictionary has been defined, the code at ❶ pulls the value associated with the key `'points'` from the dictionary. This value is then assigned to the variable `new_points`. The line at ❷ prints a statement about how many points the player just earned:

```
You just earned 5 points!
```

If you run this code every time an alien is shot down, the alien's point value will be retrieved.

Adding New Key-Value Pairs

Dictionaries are dynamic structures, and you can add new key-value pairs to a dictionary at any time. For example, to add a new key-value pair, you would give the name of the dictionary followed by the new key in square brackets along with the new value.

Let's add two new pieces of information to the `alien_0` dictionary: the alien's x- and y-coordinates, which will help us display the alien in a

particular position on the screen. Let's place the alien on the left edge of the screen, 25 pixels down from the top. Because screen coordinates usually start at the upper-left corner of the screen, we'll place the alien on the left edge of the screen by setting the x-coordinate to 0 and 25 pixels from the top by setting its y-coordinate to positive 25, as shown here:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

❶ alien_0['x_position'] = 0
❷ alien_0['y_position'] = 25
print(alien_0)
```

We start by defining the same dictionary that we've been working with. We then print this dictionary, displaying a snapshot of its information. At ❶ we add a new key-value pair to the dictionary: key `'x_position'` and value `0`. We do the same for key `'y_position'` at ❷. When we print the modified dictionary, we see the two additional key-value pairs:

```
{'color': 'green', 'points': 5}
{'color': 'green', 'points': 5, 'y_position': 25, 'x_position': 0}
```

The final version of the dictionary contains four key-value pairs. The original two specify color and point value, and two more specify the alien's position.

NOTE *As of Python 3.7, dictionaries retain the order in which they were defined. When you print a dictionary or loop through its elements, you will see the elements in the same order in which they were added to the dictionary.*

Starting with an Empty Dictionary

It's sometimes convenient, or even necessary, to start with an empty dictionary and then add each new item to it. To start filling an empty dictionary, define a dictionary with an empty set of braces and then add each key-value pair on its own line. For example, here's how to build the `alien_0` dictionary using this approach:

alien.py

```
alien_0 = {}

alien_0['color'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

Here we define an empty `alien_0` dictionary, and then add color and point values to it. The result is the dictionary we've been using in previous examples:

```
{'color': 'green', 'points': 5}
```

Typically, you'll use empty dictionaries when storing user-supplied data in a dictionary or when you write code that generates a large number of key-value pairs automatically.

Modifying Values in a Dictionary

To modify a value in a dictionary, give the name of the dictionary with the key in square brackets and then the new value you want associated with that key. For example, consider an alien that changes from green to yellow as a game progresses:

alien.py

```
alien_0 = {'color': 'green'}
print(f"The alien is {alien_0['color']}.")

alien_0['color'] = 'yellow'
print(f"The alien is now {alien_0['color']}.")
```

We first define a dictionary for `alien_0` that contains only the alien's color; then we change the value associated with the key `'color'` to `'yellow'`. The output shows that the alien has indeed changed from green to yellow:

```
The alien is green.
The alien is now yellow.
```

For a more interesting example, let's track the position of an alien that can move at different speeds. We'll store a value representing the alien's current speed and then use it to determine how far to the right the alien should move:

```
alien_0 = {'x_position': 0, 'y_position': 25, 'speed': 'medium'}
print(f"Original position: {alien_0['x_position']}")

# Move the alien to the right.
# Determine how far to move the alien based on its current speed.
❶ if alien_0['speed'] == 'slow':
    x_increment = 1
elif alien_0['speed'] == 'medium':
    x_increment = 2
else:
    # This must be a fast alien.
    x_increment = 3

# The new position is the old position plus the increment.
❷ alien_0['x_position'] = alien_0['x_position'] + x_increment

print(f"New position: {alien_0['x_position']}")
```

We start by defining an alien with an initial *x* position and *y* position, and a speed of `'medium'`. We've omitted the color and point values for the

sake of simplicity, but this example would work the same way if you included those key-value pairs as well. We also print the original value of `x_position` to see how far the alien moves to the right.

At ❶, an `if-elif-else` chain determines how far the alien should move to the right and assigns this value to the variable `x_increment`. If the alien's speed is `'slow'`, it moves one unit to the right; if the speed is `'medium'`, it moves two units to the right; and if it's `'fast'`, it moves three units to the right. Once the increment has been calculated, it's added to the value of `x_position` at ❷, and the result is stored in the dictionary's `x_position`.

Because this is a medium-speed alien, its position shifts two units to the right:

```
Original x-position: 0
New x-position: 2
```

This technique is pretty cool: by changing one value in the alien's dictionary, you can change the overall behavior of the alien. For example, to turn this medium-speed alien into a fast alien, you would add the line:

```
alien_0['speed'] = 'fast'
```

The `if-elif-else` block would then assign a larger value to `x_increment` the next time the code runs.

Removing Key-Value Pairs

When you no longer need a piece of information that's stored in a dictionary, you can use the `del` statement to completely remove a key-value pair. All `del` needs is the name of the dictionary and the key that you want to remove.

For example, let's remove the key `'points'` from the `alien_0` dictionary along with its value:

alien.py

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

❶ del alien_0['points']
print(alien_0)
```

The line at ❶ tells Python to delete the key `'points'` from the dictionary `alien_0` and to remove the value associated with that key as well. The output shows that the key `'points'` and its value of `5` are deleted from the dictionary, but the rest of the dictionary is unaffected:

```
{'color': 'green', 'points': 5}
{'color': 'green'}
```

NOTE *Be aware that the deleted key-value pair is removed permanently.*

A Dictionary of Similar Objects

The previous example involved storing different kinds of information about one object, an alien in a game. You can also use a dictionary to store one kind of information about many objects. For example, say you want to poll a number of people and ask them what their favorite programming language is. A dictionary is useful for storing the results of a simple poll, like this:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }
```

As you can see, we've broken a larger dictionary into several lines. Each key is the name of a person who responded to the poll, and each value is their language choice. When you know you'll need more than one line to define a dictionary, press ENTER after the opening brace. Then indent the next line one level (four spaces), and write the first key-value pair, followed by a comma. From this point forward when you press ENTER, your text editor should automatically indent all subsequent key-value pairs to match the first key-value pair.

Once you've finished defining the dictionary, add a closing brace on a new line after the last key-value pair and indent it one level so it aligns with the keys in the dictionary. It's good practice to include a comma after the last key-value pair as well, so you're ready to add a new key-value pair on the next line.

NOTE *Most editors have some functionality that helps you format extended lists and dictionaries in a similar manner to this example. Other acceptable ways to format long dictionaries are available as well, so you may see slightly different formatting in your editor, or in other sources.*

To use this dictionary, given the name of a person who took the poll, you can easily look up their favorite language:

favorite_languages.py

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

❶ language = favorite_languages['sarah'].title()
print(f"Sarah's favorite language is {language}.")
```

To see which language Sarah chose, we ask for the value at:

```
favorite_languages['sarah']
```

We use this syntax to pull Sarah's favorite language from the dictionary at ❶ and assign it to the variable `language`. Creating a new variable here makes for a much cleaner `print()` call. The output shows Sarah's favorite language:

```
Sarah's favorite language is C.
```

You could use this same syntax with any individual represented in the dictionary.

Using get() to Access Values

Using keys in square brackets to retrieve the value you're interested in from a dictionary might cause one potential problem: if the key you ask for doesn't exist, you'll get an error.

Let's see what happens when you ask for the point value of an alien that doesn't have a point value set:

alien_no_points.py

```
alien_0 = {'color': 'green', 'speed': 'slow'}
print(alien_0['points'])
```

This results in a traceback, showing a `KeyError`:

```
Traceback (most recent call last):
  File "alien_no_points.py", line 2, in <module>
    print(alien_0['points'])
KeyError: 'points'
```

You'll learn more about how to handle errors like this in general in Chapter 10. For dictionaries, specifically, you can use the `get()` method to set a default value that will be returned if the requested key doesn't exist.

The `get()` method requires a key as a first argument. As a second optional argument, you can pass the value to be returned if the key doesn't exist:

```
alien_0 = {'color': 'green', 'speed': 'slow'}

point_value = alien_0.get('points', 'No point value assigned.')
print(point_value)
```

If the key `'points'` exists in the dictionary, you'll get the corresponding value. If it doesn't, you get the default value. In this case, `points` doesn't exist, and we get a clean message instead of an error:

```
No point value assigned.
```

If there's a chance the key you're asking for might not exist, consider using the `get()` method instead of the square bracket notation.

NOTE *If you leave out the second argument in the call to* `get()` *and the key doesn't exist, Python will return the value* `None`*. The special value* `None` *means "no value exists." This is not an error: it's a special value meant to indicate the absence of a value. You'll see more uses for* `None` *in Chapter 8.*

TRY IT YOURSELF

6-1. Person: Use a dictionary to store information about a person you know. Store their first name, last name, age, and the city in which they live. You should have keys such as `first_name`, `last_name`, `age`, and `city`. Print each piece of information stored in your dictionary.

6-2. Favorite Numbers: Use a dictionary to store people's favorite numbers. Think of five names, and use them as keys in your dictionary. Think of a favorite number for each person, and store each as a value in your dictionary. Print each person's name and their favorite number. For even more fun, poll a few friends and get some actual data for your program.

6-3. Glossary: A Python dictionary can be used to model an actual dictionary. However, to avoid confusion, let's call it a glossary.

- Think of five programming words you've learned about in the previous chapters. Use these words as the keys in your glossary, and store their meanings as values.
- Print each word and its meaning as neatly formatted output. You might print the word followed by a colon and then its meaning, or print the word on one line and then print its meaning indented on a second line. Use the newline character (\n) to insert a blank line between each word-meaning pair in your output.

Looping Through a Dictionary

A single Python dictionary can contain just a few key-value pairs or millions of pairs. Because a dictionary can contain large amounts of data, Python lets you loop through a dictionary. Dictionaries can be used to store information in a variety of ways; therefore, several different ways exist to loop through them. You can loop through all of a dictionary's key-value pairs, through its keys, or through its values.

Looping Through All Key-Value Pairs

Before we explore the different approaches to looping, let's consider a new dictionary designed to store information about a user on a website. The

following dictionary would store one person's username, first name, and last name:

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }
```

You can access any single piece of information about `user_0` based on what you've already learned in this chapter. But what if you wanted to see everything stored in this user's dictionary? To do so, you could loop through the dictionary using a `for` loop:

user.py

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }

❶ for key, value in user_0.items():
❷     print(f"\nKey: {key}")
❸     print(f"Value: {value}")
```

As shown at ❶, to write a `for` loop for a dictionary, you create names for the two variables that will hold the key and value in each key-value pair. You can choose any names you want for these two variables. This code would work just as well if you had used abbreviations for the variable names, like this:

```
for k, v in user_0.items()
```

The second half of the `for` statement at ❶ includes the name of the dictionary followed by the method `items()`, which returns a list of key-value pairs. The `for` loop then assigns each of these pairs to the two variables provided. In the preceding example, we use the variables to print each `key` ❷, followed by the associated `value` ❸. The `"\n"` in the first `print()` call ensures that a blank line is inserted before each key-value pair in the output:

```
Key: last
Value: fermi

Key: first
Value: enrico

Key: username
Value: efermi
```

Looping through all key-value pairs works particularly well for dictionaries like the *favorite_languages.py* example on page 97, which stores the same kind of information for many different keys. If you loop through the `favorite_languages` dictionary, you get the name of each person in the dictionary and their favorite programming language. Because the keys always refer to a person's name and the value is always a language, we'll use the variables `name` and `language` in the loop instead of `key` and `value`. This will make it easier to follow what's happening inside the loop:

favorite_languages.py

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

❶ for name, language in favorite_languages.items():
❷     print(f"{name.title()}'s favorite language is {language.title()}.")
```

The code at ❶ tells Python to loop through each key-value pair in the dictionary. As it works through each pair the key is assigned to the variable `name`, and the value is assigned to the variable `language`. These descriptive names make it much easier to see what the `print()` call at ❷ is doing.

Now, in just a few lines of code, we can display all of the information from the poll:

```
Jen's favorite language is Python.
Sarah's favorite language is C.
Edward's favorite language is Ruby.
Phil's favorite language is Python.
```

This type of looping would work just as well if our dictionary stored the results from polling a thousand or even a million people.

Looping Through All the Keys in a Dictionary

The `keys()` method is useful when you don't need to work with all of the values in a dictionary. Let's loop through the `favorite_languages` dictionary and print the names of everyone who took the poll:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

❶ for name in favorite_languages.keys():
    print(name.title())
```

The line at ❶ tells Python to pull all the keys from the dictionary `favorite_languages` and assign them one at a time to the variable `name`. The output shows the names of everyone who took the poll:

```
Jen
Sarah
Edward
Phil
```

Looping through the keys is actually the default behavior when looping through a dictionary, so this code would have exactly the same output if you wrote . . .

```
for name in favorite_languages:
```

rather than . . .

```
for name in favorite_languages.keys():
```

You can choose to use the `keys()` method explicitly if it makes your code easier to read, or you can omit it if you wish.

You can access the value associated with any key you care about inside the loop by using the current key. Let's print a message to a couple of friends about the languages they chose. We'll loop through the names in the dictionary as we did previously, but when the name matches one of our friends, we'll display a message about their favorite language:

```
favorite_languages = {
    --snip--
    }

❶ friends = ['phil', 'sarah']
  for name in favorite_languages.keys():
      print(name.title())

❷     if name in friends:
❸         language = favorite_languages[name].title()
          print(f"\t{name.title()}, I see you love {language}!")
```

At ❶ we make a list of friends that we want to print a message to. Inside the loop, we print each person's name. Then at ❷ we check whether the `name` we're working with is in the list `friends`. If it is, we determine the person's favorite language using the name of the dictionary and the current value of `name` as the key ❸. We then print a special greeting, including a reference to their language of choice.

Everyone's name is printed, but our friends receive a special message:

```
Hi Jen.
Hi Sarah.
    Sarah, I see you love C!
Hi Edward.
```

```
Hi Phil.
    Phil, I see you love Python!
```

You can also use the `keys()` method to find out if a particular person was polled. This time, let's find out if Erin took the poll:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

❶ if 'erin' not in favorite_languages.keys():
    print("Erin, please take our poll!")
```

The `keys()` method isn't just for looping: it actually returns a list of all the keys, and the line at ❶ simply checks if `'erin'` is in this list. Because she's not, a message is printed inviting her to take the poll:

```
Erin, please take our poll!
```

Looping Through a Dictionary's Keys in a Particular Order

Starting in Python 3.7, looping through a dictionary returns the items in the same order they were inserted. Sometimes, though, you'll want to loop through a dictionary in a different order.

One way to do this is to sort the keys as they're returned in the `for` loop. You can use the `sorted()` function to get a copy of the keys in order:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

for name in sorted(favorite_languages.keys()):
    print(f"{name.title()}, thank you for taking the poll.")
```

This `for` statement is like other `for` statements except that we've wrapped the `sorted()` function around the `dictionary.keys()` method. This tells Python to list all keys in the dictionary and sort that list before looping through it. The output shows everyone who took the poll, with the names displayed in order:

```
Edward, thank you for taking the poll.
Jen, thank you for taking the poll.
Phil, thank you for taking the poll.
Sarah, thank you for taking the poll.
```

Looping Through All Values in a Dictionary

If you are primarily interested in the values that a dictionary contains, you can use the `values()` method to return a list of values without any keys. For example, say we simply want a list of all languages chosen in our programming language poll without the name of the person who chose each language:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

print("The following languages have been mentioned:")
for language in favorite_languages.values():
    print(language.title())
```

The `for` statement here pulls each value from the dictionary and assigns it to the variable `language`. When these values are printed, we get a list of all chosen languages:

```
The following languages have been mentioned:
Python
C
Python
Ruby
```

This approach pulls all the values from the dictionary without checking for repeats. That might work fine with a small number of values, but in a poll with a large number of respondents, this would result in a very repetitive list. To see each language chosen without repetition, we can use a set. A *set* is a collection in which each item must be unique:

```
favorite_languages = {
    --snip--
    }

print("The following languages have been mentioned:")
❶ for language in set(favorite_languages.values()):
    print(language.title())
```

When you wrap `set()` around a list that contains duplicate items, Python identifies the unique items in the list and builds a set from those items. At ❶ we use `set()` to pull out the unique languages in `favorite_languages.values()`.

The result is a nonrepetitive list of languages that have been mentioned by people taking the poll:

```
The following languages have been mentioned:
Python
```

```
C
Ruby
```

As you continue learning about Python, you'll often find a built-in feature of the language that helps you do exactly what you want with your data.

NOTE *You can build a set directly using braces and separating the elements with commas:*

```
>>> languages = {'python', 'ruby', 'python', 'c'}
>>> languages
{'ruby', 'python', 'c'}
```

It's easy to mistake sets for dictionaries because they're both wrapped in braces. When you see braces but no key-value pairs, you're probably looking at a set. Unlike lists and dictionaries, sets do not retain items in any specific order.

TRY IT YOURSELF

6-4. Glossary 2: Now that you know how to loop through a dictionary, clean up the code from Exercise 6-3 (page 99) by replacing your series of `print()` calls with a loop that runs through the dictionary's keys and values. When you're sure that your loop works, add five more Python terms to your glossary. When you run your program again, these new words and meanings should automatically be included in the output.

6-5. Rivers: Make a dictionary containing three major rivers and the country each river runs through. One key-value pair might be `'nile': 'egypt'`.

- Use a loop to print a sentence about each river, such as *The Nile runs through Egypt.*
- Use a loop to print the name of each river included in the dictionary.
- Use a loop to print the name of each country included in the dictionary.

6-6. Polling: Use the code in *favorite_languages.py* (page 97).

- Make a list of people who should take the favorite languages poll. Include some names that are already in the dictionary and some that are not.
- Loop through the list of people who should take the poll. If they have already taken the poll, print a message thanking them for responding. If they have not yet taken the poll, print a message inviting them to take the poll.

Nesting

Sometimes you'll want to store multiple dictionaries in a list, or a list of items as a value in a dictionary. This is called *nesting*. You can nest dictionaries inside a list, a list of items inside a dictionary, or even a dictionary inside another dictionary. Nesting is a powerful feature, as the following examples will demonstrate.

A List of Dictionaries

The `alien_0` dictionary contains a variety of information about one alien, but it has no room to store information about a second alien, much less a screen full of aliens. How can you manage a fleet of aliens? One way is to make a list of aliens in which each alien is a dictionary of information about that alien. For example, the following code builds a list of three aliens:

aliens.py

```
alien_0 = {'color': 'green', 'points': 5}
alien_1 = {'color': 'yellow', 'points': 10}
alien_2 = {'color': 'red', 'points': 15}

❶ aliens = [alien_0, alien_1, alien_2]

for alien in aliens:
    print(alien)
```

We first create three dictionaries, each representing a different alien. At ❶ we store each of these dictionaries in a list called `aliens`. Finally, we loop through the list and print out each alien:

```
{'color': 'green', 'points': 5}
{'color': 'yellow', 'points': 10}
{'color': 'red', 'points': 15}
```

A more realistic example would involve more than three aliens with code that automatically generates each alien. In the following example we use `range()` to create a fleet of 30 aliens:

```
# Make an empty list for storing aliens.
aliens = []

# Make 30 green aliens.
❶ for alien_number in range(30):
❷     new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
❸     aliens.append(new_alien)

# Show the first 5 aliens.
❹ for alien in aliens[:5]:
    print(alien)
print("...")

# Show how many aliens have been created.
❺ print(f"Total number of aliens: {len(aliens)}")
```

This example begins with an empty list to hold all of the aliens that will be created. At ❶ `range()` returns a series of numbers, which just tells Python how many times we want the loop to repeat. Each time the loop runs we create a new alien ❷ and then append each new alien to the list `aliens` ❸. At ❹ we use a slice to print the first five aliens, and then at ❺ we print the length of the list to prove we've actually generated the full fleet of 30 aliens:

```
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
...

Total number of aliens: 30
```

These aliens all have the same characteristics, but Python considers each one a separate object, which allows us to modify each alien individually.

How might you work with a group of aliens like this? Imagine that one aspect of a game has some aliens changing color and moving faster as the game progresses. When it's time to change colors, we can use a `for` loop and an `if` statement to change the color of aliens. For example, to change the first three aliens to yellow, medium-speed aliens worth 10 points each, we could do this:

```
# Make an empty list for storing aliens.
aliens = []

# Make 30 green aliens.
for alien_number in range (30):
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

for alien in aliens[:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10

# Show the first 5 aliens.
for alien in aliens[:5]:
    print(alien)
print("...")
```

Because we want to modify the first three aliens, we loop through a slice that includes only the first three aliens. All of the aliens are green now but that won't always be the case, so we write an `if` statement to make sure

we're only modifying green aliens. If the alien is green, we change the color to `'yellow'`, the speed to `'medium'`, and the point value to `10`, as shown in the following output:

```
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'medium', 'color': 'yellow', 'points': 10}
{'speed': 'slow', 'color': 'green', 'points': 5}
{'speed': 'slow', 'color': 'green', 'points': 5}
...
```

You could expand this loop by adding an `elif` block that turns yellow aliens into red, fast-moving ones worth 15 points each. Without showing the entire program again, that loop would look like this:

```
for alien in aliens[0:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10
    elif alien['color'] == 'yellow':
        alien['color'] = 'red'
        alien['speed'] = 'fast'
        alien['points'] = 15
```

It's common to store a number of dictionaries in a list when each dictionary contains many kinds of information about one object. For example, you might create a dictionary for each user on a website, as we did in *user.py* on page 100, and store the individual dictionaries in a list called `users`. All of the dictionaries in the list should have an identical structure so you can loop through the list and work with each dictionary object in the same way.

A List in a Dictionary

Rather than putting a dictionary inside a list, it's sometimes useful to put a list inside a dictionary. For example, consider how you might describe a pizza that someone is ordering. If you were to use only a list, all you could really store is a list of the pizza's toppings. With a dictionary, a list of toppings can be just one aspect of the pizza you're describing.

In the following example, two kinds of information are stored for each pizza: a type of crust and a list of toppings. The list of toppings is a value associated with the key `'toppings'`. To use the items in the list, we give the name of the dictionary and the key `'toppings'`, as we would any value in the dictionary. Instead of returning a single value, we get a list of toppings:

pizza.py

```
# Store information about a pizza being ordered.
❶ pizza = {
    'crust': 'thick',
    'toppings': ['mushrooms', 'extra cheese'],
    }
```

```
# Summarize the order.
❷ print(f"You ordered a {pizza['crust']}-crust pizza "
    "with the following toppings:")

❸ for topping in pizza['toppings']:
    print("\t" + topping)
```

We begin at ❶ with a dictionary that holds information about a pizza that has been ordered. One key in the dictionary is `'crust'`, and the associated value is the string `'thick'`. The next key, `'toppings'`, has a list as its value that stores all requested toppings. At ❷ we summarize the order before building the pizza. When you need to break up a long line in a `print()` call, choose an appropriate point at which to break the line being printed, and end the line with a quotation mark. Indent the next line, add an opening quotation mark, and continue the string. Python will automatically combine all of the strings it finds inside the parentheses. To print the toppings, we write a `for` loop ❸. To access the list of toppings, we use the key `'toppings'`, and Python grabs the list of toppings from the dictionary.

The following output summarizes the pizza that we plan to build:

```
You ordered a thick-crust pizza with the following toppings:
    mushrooms
    extra cheese
```

You can nest a list inside a dictionary any time you want more than one value to be associated with a single key in a dictionary. In the earlier example of favorite programming languages, if we were to store each person's responses in a list, people could choose more than one favorite language. When we loop through the dictionary, the value associated with each person would be a list of languages rather than a single language. Inside the dictionary's `for` loop, we use another `for` loop to run through the list of languages associated with each person:

favorite_languages.py

```
❶ favorite_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
    }

❷ for name, languages in favorite_languages.items():
    print(f"\n{name.title()}'s favorite languages are:")
❸     for language in languages:
        print(f"\t{language.title()}")
```

As you can see at ❶ the value associated with each name is now a list. Notice that some people have one favorite language and others have

multiple favorites. When we loop through the dictionary at ❷, we use the variable name `languages` to hold each value from the dictionary, because we know that each value will be a list. Inside the main dictionary loop, we use another `for` loop ❸ to run through each person's list of favorite languages. Now each person can list as many favorite languages as they like:

```
Jen's favorite languages are:
    Python
    Ruby

Sarah's favorite languages are:
    C

Phil's favorite languages are:
    Python
    Haskell

Edward's favorite languages are:
    Ruby
    Go
```

To refine this program even further, you could include an `if` statement at the beginning of the dictionary's `for` loop to see whether each person has more than one favorite language by examining the value of `len(languages)`. If a person has more than one favorite, the output would stay the same. If the person has only one favorite language, you could change the wording to reflect that. For example, you could say `Sarah's favorite language is C`.

NOTE *You should not nest lists and dictionaries too deeply. If you're nesting items much deeper than what you see in the preceding examples or you're working with someone else's code with significant levels of nesting, most likely a simpler way to solve the problem exists.*

A Dictionary in a Dictionary

You can nest a dictionary inside another dictionary, but your code can get complicated quickly when you do. For example, if you have several users for a website, each with a unique username, you can use the usernames as the keys in a dictionary. You can then store information about each user by using a dictionary as the value associated with their username. In the following listing, we store three pieces of information about each user: their first name, last name, and location. We'll access this information by looping through the usernames and the dictionary of information associated with each username:

many_users.py

```
users = {
    'aeinstein': {
        'first': 'albert',
```

```
        'last': 'einstein',
        'location': 'princeton',
        },

    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
        },

    }

❶ for username, user_info in users.items():
❷     print(f"\nUsername: {username}")
❸     full_name = f"{user_info['first']} {user_info['last']}"
      location = user_info['location']

❹     print(f"\tFull name: {full_name.title()}")
      print(f"\tLocation: {location.title()}")
```

We first define a dictionary called `users` with two keys: one each for the usernames `'aeinstein'` and `'mcurie'`. The value associated with each key is a dictionary that includes each user's first name, last name, and location. At ❶ we loop through the `users` dictionary. Python assigns each key to the variable `username`, and the dictionary associated with each username is assigned to the variable `user_info`. Once inside the main dictionary loop, we print the username at ❷.

At ❸ we start accessing the inner dictionary. The variable `user_info`, which contains the dictionary of user information, has three keys: `'first'`, `'last'`, and `'location'`. We use each key to generate a neatly formatted full name and location for each person, and then print a summary of what we know about each user ❹:

```
Username: aeinstein
    Full name: Albert Einstein
    Location: Princeton

Username: mcurie
    Full name: Marie Curie
    Location: Paris
```

Notice that the structure of each user's dictionary is identical. Although not required by Python, this structure makes nested dictionaries easier to work with. If each user's dictionary had different keys, the code inside the `for` loop would be more complicated.

TRY IT YOURSELF

6-7. People: Start with the program you wrote for Exercise 6-1 (page 99). Make two new dictionaries representing different people, and store all three dictionaries in a list called `people`. Loop through your list of people. As you loop through the list, print everything you know about each person.

6-8. Pets: Make several dictionaries, where each dictionary represents a different pet. In each dictionary, include the kind of animal and the owner's name. Store these dictionaries in a list called `pets`. Next, loop through your list and as you do, print everything you know about each pet.

6-9. Favorite Places: Make a dictionary called `favorite_places`. Think of three names to use as keys in the dictionary, and store one to three favorite places for each person. To make this exercise a bit more interesting, ask some friends to name a few of their favorite places. Loop through the dictionary, and print each person's name and their favorite places.

6-10. Favorite Numbers: Modify your program from Exercise 6-2 (page 99) so each person can have more than one favorite number. Then print each person's name along with their favorite numbers.

6-11. Cities: Make a dictionary called `cities`. Use the names of three cities as keys in your dictionary. Create a dictionary of information about each city and include the country that the city is in, its approximate population, and one fact about that city. The keys for each city's dictionary should be something like `country`, `population`, and `fact`. Print the name of each city and all of the information you have stored about it.

6-12. Extensions: We're now working with examples that are complex enough that they can be extended in any number of ways. Use one of the example programs from this chapter, and extend it by adding new keys and values, changing the context of the program or improving the formatting of the output.

Summary

In this chapter you learned how to define a dictionary and how to work with the information stored in a dictionary. You learned how to access and modify individual elements in a dictionary, and how to loop through all of the information in a dictionary. You learned to loop through a dictionary's key-value pairs, its keys, and its values. You also learned how to nest multiple dictionaries in a list, nest lists in a dictionary, and nest a dictionary inside a dictionary.

In the next chapter you'll learn about `while` loops and how to accept input from people who are using your programs. This will be an exciting chapter, because you'll learn to make all of your programs interactive: they'll be able to respond to user input.

7

USER INPUT AND WHILE LOOPS

Most programs are written to solve an end user's problem. To do so, you usually need to get some information from the user. For a simple example, let's say someone wants to find out whether they're old enough to vote. If you write a program to answer this question, you need to know the user's age before you can provide an answer. The program will need to ask the user to enter, or *input*, their age; once the program has this input, it can compare it to the voting age to determine if the user is old enough and then report the result.

In this chapter you'll learn how to accept user input so your program can then work with it. When your program needs a name, you'll be able to prompt the user for a name. When your program needs a list of names, you'll be able to prompt the user for a series of names. To do this, you'll use the `input()` function.

You'll also learn how to keep programs running as long as users want them to, so they can enter as much information as they need to; then, your program can work with that information. You'll use Python's `while` loop to keep programs running as long as certain conditions remain true.

With the ability to work with user input and the ability to control how long your programs run, you'll be able to write fully interactive programs.

How the input() Function Works

The `input()` function pauses your program and waits for the user to enter some text. Once Python receives the user's input, it assigns that input to a variable to make it convenient for you to work with.

For example, the following program asks the user to enter some text, then displays that message back to the user:

parrot.py

```
message = input("Tell me something, and I will repeat it back to you: ")
print(message)
```

The `input()` function takes one argument: the *prompt*, or instructions, that we want to display to the user so they know what to do. In this example, when Python runs the first line, the user sees the prompt `Tell me something, and I will repeat it back to you: `. The program waits while the user enters their response and continues after the user presses ENTER. The response is assigned to the variable `message`, then `print(message)` displays the input back to the user:

```
Tell me something, and I will repeat it back to you: Hello everyone!
Hello everyone!
```

NOTE *Sublime Text and many other editors don't run programs that prompt the user for input. You can use these editors to write programs that prompt for input, but you'll need to run these programs from a terminal. See "Running Python Programs from a Terminal" on page 12.*

Writing Clear Prompts

Each time you use the `input()` function, you should include a clear, easy-to-follow prompt that tells the user exactly what kind of information you're looking for. Any statement that tells the user what to enter should work. For example:

greeter.py

```
name = input("Please enter your name: ")
print(f"\nHello, {name}!")
```

Add a space at the end of your prompts (after the colon in the preceding example) to separate the prompt from the user's response and to make it clear to your user where to enter their text. For example:

```
Please enter your name: Eric
Hello, Eric!
```

Sometimes you'll want to write a prompt that's longer than one line. For example, you might want to tell the user why you're asking for certain input. You can assign your prompt to a variable and pass that variable to the `input()` function. This allows you to build your prompt over several lines, then write a clean `input()` statement.

greeter.py

```
prompt = "If you tell us who you are, we can personalize the messages you see."
prompt += "\nWhat is your first name? "

name = input(prompt)
print(f"\nHello, {name}!")
```

This example shows one way to build a multi-line string. The first line assigns the first part of the message to the variable `prompt`. In the second line, the operator `+=` takes the string that was assigned to `prompt` and adds the new string onto the end.

The prompt now spans two lines, again with space after the question mark for clarity:

```
If you tell us who you are, we can personalize the messages you see.
What is your first name? Eric

Hello, Eric!
```

Using int() to Accept Numerical Input

When you use the `input()` function, Python interprets everything the user enters as a string. Consider the following interpreter session, which asks for the user's age:

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age
'21'
```

The user enters the number 21, but when we ask Python for the value of `age`, it returns `'21'`, the string representation of the numerical value entered. We know Python interpreted the input as a string because the number is now enclosed in quotes. If all you want to do is print the input, this works well. But if you try to use the input as a number, you'll get an error:

```
  >>> age = input("How old are you? ")
  How old are you? 21
❶ >>> age >= 18
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
❷ TypeError: unorderable types: str() >= int()
```

When you try to use the input to do a numerical comparison ❶, Python produces an error because it can't compare a string to an integer: the string `'21'` that's assigned to `age` can't be compared to the numerical value `18` ❷.

We can resolve this issue by using the `int()` function, which tells Python to treat the input as a numerical value. The `int()` function converts a string representation of a number to a numerical representation, as shown here:

```
>>> age = input("How old are you? ")
How old are you? 21
❶ >>> age = int(age)
>>> age >= 18
True
```

In this example, when we enter `21` at the prompt, Python interprets the number as a string, but the value is then converted to a numerical representation by `int()` ❶. Now Python can run the conditional test: it compares `age` (which now represents the numerical value 21) and `18` to see if `age` is greater than or equal to 18. This test evaluates to `True`.

How do you use the `int()` function in an actual program? Consider a program that determines whether people are tall enough to ride a roller coaster:

rollercoaster.py

```
height = input("How tall are you, in inches? ")
height = int(height)

if height >= 48:
    print("\nYou're tall enough to ride!")
else:
    print("\nYou'll be able to ride when you're a little older.")
```

The program can compare `height` to `48` because `height = int(height)` converts the input value to a numerical representation before the comparison is made. If the number entered is greater than or equal to 48, we tell the user that they're tall enough:

```
How tall are you, in inches? 71

You're tall enough to ride!
```

When you use numerical input to do calculations and comparisons, be sure to convert the input value to a numerical representation first.

The Modulo Operator

A useful tool for working with numerical information is the *modulo operator* (`%`), which divides one number by another number and returns the remainder:

```
>>> 4 % 3
1
```

```
>>> 5 % 3
2
>>> 6 % 3
0
>>> 7 % 3
1
```

The modulo operator doesn't tell you how many times one number fits into another; it just tells you what the remainder is.

When one number is divisible by another number, the remainder is 0, so the modulo operator always returns 0. You can use this fact to determine if a number is even or odd:

even_or_odd.py

```
number = input("Enter a number, and I'll tell you if it's even or odd: ")
number = int(number)

if number % 2 == 0:
    print(f"\nThe number {number} is even.")
else:
    print(f"\nThe number {number} is odd.")
```

Even numbers are always divisible by two, so if the modulo of a number and two is zero (here, `if number % 2 == 0`) the number is even. Otherwise, it's odd.

```
Enter a number, and I'll tell you if it's even or odd: 42

The number 42 is even.
```

TRY IT YOURSELF

7-1. Rental Car: Write a program that asks the user what kind of rental car they would like. Print a message about that car, such as "Let me see if I can find you a Subaru."

7-2. Restaurant Seating: Write a program that asks the user how many people are in their dinner group. If the answer is more than eight, print a message saying they'll have to wait for a table. Otherwise, report that their table is ready.

7-3. Multiples of Ten: Ask the user for a number, and then report whether the number is a multiple of 10 or not.

Introducing while Loops

The `for` loop takes a collection of items and executes a block of code once for each item in the collection. In contrast, the `while` loop runs as long as, or *while*, a certain condition is true.

The while Loop in Action

You can use a `while` loop to count up through a series of numbers. For example, the following `while` loop counts from 1 to 5:

counting.py

```
current_number = 1
while current_number <= 5:
    print(current_number)
    current_number += 1
```

In the first line, we start counting from 1 by assigning `current_number` the value 1. The `while` loop is then set to keep running as long as the value of `current_number` is less than or equal to 5. The code inside the loop prints the value of `current_number` and then adds 1 to that value with `current_number += 1`. (The `+=` operator is shorthand for `current_number = current_number + 1`.)

Python repeats the loop as long as the condition `current_number <= 5` is true. Because 1 is less than 5, Python prints `1` and then adds 1, making the current number `2`. Because 2 is less than 5, Python prints `2` and adds 1 again, making the current number `3`, and so on. Once the value of `current_number` is greater than 5, the loop stops running and the program ends:

```
1
2
3
4
5
```

The programs you use every day most likely contain `while` loops. For example, a game needs a `while` loop to keep running as long as you want to keep playing, and so it can stop running as soon as you ask it to quit. Programs wouldn't be fun to use if they stopped running before we told them to or kept running even after we wanted to quit, so `while` loops are quite useful.

Letting the User Choose When to Quit

We can make the *parrot.py* program run as long as the user wants by putting most of the program inside a `while` loop. We'll define a *quit value* and then keep the program running as long as the user has not entered the quit value:

parrot.py

```
❶ prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "
```

```
❷ message = ""
❸ while message != 'quit':
      message = input(prompt)
      print(message)
```

At ❶, we define a prompt that tells the user their two options: entering a message or entering the quit value (in this case, `'quit'`). Then we set up a variable `message` ❷ to keep track of whatever value the user enters. We define `message` as an empty string, `""`, so Python has something to check the first time it reaches the `while` line. The first time the program runs and Python reaches the `while` statement, it needs to compare the value of `message` to `'quit'`, but no user input has been entered yet. If Python has nothing to compare, it won't be able to continue running the program. To solve this problem, we make sure to give `message` an initial value. Although it's just an empty string, it will make sense to Python and allow it to perform the comparison that makes the `while` loop work. This `while` loop ❸ runs as long as the value of `message` is not `'quit'`.

The first time through the loop, `message` is just an empty string, so Python enters the loop. At `message = input(prompt)`, Python displays the prompt and waits for the user to enter their input. Whatever they enter is assigned to `message` and printed; then, Python reevaluates the condition in the `while` statement. As long as the user has not entered the word `'quit'`, the prompt is displayed again and Python waits for more input. When the user finally enters `'quit'`, Python stops executing the `while` loop and the program ends:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
quit
```

This program works well, except that it prints the word `'quit'` as if it were an actual message. A simple `if` test fixes this:

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

Now the program makes a quick check before displaying the message and only prints the message if it does not match the quit value:

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello everyone!
Hello everyone!

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. Hello again.
Hello again.

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
```

Using a Flag

In the previous example, we had the program perform certain tasks while a given condition was true. But what about more complicated programs in which many different events could cause the program to stop running?

For example, in a game, several different events can end the game. When the player runs out of ships, their time runs out, or the cities they were supposed to protect are all destroyed, the game should end. It needs to end if any one of these events happens. If many possible events might occur to stop the program, trying to test all these conditions in one `while` statement becomes complicated and difficult.

For a program that should run only as long as many conditions are true, you can define one variable that determines whether or not the entire program is active. This variable, called a *flag*, acts as a signal to the program. We can write our programs so they run while the flag is set to `True` and stop running when any of several events sets the value of the flag to `False`. As a result, our overall `while` statement needs to check only one condition: whether or not the flag is currently `True`. Then, all our other tests (to see if an event has occurred that should set the flag to `False`) can be neatly organized in the rest of the program.

Let's add a flag to *parrot.py* from the previous section. This flag, which we'll call `active` (though you can call it anything), will monitor whether or not the program should continue running:

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

❶ active = True
❷ while active:
      message = input(prompt)

❸     if message == 'quit':
          active = False
❹     else:
          print(message)
```

We set the variable `active` to `True` ❶ so the program starts in an active state. Doing so makes the `while` statement simpler because no comparison is made in the `while` statement itself; the logic is taken care of in other parts of the program. As long as the `active` variable remains `True`, the loop will continue running ❷.

In the `if` statement inside the `while` loop, we check the value of `message` once the user enters their input. If the user enters `'quit'` ❸, we set `active` to `False`, and the `while` loop stops. If the user enters anything other than `'quit'` ❹, we print their input as a message.

This program has the same output as the previous example where we placed the conditional test directly in the `while` statement. But now that we have a flag to indicate whether the overall program is in an active state, it would be easy to add more tests (such as `elif` statements) for events that should cause `active` to become `False`. This is useful in complicated programs like games in which there may be many events that should each make the program stop running. When any of these events causes the active flag to become `False`, the main game loop will exit, a *Game Over* message can be displayed, and the player can be given the option to play again.

Using break to Exit a Loop

To exit a `while` loop immediately without running any remaining code in the loop, regardless of the results of any conditional test, use the `break` statement. The `break` statement directs the flow of your program; you can use it to control which lines of code are executed and which aren't, so the program only executes code that you want it to, when you want it to.

For example, consider a program that asks the user about places they've visited. We can stop the `while` loop in this program by calling `break` as soon as the user enters the `'quit'` value:

cities.py

```
prompt = "\nPlease enter the name of a city you have visited:"
prompt += "\n(Enter 'quit' when you are finished.) "

❶ while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print(f"I'd love to go to {city.title()}!")
```

A loop that starts with `while True` ❶ will run forever unless it reaches a `break` statement. The loop in this program continues asking the user to enter the names of cities they've been to until they enter `'quit'`. When they enter `'quit'`, the `break` statement runs, causing Python to exit the loop:

```
Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) New York
I'd love to go to New York!
```

```
Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) San Francisco
I'd love to go to San Francisco!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) quit
```

NOTE *You can use the* `break` *statement in any of Python's loops. For example, you could use* `break` *to quit a* `for` *loop that's working through a list or a dictionary.*

Using continue in a Loop

Rather than breaking out of a loop entirely without executing the rest of its code, you can use the `continue` statement to return to the beginning of the loop based on the result of a conditional test. For example, consider a loop that counts from 1 to 10 but prints only the odd numbers in that range:

counting.py

```
current_number = 0
while current_number < 10:
❶   current_number += 1
    if current_number % 2 == 0:
        continue

    print(current_number)
```

First we set `current_number` to 0. Because it's less than 10, Python enters the `while` loop. Once inside the loop, we increment the count by 1 at ❶, so `current_number` is 1. The `if` statement then checks the modulo of `current_number` and 2. If the modulo is 0 (which means `current_number` is divisible by 2), the `continue` statement tells Python to ignore the rest of the loop and return to the beginning. If the current number is not divisible by 2, the rest of the loop is executed and Python prints the current number:

```
1
3
5
7
9
```

Avoiding Infinite Loops

Every `while` loop needs a way to stop running so it won't continue to run forever. For example, this counting loop should count from 1 to 5:

counting.py

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

But if you accidentally omit the line `x += 1` (as shown next), the loop will run forever:

```
# This loop runs forever!
x = 1
while x <= 5:
    print(x)
```

Now the value of `x` will start at `1` but never change. As a result, the conditional test `x <= 5` will always evaluate to `True` and the `while` loop will run forever, printing a series of 1s, like this:

```
1
1
1
1
--snip--
```

Every programmer accidentally writes an infinite `while` loop from time to time, especially when a program's loops have subtle exit conditions. If your program gets stuck in an infinite loop, press CTRL-C or just close the terminal window displaying your program's output.

To avoid writing infinite loops, test every `while` loop and make sure the loop stops when you expect it to. If you want your program to end when the user enters a certain input value, run the program and enter that value. If the program doesn't end, scrutinize the way your program handles the value that should cause the loop to exit. Make sure at least one part of the program can make the loop's condition `False` or cause it to reach a `break` statement.

NOTE *Sublime Text and some other editors have an embedded output window. This can make it difficult to stop an infinite loop, and you might have to close the editor to end the loop. Try clicking in the output area of the editor before pressing CTRL-C, and you should be able to cancel an infinite loop.*

TRY IT YOURSELF

7-4. Pizza Toppings: Write a loop that prompts the user to enter a series of pizza toppings until they enter a `'quit'` value. As they enter each topping, print a message saying you'll add that topping to their pizza.

7-5. Movie Tickets: A movie theater charges different ticket prices depending on a person's age. If a person is under the age of 3, the ticket is free; if they are between 3 and 12, the ticket is $10; and if they are over age 12, the ticket is $15. Write a loop in which you ask users their age, and then tell them the cost of their movie ticket.

(continued)

7-6. Three Exits: Write different versions of either Exercise 7-4 or Exercise 7-5 that do each of the following at least once:

- Use a conditional test in the `while` statement to stop the loop.
- Use an `active` variable to control how long the loop runs.
- Use a `break` statement to exit the loop when the user enters a `'quit'` value.

7-7. Infinity: Write a loop that never ends, and run it. (To end the loop, press CTRL-C or close the window displaying the output.)

Using a while Loop with Lists and Dictionaries

So far, we've worked with only one piece of user information at a time. We received the user's input and then printed the input or a response to it. The next time through the `while` loop, we'd receive another input value and respond to that. But to keep track of many users and pieces of information, we'll need to use lists and dictionaries with our `while` loops.

A `for` loop is effective for looping through a list, but you shouldn't modify a list inside a `for` loop because Python will have trouble keeping track of the items in the list. To modify a list as you work through it, use a `while` loop. Using `while` loops with lists and dictionaries allows you to collect, store, and organize lots of input to examine and report on later.

Moving Items from One List to Another

Consider a list of newly registered but unverified users of a website. After we verify these users, how can we move them to a separate list of confirmed users? One way would be to use a `while` loop to pull users from the list of unconfirmed users as we verify them and then add them to a separate list of confirmed users. Here's what that code might look like:

confirmed_users.py

```
# Start with users that need to be verified,
#  and an empty list to hold confirmed users.
❶ unconfirmed_users = ['alice', 'brian', 'candace']
confirmed_users = []

# Verify each user until there are no more unconfirmed users.
#  Move each verified user into the list of confirmed users.
❷ while unconfirmed_users:
❸     current_user = unconfirmed_users.pop()

      print(f"Verifying user: {current_user.title()}")
❹     confirmed_users.append(current_user)
```

```
# Display all confirmed users.
print("\nThe following users have been confirmed:")
for confirmed_user in confirmed_users:
    print(confirmed_user.title())
```

We begin with a list of unconfirmed users at ❶ (Alice, Brian, and Candace) and an empty list to hold confirmed users. The `while` loop at ❷ runs as long as the list `unconfirmed_users` is not empty. Within this loop, the `pop()` function at ❸ removes unverified users one at a time from the end of `unconfirmed_users`. Here, because Candace is last in the `unconfirmed_users` list, her name will be the first to be removed, assigned to `current_user`, and added to the `confirmed_users` list at ❹. Next is Brian, then Alice.

We simulate confirming each user by printing a verification message and then adding them to the list of confirmed users. As the list of unconfirmed users shrinks, the list of confirmed users grows. When the list of unconfirmed users is empty, the loop stops and the list of confirmed users is printed:

```
Verifying user: Candace
Verifying user: Brian
Verifying user: Alice

The following users have been confirmed:
Candace
Brian
Alice
```

Removing All Instances of Specific Values from a List

In Chapter 3 we used `remove()` to remove a specific value from a list. The `remove()` function worked because the value we were interested in appeared only once in the list. But what if you want to remove all instances of a value from a list?

Say you have a list of pets with the value `'cat'` repeated several times. To remove all instances of that value, you can run a `while` loop until `'cat'` is no longer in the list, as shown here:

pets.py

```
pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

We start with a list containing multiple instances of `'cat'`. After printing the list, Python enters the `while` loop because it finds the value `'cat'` in the list

at least once. Once inside the loop, Python removes the first instance of `'cat'`, returns to the `while` line, and then reenters the loop when it finds that `'cat'` is still in the list. It removes each instance of `'cat'` until the value is no longer in the list, at which point Python exits the loop and prints the list again:

```
['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
['dog', 'dog', 'goldfish', 'rabbit']
```

Filling a Dictionary with User Input

You can prompt for as much input as you need in each pass through a `while` loop. Let's make a polling program in which each pass through the loop prompts for the participant's name and response. We'll store the data we gather in a dictionary, because we want to connect each response with a particular user:

mountain_poll.py

```
responses = {}

# Set a flag to indicate that polling is active.
polling_active = True

while polling_active:
    # Prompt for the person's name and response.
❶   name = input("\nWhat is your name? ")
    response = input("Which mountain would you like to climb someday? ")

    # Store the response in the dictionary.
❷   responses[name] = response

    # Find out if anyone else is going to take the poll.
❸   repeat = input("Would you like to let another person respond? (yes/ no) ")
    if repeat == 'no':
        polling_active = False

# Polling is complete. Show the results.
print("\n--- Poll Results ---")
❹ for name, response in responses.items():
    print(f"{name} would like to climb {response}.")
```

The program first defines an empty dictionary (`responses`) and sets a flag (`polling_active`) to indicate that polling is active. As long as `polling_active` is `True`, Python will run the code in the `while` loop.

Within the loop, the user is prompted to enter their name and a mountain they'd like to climb ❶. That information is stored in the `responses` dictionary ❷, and the user is asked whether or not to keep the poll running ❸. If they enter `yes`, the program enters the `while` loop again. If they enter `no`, the `polling_active` flag is set to `False`, the `while` loop stops running, and the final code block at ❹ displays the results of the poll.

If you run this program and enter sample responses, you should see output like this:

```
What is your name? Eric
Which mountain would you like to climb someday? Denali
Would you like to let another person respond? (yes/ no) yes

What is your name? Lynn
Which mountain would you like to climb someday? Devil's Thumb
Would you like to let another person respond? (yes/ no) no

--- Poll Results ---
Lynn would like to climb Devil's Thumb.
Eric would like to climb Denali.
```

TRY IT YOURSELF

7-8. Deli: Make a list called `sandwich_orders` and fill it with the names of various sandwiches. Then make an empty list called `finished_sandwiches`. Loop through the list of sandwich orders and print a message for each order, such as `I made your tuna sandwich`. As each sandwich is made, move it to the list of finished sandwiches. After all the sandwiches have been made, print a message listing each sandwich that was made.

7-9. No Pastrami: Using the list `sandwich_orders` from Exercise 7-8, make sure the sandwich `'pastrami'` appears in the list at least three times. Add code near the beginning of your program to print a message saying the deli has run out of pastrami, and then use a `while` loop to remove all occurrences of `'pastrami'` from `sandwich_orders`. Make sure no pastrami sandwiches end up in `finished_sandwiches`.

7-10. Dream Vacation: Write a program that polls users about their dream vacation. Write a prompt similar to *If you could visit one place in the world, where would you go?* Include a block of code that prints the results of the poll.

Summary

In this chapter you learned how to use `input()` to allow users to provide their own information in your programs. You learned to work with both text and numerical input and how to use `while` loops to make your programs run as long as your users want them to. You saw several ways to control the flow of a `while` loop by setting an `active` flag, using the `break` statement, and

using the `continue` statement. You learned how to use a `while` loop to move items from one list to another and how to remove all instances of a value from a list. You also learned how `while` loops can be used with dictionaries.

In Chapter 8 you'll learn about *functions*. Functions allow you to break your programs into small parts, each of which does one specific job. You can call a function as many times as you want, and you can store your functions in separate files. By using functions, you'll be able to write more efficient code that's easier to troubleshoot and maintain and that can be reused in many different programs.

8

FUNCTIONS

In this chapter you'll learn to write *functions*, which are named blocks of code that are designed to do one specific job.

When you want to perform a particular task that you've defined in a function, you *call* the function responsible for it. If you need to perform that task multiple times throughout your program, you don't need to type all the code for the same task again and again; you just call the function dedicated to handling that task, and the call tells Python to run the code inside the function. You'll find that using functions makes your programs easier to write, read, test, and fix.

In this chapter you'll also learn ways to pass information to functions. You'll learn how to write certain functions whose primary job is to display information and other functions designed to process data and return a value or set of values. Finally, you'll learn to store functions in separate files called *modules* to help organize your main program files.

Defining a Function

Here's a simple function named `greet_user()` that prints a greeting:

greeter.py

```
❶ def greet_user():
❷     """Display a simple greeting."""
❸     print("Hello!")

❹ greet_user()
```

This example shows the simplest structure of a function. The line at ❶ uses the keyword `def` to inform Python that you're defining a function. This is the *function definition*, which tells Python the name of the function and, if applicable, what kind of information the function needs to do its job. The parentheses hold that information. In this case, the name of the function is `greet_user()`, and it needs no information to do its job, so its parentheses are empty. (Even so, the parentheses are required.) Finally, the definition ends in a colon.

Any indented lines that follow `def greet_user():` make up the *body* of the function. The text at ❷ is a comment called a *docstring*, which describes what the function does. Docstrings are enclosed in triple quotes, which Python looks for when it generates documentation for the functions in your programs.

The line `print("Hello!")` ❸ is the only line of actual code in the body of this function, so `greet_user()` has just one job: `print("Hello!")`.

When you want to use this function, you call it. A *function call* tells Python to execute the code in the function. To *call* a function, you write the name of the function, followed by any necessary information in parentheses, as shown at ❹. Because no information is needed here, calling our function is as simple as entering `greet_user()`. As expected, it prints `Hello!`:

```
Hello!
```

Passing Information to a Function

Modified slightly, the function `greet_user()` can not only tell the user `Hello!` but also greet them by name. For the function to do this, you enter `username` in the parentheses of the function's definition at `def greet_user()`. By adding `username` here you allow the function to accept any value of `username` you specify. The function now expects you to provide a value for `username` each time you call it. When you call `greet_user()`, you can pass it a name, such as `'jesse'`, inside the parentheses:

```
def greet_user(username):
    """Display a simple greeting."""
    print(f"Hello, {username.title()}!")

greet_user('jesse')
```

Entering `greet_user('jesse')` calls `greet_user()` and gives the function the information it needs to execute the `print()` call. The function accepts the name you passed it and displays the greeting for that name:

```
Hello, Jesse!
```

Likewise, entering `greet_user('sarah')` calls `greet_user()`, passes it `'sarah'`, and prints `Hello, Sarah!` You can call `greet_user()` as often as you want and pass it any name you want to produce a predictable output every time.

Arguments and Parameters

In the preceding `greet_user()` function, we defined `greet_user()` to require a value for the variable `username`. Once we called the function and gave it the information (a person's name), it printed the right greeting.

The variable `username` in the definition of `greet_user()` is an example of a *parameter*, a piece of information the function needs to do its job. The value `'jesse'` in `greet_user('jesse')` is an example of an *argument*. An argument is a piece of information that's passed from a function call to a function. When we call the function, we place the value we want the function to work with in parentheses. In this case the argument `'jesse'` was passed to the function `greet_user()`, and the value was assigned to the parameter `username`.

NOTE *People sometimes speak of arguments and parameters interchangeably. Don't be surprised if you see the variables in a function definition referred to as arguments or the variables in a function call referred to as parameters.*

TRY IT YOURSELF

8-1. Message: Write a function called `display_message()` that prints one sentence telling everyone what you are learning about in this chapter. Call the function, and make sure the message displays correctly.

8-2. Favorite Book: Write a function called `favorite_book()` that accepts one parameter, `title`. The function should print a message, such as `One of my favorite books is Alice in Wonderland`. Call the function, making sure to include a book title as an argument in the function call.

Passing Arguments

Because a function definition can have multiple parameters, a function call may need multiple arguments. You can pass arguments to your functions in a number of ways. You can use *positional arguments*, which need to be in

the same order the parameters were written; *keyword arguments*, where each argument consists of a variable name and a value; and lists and dictionaries of values. Let's look at each of these in turn.

Positional Arguments

When you call a function, Python must match each argument in the function call with a parameter in the function definition. The simplest way to do this is based on the order of the arguments provided. Values matched up this way are called *positional arguments*.

To see how this works, consider a function that displays information about pets. The function tells us what kind of animal each pet is and the pet's name, as shown here:

pets.py

```
❶ def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

❷ describe_pet('hamster', 'harry')
```

The definition shows that this function needs a type of animal and the animal's name ❶. When we call `describe_pet()`, we need to provide an animal type and a name, in that order. For example, in the function call, the argument `'hamster'` is assigned to the parameter `animal_type` and the argument `'harry'` is assigned to the parameter `pet_name` ❷. In the function body, these two parameters are used to display information about the pet being described.

The output describes a hamster named Harry:

```
I have a hamster.
My hamster's name is Harry.
```

Multiple Function Calls

You can call a function as many times as needed. Describing a second, different pet requires just one more call to `describe_pet()`:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

In this second function call, we pass `describe_pet()` the arguments `'dog'` and `'willie'`. As with the previous set of arguments we used, Python matches `'dog'` with the parameter `animal_type` and `'willie'` with the parameter `pet_name`.

As before, the function does its job, but this time it prints values for a dog named Willie. Now we have a hamster named Harry and a dog named Willie:

```
I have a hamster.
My hamster's name is Harry.

I have a dog.
My dog's name is Willie.
```

Calling a function multiple times is a very efficient way to work. The code describing a pet is written once in the function. Then, anytime you want to describe a new pet, you call the function with the new pet's information. Even if the code for describing a pet were to expand to ten lines, you could still describe a new pet in just one line by calling the function again.

You can use as many positional arguments as you need in your functions. Python works through the arguments you provide when calling the function and matches each one with the corresponding parameter in the function's definition.

Order Matters in Positional Arguments

You can get unexpected results if you mix up the order of the arguments in a function call when using positional arguments:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('harry', 'hamster')
```

In this function call we list the name first and the type of animal second. Because the argument `'harry'` is listed first this time, that value is assigned to the parameter `animal_type`. Likewise, `'hamster'` is assigned to `pet_name`. Now we have a "harry" named "Hamster":

```
I have a harry.
My harry's name is Hamster.
```

If you get funny results like this, check to make sure the order of the arguments in your function call matches the order of the parameters in the function's definition.

Keyword Arguments

A *keyword argument* is a name-value pair that you pass to a function. You directly associate the name and the value within the argument, so when you pass the argument to the function, there's no confusion (you won't end up

with a harry named Hamster). Keyword arguments free you from having to worry about correctly ordering your arguments in the function call, and they clarify the role of each value in the function call.

Let's rewrite *pets.py* using keyword arguments to call `describe_pet()`:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(animal_type='hamster', pet_name='harry')
```

The function `describe_pet()` hasn't changed. But when we call the function, we explicitly tell Python which parameter each argument should be matched with. When Python reads the function call, it knows to assign the argument `'hamster'` to the parameter `animal_type` and the argument `'harry'` to `pet_name`. The output correctly shows that we have a hamster named Harry.

The order of keyword arguments doesn't matter because Python knows where each value should go. The following two function calls are equivalent:

```
describe_pet(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry', animal_type='hamster')
```

NOTE *When you use keyword arguments, be sure to use the exact names of the parameters in the function's definition.*

Default Values

When writing a function, you can define a *default value* for each parameter. If an argument for a parameter is provided in the function call, Python uses the argument value. If not, it uses the parameter's default value. So when you define a default value for a parameter, you can exclude the corresponding argument you'd usually write in the function call. Using default values can simplify your function calls and clarify the ways in which your functions are typically used.

For example, if you notice that most of the calls to `describe_pet()` are being used to describe dogs, you can set the default value of `animal_type` to `'dog'`. Now anyone calling `describe_pet()` for a dog can omit that information:

```
def describe_pet(pet_name, animal_type='dog'):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(pet_name='willie')
```

We changed the definition of `describe_pet()` to include a default value, `'dog'`, for `animal_type`. Now when the function is called with no `animal_type` specified, Python knows to use the value `'dog'` for this parameter:

```
I have a dog.
My dog's name is Willie.
```

Note that the order of the parameters in the function definition had to be changed. Because the default value makes it unnecessary to specify a type of animal as an argument, the only argument left in the function call is the pet's name. Python still interprets this as a positional argument, so if the function is called with just a pet's name, that argument will match up with the first parameter listed in the function's definition. This is the reason the first parameter needs to be `pet_name`.

The simplest way to use this function now is to provide just a dog's name in the function call:

```
describe_pet('willie')
```

This function call would have the same output as the previous example. The only argument provided is `'willie'`, so it is matched up with the first parameter in the definition, `pet_name`. Because no argument is provided for `animal_type`, Python uses the default value `'dog'`.

To describe an animal other than a dog, you could use a function call like this:

```
describe_pet(pet_name='harry', animal_type='hamster')
```

Because an explicit argument for `animal_type` is provided, Python will ignore the parameter's default value.

NOTE *When you use default values, any parameter with a default value needs to be listed after all the parameters that don't have default values. This allows Python to continue interpreting positional arguments correctly.*

Equivalent Function Calls

Because positional arguments, keyword arguments, and default values can all be used together, often you'll have several equivalent ways to call a function. Consider the following definition for `describe_pet()` with one default value provided:

```
def describe_pet(pet_name, animal_type='dog'):
```

With this definition, an argument always needs to be provided for `pet_name`, and this value can be provided using the positional or keyword

format. If the animal being described is not a dog, an argument for `animal_type` must be included in the call, and this argument can also be specified using the positional or keyword format.

All of the following calls would work for this function:

```
# A dog named Willie.
describe_pet('willie')
describe_pet(pet_name='willie')

# A hamster named Harry.
describe_pet('harry', 'hamster')
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster', pet_name='harry')
```

Each of these function calls would have the same output as the previous examples.

NOTE *It doesn't really matter which calling style you use. As long as your function calls produce the output you want, just use the style you find easiest to understand.*

Avoiding Argument Errors

When you start to use functions, don't be surprised if you encounter errors about unmatched arguments. Unmatched arguments occur when you provide fewer or more arguments than a function needs to do its work. For example, here's what happens if we try to call `describe_pet()` with no arguments:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet()
```

Python recognizes that some information is missing from the function call, and the traceback tells us that:

```
Traceback (most recent call last):
❶   File "pets.py", line 6, in <module>
❷     describe_pet()
❸ TypeError: describe_pet() missing 2 required positional arguments: 'animal_
type' and 'pet_name'
```

At ❶ the traceback tells us the location of the problem, allowing us to look back and see that something went wrong in our function call. At ❷ the offending function call is written out for us to see. At ❸ the traceback

tells us the call is missing two arguments and reports the names of the missing arguments. If this function were in a separate file, we could probably rewrite the call correctly without having to open that file and read the function code.

Python is helpful in that it reads the function's code for us and tells us the names of the arguments we need to provide. This is another motivation for giving your variables and functions descriptive names. If you do, Python's error messages will be more useful to you and anyone else who might use your code.

If you provide too many arguments, you should get a similar traceback that can help you correctly match your function call to the function definition.

TRY IT YOURSELF

8-3. T-Shirt: Write a function called `make_shirt()` that accepts a size and the text of a message that should be printed on the shirt. The function should print a sentence summarizing the size of the shirt and the message printed on it.

Call the function once using positional arguments to make a shirt. Call the function a second time using keyword arguments.

8-4. Large Shirts: Modify the `make_shirt()` function so that shirts are large by default with a message that reads *I love Python*. Make a large shirt and a medium shirt with the default message, and a shirt of any size with a different message.

8-5. Cities: Write a function called `describe_city()` that accepts the name of a city and its country. The function should print a simple sentence, such as `Reykjavik is in Iceland`. Give the parameter for the country a default value. Call your function for three different cities, at least one of which is not in the default country.

Return Values

A function doesn't always have to display its output directly. Instead, it can process some data and then return a value or set of values. The value the function returns is called a *return value*. The `return` statement takes a value from inside a function and sends it back to the line that called the function. Return values allow you to move much of your program's grunt work into functions, which can simplify the body of your program.

Returning a Simple Value

Let's look at a function that takes a first and last name, and returns a neatly formatted full name:

formatted_name.py

```
❶ def get_formatted_name(first_name, last_name):
      """Return a full name, neatly formatted."""
❷     full_name = f"{first_name} {last_name}"
❸     return full_name.title()

❹ musician = get_formatted_name('jimi', 'hendrix')
  print(musician)
```

The definition of `get_formatted_name()` takes as parameters a first and last name ❶. The function combines these two names, adds a space between them, and assigns the result to `full_name` ❷. The value of `full_name` is converted to title case, and then returned to the calling line at ❸.

When you call a function that returns a value, you need to provide a variable that the return value can be assigned to. In this case, the returned value is assigned to the variable `musician` at ❹. The output shows a neatly formatted name made up of the parts of a person's name:

```
Jimi Hendrix
```

This might seem like a lot of work to get a neatly formatted name when we could have just written:

```
print("Jimi Hendrix")
```

But when you consider working with a large program that needs to store many first and last names separately, functions like `get_formatted_name()` become very useful. You store first and last names separately and then call this function whenever you want to display a full name.

Making an Argument Optional

Sometimes it makes sense to make an argument optional so that people using the function can choose to provide extra information only if they want to. You can use default values to make an argument optional.

For example, say we want to expand `get_formatted_name()` to handle middle names as well. A first attempt to include middle names might look like this:

```
def get_formatted_name(first_name, middle_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {middle_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

This function works when given a first, middle, and last name. The function takes in all three parts of a name and then builds a string out of them. The function adds spaces where appropriate and converts the full name to title case:

```
John Lee Hooker
```

But middle names aren't always needed, and this function as written would not work if you tried to call it with only a first name and a last name. To make the middle name optional, we can give the `middle_name` argument an empty default value and ignore the argument unless the user provides a value. To make `get_formatted_name()` work without a middle name, we set the default value of `middle_name` to an empty string and move it to the end of the list of parameters:

```
❶ def get_formatted_name(first_name, last_name, middle_name=''):
      """Return a full name, neatly formatted."""
❷     if middle_name:
          full_name = f"{first_name} {middle_name} {last_name}"
❸     else:
          full_name = f"{first_name} {last_name}"
      return full_name.title()

  musician = get_formatted_name('jimi', 'hendrix')
  print(musician)

❹ musician = get_formatted_name('john', 'hooker', 'lee')
  print(musician)
```

In this example, the name is built from three possible parts. Because there's always a first and last name, these parameters are listed first in the function's definition. The middle name is optional, so it's listed last in the definition, and its default value is an empty string ❶.

In the body of the function, we check to see if a middle name has been provided. Python interprets non-empty strings as `True`, so `if middle_name` evaluates to `True` if a middle name argument is in the function call ❷. If a middle name is provided, the first, middle, and last names are combined to form a full name. This name is then changed to title case and returned to the function call line where it's assigned to the variable `musician` and printed. If no middle name is provided, the empty string fails the `if` test and the `else` block runs ❸. The full name is made with just a first and last name, and the formatted name is returned to the calling line where it's assigned to `musician` and printed.

Calling this function with a first and last name is straightforward. If we're using a middle name, however, we have to make sure the middle name is the last argument passed so Python will match up the positional arguments correctly ❹.

This modified version of our function works for people with just a first and last name, and it works for people who have a middle name as well:

```
Jimi Hendrix
John Lee Hooker
```

Optional values allow functions to handle a wide range of use cases while letting function calls remain as simple as possible.

Returning a Dictionary

A function can return any kind of value you need it to, including more complicated data structures like lists and dictionaries. For example, the following function takes in parts of a name and returns a dictionary representing a person:

person.py

```
def build_person(first_name, last_name):
    """Return a dictionary of information about a person."""
❶   person = {'first': first_name, 'last': last_name}
❷   return person

musician = build_person('jimi', 'hendrix')
❸ print(musician)
```

The function `build_person()` takes in a first and last name, and puts these values into a dictionary at ❶. The value of `first_name` is stored with the key `'first'`, and the value of `last_name` is stored with the key `'last'`. The entire dictionary representing the person is returned at ❷. The return value is printed at ❸ with the original two pieces of textual information now stored in a dictionary:

```
{'first': 'jimi', 'last': 'hendrix'}
```

This function takes in simple textual information and puts it into a more meaningful data structure that lets you work with the information beyond just printing it. The strings `'jimi'` and `'hendrix'` are now labeled as a first name and last name. You can easily extend this function to accept optional values like a middle name, an age, an occupation, or any other information you want to store about a person. For example, the following change allows you to store a person's age as well:

```
def build_person(first_name, last_name, age=None):
    """Return a dictionary of information about a person."""
    person = {'first': first_name, 'last': last_name}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', age=27)
print(musician)
```

We add a new optional parameter age to the function definition and assign the parameter the special value None, which is used when a variable has no specific value assigned to it. You can think of None as a placeholder value. In conditional tests, None evaluates to False. If the function call includes a value for age, that value is stored in the dictionary. This function always stores a person's name, but it can also be modified to store any other information you want about a person.

Using a Function with a while Loop

You can use functions with all the Python structures you've learned about so far. For example, let's use the get_formatted_name() function with a while loop to greet users more formally. Here's a first attempt at greeting people using their first and last names:

greeter.py

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

# This is an infinite loop!
while True:
❶   print("\nPlease tell me your name:")
    f_name = input("First name: ")
    l_name = input("Last name: ")

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

For this example, we use a simple version of get_formatted_name() that doesn't involve middle names. The while loop asks the user to enter their name, and we prompt for their first and last name separately ❶.

But there's one problem with this while loop: We haven't defined a quit condition. Where do you put a quit condition when you ask for a series of inputs? We want the user to be able to quit as easily as possible, so each prompt should offer a way to quit. The break statement offers a straightforward way to exit the loop at either prompt:

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

while True:
    print("\nPlease tell me your name:")
    print("(enter 'q' at any time to quit)")

    f_name = input("First name: ")
    if f_name == 'q':
        break
```

```
    l_name = input("Last name: ")
    if l_name == 'q':
        break

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

We add a message that informs the user how to quit, and then we break out of the loop if the user enters the quit value at either prompt. Now the program will continue greeting people until someone enters `'q'` for either name:

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: eric
Last name: matthes

Hello, Eric Matthes!

Please tell me your name:
(enter 'q' at any time to quit)
First name: q
```

TRY IT YOURSELF

8-6. City Names: Write a function called `city_country()` that takes in the name of a city and its country. The function should return a string formatted like this:

```
"Santiago, Chile"
```

Call your function with at least three city-country pairs, and print the values that are returned.

8-7. Album: Write a function called `make_album()` that builds a dictionary describing a music album. The function should take in an artist name and an album title, and it should return a dictionary containing these two pieces of information. Use the function to make three dictionaries representing different albums. Print each return value to show that the dictionaries are storing the album information correctly.

Use `None` to add an optional parameter to `make_album()` that allows you to store the number of songs on an album. If the calling line includes a value for the number of songs, add that value to the album's dictionary. Make at least one new function call that includes the number of songs on an album.

8-8. User Albums: Start with your program from Exercise 8-7. Write a `while` loop that allows users to enter an album's artist and title. Once you have that information, call `make_album()` with the user's input and print the dictionary that's created. Be sure to include a quit value in the `while` loop.

Passing a List

You'll often find it useful to pass a list to a function, whether it's a list of names, numbers, or more complex objects, such as dictionaries. When you pass a list to a function, the function gets direct access to the contents of the list. Let's use functions to make working with lists more efficient.

Say we have a list of users and want to print a greeting to each. The following example sends a list of names to a function called `greet_users()`, which greets each person in the list individually:

greet_users.py

```
def greet_users(names):
    """Print a simple greeting to each user in the list."""
    for name in names:
        msg = f"Hello, {name.title()}!"
        print(msg)

❶ usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

We define `greet_users()` so it expects a list of names, which it assigns to the parameter `names`. The function loops through the list it receives and prints a greeting to each user. At ❶ we define a list of users and then pass the list `usernames` to `greet_users()` in our function call:

```
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

This is the output we wanted. Every user sees a personalized greeting, and you can call the function any time you want to greet a specific set of users.

Modifying a List in a Function

When you pass a list to a function, the function can modify the list. Any changes made to the list inside the function's body are permanent, allowing you to work efficiently even when you're dealing with large amounts of data.

Consider a company that creates 3D printed models of designs that users submit. Designs that need to be printed are stored in a list, and after being printed they're moved to a separate list. The following code does this without using functions:

printing_models.py

```
# Start with some designs that need to be printed.
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

# Simulate printing each design, until none are left.
#  Move each design to completed_models after printing.
while unprinted_designs:
    current_design = unprinted_designs.pop()
```

```
    print(f"Printing model: {current_design}")
    completed_models.append(current_design)

# Display all completed models.
print("\nThe following models have been printed:")
for completed_model in completed_models:
    print(completed_model)
```

This program starts with a list of designs that need to be printed and an empty list called `completed_models` that each design will be moved to after it has been printed. As long as designs remain in `unprinted_designs`, the `while` loop simulates printing each design by removing a design from the end of the list, storing it in `current_design`, and displaying a message that the current design is being printed. It then adds the design to the list of completed models. When the loop is finished running, a list of the designs that have been printed is displayed:

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: phone case

The following models have been printed:
dodecahedron
robot pendant
phone case
```

We can reorganize this code by writing two functions, each of which does one specific job. Most of the code won't change; we're just making it more carefully structured. The first function will handle printing the designs, and the second will summarize the prints that have been made:

```
❶ def print_models(unprinted_designs, completed_models):
    """
    Simulate printing each design, until none are left.
    Move each design to completed_models after printing.
    """
    while unprinted_designs:
        current_design = unprinted_designs.pop()
        print(f"Printing model: {current_design}")
        completed_models.append(current_design)

❷ def show_completed_models(completed_models):
    """Show all the models that were printed."""
    print("\nThe following models have been printed:")
    for completed_model in completed_models:
        print(completed_model)

unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

At ❶ we define the function `print_models()` with two parameters: a list of designs that need to be printed and a list of completed models. Given these two lists, the function simulates printing each design by emptying the list of unprinted designs and filling up the list of completed models. At ❷ we define the function `show_completed_models()` with one parameter: the list of completed models. Given this list, `show_completed_models()` displays the name of each model that was printed.

This program has the same output as the version without functions, but the code is much more organized. The code that does most of the work has been moved to two separate functions, which makes the main part of the program easier to understand. Look at the body of the program to see how much easier it is to understand what this program is doing:

```
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

We set up a list of unprinted designs and an empty list that will hold the completed models. Then, because we've already defined our two functions, all we have to do is call them and pass them the right arguments. We call `print_models()` and pass it the two lists it needs; as expected, `print_models()` simulates printing the designs. Then we call `show_completed_models()` and pass it the list of completed models so it can report the models that have been printed. The descriptive function names allow others to read this code and understand it, even without comments.

This program is easier to extend and maintain than the version without functions. If we need to print more designs later on, we can simply call `print_models()` again. If we realize the printing code needs to be modified, we can change the code once, and our changes will take place everywhere the function is called. This technique is more efficient than having to update code separately in several places in the program.

This example also demonstrates the idea that every function should have one specific job. The first function prints each design, and the second displays the completed models. This is more beneficial than using one function to do both jobs. If you're writing a function and notice the function is doing too many different tasks, try to split the code into two functions. Remember that you can always call a function from another function, which can be helpful when splitting a complex task into a series of steps.

Preventing a Function from Modifying a List

Sometimes you'll want to prevent a function from modifying a list. For example, say that you start with a list of unprinted designs and write a function to move them to a list of completed models, as in the previous example. You may decide that even though you've printed all the designs, you want to keep the original list of unprinted designs for your records.

But because you moved all the design names out of `unprinted_designs`, the list is now empty, and the empty list is the only version you have; the original is gone. In this case, you can address this issue by passing the function a copy of the list, not the original. Any changes the function makes to the list will affect only the copy, leaving the original list intact.

You can send a copy of a list to a function like this:

```
function_name(list_name[:])
```

The slice notation `[:]` makes a copy of the list to send to the function. If we didn't want to empty the list of unprinted designs in *printing_models.py*, we could call `print_models()` like this:

```
print_models(unprinted_designs[:], completed_models)
```

The function `print_models()` can do its work because it still receives the names of all unprinted designs. But this time it uses a copy of the original unprinted designs list, not the actual `unprinted_designs` list. The list `completed_models` will fill up with the names of printed models like it did before, but the original list of unprinted designs will be unaffected by the function.

Even though you can preserve the contents of a list by passing a copy of it to your functions, you should pass the original list to functions unless you have a specific reason to pass a copy. It's more efficient for a function to work with an existing list to avoid using the time and memory needed to make a separate copy, especially when you're working with large lists.

TRY IT YOURSELF

8-9. Messages: Make a list containing a series of short text messages. Pass the list to a function called `show_messages()`, which prints each text message.

8-10. Sending Messages: Start with a copy of your program from Exercise 8-9. Write a function called `send_messages()` that prints each text message and moves each message to a new list called `sent_messages` as it's printed. After calling the function, print both of your lists to make sure the messages were moved correctly.

8-11. Archived Messages: Start with your work from Exercise 8-10. Call the function `send_messages()` with a copy of the list of messages. After calling the function, print both of your lists to show that the original list has retained its messages.

Passing an Arbitrary Number of Arguments

Sometimes you won't know ahead of time how many arguments a function needs to accept. Fortunately, Python allows a function to collect an arbitrary number of arguments from the calling statement.

For example, consider a function that builds a pizza. It needs to accept a number of toppings, but you can't know ahead of time how many toppings a person will want. The function in the following example has one parameter, `*toppings`, but this parameter collects as many arguments as the calling line provides:

pizza.py

```
def make_pizza(*toppings):
    """Print the list of toppings that have been requested."""
    print(toppings)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

The asterisk in the parameter name `*toppings` tells Python to make an empty tuple called `toppings` and pack whatever values it receives into this tuple. The `print()` call in the function body produces output showing that Python can handle a function call with one value and a call with three values. It treats the different calls similarly. Note that Python packs the arguments into a tuple, even if the function receives only one value:

```
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')
```

Now we can replace the `print()` call with a loop that runs through the list of toppings and describes the pizza being ordered:

```
def make_pizza(*toppings):
    """Summarize the pizza we are about to make."""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

The function responds appropriately, whether it receives one value or three values:

```
Making a pizza with the following toppings:
- pepperoni

Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

This syntax works no matter how many arguments the function receives.

Mixing Positional and Arbitrary Arguments

If you want a function to accept several different kinds of arguments, the parameter that accepts an arbitrary number of arguments must be placed last in the function definition. Python matches positional and keyword arguments first and then collects any remaining arguments in the final parameter.

For example, if the function needs to take in a size for the pizza, that parameter must come before the parameter `*toppings`:

```
def make_pizza(size, *toppings):
    """Summarize the pizza we are about to make."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

In the function definition, Python assigns the first value it receives to the parameter `size`. All other values that come after are stored in the tuple `toppings`. The function calls include an argument for the size first, followed by as many toppings as needed.

Now each pizza has a size and a number of toppings, and each piece of information is printed in the proper place, showing size first and toppings after:

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

NOTE *You'll often see the generic parameter name `*args`, which collects arbitrary positional arguments like this.*

Using Arbitrary Keyword Arguments

Sometimes you'll want to accept an arbitrary number of arguments, but you won't know ahead of time what kind of information will be passed to the function. In this case, you can write functions that accept as many key-value pairs as the calling statement provides. One example involves building user profiles: you know you'll get information about a user, but you're not sure what kind of information you'll receive. The function `build_profile()` in the

following example always takes in a first and last name, but it accepts an arbitrary number of keyword arguments as well:

user_profile.py

```
def build_profile(first, last, **user_info):
    """Build a dictionary containing everything we know about a user."""
❶   user_info['first_name'] = first
    user_info['last_name'] = last
    return user_info

user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='physics')
print(user_profile)
```

The definition of `build_profile()` expects a first and last name, and then it allows the user to pass in as many name-value pairs as they want. The double asterisks before the parameter `**user_info` cause Python to create an empty dictionary called `user_info` and pack whatever name-value pairs it receives into this dictionary. Within the function, you can access the key-value pairs in `user_info` just as you would for any dictionary.

In the body of `build_profile()`, we add the first and last names to the `user_info` dictionary because we'll always receive these two pieces of information from the user ❶, and they haven't been placed into the dictionary yet. Then we return the `user_info` dictionary to the function call line.

We call `build_profile()`, passing it the first name `'albert'`, the last name `'einstein'`, and the two key-value pairs `location='princeton'` and `field='physics'`. We assign the returned `profile` to `user_profile` and print `user_profile`:

```
{'location': 'princeton', 'field': 'physics',
'first_name': 'albert', 'last_name': 'einstein'}
```

The returned dictionary contains the user's first and last names and, in this case, the location and field of study as well. The function would work no matter how many additional key-value pairs are provided in the function call.

You can mix positional, keyword, and arbitrary values in many different ways when writing your own functions. It's useful to know that all these argument types exist because you'll see them often when you start reading other people's code. It takes practice to learn to use the different types correctly and to know when to use each type. For now, remember to use the simplest approach that gets the job done. As you progress you'll learn to use the most efficient approach each time.

NOTE *You'll often see the parameter name `**kwargs` used to collect non-specific keyword arguments.*

TRY IT YOURSELF

8-12. Sandwiches: Write a function that accepts a list of items a person wants on a sandwich. The function should have one parameter that collects as many items as the function call provides, and it should print a summary of the sandwich that's being ordered. Call the function three times, using a different number of arguments each time.

8-13. User Profile: Start with a copy of *user_profile.py* from page 149. Build a profile of yourself by calling `build_profile()`, using your first and last names and three other key-value pairs that describe you.

8-14. Cars: Write a function that stores information about a car in a dictionary. The function should always receive a manufacturer and a model name. It should then accept an arbitrary number of keyword arguments. Call the function with the required information and two other name-value pairs, such as a color or an optional feature. Your function should work for a call like this one:

```
car = make_car('subaru', 'outback', color='blue', tow_package=True)
```

Print the dictionary that's returned to make sure all the information was stored correctly.

Storing Your Functions in Modules

One advantage of functions is the way they separate blocks of code from your main program. By using descriptive names for your functions, your main program will be much easier to follow. You can go a step further by storing your functions in a separate file called a *module* and then *importing* that module into your main program. An `import` statement tells Python to make the code in a module available in the currently running program file.

Storing your functions in a separate file allows you to hide the details of your program's code and focus on its higher-level logic. It also allows you to reuse functions in many different programs. When you store your functions in separate files, you can share those files with other programmers without having to share your entire program. Knowing how to import functions also allows you to use libraries of functions that other programmers have written.

There are several ways to import a module, and I'll show you each of these briefly.

Importing an Entire Module

To start importing functions, we first need to create a module. A *module* is a file ending in *.py* that contains the code you want to import into your

program. Let's make a module that contains the function `make_pizza()`. To make this module, we'll remove everything from the file *pizza.py* except the function `make_pizza()`:

pizza.py

```
def make_pizza(size, *toppings):
    """Summarize the pizza we are about to make."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")
```

Now we'll make a separate file called *making_pizzas.py* in the same directory as *pizza.py*. This file imports the module we just created and then makes two calls to `make_pizza()`:

making_pizzas.py

```
import pizza

❶ pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

When Python reads this file, the line `import pizza` tells Python to open the file *pizza.py* and copy all the functions from it into this program. You don't actually see code being copied between files because Python copies the code behind the scenes just before the program runs. All you need to know is that any function defined in *pizza.py* will now be available in *making_pizzas.py*.

To call a function from an imported module, enter the name of the module you imported, `pizza`, followed by the name of the function, `make_pizza()`, separated by a dot ❶. This code produces the same output as the original program that didn't import a module:

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

This first approach to importing, in which you simply write `import` followed by the name of the module, makes every function from the module available in your program. If you use this kind of `import` statement to import an entire module named *module_name.py*, each function in the module is available through the following syntax:

```
module_name.function_name()
```

Importing Specific Functions

You can also import a specific function from a module. Here's the general syntax for this approach:

```
from module_name import function_name
```

You can import as many functions as you want from a module by separating each function's name with a comma:

```
from module_name import function_0, function_1, function_2
```

The *making_pizzas.py* example would look like this if we want to import just the function we're going to use:

```
from pizza import make_pizza

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

With this syntax, you don't need to use the dot notation when you call a function. Because we've explicitly imported the function `make_pizza()` in the `import` statement, we can call it by name when we use the function.

Using as to Give a Function an Alias

If the name of a function you're importing might conflict with an existing name in your program or if the function name is long, you can use a short, unique *alias*—an alternate name similar to a nickname for the function. You'll give the function this special nickname when you import the function.

Here we give the function `make_pizza()` an alias, `mp()`, by importing `make_pizza as mp`. The `as` keyword renames a function using the alias you provide:

```
from pizza import make_pizza as mp

mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

The `import` statement shown here renames the function `make_pizza()` to `mp()` in this program. Any time we want to call `make_pizza()` we can simply write `mp()` instead, and Python will run the code in `make_pizza()` while avoiding any confusion with another `make_pizza()` function you might have written in this program file.

The general syntax for providing an alias is:

```
from module_name import function_name as fn
```

Using as to Give a Module an Alias

You can also provide an alias for a module name. Giving a module a short alias, like `p` for `pizza`, allows you to call the module's functions more quickly. Calling `p.make_pizza()` is more concise than calling `pizza.make_pizza()`:

```
import pizza as p

p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

The module `pizza` is given the alias `p` in the `import` statement, but all of the module's functions retain their original names. Calling the functions by writing `p.make_pizza()` is not only more concise than writing `pizza.make_pizza()`, but also redirects your attention from the module name and allows you to focus on the descriptive names of its functions. These function names, which clearly tell you what each function does, are more important to the readability of your code than using the full module name.

The general syntax for this approach is:

```
import module_name as mn
```

Importing All Functions in a Module

You can tell Python to import every function in a module by using the asterisk (*) operator:

```
from pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

The asterisk in the `import` statement tells Python to copy every function from the module `pizza` into this program file. Because every function is imported, you can call each function by name without using the dot notation. However, it's best not to use this approach when you're working with larger modules that you didn't write: if the module has a function name that matches an existing name in your project, you can get some unexpected results. Python may see several functions or variables with the same name, and instead of importing all the functions separately, it will overwrite the functions.

The best approach is to import the function or functions you want, or import the entire module and use the dot notation. This leads to clear code that's easy to read and understand. I include this section so you'll recognize `import` statements like the following when you see them in other people's code:

```
from module_name import *
```

Styling Functions

You need to keep a few details in mind when you're styling functions. Functions should have descriptive names, and these names should use lowercase letters and underscores. Descriptive names help you and others understand what your code is trying to do. Module names should use these conventions as well.

Every function should have a comment that explains concisely what the function does. This comment should appear immediately after the function definition and use the docstring format. In a well-documented function, other programmers can use the function by reading only the description in the docstring. They should be able to trust that the code works as described, and as long as they know the name of the function, the arguments it needs, and the kind of value it returns, they should be able to use it in their programs.

If you specify a default value for a parameter, no spaces should be used on either side of the equal sign:

```
def function_name(parameter_0, parameter_1='default value')
```

The same convention should be used for keyword arguments in function calls:

```
function_name(value_0, parameter_1='value')
```

PEP 8 (*https://www.python.org/dev/peps/pep-0008/*) recommends that you limit lines of code to 79 characters so every line is visible in a reasonably sized editor window. If a set of parameters causes a function's definition to be longer than 79 characters, press ENTER after the opening parenthesis on the definition line. On the next line, press TAB twice to separate the list of arguments from the body of the function, which will only be indented one level.

Most editors automatically line up any additional lines of parameters to match the indentation you have established on the first line:

```
def function_name(
        parameter_0, parameter_1, parameter_2,
        parameter_3, parameter_4, parameter_5):
    function body...
```

If your program or module has more than one function, you can separate each by two blank lines to make it easier to see where one function ends and the next one begins.

All `import` statements should be written at the beginning of a file. The only exception is if you use comments at the beginning of your file to describe the overall program.

TRY IT YOURSELF

8-15. Printing Models: Put the functions for the example *printing_models.py* in a separate file called *printing_functions.py*. Write an `import` statement at the top of *printing_models.py*, and modify the file to use the imported functions.

8-16. Imports: Using a program you wrote that has one function in it, store that function in a separate file. Import the function into your main program file, and call the function using each of these approaches:

```
import module_name
from module_name import function_name
from module_name import function_name as fn
import module_name as mn
from module_name import *
```

8-17. Styling Functions: Choose any three programs you wrote for this chapter, and make sure they follow the styling guidelines described in this section.

Summary

In this chapter you learned how to write functions and to pass arguments so that your functions have access to the information they need to do their work. You learned how to use positional and keyword arguments, and how to accept an arbitrary number of arguments. You saw functions that display output and functions that return values. You learned how to use functions with lists, dictionaries, `if` statements, and `while` loops. You also saw how to store your functions in separate files called *modules*, so your program files will be simpler and easier to understand. Finally, you learned to style your functions so your programs will continue to be well-structured and as easy as possible for you and others to read.

One of your goals as a programmer should be to write simple code that does what you want it to, and functions help you do this. They allow you to write blocks of code and leave them alone once you know they work. When you know a function does its job correctly, you can trust that it will continue to work and move on to your next coding task.

Functions allow you to write code once and then reuse that code as many times as you want. When you need to run the code in a function, all you need to do is write a one-line call and the function does its job. When you need to modify a function's behavior, you only have to modify one block of code, and your change takes effect everywhere you've made a call to that function.

Using functions makes your programs easier to read, and good function names summarize what each part of a program does. Reading a series of function calls gives you a much quicker sense of what a program does than reading a long series of code blocks.

Functions also make your code easier to test and debug. When the bulk of your program's work is done by a set of functions, each of which has a specific job, it's much easier to test and maintain the code you've written. You can write a separate program that calls each function and tests whether each function works in all the situations it may encounter. When you do this, you can be confident that your functions will work properly each time you call them.

In Chapter 9 you'll learn to write classes. *Classes* combine functions and data into one neat package that can be used in flexible and efficient ways.

9

CLASSES

Object-oriented programming is one of the most effective approaches to writing software. In object-oriented programming you write *classes* that represent real-world things and situations, and you create *objects* based on these classes. When you write a class, you define the general behavior that a whole category of objects can have. When you create individual objects from the class, each object is automatically equipped with the general behavior; you can then give each object whatever unique traits you desire. You'll be amazed how well real-world situations can be modeled with object-oriented programming.

Making an object from a class is called *instantiation*, and you work with *instances* of a class. In this chapter you'll write classes and create instances of those classes. You'll specify the kind of information that can be stored in instances, and you'll define actions that can be taken with these instances. You'll also write classes that extend the functionality of existing classes, so

similar classes can share code efficiently. You'll store your classes in modules and import classes written by other programmers into your own program files.

Understanding object-oriented programming will help you see the world as a programmer does. It'll help you really know your code, not just what's happening line by line, but also the bigger concepts behind it. Knowing the logic behind classes will train you to think logically so you can write programs that effectively address almost any problem you encounter.

Classes also make life easier for you and the other programmers you'll work with as you take on increasingly complex challenges. When you and other programmers write code based on the same kind of logic, you'll be able to understand each other's work. Your programs will make sense to many collaborators, allowing everyone to accomplish more.

Creating and Using a Class

You can model almost anything using classes. Let's start by writing a simple class, `Dog`, that represents a dog—not one dog in particular, but any dog. What do we know about most pet dogs? Well, they all have a name and age. We also know that most dogs sit and roll over. Those two pieces of information (name and age) and those two behaviors (sit and roll over) will go in our `Dog` class because they're common to most dogs. This class will tell Python how to make an object representing a dog. After our class is written, we'll use it to make individual instances, each of which represents one specific dog.

Creating the Dog Class

Each instance created from the `Dog` class will store a `name` and an `age`, and we'll give each dog the ability to `sit()` and `roll_over()`:

dog.py

```
❶ class Dog:
❷     """A simple attempt to model a dog."""

❸     def __init__(self, name, age):
          """Initialize name and age attributes."""
❹         self.name = name
          self.age = age

❺     def sit(self):
          """Simulate a dog sitting in response to a command."""
          print(f"{self.name} is now sitting.")

      def roll_over(self):
          """Simulate rolling over in response to a command."""
          print(f"{self.name} rolled over!")
```

There's a lot to notice here, but don't worry. You'll see this structure throughout this chapter and have lots of time to get used to it. At ❶ we

define a class called Dog. By convention, capitalized names refer to classes in Python. There are no parentheses in the class definition because we're creating this class from scratch. At ❷ we write a docstring describing what this class does.

The __init__() Method

A function that's part of a class is a *method*. Everything you learned about functions applies to methods as well; the only practical difference for now is the way we'll call methods. The __init__() method at ❸ is a special method that Python runs automatically whenever we create a new instance based on the Dog class. This method has two leading underscores and two trailing underscores, a convention that helps prevent Python's default method names from conflicting with your method names. Make sure to use two underscores on each side of __init__(). If you use just one on each side, the method won't be called automatically when you use your class, which can result in errors that are difficult to identify.

We define the __init__() method to have three parameters: self, name, and age. The self parameter is required in the method definition, and it must come first before the other parameters. It must be included in the definition because when Python calls this method later (to create an instance of Dog), the method call will automatically pass the self argument. Every method call associated with an instance automatically passes self, which is a reference to the instance itself; it gives the individual instance access to the attributes and methods in the class. When we make an instance of Dog, Python will call the __init__() method from the Dog class. We'll pass Dog() a name and an age as arguments; self is passed automatically, so we don't need to pass it. Whenever we want to make an instance from the Dog class, we'll provide values for only the last two parameters, name and age.

The two variables defined at ❹ each have the prefix self. Any variable prefixed with self is available to every method in the class, and we'll also be able to access these variables through any instance created from the class. The line self.name = name takes the value associated with the parameter name and assigns it to the variable name, which is then attached to the instance being created. The same process happens with self.age = age. Variables that are accessible through instances like this are called *attributes*.

The Dog class has two other methods defined: sit() and roll_over() ❺. Because these methods don't need additional information to run, we just define them to have one parameter, self. The instances we create later will have access to these methods. In other words, they'll be able to sit and roll over. For now, sit() and roll_over() don't do much. They simply print a message saying the dog is sitting or rolling over. But the concept can be extended to realistic situations: if this class were part of an actual computer game, these methods would contain code to make an animated dog sit and roll over. If this class was written to control a robot, these methods would direct movements that cause a robotic dog to sit and roll over.

Making an Instance from a Class

Think of a class as a set of instructions for how to make an instance. The class `Dog` is a set of instructions that tells Python how to make individual instances representing specific dogs.

Let's make an instance representing a specific dog:

```
class Dog:
    --snip--

❶ my_dog = Dog('Willie', 6)

❷ print(f"My dog's name is {my_dog.name}.")
❸ print(f"My dog is {my_dog.age} years old.")
```

The `Dog` class we're using here is the one we just wrote in the previous example. At ❶ we tell Python to create a dog whose name is `'Willie'` and whose age is `6`. When Python reads this line, it calls the `__init__()` method in `Dog` with the arguments `'Willie'` and `6`. The `__init__()` method creates an instance representing this particular dog and sets the `name` and `age` attributes using the values we provided. Python then returns an instance representing this dog. We assign that instance to the variable `my_dog`. The naming convention is helpful here: we can usually assume that a capitalized name like `Dog` refers to a class, and a lowercase name like `my_dog` refers to a single instance created from a class.

Accessing Attributes

To access the attributes of an instance, you use dot notation. At ❷ we access the value of `my_dog`'s attribute `name` by writing:

```
my_dog.name
```

Dot notation is used often in Python. This syntax demonstrates how Python finds an attribute's value. Here Python looks at the instance `my_dog` and then finds the attribute `name` associated with `my_dog`. This is the same attribute referred to as `self.name` in the class `Dog`. At ❸ we use the same approach to work with the attribute `age`.

The output is a summary of what we know about `my_dog`:

```
My dog's name is Willie.
My dog is 6 years old.
```

Calling Methods

After we create an instance from the class `Dog`, we can use dot notation to call any method defined in `Dog`. Let's make our dog sit and roll over:

```
class Dog:
    --snip--
```

```
my_dog = Dog('Willie', 6)
my_dog.sit()
my_dog.roll_over()
```

To call a method, give the name of the instance (in this case, `my_dog`) and the method you want to call, separated by a dot. When Python reads `my_dog.sit()`, it looks for the method `sit()` in the class `Dog` and runs that code. Python interprets the line `my_dog.roll_over()` in the same way.

Now Willie does what we tell him to:

```
Willie is now sitting.
Willie rolled over!
```

This syntax is quite useful. When attributes and methods have been given appropriately descriptive names like `name`, `age`, `sit()`, and `roll_over()`, we can easily infer what a block of code, even one we've never seen before, is supposed to do.

Creating Multiple Instances

You can create as many instances from a class as you need. Let's create a second dog called `your_dog`:

```
class Dog:
    --snip--

my_dog = Dog('Willie', 6)
your_dog = Dog('Lucy', 3)

print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
my_dog.sit()

print(f"\nYour dog's name is {your_dog.name}.")
print(f"Your dog is {your_dog.age} years old.")
your_dog.sit()
```

In this example we create a dog named Willie and a dog named Lucy. Each dog is a separate instance with its own set of attributes, capable of the same set of actions:

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.

Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

Even if we used the same name and age for the second dog, Python would still create a separate instance from the `Dog` class. You can make

as many instances from one class as you need, as long as you give each instance a unique variable name or it occupies a unique spot in a list or dictionary.

TRY IT YOURSELF

9-1. Restaurant: Make a class called `Restaurant`. The `__init__()` method for `Restaurant` should store two attributes: a `restaurant_name` and a `cuisine_type`. Make a method called `describe_restaurant()` that prints these two pieces of information, and a method called `open_restaurant()` that prints a message indicating that the restaurant is open.

Make an instance called `restaurant` from your class. Print the two attributes individually, and then call both methods.

9-2. Three Restaurants: Start with your class from Exercise 9-1. Create three different instances from the class, and call `describe_restaurant()` for each instance.

9-3. Users: Make a class called `User`. Create two attributes called `first_name` and `last_name`, and then create several other attributes that are typically stored in a user profile. Make a method called `describe_user()` that prints a summary of the user's information. Make another method called `greet_user()` that prints a personalized greeting to the user.

Create several instances representing different users, and call both methods for each user.

Working with Classes and Instances

You can use classes to represent many real-world situations. Once you write a class, you'll spend most of your time working with instances created from that class. One of the first tasks you'll want to do is modify the attributes associated with a particular instance. You can modify the attributes of an instance directly or write methods that update attributes in specific ways.

The Car Class

Let's write a new class representing a car. Our class will store information about the kind of car we're working with, and it will have a method that summarizes this information:

car.py

```
class Car:
    """A simple attempt to represent a car."""

❶   def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
```

```
        self.model = model
        self.year = year

❷     def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f"{self.year} {self.manufacturer} {self.model}"
        return long_name.title()

❸ my_new_car = Car('audi', 'a4', 2019)
  print(my_new_car.get_descriptive_name())
```

At ❶ in the `Car` class, we define the `__init__()` method with the `self` parameter first, just like we did before with our `Dog` class. We also give it three other parameters: `make`, `model`, and `year`. The `__init__()` method takes in these parameters and assigns them to the attributes that will be associated with instances made from this class. When we make a new `Car` instance, we'll need to specify a make, model, and year for our instance.

At ❷ we define a method called `get_descriptive_name()` that puts a car's `year`, `make`, and `model` into one string neatly describing the car. This will spare us from having to print each attribute's value individually. To work with the attribute values in this method, we use `self.make`, `self.model`, and `self.year`. At ❸ we make an instance from the `Car` class and assign it to the variable `my_new_car`. Then we call `get_descriptive_name()` to show what kind of car we have:

```
2019 Audi A4
```

To make the class more interesting, let's add an attribute that changes over time. We'll add an attribute that stores the car's overall mileage.

Setting a Default Value for an Attribute

When an instance is created, attributes can be defined without being passed in as parameters. These attributes can be defined in the `__init__()` method, where they are assigned a default value.

Let's add an attribute called `odometer_reading` that always starts with a value of 0. We'll also add a method `read_odometer()` that helps us read each car's odometer:

```
class Car:

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
❶       self.odometer_reading = 0

    def get_descriptive_name(self):
        --snip--
```

```
❷     def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f"This car has {self.odometer_reading} miles on it.")

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

This time when Python calls the `__init__()` method to create a new instance, it stores the make, model, and year values as attributes like it did in the previous example. Then Python creates a new attribute called `odometer_reading` and sets its initial value to 0 ❶. We also have a new method called `read_odometer()` at ❷ that makes it easy to read a car's mileage.

Our car starts with a mileage of 0:

```
2019 Audi A4
This car has 0 miles on it.
```

Not many cars are sold with exactly 0 miles on the odometer, so we need a way to change the value of this attribute.

Modifying Attribute Values

You can change an attribute's value in three ways: you can change the value directly through an instance, set the value through a method, or increment the value (add a certain amount to it) through a method. Let's look at each of these approaches.

Modifying an Attribute's Value Directly

The simplest way to modify the value of an attribute is to access the attribute directly through an instance. Here we set the odometer reading to 23 directly:

```
class Car:
    --snip--

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

❶ my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

At ❶ we use dot notation to access the car's `odometer_reading` attribute and set its value directly. This line tells Python to take the instance `my_new_car`, find the attribute `odometer_reading` associated with it, and set the value of that attribute to 23:

```
2019 Audi A4
This car has 23 miles on it.
```

Sometimes you'll want to access attributes directly like this, but other times you'll want to write a method that updates the value for you.

Modifying an Attribute's Value Through a Method

It can be helpful to have methods that update certain attributes for you. Instead of accessing the attribute directly, you pass the new value to a method that handles the updating internally.

Here's an example showing a method called `update_odometer()`:

```
class Car:
    --snip--

❶   def update_odometer(self, mileage):
        """Set the odometer reading to the given value."""
        self.odometer_reading = mileage

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

❷ my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

The only modification to `Car` is the addition of `update_odometer()` at ❶. This method takes in a mileage value and assigns it to `self.odometer_reading`. At ❷ we call `update_odometer()` and give it `23` as an argument (corresponding to the `mileage` parameter in the method definition). It sets the odometer reading to 23, and `read_odometer()` prints the reading:

```
2019 Audi A4
This car has 23 miles on it.
```

We can extend the method `update_odometer()` to do additional work every time the odometer reading is modified. Let's add a little logic to make sure no one tries to roll back the odometer reading:

```
class Car:
    --snip--

    def update_odometer(self, mileage):
        """
        Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back.
        """
❶       if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
❷           print("You can't roll back an odometer!")
```

Now `update_odometer()` checks that the new reading makes sense before modifying the attribute. If the new mileage, `mileage`, is greater than or equal

to the existing mileage, `self.odometer_reading`, you can update the odometer reading to the new mileage ❶. If the new mileage is less than the existing mileage, you'll get a warning that you can't roll back an odometer ❷.

Incrementing an Attribute's Value Through a Method

Sometimes you'll want to increment an attribute's value by a certain amount rather than set an entirely new value. Say we buy a used car and put 100 miles on it between the time we buy it and the time we register it. Here's a method that allows us to pass this incremental amount and add that value to the odometer reading:

```
class Car:
    --snip--

    def update_odometer(self, mileage):
        --snip--

❶   def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles

❷ my_used_car = Car('subaru', 'outback', 2015)
  print(my_used_car.get_descriptive_name())

❸ my_used_car.update_odometer(23_500)
  my_used_car.read_odometer()

❹ my_used_car.increment_odometer(100)
  my_used_car.read_odometer()
```

The new method `increment_odometer()` at ❶ takes in a number of miles, and adds this value to `self.odometer_reading`. At ❷ we create a used car, `my_used_car`. We set its odometer to 23,500 by calling `update_odometer()` and passing it `23_500` at ❸. At ❹ we call `increment_odometer()` and pass it `100` to add the 100 miles that we drove between buying the car and registering it:

```
2015 Subaru Outback
This car has 23500 miles on it.
This car has 23600 miles on it.
```

You can easily modify this method to reject negative increments so no one uses this function to roll back an odometer.

NOTE *You can use methods like this to control how users of your program update values such as an odometer reading, but anyone with access to the program can set the odometer reading to any value by accessing the attribute directly. Effective security takes extreme attention to detail in addition to basic checks like those shown here.*

TRY IT YOURSELF

9-4. Number Served: Start with your program from Exercise 9-1 (page 162). Add an attribute called `number_served` with a default value of 0. Create an instance called `restaurant` from this class. Print the number of customers the restaurant has served, and then change this value and print it again.

Add a method called `set_number_served()` that lets you set the number of customers that have been served. Call this method with a new number and print the value again.

Add a method called `increment_number_served()` that lets you increment the number of customers who've been served. Call this method with any number you like that could represent how many customers were served in, say, a day of business.

9-5. Login Attempts: Add an attribute called `login_attempts` to your `User` class from Exercise 9-3 (page 162). Write a method called `increment_login_attempts()` that increments the value of `login_attempts` by 1. Write another method called `reset_login_attempts()` that resets the value of `login_attempts` to 0.

Make an instance of the `User` class and call `increment_login_attempts()` several times. Print the value of `login_attempts` to make sure it was incremented properly, and then call `reset_login_attempts()`. Print `login_attempts` again to make sure it was reset to 0.

Inheritance

You don't always have to start from scratch when writing a class. If the class you're writing is a specialized version of another class you wrote, you can use *inheritance*. When one class *inherits* from another, it takes on the attributes and methods of the first class. The original class is called the *parent class*, and the new class is the *child class*. The child class can inherit any or all of the attributes and methods of its parent class, but it's also free to define new attributes and methods of its own.

The __init__() Method for a Child Class

When you're writing a new class based on an existing class, you'll often want to call the `__init__()` method from the parent class. This will initialize any attributes that were defined in the parent `__init__()` method and make them available in the child class.

As an example, let's model an electric car. An electric car is just a specific kind of car, so we can base our new `ElectricCar` class on the `Car` class we wrote earlier. Then we'll only have to write code for the attributes and behavior specific to electric cars.

Let's start by making a simple version of the `ElectricCar` class, which does everything the `Car` class does:

electric_car.py

```
❶ class Car:
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = f"{self.year} {self.manufacturer} {self.model}"
        return long_name.title()

    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        self.odometer_reading += miles

❷ class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

❸     def __init__(self, make, model, year):
        """Initialize attributes of the parent class."""
❹         super().__init__(make, model, year)


❺ my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
```

At ❶ we start with `Car`. When you create a child class, the parent class must be part of the current file and must appear before the child class in the file. At ❷ we define the child class, `ElectricCar`. The name of the parent class must be included in parentheses in the definition of a child class. The `__init__()` method at ❸ takes in the information required to make a `Car` instance.

The `super()` function at ❹ is a special function that allows you to call a method from the parent class. This line tells Python to call the `__init__()` method from `Car`, which gives an `ElectricCar` instance all the attributes defined in that method. The name *super* comes from a convention of calling the parent class a *superclass* and the child class a *subclass*.

We test whether inheritance is working properly by trying to create an electric car with the same kind of information we'd provide when making a regular car. At ❺ we make an instance of the `ElectricCar` class and assign it to `my_tesla`. This line calls the `__init__()` method defined in `ElectricCar`, which in turn tells Python to call the `__init__()` method defined in the parent class `Car`. We provide the arguments `'tesla'`, `'model s'`, and `2019`.

Aside from `__init__()`, there are no attributes or methods yet that are particular to an electric car. At this point we're just making sure the electric car has the appropriate `Car` behaviors:

```
2019 Tesla Model S
```

The `ElectricCar` instance works just like an instance of `Car`, so now we can begin defining attributes and methods specific to electric cars.

Defining Attributes and Methods for the Child Class

Once you have a child class that inherits from a parent class, you can add any new attributes and methods necessary to differentiate the child class from the parent class.

Let's add an attribute that's specific to electric cars (a battery, for example) and a method to report on this attribute. We'll store the battery size and write a method that prints a description of the battery:

```
class Car:
    --snip--

class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
❶       self.battery_size = 75

❷   def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.describe_battery()
```

At ❶ we add a new attribute `self.battery_size` and set its initial value to, say, `75`. This attribute will be associated with all instances created from the `ElectricCar` class but won't be associated with any instances of `Car`. We also

add a method called `describe_battery()` that prints information about the battery at ❷. When we call this method, we get a description that is clearly specific to an electric car:

```
2019 Tesla Model S
This car has a 75-kWh battery.
```

There's no limit to how much you can specialize the `ElectricCar` class. You can add as many attributes and methods as you need to model an electric car to whatever degree of accuracy you need. An attribute or method that could belong to any car, rather than one that's specific to an electric car, should be added to the `Car` class instead of the `ElectricCar` class. Then anyone who uses the `Car` class will have that functionality available as well, and the `ElectricCar` class will only contain code for the information and behavior specific to electric vehicles.

Overriding Methods from the Parent Class

You can override any method from the parent class that doesn't fit what you're trying to model with the child class. To do this, you define a method in the child class with the same name as the method you want to override in the parent class. Python will disregard the parent class method and only pay attention to the method you define in the child class.

Say the class `Car` had a method called `fill_gas_tank()`. This method is meaningless for an all-electric vehicle, so you might want to override this method. Here's one way to do that:

```
class ElectricCar(Car):
    --snip--

    def fill_gas_tank(self):
        """Electric cars don't have gas tanks."""
        print("This car doesn't need a gas tank!")
```

Now if someone tries to call `fill_gas_tank()` with an electric car, Python will ignore the method `fill_gas_tank()` in `Car` and run this code instead. When you use inheritance, you can make your child classes retain what you need and override anything you don't need from the parent class.

Instances as Attributes

When modeling something from the real world in code, you may find that you're adding more and more detail to a class. You'll find that you have a growing list of attributes and methods and that your files are becoming lengthy. In these situations, you might recognize that part of one class can be written as a separate class. You can break your large class into smaller classes that work together.

For example, if we continue adding detail to the `ElectricCar` class, we might notice that we're adding many attributes and methods specific to

the car's battery. When we see this happening, we can stop and move those attributes and methods to a separate class called `Battery`. Then we can use a `Battery` instance as an attribute in the `ElectricCar` class:

```
class Car:
    --snip--

❶ class Battery:
    """A simple attempt to model a battery for an electric car."""

❷     def __init__(self, battery_size=75):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size

❸     def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")


class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
❹       self.battery = Battery()


my_tesla = ElectricCar('tesla', 'model s', 2019)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
```

At ❶ we define a new class called `Battery` that doesn't inherit from any other class. The `__init__()` method at ❷ has one parameter, `battery_size`, in addition to `self`. This is an optional parameter that sets the battery's size to 75 if no value is provided. The method `describe_battery()` has been moved to this class as well ❸.

In the `ElectricCar` class, we now add an attribute called `self.battery` ❹. This line tells Python to create a new instance of `Battery` (with a default size of 75, because we're not specifying a value) and assign that instance to the attribute `self.battery`. This will happen every time the `__init__()` method is called; any `ElectricCar` instance will now have a `Battery` instance created automatically.

We create an electric car and assign it to the variable `my_tesla`. When we want to describe the battery, we need to work through the car's `battery` attribute:

```
my_tesla.battery.describe_battery()
```

This line tells Python to look at the instance `my_tesla`, find its `battery` attribute, and call the method `describe_battery()` that's associated with the `Battery` instance stored in the attribute.

The output is identical to what we saw previously:

```
2019 Tesla Model S
This car has a 75-kWh battery.
```

This looks like a lot of extra work, but now we can describe the battery in as much detail as we want without cluttering the `ElectricCar` class. Let's add another method to `Battery` that reports the range of the car based on the battery size:

```
class Car:
    --snip--

class Battery:
    --snip--

❶     def get_range(self):
        """Print a statement about the range this battery provides."""
       if self.battery_size == 75:
            range = 260
        elif self.battery_size == 100:
            range = 315

        print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    --snip--

my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
❷ my_tesla.battery.get_range()
```

The new method `get_range()` at ❶ performs some simple analysis. If the battery's capacity is 75 kWh, `get_range()` sets the range to 260 miles, and if the capacity is 100 kWh, it sets the range to 315 miles. It then reports this value. When we want to use this method, we again have to call it through the car's `battery` attribute at ❷.

The output tells us the range of the car based on its battery size:

```
2019 Tesla Model S
This car has a 75-kWh battery.
This car can go about 260 miles on a full charge.
```

Modeling Real-World Objects

As you begin to model more complicated things like electric cars, you'll wrestle with interesting questions. Is the range of an electric car a property of the battery or of the car? If we're only describing one car, it's probably fine to maintain the association of the method `get_range()` with the `Battery` class. But if we're describing a manufacturer's entire line of cars, we probably want to move `get_range()` to the `ElectricCar` class. The `get_range()` method would still check the battery size before determining the range, but it would report a range specific to the kind of car it's associated with. Alternatively, we could maintain the association of the `get_range()` method with the battery but pass it a parameter such as `car_model`. The `get_range()` method would then report a range based on the battery size and car model.

This brings you to an interesting point in your growth as a programmer. When you wrestle with questions like these, you're thinking at a higher logical level rather than a syntax-focused level. You're thinking not about Python, but about how to represent the real world in code. When you reach this point, you'll realize there are often no right or wrong approaches to modeling real-world situations. Some approaches are more efficient than others, but it takes practice to find the most efficient representations. If your code is working as you want it to, you're doing well! Don't be discouraged if you find you're ripping apart your classes and rewriting them several times using different approaches. In the quest to write accurate, efficient code, everyone goes through this process.

TRY IT YOURSELF

9-6. Ice Cream Stand: An ice cream stand is a specific kind of restaurant. Write a class called `IceCreamStand` that inherits from the `Restaurant` class you wrote in Exercise 9-1 (page 162) or Exercise 9-4 (page 167). Either version of the class will work; just pick the one you like better. Add an attribute called `flavors` that stores a list of ice cream flavors. Write a method that displays these flavors. Create an instance of `IceCreamStand`, and call this method.

9-7. Admin: An administrator is a special kind of user. Write a class called `Admin` that inherits from the `User` class you wrote in Exercise 9-3 (page 162) or Exercise 9-5 (page 167). Add an attribute, `privileges`, that stores a list of strings like `"can add post"`, `"can delete post"`, `"can ban user"`, and so on. Write a method called `show_privileges()` that lists the administrator's set of privileges. Create an instance of `Admin`, and call your method.

9-8. Privileges: Write a separate `Privileges` class. The class should have one attribute, `privileges`, that stores a list of strings as described in Exercise 9-7. Move the `show_privileges()` method to this class. Make a `Privileges` instance as an attribute in the `Admin` class. Create a new instance of `Admin` and use your method to show its privileges.

(continued)

9-9. Battery Upgrade: Use the final version of *electric_car.py* from this section. Add a method to the `Battery` class called `upgrade_battery()`. This method should check the battery size and set the capacity to 100 if it isn't already. Make an electric car with a default battery size, call `get_range()` once, and then call `get_range()` a second time after upgrading the battery. You should see an increase in the car's range.

Importing Classes

As you add more functionality to your classes, your files can get long, even when you use inheritance properly. In keeping with the overall philosophy of Python, you'll want to keep your files as uncluttered as possible. To help, Python lets you store classes in modules and then import the classes you need into your main program.

Importing a Single Class

Let's create a module containing just the `Car` class. This brings up a subtle naming issue: we already have a file named *car.py* in this chapter, but this module should be named *car.py* because it contains code representing a car. We'll resolve this naming issue by storing the `Car` class in a module named *car.py*, replacing the *car.py* file we were previously using. From now on, any program that uses this module will need a more specific filename, such as *my_car.py*. Here's *car.py* with just the code from the class `Car`:

car.py

```
❶ """A class that can be used to represent a car."""

class Car:
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f"{self.year} {self.manufacturer} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f"This car has {self.odometer_reading} miles on it.")
```

```
    def update_odometer(self, mileage):
        """
        Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles
```

At ❶ we include a module-level docstring that briefly describes the contents of this module. You should write a docstring for each module you create.

Now we make a separate file called *my_car.py*. This file will import the `Car` class and then create an instance from that class:

my_car.py

```
❶ from car import Car

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

The `import` statement at ❶ tells Python to open the `car` module and import the class `Car`. Now we can use the `Car` class as if it were defined in this file. The output is the same as we saw earlier:

```
2019 Audi A4
This car has 23 miles on it.
```

Importing classes is an effective way to program. Picture how long this program file would be if the entire `Car` class were included. When you instead move the class to a module and import the module, you still get all the same functionality, but you keep your main program file clean and easy to read. You also store most of the logic in separate files; once your classes work as you want them to, you can leave those files alone and focus on the higher-level logic of your main program.

Storing Multiple Classes in a Module

You can store as many classes as you need in a single module, although each class in a module should be related somehow. The classes `Battery` and `ElectricCar` both help represent cars, so let's add them to the module *car.py*.

car.py

```python
"""A set of classes used to represent gas and electric cars."""

class Car:
    --snip--

class Battery:
    """A simple attempt to model a battery for an electric car."""

    def __init__(self, battery_size=70):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

    def get_range(self):
        """Print a statement about the range this battery provides."""
        if self.battery_size == 75:
            range = 260
        elif self.battery_size == 100:
            range = 315

        print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):
    """Models aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
        self.battery = Battery()
```

Now we can make a new file called *my_electric_car.py*, import the `ElectricCar` class, and make an electric car:

my_electric_car.py

```python
from car import ElectricCar

my_tesla = ElectricCar('tesla', 'model s', 2019)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

This has the same output we saw earlier, even though most of the logic is hidden away in a module:

```
2019 Tesla Model S
This car has a 75-kWh battery.
This car can go about 260 miles on a full charge.
```

Importing Multiple Classes from a Module

You can import as many classes as you need into a program file. If we want to make a regular car and an electric car in the same file, we need to import both classes, `Car` and `ElectricCar`:

my_cars.py

```
❶ from car import Car, ElectricCar

❷ my_beetle = Car('volkswagen', 'beetle', 2019)
  print(my_beetle.get_descriptive_name())

❸ my_tesla = ElectricCar('tesla', 'roadster', 2019)
  print(my_tesla.get_descriptive_name())
```

You import multiple classes from a module by separating each class with a comma ❶. Once you've imported the necessary classes, you're free to make as many instances of each class as you need.

In this example we make a regular Volkswagen Beetle at ❷ and an electric Tesla Roadster at ❸:

```
2019 Volkswagen Beetle
2019 Tesla Roadster
```

Importing an Entire Module

You can also import an entire module and then access the classes you need using dot notation. This approach is simple and results in code that is easy to read. Because every call that creates an instance of a class includes the module name, you won't have naming conflicts with any names used in the current file.

Here's what it looks like to import the entire `car` module and then create a regular car and an electric car:

my_cars.py

```
❶ import car

❷ my_beetle = car.Car('volkswagen', 'beetle', 2019)
  print(my_beetle.get_descriptive_name())

❸ my_tesla = car.ElectricCar('tesla', 'roadster', 2019)
  print(my_tesla.get_descriptive_name())
```

At ❶ we import the entire `car` module. We then access the classes we need through the *`module_name.ClassName`* syntax. At ❷ we again create a Volkswagen Beetle, and at ❸ we create a Tesla Roadster.

Importing All Classes from a Module

You can import every class from a module using the following syntax:

```
from module_name import *
```

This method is not recommended for two reasons. First, it's helpful to be able to read the `import` statements at the top of a file and get a clear sense of which classes a program uses. With this approach it's unclear which classes you're using from the module. This approach can also lead to confusion with names in the file. If you accidentally import a class with the same name as something else in your program file, you can create errors that are hard to diagnose. I show this here because even though it's not a recommended approach, you're likely to see it in other people's code at some point.

If you need to import many classes from a module, you're better off importing the entire module and using the *`module_name.ClassName`* syntax. You won't see all the classes used at the top of the file, but you'll see clearly where the module is used in the program. You'll also avoid the potential naming conflicts that can arise when you import every class in a module.

Importing a Module into a Module

Sometimes you'll want to spread out your classes over several modules to keep any one file from growing too large and avoid storing unrelated classes in the same module. When you store your classes in several modules, you may find that a class in one module depends on a class in another module. When this happens, you can import the required class into the first module.

For example, let's store the `Car` class in one module and the `ElectricCar` and `Battery` classes in a separate module. We'll make a new module called *electric_car.py*—replacing the *electric_car.py* file we created earlier—and copy just the `Battery` and `ElectricCar` classes into this file:

electric_car.py

```
"""A set of classes that can be used to represent electric cars."""

❶ from car import Car

class Battery:
    --snip--

class ElectricCar(Car):
    --snip--
```

The class `ElectricCar` needs access to its parent class `Car`, so we import `Car` directly into the module at ❶. If we forget this line, Python will raise an error when we try to import the `electric_car` module. We also need to update the `Car` module so it contains only the `Car` class:

car.py

```
"""A class that can be used to represent a car."""

class Car:
    --snip--
```

Now we can import from each module separately and create whatever kind of car we need:

my_cars.py

```
❶ from car import Car
from electric_car import ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2019)
print(my_beetle.get_descriptive_name())

my_tesla = ElectricCar('tesla', 'roadster', 2019)
print(my_tesla.get_descriptive_name())
```

At ❶ we import `Car` from its module, and `ElectricCar` from its module. We then create one regular car and one electric car. Both kinds of cars are created correctly:

```
2019 Volkswagen Beetle
2019 Tesla Roadster
```

Using Aliases

As you saw in Chapter 8, aliases can be quite helpful when using modules to organize your projects' code. You can use aliases when importing classes as well.

As an example, consider a program where you want to make a bunch of electric cars. It might get tedious to type (and read) `ElectricCar` over and over again. You can give `ElectricCar` an alias in the import statement:

```
from electric_car import ElectricCar as EC
```

Now you can use this alias whenever you want to make an electric car:

```
my_tesla = EC('tesla', 'roadster', 2019)
```

Finding Your Own Workflow

As you can see, Python gives you many options for how to structure code in a large project. It's important to know all these possibilities so you can determine the best ways to organize your projects as well as understand other people's projects.

When you're starting out, keep your code structure simple. Try doing everything in one file and moving your classes to separate modules once everything is working. If you like how modules and files interact, try storing your classes in modules when you start a project. Find an approach that lets you write code that works, and go from there.

TRY IT YOURSELF

9-10. Imported Restaurant: Using your latest `Restaurant` class, store it in a module. Make a separate file that imports `Restaurant`. Make a `Restaurant` instance, and call one of `Restaurant`'s methods to show that the `import` statement is working properly.

9-11. Imported Admin: Start with your work from Exercise 9-8 (page 173). Store the classes `User`, `Privileges`, and `Admin` in one module. Create a separate file, make an `Admin` instance, and call `show_privileges()` to show that everything is working correctly.

9-12. Multiple Modules: Store the `User` class in one module, and store the `Privileges` and `Admin` classes in a separate module. In a separate file, create an `Admin` instance and call `show_privileges()` to show that everything is still working correctly.

The Python Standard Library

The *Python standard library* is a set of modules included with every Python installation. Now that you have a basic understanding of how functions and classes work, you can start to use modules like these that other programmers have written. You can use any function or class in the standard library by including a simple `import` statement at the top of your file. Let's look at one module, `random`, which can be useful in modeling many real-world situations.

One interesting function from the random module is `randint()`. This function takes two integer arguments and returns a randomly selected integer between (and including) those numbers.

Here's how to generate a random number between 1 and 6:

```
>>> from random import randint
>>> randint(1, 6)
3
```

Another useful function is `choice()`. This function takes in a list or tuple and returns a randomly chosen element:

```
>>> from random import choice
>>> players = ['charles', 'martina', 'michael', 'florence', 'eli']
>>> first_up = choice(players)
>>> first_up
'florence'
```

The `random` module shouldn't be used when building security-related applications, but it's good enough for many fun and interesting projects.

NOTE *You can also download modules from external sources. You'll see a number of these examples in Part II, where we'll need external modules to complete each project.*

TRY IT YOURSELF

9-13. Dice: Make a class `Die` with one attribute called `sides`, which has a default value of 6. Write a method called `roll_die()` that prints a random number between 1 and the number of sides the die has. Make a 6-sided die and roll it 10 times.

Make a 10-sided die and a 20-sided die. Roll each die 10 times.

9-14. Lottery: Make a list or tuple containing a series of 10 numbers and five letters. Randomly select four numbers or letters from the list and print a message saying that any ticket matching these four numbers or letters wins a prize.

9-15. Lottery Analysis: You can use a loop to see how hard it might be to win the kind of lottery you just modeled. Make a list or tuple called `my_ticket`. Write a loop that keeps pulling numbers until your ticket wins. Print a message reporting how many times the loop had to run to give you a winning ticket.

9-16. Python Module of the Week: One excellent resource for exploring the Python standard library is a site called *Python Module of the Week*. Go to *https://pymotw.com/* and look at the table of contents. Find a module that looks interesting to you and read about it, perhaps starting with the `random` module.

Styling Classes

A few styling issues related to classes are worth clarifying, especially as your programs become more complicated.

Class names should be written in *CamelCase*. To do this, capitalize the first letter of each word in the name, and don't use underscores. Instance and module names should be written in lowercase with underscores between words.

Every class should have a docstring immediately following the class definition. The docstring should be a brief description of what the class does, and you should follow the same formatting conventions you used for writing docstrings in functions. Each module should also have a docstring describing what the classes in a module can be used for.

You can use blank lines to organize code, but don't use them excessively. Within a class you can use one blank line between methods, and within a module you can use two blank lines to separate classes.

If you need to import a module from the standard library and a module that you wrote, place the import statement for the standard library module first. Then add a blank line and the import statement for the module you wrote. In programs with multiple import statements, this convention makes it easier to see where the different modules used in the program come from.

Summary

In this chapter you learned how to write your own classes. You learned how to store information in a class using attributes and how to write methods that give your classes the behavior they need. You learned to write `__init__()` methods that create instances from your classes with exactly the attributes you want. You saw how to modify the attributes of an instance directly and through methods. You learned that inheritance can simplify the creation of classes that are related to each other, and you learned to use instances of one class as attributes in another class to keep each class simple.

You saw how storing classes in modules and importing classes you need into the files where they'll be used can keep your projects organized. You started learning about the Python standard library, and you saw an example based on the `random` module. Finally, you learned to style your classes using Python conventions.

In Chapter 10 you'll learn to work with files so you can save the work you've done in a program and the work you've allowed users to do. You'll also learn about *exceptions*, a special Python class designed to help you respond to errors when they arise.

10

FILES AND EXCEPTIONS

Now that you've mastered the basic skills you need to write organized programs that are easy to use, it's time to think about making your programs even more relevant and usable. In this chapter you'll learn to work with files so your programs can quickly analyze lots of data. You'll learn to handle errors so your programs don't crash when they encounter unexpected situations. You'll learn about *exceptions*, which are special objects Python creates to manage errors that arise while a program is running. You'll also learn about the `json` module, which allows you to save user data so it isn't lost when your program stops running.

Learning to work with files and save data will make your programs easier for people to use. Users will be able to choose what data to enter and when to enter it. People can run your program, do some work, and then close the program and pick up where they left off later. Learning to handle exceptions will help you deal with situations in which files don't exist and deal with other problems that can cause your programs to crash. This will make your programs more robust when they encounter bad data, whether

it comes from innocent mistakes or from malicious attempts to break your programs. With the skills you'll learn in this chapter, you'll make your programs more applicable, usable, and stable.

Reading from a File

An incredible amount of data is available in text files. Text files can contain weather data, traffic data, socioeconomic data, literary works, and more. Reading from a file is particularly useful in data analysis applications, but it's also applicable to any situation in which you want to analyze or modify information stored in a file. For example, you can write a program that reads in the contents of a text file and rewrites the file with formatting that allows a browser to display it.

When you want to work with the information in a text file, the first step is to read the file into memory. You can read the entire contents of a file, or you can work through the file one line at a time.

Reading an Entire File

To begin, we need a file with a few lines of text in it. Let's start with a file that contains *pi* to 30 decimal places, with 10 decimal places per line:

pi_digits.txt

```
3.1415926535
  8979323846
  2643383279
```

To try the following examples yourself, you can enter these lines in an editor and save the file as *pi_digits.txt*, or you can download the file from the book's resources through *https://nostarch.com/pythoncrashcourse2e/*. Save the file in the same directory where you'll store this chapter's programs.

Here's a program that opens this file, reads it, and prints the contents of the file to the screen:

file_reader.py

```
with open('pi_digits.txt') as file_object:
    contents = file_object.read()
print(contents)
```

The first line of this program has a lot going on. Let's start by looking at the `open()` function. To do any work with a file, even just printing its contents, you first need to *open* the file to access it. The `open()` function needs one argument: the name of the file you want to open. Python looks for this file in the directory where the program that's currently being executed is stored. In this example, *file_reader.py* is currently running, so Python looks for *pi_digits.txt* in the directory where *file_reader.py* is stored. The `open()` function returns an object representing the file. Here, `open('pi_digits.txt')` returns an object representing *pi_digits.txt*. Python assigns this object to `file_object`, which we'll work with later in the program.

The keyword `with` closes the file once access to it is no longer needed. Notice how we call `open()` in this program but not `close()`. You could open and close the file by calling `open()` and `close()`, but if a bug in your program prevents the `close()` method from being executed, the file may never close. This may seem trivial, but improperly closed files can cause data to be lost or corrupted. And if you call `close()` too early in your program, you'll find yourself trying to work with a *closed* file (a file you can't access), which leads to more errors. It's not always easy to know exactly when you should close a file, but with the structure shown here, Python will figure that out for you. All you have to do is open the file and work with it as desired, trusting that Python will close it automatically when the `with` block finishes execution.

Once we have a file object representing *pi_digits.txt*, we use the `read()` method in the second line of our program to read the entire contents of the file and store it as one long string in `contents`. When we print the value of `contents`, we get the entire text file back:

```
3.1415926535
  8979323846
  2643383279
```

The only difference between this output and the original file is the extra blank line at the end of the output. The blank line appears because `read()` returns an empty string when it reaches the end of the file; this empty string shows up as a blank line. If you want to remove the extra blank line, you can use `rstrip()` in the call to `print()`:

```
with open('pi_digits.txt') as file_object:
    contents = file_object.read()
    print(contents.rstrip())
```

Recall that Python's `rstrip()` method removes, or strips, any whitespace characters from the right side of a string. Now the output matches the contents of the original file exactly:

```
3.1415926535
  8979323846
  2643383279
```

File Paths

When you pass a simple filename like *pi_digits.txt* to the `open()` function, Python looks in the directory where the file that's currently being executed (that is, your *.py* program file) is stored.

Sometimes, depending on how you organize your work, the file you want to open won't be in the same directory as your program file. For example, you might store your program files in a folder called

python_work; inside *python_work*, you might have another folder called *text_files* to distinguish your program files from the text files they're manipulating. Even though *text_files* is in *python_work*, just passing `open()` the name of a file in *text_files* won't work, because Python will only look in *python_work* and stop there; it won't go on and look in *text_files*. To get Python to open files from a directory other than the one where your program file is stored, you need to provide a *file path*, which tells Python to look in a specific location on your system.

Because *text_files* is inside *python_work*, you could use a relative file path to open a file from *text_files*. A *relative file path* tells Python to look for a given location relative to the directory where the currently running program file is stored. For example, you'd write:

```
with open('text_files/filename.txt') as file_object:
```

This line tells Python to look for the desired *.txt* file in the folder *text_files* and assumes that *text_files* is located inside *python_work* (which it is).

NOTE *Windows systems use a backslash (\) instead of a forward slash (/) when displaying file paths, but you can still use forward slashes in your code.*

You can also tell Python exactly where the file is on your computer regardless of where the program that's being executed is stored. This is called an *absolute file path*. You use an absolute path if a relative path doesn't work. For instance, if you've put *text_files* in some folder other than *python_work*—say, a folder called *other_files*—then just passing `open()` the path `'text_files/filename.txt'` won't work because Python will only look for that location inside *python_work*. You'll need to write out a full path to clarify where you want Python to look.

Absolute paths are usually longer than relative paths, so it's helpful to assign them to a variable and then pass that variable to `open()`:

```
file_path = '/home/ehmatthes/other_files/text_files/filename.txt'
with open(file_path) as file_object:
```

Using absolute paths, you can read files from any location on your system. For now it's easiest to store files in the same directory as your program files or in a folder such as *text_files* within the directory that stores your program files.

NOTE *If you try to use backslashes in a file path, you'll get an error because the backslash is used to escape characters in strings. For example, in the path* `"C:\path\to\file.txt"`*, the sequence* `\t` *is interpreted as a tab. If you need to use backslashes, you can escape each one in the path, like this:* `"C:\\path\\to\\file.txt"`*.*

Reading Line by Line

When you're reading a file, you'll often want to examine each line of the file. You might be looking for certain information in the file, or you might want to modify the text in the file in some way. For example, you might want to read through a file of weather data and work with any line that includes the word *sunny* in the description of that day's weather. In a news report, you might look for any line with the tag `<headline>` and rewrite that line with a specific kind of formatting.

You can use a `for` loop on the file object to examine each line from a file one at a time:

file_reader.py

```
❶ filename = 'pi_digits.txt'

❷ with open(filename) as file_object:
❸     for line in file_object:
          print(line)
```

At ❶ we assign the name of the file we're reading from to the variable `filename`. This is a common convention when working with files. Because the variable `filename` doesn't represent the actual file—it's just a string telling Python where to find the file—you can easily swap out `'pi_digits.txt'` for the name of another file you want to work with. After we call `open()`, an object representing the file and its contents is assigned to the variable `file_object` ❷. We again use the `with` syntax to let Python open and close the file properly. To examine the file's contents, we work through each line in the file by looping over the file object ❸.

When we print each line, we find even more blank lines:

```
3.1415926535

  8979323846

  2643383279
```

These blank lines appear because an invisible newline character is at the end of each line in the text file. The `print` function adds its own newline each time we call it, so we end up with two newline characters at the end of each line: one from the file and one from `print()`. Using `rstrip()` on each line in the `print()` call eliminates these extra blank lines:

```
filename = 'pi_digits.txt'

with open(filename) as file_object:
    for line in file_object:
        print(line.rstrip())
```

Now the output matches the contents of the file once again:

```
3.1415926535
  8979323846
  2643383279
```

Making a List of Lines from a File

When you use `with`, the file object returned by `open()` is only available inside the `with` block that contains it. If you want to retain access to a file's contents outside the `with` block, you can store the file's lines in a list inside the block and then work with that list. You can process parts of the file immediately and postpone some processing for later in the program.

The following example stores the lines of *pi_digits.txt* in a list inside the `with` block and then prints the lines outside the `with` block:

```
filename = 'pi_digits.txt'

with open(filename) as file_object:
❶     lines = file_object.readlines()

❷ for line in lines:
    print(line.rstrip())
```

At ❶ the `readlines()` method takes each line from the file and stores it in a list. This list is then assigned to `lines`, which we can continue to work with after the `with` block ends. At ❷ we use a simple `for` loop to print each line from `lines`. Because each item in `lines` corresponds to each line in the file, the output matches the contents of the file exactly.

Working with a File's Contents

After you've read a file into memory, you can do whatever you want with that data, so let's briefly explore the digits of *pi*. First, we'll attempt to build a single string containing all the digits in the file with no whitespace in it:

pi_string.py

```
filename = 'pi_digits.txt'

with open(filename) as file_object:
    lines = file_object.readlines()

❶ pi_string = ''
❷ for line in lines:
    pi_string += line.rstrip()

❸ print(pi_string)
  print(len(pi_string))
```

We start by opening the file and storing each line of digits in a list, just as we did in the previous example. At ❶ we create a variable, `pi_string`, to hold the digits of *pi*. We then create a loop that adds each line of digits to `pi_string` and removes the newline character from each line ❷. At ❸ we print this string and also show how long the string is:

```
3.1415926535  8979323846  2643383279
36
```

The variable `pi_string` contains the whitespace that was on the left side of the digits in each line, but we can get rid of that by using `strip()` instead of `rstrip()`:

```
--snip--
for line in lines:
    pi_string += line.strip()

print(pi_string)
print(len(pi_string))
```

Now we have a string containing *pi* to 30 decimal places. The string is 32 characters long because it also includes the leading 3 and a decimal point:

```
3.14159265358979323846264338327 9
32
```

NOTE *When Python reads from a text file, it interprets all text in the file as a string. If you read in a number and want to work with that value in a numerical context, you'll have to convert it to an integer using the `int()` function or convert it to a float using the `float()` function.*

Large Files: One Million Digits

So far we've focused on analyzing a text file that contains only three lines, but the code in these examples would work just as well on much larger files. If we start with a text file that contains *pi* to 1,000,000 decimal places instead of just 30, we can create a single string containing all these digits. We don't need to change our program at all except to pass it a different file. We'll also print just the first 50 decimal places, so we don't have to watch a million digits scroll by in the terminal:

pi_string.py

```
filename = 'pi_million_digits.txt'

with open(filename) as file_object:
    lines = file_object.readlines()
```

```
pi_string = ''
for line in lines:
    pi_string += line.strip()

print(f"{pi_string[:52]}...")
print(len(pi_string))
```

The output shows that we do indeed have a string containing *pi* to 1,000,000 decimal places:

```
3.14159265358979323846264338327950288419716939937510...
1000002
```

Python has no inherent limit to how much data you can work with; you can work with as much data as your system's memory can handle.

NOTE *To run this program (and many of the examples that follow), you'll need to download the resources available at* https://nostarch.com/pythoncrashcourse2e/.

Is Your Birthday Contained in Pi?

I've always been curious to know if my birthday appears anywhere in the digits of *pi*. Let's use the program we just wrote to find out if someone's birthday appears anywhere in the first million digits of *pi*. We can do this by expressing each birthday as a string of digits and seeing if that string appears anywhere in `pi_string`:

```
--snip--
for line in lines:
    pi_string += line.strip()

❶ birthday = input("Enter your birthday, in the form mmddyy: ")
❷ if birthday in pi_string:
    print("Your birthday appears in the first million digits of pi!")
else:
    print("Your birthday does not appear in the first million digits of pi.")
```

At ❶ we prompt for the user's birthday, and then at ❷ we check if that string is in `pi_string`. Let's try it:

```
Enter your birthdate, in the form mmddyy: 120372
Your birthday appears in the first million digits of pi!
```

My birthday does appear in the digits of *pi*! Once you've read from a file, you can analyze its contents in just about any way you can imagine.

TRY IT YOURSELF

10-1. Learning Python: Open a blank file in your text editor and write a few lines summarizing what you've learned about Python so far. Start each line with the phrase *In Python you can. . . .* Save the file as *learning_python.txt* in the same directory as your exercises from this chapter. Write a program that reads the file and prints what you wrote three times. Print the contents once by reading in the entire file, once by looping over the file object, and once by storing the lines in a list and then working with them outside the `with` block.

10-2. Learning C: You can use the `replace()` method to replace any word in a string with a different word. Here's a quick example showing how to replace `'dog'` with `'cat'` in a sentence:

```
>>> message = "I really like dogs."
>>> message.replace('dog', 'cat')
'I really like cats.'
```

Read in each line from the file you just created, *learning_python.txt*, and replace the word *Python* with the name of another language, such as *C*. Print each modified line to the screen.

Writing to a File

One of the simplest ways to save data is to write it to a file. When you write text to a file, the output will still be available after you close the terminal containing your program's output. You can examine output after a program finishes running, and you can share the output files with others as well. You can also write programs that read the text back into memory and work with it again later.

Writing to an Empty File

To write text to a file, you need to call `open()` with a second argument telling Python that you want to write to the file. To see how this works, let's write a simple message and store it in a file instead of printing it to the screen:

write_message.py

```
filename = 'programming.txt'

❶ with open(filename, 'w') as file_object:
❷     file_object.write("I love programming.")
```

The call to `open()` in this example has two arguments ❶. The first argument is still the name of the file we want to open. The second argument, `'w'`, tells Python that we want to open the file in *write mode*. You can open a file

in *read mode* (`'r'`), *write mode* (`'w'`), *append mode* (`'a'`), or a mode that allows you to read and write to the file (`'r+'`). If you omit the mode argument, Python opens the file in read-only mode by default.

The `open()` function automatically creates the file you're writing to if it doesn't already exist. However, be careful opening a file in write mode (`'w'`) because if the file does exist, Python will erase the contents of the file before returning the file object.

At ❷ we use the `write()` method on the file object to write a string to the file. This program has no terminal output, but if you open the file *programming.txt*, you'll see one line:

programming.txt

```
I love programming.
```

This file behaves like any other file on your computer. You can open it, write new text in it, copy from it, paste to it, and so forth.

NOTE *Python can only write strings to a text file. If you want to store numerical data in a text file, you'll have to convert the data to string format first using the `str()` function.*

Writing Multiple Lines

The `write()` function doesn't add any newlines to the text you write. So if you write more than one line without including newline characters, your file may not look the way you want it to:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
    file_object.write("I love creating new games.")
```

If you open *programming.txt*, you'll see the two lines squished together:

```
I love programming.I love creating new games.
```

Including newlines in your calls to `write()` makes each string appear on its own line:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:
    file_object.write("I love programming.\n")
    file_object.write("I love creating new games.\n")
```

The output now appears on separate lines:

```
I love programming.
I love creating new games.
```

You can also use spaces, tab characters, and blank lines to format your output, just as you've been doing with terminal-based output.

Appending to a File

If you want to add content to a file instead of writing over existing content, you can open the file in *append mode*. When you open a file in append mode, Python doesn't erase the contents of the file before returning the file object. Any lines you write to the file will be added at the end of the file. If the file doesn't exist yet, Python will create an empty file for you.

Let's modify *write_message.py* by adding some new reasons we love programming to the existing file *programming.txt*:

write_message.py

```
filename = 'programming.txt'

❶ with open(filename, 'a') as file_object:
❷     file_object.write("I also love finding meaning in large datasets.\n")
      file_object.write("I love creating apps that can run in a browser.\n")
```

At ❶ we use the `'a'` argument to open the file for appending rather than writing over the existing file. At ❷ we write two new lines, which are added to *programming.txt*:

programming.txt

```
I love programming.
I love creating new games.
I also love finding meaning in large datasets.
I love creating apps that can run in a browser.
```

We end up with the original contents of the file, followed by the new content we just added.

TRY IT YOURSELF

10-3. Guest: Write a program that prompts the user for their name. When they respond, write their name to a file called *guest.txt*.

10-4. Guest Book: Write a `while` loop that prompts users for their name. When they enter their name, print a greeting to the screen and add a line recording their visit in a file called *guest_book.txt*. Make sure each entry appears on a new line in the file.

10-5. Programming Poll: Write a `while` loop that asks people why they like programming. Each time someone enters a reason, add their reason to a file that stores all the responses.

Exceptions

Python uses special objects called *exceptions* to manage errors that arise during a program's execution. Whenever an error occurs that makes Python unsure what to do next, it creates an exception object. If you write code that handles the exception, the program will continue running. If you don't handle the exception, the program will halt and show a *traceback*, which includes a report of the exception that was raised.

Exceptions are handled with `try-except` blocks. A `try-except` block asks Python to do something, but it also tells Python what to do if an exception is raised. When you use `try-except` blocks, your programs will continue running even if things start to go wrong. Instead of tracebacks, which can be confusing for users to read, users will see friendly error messages that you write.

Handling the ZeroDivisionError Exception

Let's look at a simple error that causes Python to raise an exception. You probably know that it's impossible to divide a number by zero, but let's ask Python to do it anyway:

division_calculator.py

```
print(5/0)
```

Of course Python can't do this, so we get a traceback:

```
Traceback (most recent call last):
  File "division_calculator.py", line 1, in <module>
    print(5/0)
❶ ZeroDivisionError: division by zero
```

The error reported at ❶ in the traceback, `ZeroDivisionError`, is an exception object. Python creates this kind of object in response to a situation where it can't do what we ask it to. When this happens, Python stops the program and tells us the kind of exception that was raised. We can use this information to modify our program. We'll tell Python what to do when this kind of exception occurs; that way, if it happens again, we're prepared.

Using try-except Blocks

When you think an error may occur, you can write a `try-except` block to handle the exception that might be raised. You tell Python to try running some code, and you tell it what to do if the code results in a particular kind of exception.

Here's what a `try-except` block for handling the `ZeroDivisionError` exception looks like:

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

We put `print(5/0)`, the line that caused the error, inside a `try` block. If the code in a `try` block works, Python skips over the `except` block. If the code in the `try` block causes an error, Python looks for an `except` block whose error matches the one that was raised and runs the code in that block.

In this example, the code in the `try` block produces a `ZeroDivisionError`, so Python looks for an `except` block telling it how to respond. Python then runs the code in that block, and the user sees a friendly error message instead of a traceback:

```
You can't divide by zero!
```

If more code followed the `try-except` block, the program would continue running because we told Python how to handle the error. Let's look at an example where catching an error can allow a program to continue running.

Using Exceptions to Prevent Crashes

Handling errors correctly is especially important when the program has more work to do after the error occurs. This happens often in programs that prompt users for input. If the program responds to invalid input appropriately, it can prompt for more valid input instead of crashing.

Let's create a simple calculator that does only division:

division_calculator.py

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")

while True:
❶   first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
❷   second_number = input("Second number: ")
    if second_number == 'q':
        break
❸   answer = int(first_number) / int(second_number)
    print(answer)
```

This program prompts the user to input a `first_number` ❶ and, if the user does not enter *q* to quit, a `second_number` ❷. We then divide these two numbers to get an `answer` ❸. This program does nothing to handle errors, so asking it to divide by zero causes it to crash:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 5
Second number: 0
Traceback (most recent call last):
  File "division_calculator.py", line 9, in <module>
    answer = int(first_number) / int(second_number)
ZeroDivisionError: division by zero
```

It's bad that the program crashed, but it's also not a good idea to let users see tracebacks. Nontechnical users will be confused by them, and in a malicious setting, attackers will learn more than you want them to know from a traceback. For example, they'll know the name of your program file, and they'll see a part of your code that isn't working properly. A skilled attacker can sometimes use this information to determine which kind of attacks to use against your code.

The else Block

We can make this program more error resistant by wrapping the line that might produce errors in a `try-except` block. The error occurs on the line that performs the division, so that's where we'll put the `try-except` block. This example also includes an `else` block. Any code that depends on the `try` block executing successfully goes in the `else` block:

```
--snip--
while True:
    --snip--
    if second_number == 'q':
        break
❶   try:
        answer = int(first_number) / int(second_number)
❷   except ZeroDivisionError:
        print("You can't divide by 0!")
❸   else:
        print(answer)
```

We ask Python to try to complete the division operation in a `try` block ❶, which includes only the code that might cause an error. Any code that depends on the `try` block succeeding is added to the `else` block. In this case if the division operation is successful, we use the `else` block to print the result ❸.

The `except` block tells Python how to respond when a `ZeroDivisionError` arises ❷. If the `try` block doesn't succeed because of a division by zero error, we print a friendly message telling the user how to avoid this kind of error. The program continues to run, and the user never sees a traceback:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 5
Second number: 0
You can't divide by 0!
```

```
First number: 5
Second number: 2
2.5

First number: q
```

The try-except-else block works like this: Python attempts to run the code in the try block. The only code that should go in a try block is code that might cause an exception to be raised. Sometimes you'll have additional code that should run only if the try block was successful; this code goes in the else block. The except block tells Python what to do in case a certain exception arises when it tries to run the code in the try block.

By anticipating likely sources of errors, you can write robust programs that continue to run even when they encounter invalid data and missing resources. Your code will be resistant to innocent user mistakes and malicious attacks.

Handling the FileNotFoundError Exception

One common issue when working with files is handling missing files. The file you're looking for might be in a different location, the filename may be misspelled, or the file may not exist at all. You can handle all of these situations in a straightforward way with a try-except block.

Let's try to read a file that doesn't exist. The following program tries to read in the contents of *Alice in Wonderland*, but I haven't saved the file *alice.txt* in the same directory as *alice.py*:

alice.py

```
filename = 'alice.txt'

with open(filename, encoding='utf-8') as f:
    contents = f.read()
```

There are two changes here. One is the use of the variable f to represent the file object, which is a common convention. The second is the use of the encoding argument. This argument is needed when your system's default encoding doesn't match the encoding of the file that's being read.

Python can't read from a missing file, so it raises an exception:

```
Traceback (most recent call last):
  File "alice.py", line 3, in <module>
    with open(filename, encoding='utf-8') as f:
FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'
```

The last line of the traceback reports a FileNotFoundError: this is the exception Python creates when it can't find the file it's trying to open.

In this example, the `open()` function produces the error, so to handle it, the `try` block will begin with the line that contains `open()`:

```
filename = 'alice.txt'

try:
    with open(filename, encoding='utf-8') as f:
        contents = f.read()
except FileNotFoundError:
    print(f"Sorry, the file {filename} does not exist.")
```

In this example, the code in the `try` block produces a `FileNotFoundError`, so Python looks for an `except` block that matches that error. Python then runs the code in that block, and the result is a friendly error message instead of a traceback:

```
Sorry, the file alice.txt does not exist.
```

The program has nothing more to do if the file doesn't exist, so the error-handling code doesn't add much to this program. Let's build on this example and see how exception handling can help when you're working with more than one file.

Analyzing Text

You can analyze text files containing entire books. Many classic works of literature are available as simple text files because they are in the public domain. The texts used in this section come from Project Gutenberg (*http://gutenberg.org/*). Project Gutenberg maintains a collection of literary works that are available in the public domain, and it's a great resource if you're interested in working with literary texts in your programming projects.

Let's pull in the text of *Alice in Wonderland* and try to count the number of words in the text. We'll use the string method `split()`, which can build a list of words from a string. Here's what `split()` does with a string containing just the title `"Alice in Wonderland"`:

```
>>> title = "Alice in Wonderland"
>>> title.split()
['Alice', 'in', 'Wonderland']
```

The `split()` method separates a string into parts wherever it finds a space and stores all the parts of the string in a list. The result is a list of words from the string, although some punctuation may also appear with some of the words. To count the number of words in *Alice in Wonderland*, we'll use `split()` on the entire text. Then we'll count the items in the list to get a rough idea of the number of words in the text:

```
filename = 'alice.txt'

try:
```

```
    with open(filename, encoding='utf-8') as f:
        contents = f.read()
except FileNotFoundError:
    print(f"Sorry, the file {filename} does not exist.")
else:
    # Count the approximate number of words in the file.
❶   words = contents.split()
❷   num_words = len(words)
❸   print(f"The file {filename} has about {num_words} words.")
```

I moved the file *alice.txt* to the correct directory, so the `try` block will work this time. At ❶ we take the string `contents`, which now contains the entire text of *Alice in Wonderland* as one long string, and use the `split()` method to produce a list of all the words in the book. When we use `len()` on this list to examine its length, we get a good approximation of the number of words in the original string ❷. At ❸ we print a statement that reports how many words were found in the file. This code is placed in the `else` block because it will work only if the code in the `try` block was executed successfully. The output tells us how many words are in *alice.txt*:

```
The file alice.txt has about 29465 words.
```

The count is a little high because extra information is provided by the publisher in the text file used here, but it's a good approximation of the length of *Alice in Wonderland*.

Working with Multiple Files

Let's add more books to analyze. But before we do, let's move the bulk of this program to a function called `count_words()`. By doing so, it will be easier to run the analysis for multiple books:

word_count.py

```
def count_words(filename):
❶   """Count the approximate number of words in a file."""
    try:
        with open(filename, encoding='utf-8') as f:
            contents = f.read()
    except FileNotFoundError:
        print(f"Sorry, the file {filename} does not exist.")
    else:
        words = contents.split()
        num_words = len(words)
        print(f"The file {filename} has about {num_words} words.")

filename = 'alice.txt'
count_words(filename)
```

Most of this code is unchanged. We simply indented it and moved it into the body of `count_words()`. It's a good habit to keep comments up to date when you're modifying a program, so we changed the comment to a docstring and reworded it slightly ❶.

Now we can write a simple loop to count the words in any text we want to analyze. We do this by storing the names of the files we want to analyze in a list, and then we call `count_words()` for each file in the list. We'll try to count the words for *Alice in Wonderland*, *Siddhartha*, *Moby Dick*, and *Little Women*, which are all available in the public domain. I've intentionally left *siddhartha.txt* out of the directory containing *word_count.py*, so we can see how well our program handles a missing file:

```
def count_words(filename):
    --snip--

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'little_women.txt']
for filename in filenames:
    count_words(filename)
```

The missing *siddhartha.txt* file has no effect on the rest of the program's execution:

```
The file alice.txt has about 29465 words.
Sorry, the file siddhartha.txt does not exist.
The file moby_dick.txt has about 215830 words.
The file little_women.txt has about 189079 words.
```

Using the `try-except` block in this example provides two significant advantages. We prevent our users from seeing a traceback, and we let the program continue analyzing the texts it's able to find. If we don't catch the `FileNotFoundError` that *siddhartha.txt* raised, the user would see a full traceback, and the program would stop running after trying to analyze *Siddhartha*. It would never analyze *Moby Dick* or *Little Women*.

Failing Silently

In the previous example, we informed our users that one of the files was unavailable. But you don't need to report every exception you catch. Sometimes you'll want the program to fail silently when an exception occurs and continue on as if nothing happened. To make a program fail silently, you write a `try` block as usual, but you explicitly tell Python to do nothing in the `except` block. Python has a `pass` statement that tells it to do nothing in a block:

```
def count_words(filename):
    """Count the approximate number of words in a file."""
    try:
        --snip--
    except FileNotFoundError:
❶       pass
    else:
        --snip--

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'little_women.txt']
for filename in filenames:
    count_words(filename)
```

The only difference between this listing and the previous one is the `pass` statement at ❶. Now when a `FileNotFoundError` is raised, the code in the `except` block runs, but nothing happens. No traceback is produced, and there's no output in response to the error that was raised. Users see the word counts for each file that exists, but they don't see any indication that a file wasn't found:

```
The file alice.txt has about 29465 words.
The file moby_dick.txt has about 215830 words.
The file little_women.txt has about 189079 words.
```

The `pass` statement also acts as a placeholder. It's a reminder that you're choosing to do nothing at a specific point in your program's execution and that you might want to do something there later. For example, in this program we might decide to write any missing filenames to a file called *missing_files.txt*. Our users wouldn't see this file, but we'd be able to read the file and deal with any missing texts.

Deciding Which Errors to Report

How do you know when to report an error to your users and when to fail silently? If users know which texts are supposed to be analyzed, they might appreciate a message informing them why some texts were not analyzed. If users expect to see some results but don't know which books are supposed to be analyzed, they might not need to know that some texts were unavailable. Giving users information they aren't looking for can decrease the usability of your program. Python's error-handling structures give you fine-grained control over how much to share with users when things go wrong; it's up to you to decide how much information to share.

Well-written, properly tested code is not very prone to internal errors, such as syntax or logical errors. But every time your program depends on something external, such as user input, the existence of a file, or the availability of a network connection, there is a possibility of an exception being raised. A little experience will help you know where to include exception handling blocks in your program and how much to report to users about errors that arise.

TRY IT YOURSELF

10-6. Addition: One common problem when prompting for numerical input occurs when people provide text instead of numbers. When you try to convert the input to an `int`, you'll get a `ValueError`. Write a program that prompts for two numbers. Add them together and print the result. Catch the `ValueError` if either input value is not a number, and print a friendly error message. Test your program by entering two numbers and then by entering some text instead of a number.

(continued)

10-7. Addition Calculator: Wrap your code from Exercise 10-6 in a `while` loop so the user can continue entering numbers even if they make a mistake and enter text instead of a number.

10-8. Cats and Dogs: Make two files, *cats.txt* and *dogs.txt*. Store at least three names of cats in the first file and three names of dogs in the second file. Write a program that tries to read these files and print the contents of the file to the screen. Wrap your code in a `try-except` block to catch the `FileNotFound` error, and print a friendly message if a file is missing. Move one of the files to a different location on your system, and make sure the code in the `except` block executes properly.

10-9. Silent Cats and Dogs: Modify your `except` block in Exercise 10-8 to fail silently if either file is missing.

10-10. Common Words: Visit Project Gutenberg (*https://gutenberg.org/*) and find a few texts you'd like to analyze. Download the text files for these works, or copy the raw text from your browser into a text file on your computer.

You can use the `count()` method to find out how many times a word or phrase appears in a string. For example, the following code counts the number of times `'row'` appears in a string:

```
>>> line = "Row, row, row your boat"
>>> line.count('row')
2
>>> line.lower().count('row')
3
```

Notice that converting the string to lowercase using `lower()` catches all appearances of the word you're looking for, regardless of how it's formatted.

Write a program that reads the files you found at Project Gutenberg and determines how many times the word `'the'` appears in each text. This will be an approximation because it will also count words such as `'then'` and `'there'`. Try counting `'the '`, with a space in the string, and see how much lower your count is.

Storing Data

Many of your programs will ask users to input certain kinds of information. You might allow users to store preferences in a game or provide data for a visualization. Whatever the focus of your program is, you'll store the information users provide in data structures such as lists and dictionaries. When users close a program, you'll almost always want to save the information they entered. A simple way to do this involves storing your data using the `json` module.

The `json` module allows you to dump simple Python data structures into a file and load the data from that file the next time the program runs. You can also use `json` to share data between different Python programs. Even better, the JSON data format is not specific to Python, so you can share data you store in the JSON format with people who work in many other programming languages. It's a useful and portable format, and it's easy to learn.

NOTE *The JSON (JavaScript Object Notation) format was originally developed for JavaScript. However, it has since become a common format used by many languages, including Python.*

Using json.dump() and json.load()

Let's write a short program that stores a set of numbers and another program that reads these numbers back into memory. The first program will use `json.dump()` to store the set of numbers, and the second program will use `json.load()`.

The `json.dump()` function takes two arguments: a piece of data to store and a file object it can use to store the data. Here's how you can use `json.dump()` to store a list of numbers:

number_writer.py

```
import json

numbers = [2, 3, 5, 7, 11, 13]

❶ filename = 'numbers.json'
❷ with open(filename, 'w') as f:
❸     json.dump(numbers, f)
```

We first import the `json` module and then create a list of numbers to work with. At ❶ we choose a filename in which to store the list of numbers. It's customary to use the file extension *.json* to indicate that the data in the file is stored in the JSON format. Then we open the file in write mode, which allows `json` to write the data to the file ❷. At ❸ we use the `json.dump()` function to store the list `numbers` in the file *numbers.json*.

This program has no output, but let's open the file *numbers.json* and look at it. The data is stored in a format that looks just like Python:

```
[2, 3, 5, 7, 11, 13]
```

Now we'll write a program that uses `json.load()` to read the list back into memory:

number_reader.py

```
import json

❶ filename = 'numbers.json'
❷ with open(filename) as f:
❸     numbers = json.load(f)

print(numbers)
```

At ❶ we make sure to read from the same file we wrote to. This time when we open the file, we open it in read mode because Python only needs to read from the file ❷. At ❸ we use the `json.load()` function to load the information stored in *numbers.json*, and we assign it to the variable `numbers`. Finally we print the recovered list of numbers and see that it's the same list created in *number_writer.py*:

```
[2, 3, 5, 7, 11, 13]
```

This is a simple way to share data between two programs.

Saving and Reading User-Generated Data

Saving data with `json` is useful when you're working with user-generated data, because if you don't store your user's information somehow, you'll lose it when the program stops running. Let's look at an example where we prompt the user for their name the first time they run a program and then remember their name when they run the program again.

Let's start by storing the user's name:

remember_me.py

```
import json

❶ username = input("What is your name? ")

filename = 'username.json'
with open(filename, 'w') as f:
❷     json.dump(username, f)
❸     print(f"We'll remember you when you come back, {username}!")
```

At ❶ we prompt for a username to store. Next, we use `json.dump()`, passing it a username and a file object, to store the username in a file ❷. Then we print a message informing the user that we've stored their information ❸:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Now let's write a new program that greets a user whose name has already been stored:

greet_user.py

```
import json

filename = 'username.json'

with open(filename) as f:
❶     username = json.load(f)
❷     print(f"Welcome back, {username}!")
```

At ❶ we use `json.load()` to read the information stored in *username.json* and assign it to the variable `username`. Now that we've recovered the username, we can welcome them back ❷:

```
Welcome back, Eric!
```

We need to combine these two programs into one file. When someone runs *remember_me.py*, we want to retrieve their username from memory if possible; therefore, we'll start with a `try` block that attempts to recover the username. If the file *username.json* doesn't exist, we'll have the `except` block prompt for a username and store it in *username.json* for next time:

remember_me.py

```
import json

# Load the username, if it has been stored previously.
#  Otherwise, prompt for the username and store it.
filename = 'username.json'
try:
❶     with open(filename) as f:
❷         username = json.load(f)
❸ except FileNotFoundError:
❹     username = input("What is your name? ")
❺     with open(filename, 'w') as f:
          json.dump(username, f)
          print(f"We'll remember you when you come back, {username}!")
else:
    print(f"Welcome back, {username}!")
```

There's no new code here; blocks of code from the last two examples are just combined into one file. At ❶ we try to open the file *username.json*. If this file exists, we read the username back into memory ❷ and print a message welcoming back the user in the `else` block. If this is the first time the user runs the program, *username.json* won't exist and a `FileNotFoundError` will occur ❸. Python will move on to the `except` block where we prompt the user to enter their username ❹. We then use `json.dump()` to store the username and print a greeting ❺.

Whichever block executes, the result is a username and an appropriate greeting. If this is the first time the program runs, this is the output:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Otherwise:

```
Welcome back, Eric!
```

This is the output you see if the program was already run at least once.

Refactoring

Often, you'll come to a point where your code will work, but you'll recognize that you could improve the code by breaking it up into a series of functions that have specific jobs. This process is called *refactoring*. Refactoring makes your code cleaner, easier to understand, and easier to extend.

We can refactor *remember_me.py* by moving the bulk of its logic into one or more functions. The focus of *remember_me.py* is on greeting the user, so let's move all of our existing code into a function called `greet_user()`:

remember_me.py

```
import json

def greet_user():
❶    """Greet the user by name."""
    filename = 'username.json'
    try:
        with open(filename) as f:
            username = json.load(f)
    except FileNotFoundError:
        username = input("What is your name? ")
        with open(filename, 'w') as f:
            json.dump(username, f)
            print(f"We'll remember you when you come back, {username}!")
    else:
        print(f"Welcome back, {username}!")

greet_user()
```

Because we're using a function now, we update the comments with a docstring that reflects how the program currently works ❶. This file is a little cleaner, but the function `greet_user()` is doing more than just greeting the user—it's also retrieving a stored username if one exists and prompting for a new username if one doesn't exist.

Let's refactor `greet_user()` so it's not doing so many different tasks. We'll start by moving the code for retrieving a stored username to a separate function:

```
import json

def get_stored_username():
❶    """Get stored username if available."""
    filename = 'username.json'
    try:
        with open(filename) as f:
            username = json.load(f)
    except FileNotFoundError:
❷        return None
    else:
        return username
```

```
def greet_user():
    """Greet the user by name."""
    username = get_stored_username()
❸   if username:
        print(f"Welcome back, {username}!")
    else:
        username = input("What is your name? ")
        filename = 'username.json'
        with open(filename, 'w') as f:
            json.dump(username, f)
            print(f"We'll remember you when you come back, {username}!")

greet_user()
```

The new function `get_stored_username()` has a clear purpose, as stated in the docstring at ❶. This function retrieves a stored username and returns the username if it finds one. If the file *username.json* doesn't exist, the function returns `None` ❷. This is good practice: a function should either return the value you're expecting, or it should return `None`. This allows us to perform a simple test with the return value of the function. At ❸ we print a welcome back message to the user if the attempt to retrieve a username was successful, and if it doesn't, we prompt for a new username.

We should factor one more block of code out of `greet_user()`. If the username doesn't exist, we should move the code that prompts for a new username to a function dedicated to that purpose:

```
import json

def get_stored_username():
    """Get stored username if available."""
    --snip--

def get_new_username():
    """Prompt for a new username."""
    username = input("What is your name? ")
    filename = 'username.json'
    with open(filename, 'w') as f:
        json.dump(username, f)
    return username

def greet_user():
    """Greet the user by name."""
    username = get_stored_username()
    if username:
        print(f"Welcome back, {username}!")
    else:
        username = get_new_username()
        print(f"We'll remember you when you come back, {username}!")

greet_user()
```

Each function in this final version of *remember_me.py* has a single, clear purpose. We call `greet_user()`, and that function prints an appropriate message: it either welcomes back an existing user or greets a new user. It does this by calling `get_stored_username()`, which is responsible only for retrieving a stored username if one exists. Finally, `greet_user()` calls `get_new_username()` if necessary, which is responsible only for getting a new username and storing it. This compartmentalization of work is an essential part of writing clear code that will be easy to maintain and extend.

TRY IT YOURSELF

10-11. Favorite Number: Write a program that prompts for the user's favorite number. Use `json.dump()` to store this number in a file. Write a separate program that reads in this value and prints the message, "I know your favorite number! It's _____."

10-12. Favorite Number Remembered: Combine the two programs from Exercise 10-11 into one file. If the number is already stored, report the favorite number to the user. If not, prompt for the user's favorite number and store it in a file. Run the program twice to see that it works.

10-13. Verify User: The final listing for *remember_me.py* assumes either that the user has already entered their username or that the program is running for the first time. We should modify it in case the current user is not the person who last used the program.

Before printing a welcome back message in `greet_user()`, ask the user if this is the correct username. If it's not, call `get_new_username()` to get the correct username.

Summary

In this chapter, you learned how to work with files. You learned to read an entire file at once and read through a file's contents one line at a time. You learned to write to a file and append text onto the end of a file. You read about exceptions and how to handle the exceptions you're likely to see in your programs. Finally, you learned how to store Python data structures so you can save information your users provide, preventing them from having to start over each time they run a program.

In Chapter 11 you'll learn efficient ways to test your code. This will help you trust that the code you develop is correct, and it will help you identify bugs that are introduced as you continue to build on the programs you've written.

11

TESTING YOUR CODE

When you write a function or a class, you can also write tests for that code. Testing proves that your code works as it's supposed to in response to all the input types it's designed to receive. When you write tests, you can be confident that your code will work correctly as more people begin to use your programs. You'll also be able to test new code as you add it to make sure your changes don't break your program's existing behavior. Every programmer makes mistakes, so every programmer must test their code often, catching problems before users encounter them.

In this chapter you'll learn to test your code using tools in Python's `unittest` module. You'll learn to build a test case and check that a set of inputs results in the output you want. You'll see what a passing test looks like and what a failing test looks like, and you'll learn how a failing test can help you improve your code. You'll learn to test functions and classes, and you'll start to understand how many tests to write for a project.

Testing a Function

To learn about testing, we need code to test. Here's a simple function that takes in a first and last name, and returns a neatly formatted full name:

name_function.py

```
def get_formatted_name(first, last):
    """Generate a neatly formatted full name."""
    full_name = f"{first} {last}"
    return full_name.title()
```

The function `get_formatted_name()` combines the first and last name with a space in between to complete a full name, and then capitalizes and returns the full name. To check that `get_formatted_name()` works, let's make a program that uses this function. The program *names.py* lets users enter a first and last name, and see a neatly formatted full name:

names.py

```
from name_function import get_formatted_name

print("Enter 'q' at any time to quit.")
while True:
    first = input("\nPlease give me a first name: ")
    if first == 'q':
        break
    last = input("Please give me a last name: ")
    if last == 'q':
        break

    formatted_name = get_formatted_name(first, last)
    print(f"\tNeatly formatted name: {formatted_name}.")
```

This program imports `get_formatted_name()` from *name_function.py*. The user can enter a series of first and last names, and see the formatted full names that are generated:

```
Enter 'q' at any time to quit.

Please give me a first name: janis
Please give me a last name: joplin
        Neatly formatted name: Janis Joplin.

Please give me a first name: bob
Please give me a last name: dylan
        Neatly formatted name: Bob Dylan.

Please give me a first name: q
```

We can see that the names generated here are correct. But let's say we want to modify `get_formatted_name()` so it can also handle middle names. As we do so, we want to make sure we don't break the way the function handles names that have only a first and last name. We could test our code by running *names.py* and entering a name like `Janis Joplin` every time we modify `get_formatted_name()`, but that would become tedious. Fortunately,

Python provides an efficient way to automate the testing of a function's output. If we automate the testing of `get_formatted_name()`, we can always be confident that the function will work when given the kinds of names we've written tests for.

Unit Tests and Test Cases

The module `unittest` from the Python standard library provides tools for testing your code. A *unit test* verifies that one specific aspect of a function's behavior is correct. A *test case* is a collection of unit tests that together prove that a function behaves as it's supposed to, within the full range of situations you expect it to handle. A good test case considers all the possible kinds of input a function could receive and includes tests to represent each of these situations. A test case with *full coverage* includes a full range of unit tests covering all the possible ways you can use a function. Achieving full coverage on a large project can be daunting. It's often good enough to write tests for your code's critical behaviors and then aim for full coverage only if the project starts to see widespread use.

A Passing Test

The syntax for setting up a test case takes some getting used to, but once you've set up the test case it's straightforward to add more unit tests for your functions. To write a test case for a function, import the `unittest` module and the function you want to test. Then create a class that inherits from `unittest.TestCase`, and write a series of methods to test different aspects of your function's behavior.

Here's a test case with one method that verifies that the function `get_formatted_name()` works correctly when given a first and last name:

test_name_function.py

```
  import unittest
  from name_function import get_formatted_name

❶ class NamesTestCase(unittest.TestCase):
      """Tests for 'name_function.py'."""

      def test_first_last_name(self):
          """Do names like 'Janis Joplin' work?"""
❷         formatted_name = get_formatted_name('janis', 'joplin')
❸         self.assertEqual(formatted_name, 'Janis Joplin')

❹ if __name__ == '__main__':
      unittest.main()
```

First, we import `unittest` and the function we want to test, `get_formatted_name()`. At ❶ we create a class called `NamesTestCase`, which will contain a series of unit tests for `get_formatted_name()`. You can name the class anything you want, but it's best to call it something related to the function you're about to test and to use the word *Test* in the class name. This class must inherit from the class `unittest.TestCase` so Python knows how to run the tests you write.

NamesTestCase contains a single method that tests one aspect of get_formatted_name(). We call this method test_first_last_name() because we're verifying that names with only a first and last name are formatted correctly. Any method that starts with test_ will be run automatically when we run *test_name_function.py*. Within this test method, we call the function we want to test. In this example we call get_formatted_name() with the arguments 'janis' and 'joplin', and assign the result to formatted_name ❷.

At ❸ we use one of unittest's most useful features: an *assert* method. Assert methods verify that a result you received matches the result you expected to receive. In this case, because we know get_formatted_name() is supposed to return a capitalized, properly spaced full name, we expect the value of formatted_name to be Janis Joplin. To check if this is true, we use unittest's assertEqual() method and pass it formatted_name and 'Janis Joplin'. The line

```
self.assertEqual(formatted_name, 'Janis Joplin')
```

says, "Compare the value in formatted_name to the string 'Janis Joplin'. If they are equal as expected, fine. But if they don't match, let me know!"

We're going to run this file directly, but it's important to note that many testing frameworks import your test files before running them. When a file is imported, the interpreter executes the file as it's being imported. The if block at ❹ looks at a special variable, __name__, which is set when the program is executed. If this file is being run as the main program, the value of __name__ is set to '__main__'. In this case, we call unittest.main(), which runs the test case. When a testing framework imports this file, the value of __name__ won't be '__main__' and this block will not be executed.

When we run *test_name_function.py*, we get the following output:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

The dot on the first line of output tells us that a single test passed. The next line tells us that Python ran one test, and it took less than 0.001 seconds to run. The final OK tells us that all unit tests in the test case passed.

This output indicates that the function get_formatted_name() will always work for names that have a first and last name unless we modify the function. When we modify get_formatted_name(), we can run this test again. If the test case passes, we know the function will still work for names like Janis Joplin.

A Failing Test

What does a failing test look like? Let's modify get_formatted_name() so it can handle middle names, but we'll do so in a way that breaks the function for names with just a first and last name, like Janis Joplin.

Here's a new version of `get_formatted_name()` that requires a middle name argument:

name_function.py

```
def get_formatted_name(first, middle, last):
    """Generate a neatly formatted full name."""
    full_name = f"{first} {middle} {last}"
    return full_name.title()
```

This version should work for people with middle names, but when we test it, we see that we've broken the function for people with just a first and last name. This time, running the file *test_name_function.py* gives this output:

```
❶ E
  ======================================================================
❷ ERROR: test_first_last_name (__main__.NamesTestCase)
  ----------------------------------------------------------------------
❸ Traceback (most recent call last):
    File "test_name_function.py", line 8, in test_first_last_name
      formatted_name = get_formatted_name('janis', 'joplin')
  TypeError: get_formatted_name() missing 1 required positional argument: 'last'

  ----------------------------------------------------------------------
❹ Ran 1 test in 0.000s

❺ FAILED (errors=1)
```

There's a lot of information here because there's a lot you might need to know when a test fails. The first item in the output is a single `E` ❶, which tells us one unit test in the test case resulted in an error. Next, we see that `test_first_last_name()` in `NamesTestCase` caused an error ❷. Knowing which test failed is critical when your test case contains many unit tests. At ❸ we see a standard traceback, which reports that the function call `get_formatted_name('janis', 'joplin')` no longer works because it's missing a required positional argument.

We also see that one unit test was run ❹. Finally, we see an additional message that the overall test case failed and that one error occurred when running the test case ❺. This information appears at the end of the output so you see it right away; you don't need to scroll up through a long output listing to find out how many tests failed.

Responding to a Failed Test

What do you do when a test fails? Assuming you're checking the right conditions, a passing test means the function is behaving correctly and a failing test means there's an error in the new code you wrote. So when a test fails, don't change the test. Instead, fix the code that caused the test to fail. Examine the changes you just made to the function, and figure out how those changes broke the desired behavior.

In this case `get_formatted_name()` used to require only two parameters: a first name and a last name. Now it requires a first name, middle name, and

last name. The addition of that mandatory middle name parameter broke the desired behavior of `get_formatted_name()`. The best option here is to make the middle name optional. Once we do, our test for names like `Janis Joplin` should pass again, and we should be able to accept middle names as well. Let's modify `get_formatted_name()` so middle names are optional and then run the test case again. If it passes, we'll move on to making sure the function handles middle names properly.

To make middle names optional, we move the parameter `middle` to the end of the parameter list in the function definition and give it an empty default value. We also add an `if` test that builds the full name properly, depending on whether or not a middle name is provided:

name_function.py

```
def get_formatted_name(first, last, middle=''):
    """Generate a neatly formatted full name."""
    if middle:
        full_name = f"{first} {middle} {last}"
    else:
        full_name = f"{first} {last}"
    return full_name.title()
```

In this new version of `get_formatted_name()`, the middle name is optional. If a middle name is passed to the function, the full name will contain a first, middle, and last name. Otherwise, the full name will consist of just a first and last name. Now the function should work for both kinds of names. To find out if the function still works for names like `Janis Joplin`, let's run *test_name_function.py* again:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

The test case passes now. This is ideal; it means the function works for names like `Janis Joplin` again without us having to test the function manually. Fixing our function was easy because the failed test helped us identify the new code that broke existing behavior.

Adding New Tests

Now that we know `get_formatted_name()` works for simple names again, let's write a second test for people who include a middle name. We do this by adding another method to the class `NamesTestCase`:

test_name_function.py

```
--snip--

class NamesTestCase(unittest.TestCase):
    """Tests for 'name_function.py'."""

    def test_first_last_name(self):
        --snip--
```

```
    def test_first_last_middle_name(self):
        """Do names like 'Wolfgang Amadeus Mozart' work?"""
❶       formatted_name = get_formatted_name(
            'wolfgang', 'mozart', 'amadeus')
        self.assertEqual(formatted_name, 'Wolfgang Amadeus Mozart')

if __name__ == '__main__':
    unittest.main()
```

We name this new method `test_first_last_middle_name()`. The method name must start with *test_* so the method runs automatically when we run *test_name_function.py*. We name the method to make it clear which behavior of `get_formatted_name()` we're testing. As a result, if the test fails, we know right away what kinds of names are affected. It's fine to have long method names in your `TestCase` classes. They need to be descriptive so you can make sense of the output when your tests fail, and because Python calls them automatically, you'll never have to write code that calls these methods.

To test the function, we call `get_formatted_name()` with a first, last, and middle name ❶, and then we use `assertEqual()` to check that the returned full name matches the full name (first, middle, and last) that we expect. When we run *test_name_function.py* again, both tests pass:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Great! We now know that the function still works for names like `Janis Joplin`, and we can be confident that it will work for names like `Wolfgang Amadeus Mozart` as well.

TRY IT YOURSELF

11-1. City, Country: Write a function that accepts two parameters: a city name and a country name. The function should return a single string of the form `City, Country`, such as `Santiago, Chile`. Store the function in a module called *city_functions.py*.

Create a file called *test_cities.py* that tests the function you just wrote (remember that you need to import `unittest` and the function you want to test). Write a method called `test_city_country()` to verify that calling your function with values such as `'santiago'` and `'chile'` results in the correct string. Run *test_cities.py*, and make sure `test_city_country()` passes.

(continued)

11-2. Population: Modify your function so it requires a third parameter, `population`. It should now return a single string of the form *`City, Country - population xxx`*, such as `Santiago, Chile - population 5000000`. Run *test_cities.py* again. Make sure `test_city_country()` fails this time.

Modify the function so the `population` parameter is optional. Run *test_cities.py* again, and make sure `test_city_country()` passes again.

Write a second test called `test_city_country_population()` that verifies you can call your function with the values `'santiago'`, `'chile'`, and `'population=5000000'`. Run *test_cities.py* again, and make sure this new test passes.

Testing a Class

In the first part of this chapter, you wrote tests for a single function. Now you'll write tests for a class. You'll use classes in many of your own programs, so it's helpful to be able to prove that your classes work correctly. If you have passing tests for a class you're working on, you can be confident that improvements you make to the class won't accidentally break its current behavior.

A Variety of Assert Methods

Python provides a number of assert methods in the `unittest.TestCase` class. As mentioned earlier, assert methods test whether a condition you believe is true at a specific point in your code is indeed true. If the condition is true as expected, your assumption about how that part of your program behaves is confirmed; you can be confident that no errors exist. If the condition you assume is true is actually not true, Python raises an exception.

Table 11-1 describes six commonly used assert methods. With these methods you can verify that returned values equal or don't equal expected values, that values are `True` or `False`, and that values are `in` or `not in` a given list. You can use these methods only in a class that inherits from `unittest.TestCase`, so let's look at how we can use one of these methods in the context of testing an actual class.

Table 11-1: Assert Methods Available from the `unittest` Module

Method	Use
`assertEqual(a, b)`	Verify that `a == b`
`assertNotEqual(a, b)`	Verify that `a != b`
`assertTrue(x)`	Verify that `x` is `True`
`assertFalse(x)`	Verify that `x` is `False`
`assertIn(`*`item, list`*`)`	Verify that *`item`* is in *`list`*
`assertNotIn(`*`item, list`*`)`	Verify that *`item`* is not in *`list`*

A Class to Test

Testing a class is similar to testing a function—much of your work involves testing the behavior of the methods in the class. But there are a few differences, so let's write a class to test. Consider a class that helps administer anonymous surveys:

survey.py

```
class AnonymousSurvey:
    """Collect anonymous answers to a survey question."""

❶   def __init__(self, question):
        """Store a question, and prepare to store responses."""
        self.question = question
        self.responses = []

❷   def show_question(self):
        """Show the survey question."""
        print(self.question)

❸   def store_response(self, new_response):
        """Store a single response to the survey."""
        self.responses.append(new_response)

❹   def show_results(self):
        """Show all the responses that have been given."""
        print("Survey results:")
        for response in self.responses:
            print(f"- {response}")
```

This class starts with a survey question that you provide ❶ and includes an empty list to store responses. The class has methods to print the survey question ❷, add a new response to the response list ❸, and print all the responses stored in the list ❹. To create an instance from this class, all you have to provide is a question. Once you have an instance representing a particular survey, you display the survey question with `show_question()`, store a response using `store_response()`, and show results with `show_results()`.

To show that the `AnonymousSurvey` class works, let's write a program that uses the class:

language_survey.py

```
from survey import AnonymousSurvey

# Define a question, and make a survey.
question = "What language did you first learn to speak?"
my_survey = AnonymousSurvey(question)

# Show the question, and store responses to the question.
my_survey.show_question()
print("Enter 'q' at any time to quit.\n")
while True:
    response = input("Language: ")
    if response == 'q':
        break
    my_survey.store_response(response)
```

```
# Show the survey results.
print("\nThank you to everyone who participated in the survey!")
my_survey.show_results()
```

This program defines a question (`"What language did you first learn to speak?"`) and creates an `AnonymousSurvey` object with that question. The program calls `show_question()` to display the question and then prompts for responses. Each response is stored as it is received. When all responses have been entered (the user inputs `q` to quit), `show_results()` prints the survey results:

```
What language did you first learn to speak?
Enter 'q' at any time to quit.

Language: English
Language: Spanish
Language: English
Language: Mandarin
Language: q

Thank you to everyone who participated in the survey!
Survey results:
- English
- Spanish
- English
- Mandarin
```

This class works for a simple anonymous survey. But let's say we want to improve `AnonymousSurvey` and the module it's in, `survey`. We could allow each user to enter more than one response. We could write a method to list only unique responses and to report how many times each response was given. We could write another class to manage nonanonymous surveys.

Implementing such changes would risk affecting the current behavior of the class `AnonymousSurvey`. For example, it's possible that while trying to allow each user to enter multiple responses, we could accidentally change how single responses are handled. To ensure we don't break existing behavior as we develop this module, we can write tests for the class.

Testing the AnonymousSurvey Class

Let's write a test that verifies one aspect of the way `AnonymousSurvey` behaves. We'll write a test to verify that a single response to the survey question is stored properly. We'll use the `assertIn()` method to verify that the response is in the list of responses after it's been stored:

test_survey.py

```
import unittest
from survey import AnonymousSurvey

❶ class TestAnonymousSurvey(unittest.TestCase):
    """Tests for the class AnonymousSurvey"""
```

```
❷     def test_store_single_response(self):
          """Test that a single response is stored properly."""
          question = "What language did you first learn to speak?"
❸         my_survey = AnonymousSurvey(question)
          my_survey.store_response('English')
❹         self.assertIn('English', my_survey.responses)

if __name__ == '__main__':
    unittest.main()
```

We start by importing the `unittest` module and the class we want to test, `AnonymousSurvey`. We call our test case `TestAnonymousSurvey`, which again inherits from `unittest.TestCase` ❶. The first test method will verify that when we store a response to the survey question, the response ends up in the survey's list of responses. A good descriptive name for this method is `test_store_single_response()` ❷. If this test fails, we'll know from the method name shown in the output of the failing test that there was a problem storing a single response to the survey.

To test the behavior of a class, we need to make an instance of the class. At ❸ we create an instance called `my_survey` with the question `"What language did you first learn to speak?"` We store a single response, `English`, using the `store_response()` method. Then we verify that the response was stored correctly by asserting that `English` is in the list `my_survey.responses` ❹.

When we run *test_survey.py*, the test passes:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

This is good, but a survey is useful only if it generates more than one response. Let's verify that three responses can be stored correctly. To do this, we add another method to `TestAnonymousSurvey`:

```
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Tests for the class AnonymousSurvey"""

    def test_store_single_response(self):
        --snip--

    def test_store_three_responses(self):
        """Test that three individual responses are stored properly."""
        question = "What language did you first learn to speak?"
        my_survey = AnonymousSurvey(question)
❶       responses = ['English', 'Spanish', 'Mandarin']
        for response in responses:
            my_survey.store_response(response)

❷       for response in responses:
```

```
            self.assertIn(response, my_survey.responses)

if __name__ == '__main__':
    unittest.main()
```

We call the new method `test_store_three_responses()`. We create a survey object just like we did in `test_store_single_response()`. We define a list containing three different responses ❶, and then we call `store_response()` for each of these responses. Once the responses have been stored, we write another loop and assert that each response is now in `my_survey.responses` ❷.

When we run *test_survey.py* again, both tests (for a single response and for three responses) pass:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

This works perfectly. However, these tests are a bit repetitive, so we'll use another feature of `unittest` to make them more efficient.

The setUp() Method

In *test_survey.py* we created a new instance of `AnonymousSurvey` in each test method, and we created new responses in each method. The `unittest.TestCase` class has a `setUp()` method that allows you to create these objects once and then use them in each of your test methods. When you include a `setUp()` method in a `TestCase` class, Python runs the `setUp()` method before running each method starting with *test_*. Any objects created in the `setUp()` method are then available in each test method you write.

Let's use `setUp()` to create a survey instance and a set of responses that can be used in `test_store_single_response()` and `test_store_three_responses()`:

```
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Tests for the class AnonymousSurvey."""

    def setUp(self):
        """
        Create a survey and a set of responses for use in all test methods.
        """
        question = "What language did you first learn to speak?"
❶       self.my_survey = AnonymousSurvey(question)
❷       self.responses = ['English', 'Spanish', 'Mandarin']

    def test_store_single_response(self):
        """Test that a single response is stored properly."""
        self.my_survey.store_response(self.responses[0])
        self.assertIn(self.responses[0], self.my_survey.responses)
```

```
    def test_store_three_responses(self):
        """Test that three individual responses are stored properly."""
        for response in self.responses:
            self.my_survey.store_response(response)
        for response in self.responses:
            self.assertIn(response, self.my_survey.responses)

if __name__ == '__main__':
    unittest.main()
```

The method `setUp()` does two things: it creates a survey instance ❶, and it creates a list of responses ❷. Each of these is prefixed by `self`, so they can be used anywhere in the class. This makes the two test methods simpler, because neither one has to make a survey instance or a response. The method `test_store_single_response()` verifies that the first response in `self.responses`—`self.responses[0]`—can be stored correctly, and `test_store_three_responses()` verifies that all three responses in `self.responses` can be stored correctly.

When we run *test_survey.py* again, both tests still pass. These tests would be particularly useful when trying to expand `AnonymousSurvey` to handle multiple responses for each person. After modifying the code to accept multiple responses, you could run these tests and make sure you haven't affected the ability to store a single response or a series of individual responses.

When you're testing your own classes, the `setUp()` method can make your test methods easier to write. You make one set of instances and attributes in `setUp()` and then use these instances in all your test methods. This is much easier than making a new set of instances and attributes in each test method.

NOTE *When a test case is running, Python prints one character for each unit test as it is completed. A passing test prints a dot, a test that results in an error prints an* `E`*, and a test that results in a failed assertion prints an* `F`*. This is why you'll see a different number of dots and characters on the first line of output when you run your test cases. If a test case takes a long time to run because it contains many unit tests, you can watch these results to get a sense of how many tests are passing.*

TRY IT YOURSELF

11-3. Employee: Write a class called `Employee`. The `__init__()` method should take in a first name, a last name, and an annual salary, and store each of these as attributes. Write a method called `give_raise()` that adds $5,000 to the annual salary by default but also accepts a different raise amount.

Write a test case for `Employee`. Write two test methods, `test_give_default_raise()` and `test_give_custom_raise()`. Use the `setUp()` method so you don't have to create a new employee instance in each test method. Run your test case, and make sure both tests pass.

Summary

In this chapter you learned to write tests for functions and classes using tools in the `unittest` module. You learned to write a class that inherits from `unittest.TestCase`, and you learned to write test methods that verify specific behaviors your functions and classes should exhibit. You learned to use the `setUp()` method to efficiently create instances and attributes from your classes that can be used in all the test methods for a class.

Testing is an important topic that many beginners don't learn. You don't have to write tests for all the simple projects you try as a beginner. But as soon as you start to work on projects that involve significant development effort, you should test the critical behaviors of your functions and classes. You'll be more confident that new work on your project won't break the parts that work, and this will give you the freedom to make improvements to your code. If you accidentally break existing functionality, you'll know right away, so you can still fix the problem easily. Responding to a failed test that you ran is much easier than responding to a bug report from an unhappy user.

Other programmers respect your projects more if you include some initial tests. They'll feel more comfortable experimenting with your code and be more willing to work with you on projects. If you want to contribute to a project that other programmers are working on, you'll be expected to show that your code passes existing tests and you'll usually be expected to write tests for new behavior you introduce to the project.

Play around with tests to become familiar with the process of testing your code. Write tests for the most critical behaviors of your functions and classes, but don't aim for full coverage in early projects unless you have a specific reason to do so.

PART II

PROJECTS

Congratulations! You now know enough about Python to start building interactive and meaningful projects. Creating your own projects will teach you new skills and solidify your understanding of the concepts introduced in Part I.

Part II contains three types of projects, and you can choose to do any or all of these projects in whichever order you like. Here's a brief description of each project to help you decide which to dig into first.

Alien Invasion: Making a Game with Python

In the *Alien Invasion* project (Chapters 12, 13, and 14), you'll use the Pygame package to develop a 2D game in which the aim is to shoot down a fleet of aliens as they drop down the screen in levels that increase in speed and difficulty. At the end of the project, you'll have learned skills that will enable you to develop your own 2D games in Pygame.

Data Visualization

The Data Visualization project starts in Chapter 15, in which you'll learn to generate data and create a series of functional and beautiful visualizations of that data using Matplotlib and Plotly. Chapter 16 teaches you to access data from online sources and feed it into a visualization package to create plots of weather data and a map of global earthquake activity. Finally, Chapter 17 shows you how to write a program to automatically download

and visualize data. Learning to make visualizations allows you to explore the field of data mining, which is a highly sought-after skill in the world today.

Web Applications

In the Web Applications project (Chapters 18, 19, and 20), you'll use the Django package to create a simple web application that allows users to keep a journal about any number of topics they've been learning about. Users will create an account with a username and password, enter a topic, and then make entries about what they're learning. You'll also learn how to deploy your app so anyone in the world can access it.

After completing this project, you'll be able to start building your own simple web applications, and you'll be ready to delve into more thorough resources on building applications with Django.

PROJECT 1

ALIEN INVASION

12

A SHIP THAT FIRES BULLETS

Let's build a game called *Alien Invasion*! We'll use Pygame, a collection of fun, powerful Python modules that manage graphics, animation, and even sound, making it easier for you to build sophisticated games. With Pygame handling tasks like drawing images to the screen, you can focus on the higher-level logic of game dynamics.

In this chapter, you'll set up Pygame, and then create a rocket ship that moves right and left and fires bullets in response to player input. In the next two chapters, you'll create a fleet of aliens to destroy, and then continue to refine the game by setting limits on the number of ships you can use and adding a scoreboard.

While building this game, you'll also learn how to manage large projects that span multiple files. We'll refactor a lot of code and manage file contents to organize the project and make the code efficient.

Making games is an ideal way to have fun while learning a language. It's deeply satisfying to play a game you wrote, and writing a simple game will help you comprehend how professionals develop games. As you work through this chapter, enter and run the code to identify how each code block contributes to overall gameplay. Experiment with different values and settings to better understand how to refine interactions in your games.

NOTE Alien Invasion *spans a number of different files, so make a new* alien_invasion *folder on your system. Be sure to save all files for the project to this folder so your* `import` *statements will work correctly.*

Also, if you feel comfortable using version control, you might want to use it for this project. If you haven't used version control before, see Appendix D for an overview.

Planning Your Project

When you're building a large project, it's important to prepare a plan before you begin to write code. Your plan will keep you focused and make it more likely that you'll complete the project.

Let's write a description of the general gameplay. Although the following description doesn't cover every detail of *Alien Invasion*, it provides a clear idea of how to start building the game:

> In *Alien Invasion*, the player controls a rocket ship that appears at the bottom center of the screen. The player can move the ship right and left using the arrow keys and shoot bullets using the spacebar. When the game begins, a fleet of aliens fills the sky and moves across and down the screen. The player shoots and destroys the aliens. If the player shoots all the aliens, a new fleet appears that moves faster than the previous fleet. If any alien hits the player's ship or reaches the bottom of the screen, the player loses a ship. If the player loses three ships, the game ends.

For the first development phase, we'll make a ship that can move right and left and fires bullets when the player presses the spacebar. After setting up this behavior, we can create the aliens and refine the gameplay.

Installing Pygame

Before you begin coding, install Pygame. The `pip` module helps you download and install Python packages. To install Pygame, enter the following command at a terminal prompt:

```
$ python -m pip install --user pygame
```

This command tells Python to run the `pip` module and install the `pygame` package to the current user's Python installation. If you use a command

other than python to run programs or start a terminal session, such as python3, your command will look like this:

```
$ python3 -m pip install --user pygame
```

NOTE *If this command doesn't work on macOS, try running the command again without the --user flag.*

Starting the Game Project

We'll begin building the game by creating an empty Pygame window. Later, we'll draw the game elements, such as the ship and the aliens, on this window. We'll also make our game respond to user input, set the background color, and load a ship image.

Creating a Pygame Window and Responding to User Input

We'll make an empty Pygame window by creating a class to represent the game. In your text editor, create a new file and save it as *alien_invasion.py*; then enter the following:

alien_invasion.py

```
import sys

import pygame

class AlienInvasion:
    """Overall class to manage game assets and behavior."""

    def __init__(self):
        """Initialize the game, and create game resources."""
❶       pygame.init()

❷       self.screen = pygame.display.set_mode((1200, 800))
        pygame.display.set_caption("Alien Invasion")

    def run_game(self):
        """Start the main loop for the game."""
❸       while True:
            # Watch for keyboard and mouse events.
❹           for event in pygame.event.get():
❺               if event.type == pygame.QUIT:
                    sys.exit()

            # Make the most recently drawn screen visible.
❻           pygame.display.flip()

if __name__ == '__main__':
    # Make a game instance, and run the game.
    ai = AlienInvasion()
    ai.run_game()
```

First, we import the `sys` and `pygame` modules. The `pygame` module contains the functionality we need to make a game. We'll use tools in the `sys` module to exit the game when the player quits.

Alien Invasion starts as a class called `AlienInvasion`. In the `__init__()` method, the `pygame.init()` function initializes the background settings that Pygame needs to work properly ❶. At ❷, we call `pygame.display.set_mode()` to create a display window, on which we'll draw all the game's graphical elements. The argument `(1200, 800)` is a tuple that defines the dimensions of the game window, which will be 1200 pixels wide by 800 pixels high. (You can adjust these values depending on your display size.) We assign this display window to the attribute `self.screen`, so it will be available in all methods in the class.

The object we assigned to `self.screen` is called a *surface*. A surface in Pygame is a part of the screen where a game element can be displayed. Each element in the game, like an alien or a ship, is its own surface. The surface returned by `display.set_mode()` represents the entire game window. When we activate the game's animation loop, this surface will be redrawn on every pass through the loop, so it can be updated with any changes triggered by user input.

The game is controlled by the `run_game()` method. This method contains a `while` loop ❸ that runs continually. The `while` loop contains an event loop and code that manages screen updates. An *event* is an action that the user performs while playing the game, such as pressing a key or moving the mouse. To make our program respond to events, we write this *event loop* to *listen* for events and perform appropriate tasks depending on the kinds of events that occur. The `for` loop at ❹ is an event loop.

To access the events that Pygame detects, we'll use the `pygame.event.get()` function. This function returns a list of events that have taken place since the last time this function was called. Any keyboard or mouse event will cause this `for` loop to run. Inside the loop, we'll write a series of `if` statements to detect and respond to specific events. For example, when the player clicks the game window's close button, a `pygame.QUIT` event is detected and we call `sys.exit()` to exit the game ❺.

The call to `pygame.display.flip()` at ❻ tells Pygame to make the most recently drawn screen visible. In this case, it simply draws an empty screen on each pass through the `while` loop, erasing the old screen so only the new screen is visible. When we move the game elements around, `pygame.display.flip()` continually updates the display to show the new positions of game elements and hides the old ones, creating the illusion of smooth movement.

At the end of the file, we create an instance of the game, and then call `run_game()`. We place `run_game()` in an `if` block that only runs if the file is called directly. When you run this *alien_invasion.py* file, you should see an empty Pygame window.

Setting the Background Color

Pygame creates a black screen by default, but that's boring. Let's set a different background color. We'll do this at the end of the `__init__()` method.

alien_invasion.py

```
    def __init__(self):
        --snip--
        pygame.display.set_caption("Alien Invasion")

        # Set the background color.
❶       self.bg_color = (230, 230, 230)

    def run_game(self):
        --snip--
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    sys.exit()

            # Redraw the screen during each pass through the loop.
❷           self.screen.fill(self.bg_color)

            # Make the most recently drawn screen visible.
            pygame.display.flip()
```

Colors in Pygame are specified as RGB colors: a mix of red, green, and blue. Each color value can range from 0 to 255. The color value (255, 0, 0) is red, (0, 255, 0) is green, and (0, 0, 255) is blue. You can mix different RGB values to create up to 16 million colors. The color value (230, 230, 230) mixes equal amounts of red, blue, and green, which produces a light gray background color. We assign this color to `self.bg_color` ❶.

At ❷, we fill the screen with the background color using the `fill()` method, which acts on a surface and takes only one argument: a color.

Creating a Settings Class

Each time we introduce new functionality into the game, we'll typically create some new settings as well. Instead of adding settings throughout the code, let's write a module called `settings` that contains a class called `Settings` to store all these values in one place. This approach allows us to work with just one settings object any time we need to access an individual setting. This also makes it easier to modify the game's appearance and behavior as our project grows: to modify the game, we'll simply change some values in *settings.py*, which we'll create next, instead of searching for different settings throughout the project.

Create a new file named *settings.py* inside your *alien_invasion* folder, and add this initial `Settings` class:

settings.py

```
class Settings:
    """A class to store all settings for Alien Invasion."""

    def __init__(self):
        """Initialize the game's settings."""
        # Screen settings
        self.screen_width = 1200
        self.screen_height = 800
        self.bg_color = (230, 230, 230)
```

To make an instance of `Settings` in the project and use it to access our settings, we need to modify *alien_invasion.py* as follows:

alien_invasion.py

```
--snip--
import pygame

from settings import Settings

class AlienInvasion:
    """Overall class to manage game assets and behavior."""

    def __init__(self):
        """Initialize the game, and create game resources."""
        pygame.init()
❶       self.settings = Settings()

❷       self.screen = pygame.display.set_mode(
            (self.settings.screen_width, self.settings.screen_height))
        pygame.display.set_caption("Alien Invasion")

    def run_game(self):
            --snip--
            # Redraw the screen during each pass through the loop.
❸           self.screen.fill(self.settings.bg_color)

            # Make the most recently drawn screen visible.
            pygame.display.flip()
--snip--
```

We import `Settings` into the main program file. Then we create an instance of `Settings` and assign it to `self.settings` ❶, after making the call to `pygame.init()`. When we create a screen ❷, we use the `screen_width` and `screen_height` attributes of `self.settings`, and then we use `self.settings` to access the background color when filling the screen at ❸ as well.

When you run *alien_invasion.py* now you won't yet see any changes, because all we've done is move the settings we were already using elsewhere. Now we're ready to start adding new elements to the screen.

Adding the Ship Image

Let's add the ship to our game. To draw the player's ship on the screen, we'll load an image and then use the Pygame `blit()` method to draw the image.

When you're choosing artwork for your games, be sure to pay attention to licensing. The safest and cheapest way to start is to use freely licensed graphics that you can use and modify, from a website like *https://pixabay.com/*.

You can use almost any type of image file in your game, but it's easiest when you use a bitmap (*.bmp*) file because Pygame loads bitmaps by default. Although you can configure Pygame to use other file types, some file types

depend on certain image libraries that must be installed on your computer. Most images you'll find are in *.jpg* or *.png* formats, but you can convert them to bitmaps using tools like Photoshop, GIMP, and Paint.

Pay particular attention to the background color in your chosen image. Try to find a file with a transparent or solid background that you can replace with any background color using an image editor. Your games will look best if the image's background color matches your game's background color. Alternatively, you can match your game's background to the image's background.

For *Alien Invasion*, you can use the file *ship.bmp* (Figure 12-1), which is available in the book's resources at *https://nostarch.com/pythoncrashcourse2e/*. The file's background color matches the settings we're using in this project. Make a folder called *images* inside your main *alien_invasion* project folder. Save the file *ship.bmp* in the *images* folder.

Figure 12-1: The ship for Alien Invasion

Creating the Ship Class

After choosing an image for the ship, we need to display it on the screen. To use our ship, we'll create a new `ship` module that will contain the class `Ship`. This class will manage most of the behavior of the player's ship:

ship.py

```
import pygame

class Ship:
    """A class to manage the ship."""

    def __init__(self, ai_game):
        """Initialize the ship and set its starting position."""
❶       self.screen = ai_game.screen
❷       self.screen_rect = ai_game.screen.get_rect()

        # Load the ship image and get its rect.
❸       self.image = pygame.image.load('images/ship.bmp')
        self.rect = self.image.get_rect()
```

```
        # Start each new ship at the bottom center of the screen.
❹       self.rect.midbottom = self.screen_rect.midbottom

❺   def blitme(self):
        """Draw the ship at its current location."""
        self.screen.blit(self.image, self.rect)
```

Pygame is efficient because it lets you treat all game elements like rectangles (*rects*), even if they're not exactly shaped like rectangles. Treating an element as a rectangle is efficient because rectangles are simple geometric shapes. When Pygame needs to figure out whether two game elements have collided, for example, it can do this more quickly if it treats each object as a rectangle. This approach usually works well enough that no one playing the game will notice that we're not working with the exact shape of each game element. We'll treat the ship and the screen as rectangles in this class.

We import the `pygame` module before defining the class. The `__init__()` method of `Ship` takes two parameters: the `self` reference and a reference to the current instance of the `AlienInvasion` class. This will give `Ship` access to all the game resources defined in `AlienInvasion`. At ❶ we assign the screen to an attribute of `Ship`, so we can access it easily in all the methods in this class. At ❷ we access the screen's `rect` attribute using the `get_rect()` method and assign it to `self.screen_rect`. Doing so allows us to place the ship in the correct location on the screen.

To load the image, we call `pygame.image.load()` ❸ and give it the location of our ship image. This function returns a surface representing the ship, which we assign to `self.image`. When the image is loaded, we call `get_rect()` to access the ship surface's `rect` attribute so we can later use it to place the ship.

When you're working with a `rect` object, you can use the x- and y-coordinates of the top, bottom, left, and right edges of the rectangle, as well as the center, to place the object. You can set any of these values to establish the current position of the rect. When you're centering a game element, work with the `center`, `centerx`, or `centery` attributes of a rect. When you're working at an edge of the screen, work with the `top`, `bottom`, `left`, or `right` attributes. There are also attributes that combine these properties, such as `midbottom`, `midtop`, `midleft`, and `midright`. When you're adjusting the horizontal or vertical placement of the rect, you can just use the `x` and `y` attributes, which are the x- and y-coordinates of its top-left corner. These attributes spare you from having to do calculations that game developers formerly had to do manually, and you'll use them often.

NOTE *In Pygame, the origin (0, 0) is at the top-left corner of the screen, and coordinates increase as you go down and to the right. On a 1200 by 800 screen, the origin is at the top-left corner, and the bottom-right corner has the coordinates (1200, 800). These coordinates refer to the game window, not the physical screen.*

We'll position the ship at the bottom center of the screen. To do so, make the value of `self.rect.midbottom` match the `midbottom` attribute of the screen's rect ❹. Pygame uses these `rect` attributes to position the ship image so it's centered horizontally and aligned with the bottom of the screen.

At ❺, we define the `blitme()` method, which draws the image to the screen at the position specified by `self.rect`.

Drawing the Ship to the Screen

Now let's update *alien_invasion.py* so it creates a ship and calls the ship's `blitme()` method:

alien_invasion.py

```
--snip--
from settings import Settings
from ship import Ship

class AlienInvasion:
    """Overall class to manage game assets and behavior."""

    def __init__(self):
        --snip--
        pygame.display.set_caption("Alien Invasion")

❶       self.ship = Ship(self)

    def run_game(self):
            --snip--
            # Redraw the screen during each pass through the loop.
            self.screen.fill(self.settings.bg_color)
❷           self.ship.blitme()

            # Make the most recently drawn screen visible.
            pygame.display.flip()
--snip--
```

We import `Ship` and then make an instance of `Ship` after the screen has been created ❶. The call to `Ship()` requires one argument, an instance of `AlienInvasion`. The `self` argument here refers to the current instance of `AlienInvasion`. This is the parameter that gives `Ship` access to the game's resources, such as the `screen` object. We assign this `Ship` instance to `self.ship`.

After filling the background, we draw the ship on the screen by calling `ship.blitme()`, so the ship appears on top of the background ❷.

When you run *alien_invasion.py* now, you should see an empty game screen with the rocket ship sitting at the bottom center, as shown in Figure 12-2.

Figure 12-2: Alien Invasion *with the ship at the bottom center of the screen*

Refactoring: The _check_events() and _update_screen() Methods

In large projects, you'll often refactor code you've written before adding more code. Refactoring simplifies the structure of the code you've already written, making it easier to build on. In this section, we'll break the `run_game()` method, which is getting lengthy, into two helper methods. A *helper method* does work inside a class but isn't meant to be called through an instance. In Python, a single leading underscore indicates a helper method.

The _check_events() Method

We'll move the code that manages events to a separate method called `_check_events()`. This will simplify `run_game()` and isolate the event management loop. Isolating the event loop allows you to manage events separately from other aspects of the game, such as updating the screen.

Here's the `AlienInvasion` class with the new `_check_events()` method, which only affects the code in `run_game()`:

alien_invasion.py

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
❶           self._check_events()
```

```
        # Redraw the screen during each pass through the loop.
        --snip--

❷   def _check_events(self):
        """Respond to keypresses and mouse events."""
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
```

We make a new `_check_events()` method ❷ and move the lines that check whether the player has clicked to close the window into this new method.

To call a method from within a class, use dot notation with the variable `self` and the name of the method ❶. We call the method from inside the `while` loop in `run_game()`.

The _update_screen() Method

To further simplify `run_game()`, we'll move the code for updating the screen to a separate method called `_update_screen()`:

alien_invasion.py

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
            self._check_events()
            self._update_screen()

    def _check_events(self):
        --snip--

    def _update_screen(self):
        """Update images on the screen, and flip to the new screen."""
        self.screen.fill(self.settings.bg_color)
        self.ship.blitme()

        pygame.display.flip()
```

We moved the code that draws the background and the ship and flips the screen to `_update_screen()`. Now the body of the main loop in `run_game()` is much simpler. It's easy to see that we're looking for new events and updating the screen on each pass through the loop.

If you've already built a number of games, you'll probably start out by breaking your code into methods like these. But if you've never tackled a project like this, you probably won't know how to structure your code. This approach of writing code that works and then restructuring it as it grows more complex gives you an idea of a realistic development process: you start out writing your code as simply as possible, and then refactor it as your project becomes more complex.

Now that we've restructured the code to make it easier to add to, we can work on the dynamic aspects of the game!

TRY IT YOURSELF

12-1. Blue Sky: Make a Pygame window with a blue background.

12-2. Game Character: Find a bitmap image of a game character you like or convert an image to a bitmap. Make a class that draws the character at the center of the screen and match the background color of the image to the background color of the screen, or vice versa.

Piloting the Ship

Next, we'll give the player the ability to move the ship right and left. We'll write code that responds when the player presses the right or left arrow key. We'll focus on movement to the right first, and then we'll apply the same principles to control movement to the left. As we add this code, you'll learn how to control the movement of images on the screen and respond to user input.

Responding to a Keypress

Whenever the player presses a key, that keypress is registered in Pygame as an event. Each event is picked up by the `pygame.event.get()` method. We need to specify in our `_check_events()` method what kind of events we want the game to check for. Each keypress is registered as a `KEYDOWN` event.

When Pygame detects a `KEYDOWN` event, we need to check whether the key that was pressed is one that triggers a certain action. For example, if the player presses the right arrow key, we want to increase the ship's `rect.x` value to move the ship to the right:

alien_invasion.py

```
    def _check_events(self):
        """Respond to keypresses and mouse events."""
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
❶           elif event.type == pygame.KEYDOWN:
❷               if event.key == pygame.K_RIGHT:
                    # Move the ship to the right.
❸                   self.ship.rect.x += 1
```

Inside `_check_events()` we add an `elif` block to the event loop to respond when Pygame detects a `KEYDOWN` event ❶. We check whether the key pressed, `event.key`, is the right arrow key ❷. The right arrow key is represented by `pygame.K_RIGHT`. If the right arrow key was pressed, we move the ship to the right by increasing the value of `self.ship.rect.x` by 1 ❸.

When you run *alien_invasion.py* now, the ship should move to the right one pixel every time you press the right arrow key. That's a start, but it's not an efficient way to control the ship. Let's improve this control by allowing continuous movement.

Allowing Continuous Movement

When the player holds down the right arrow key, we want the ship to continue moving right until the player releases the key. We'll have the game detect a `pygame.KEYUP` event so we'll know when the right arrow key is released; then we'll use the `KEYDOWN` and `KEYUP` events together with a flag called `moving_right` to implement continuous motion.

When the `moving_right` flag is `False`, the ship will be motionless. When the player presses the right arrow key, we'll set the flag to `True`, and when the player releases the key, we'll set the flag to `False` again.

The `Ship` class controls all attributes of the ship, so we'll give it an attribute called `moving_right` and an `update()` method to check the status of the `moving_right` flag. The `update()` method will change the position of the ship if the flag is set to `True`. We'll call this method once on each pass through the `while` loop to update the position of the ship.

Here are the changes to `Ship`:

ship.py

```
class Ship:
    """A class to manage the ship."""

    def __init__(self, ai_game):
        --snip--
        # Start each new ship at the bottom center of the screen.
        self.rect.midbottom = self.screen_rect.midbottom

        # Movement flag
❶       self.moving_right = False

❷   def update(self):
        """Update the ship's position based on the movement flag."""
        if self.moving_right:
            self.rect.x += 1

    def blitme(self):
        --snip--
```

We add a `self.moving_right` attribute in the `__init__()` method and set it to `False` initially ❶. Then we add `update()`, which moves the ship right if the flag is `True` ❷. The `update()` method will be called through an instance of `Ship`, so it's not considered a helper method.

Now we need to modify `_check_events()` so that `moving_right` is set to `True` when the right arrow key is pressed and `False` when the key is released:

alien_invasion.py

```
    def _check_events(self):
        """Respond to keypresses and mouse events."""
        for event in pygame.event.get():
            --snip--
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_RIGHT:
❶                   self.ship.moving_right = True
❷           elif event.type == pygame.KEYUP:
```

```
            if event.key == pygame.K_RIGHT:
                self.ship.moving_right = False
```

At ❶, we modify how the game responds when the player presses the right arrow key: instead of changing the ship's position directly, we merely set `moving_right` to `True`. At ❷, we add a new `elif` block, which responds to `KEYUP` events. When the player releases the right arrow key (`K_RIGHT`), we set `moving_right` to `False`.

Next, we modify the `while` loop in `run_game()` so it calls the ship's `update()` method on each pass through the loop:

alien_invasion.py

```
def run_game(self):
    """Start the main loop for the game."""
    while True:
        self._check_events()
        self.ship.update()
        self._update_screen()
```

The ship's position will be updated after we've checked for keyboard events and before we update the screen. This allows the ship's position to be updated in response to player input and ensures the updated position will be used when drawing the ship to the screen.

When you run *alien_invasion.py* and hold down the right arrow key, the ship should move continuously to the right until you release the key.

Moving Both Left and Right

Now that the ship can move continuously to the right, adding movement to the left is straightforward. Again, we'll modify the `Ship` class and the `_check_events()` method. Here are the relevant changes to `__init__()` and `update()` in `Ship`:

ship.py

```
def __init__(self, ai_game):
    --snip--
    # Movement flags
    self.moving_right = False
    self.moving_left = False

def update(self):
    """Update the ship's position based on movement flags."""
    if self.moving_right:
        self.rect.x += 1
    if self.moving_left:
        self.rect.x -= 1
```

In `__init__()`, we add a `self.moving_left` flag. In `update()`, we use two separate `if` blocks rather than an `elif` to allow the ship's `rect.x` value to be increased and then decreased when both arrow keys are held down. This results in the ship standing still. If we used `elif` for motion to the left, the

right arrow key would always have priority. Doing it this way makes the movements more accurate when switching from right to left when the player might momentarily hold down both keys.

We have to make two adjustments to `_check_events()`:

alien_invasion.py

```
    def _check_events(self):
        """Respond to keypresses and mouse events."""
        for event in pygame.event.get():
            --snip--
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_RIGHT:
                    self.ship.moving_right = True
                elif event.key == pygame.K_LEFT:
                    self.ship.moving_left = True

            elif event.type == pygame.KEYUP:
                if event.key == pygame.K_RIGHT:
                    self.ship.moving_right = False
                elif event.key == pygame.K_LEFT:
                    self.ship.moving_left = False
```

If a `KEYDOWN` event occurs for the `K_LEFT` key, we set `moving_left` to `True`. If a `KEYUP` event occurs for the `K_LEFT` key, we set `moving_left` to `False`. We can use `elif` blocks here because each event is connected to only one key. If the player presses both keys at once, two separate events will be detected.

When you run *alien_invasion.py* now, you should be able to move the ship continuously to the right and left. If you hold down both keys, the ship should stop moving.

Next, we'll further refine the ship's movement. Let's adjust the ship's speed and limit how far the ship can move so it can't disappear off the sides of the screen.

Adjusting the Ship's Speed

Currently, the ship moves one pixel per cycle through the `while` loop, but we can take finer control of the ship's speed by adding a `ship_speed` attribute to the `Settings` class. We'll use this attribute to determine how far to move the ship on each pass through the loop. Here's the new attribute in *settings.py*:

settings.py

```
class Settings:
    """A class to store all settings for Alien Invasion."""

    def __init__(self):
        --snip--

        # Ship settings
        self.ship_speed = 1.5
```

We set the initial value of `ship_speed` to `1.5`. When the ship moves now, its position is adjusted by 1.5 pixels rather than 1 pixel on each pass through the loop.

We're using decimal values for the speed setting to give us finer control of the ship's speed when we increase the tempo of the game later on. However, rect attributes such as `x` store only integer values, so we need to make some modifications to `Ship`:

ship.py

```
class Ship:
    """A class to manage the ship."""

❶   def __init__(self, ai_game):
        """Initialize the ship and set its starting position."""
        self.screen = ai_game.screen
        self.settings = ai_game.settings
        --snip--

        # Start each new ship at the bottom center of the screen.
        --snip--

        # Store a decimal value for the ship's horizontal position.
❷       self.x = float(self.rect.x)

        # Movement flags
        self.moving_right = False
        self.moving_left = False

    def update(self):
        """Update the ship's position based on movement flags."""
        # Update the ship's x value, not the rect.
        if self.moving_right:
❸           self.x += self.settings.ship_speed
        if self.moving_left:
            self.x -= self.settings.ship_speed

        # Update rect object from self.x.
❹       self.rect.x = self.x

    def blitme(self):
        --snip--
```

We create a `settings` attribute for `Ship`, so we can use it in `update()` ❶. Because we're adjusting the position of the ship by fractions of a pixel, we need to assign the position to a variable that can store a decimal value. You can use a decimal value to set an attribute of `rect`, but the `rect` will only keep the integer portion of that value. To keep track of the ship's position accurately, we define a new `self.x` attribute that can hold decimal values ❷. We use the `float()` function to convert the value of `self.rect.x` to a decimal and assign this value to `self.x`.

Now when we change the ship's position in `update()`, the value of `self.x` is adjusted by the amount stored in `settings.ship_speed` ❸. After `self.x` has been updated, we use the new value to update `self.rect.x`, which controls

the position of the ship ❹. Only the integer portion of `self.x` will be stored in `self.rect.x`, but that's fine for displaying the ship.

Now we can change the value of `ship_speed`, and any value greater than one will make the ship move faster. This will help make the ship respond quickly enough to shoot down aliens, and it will let us change the tempo of the game as the player progresses in gameplay.

NOTE *If you're using macOS, you might notice that the ship moves very slowly, even with a high speed setting. You can remedy this problem by running the game in fullscreen mode, which we'll implement shortly.*

Limiting the Ship's Range

At this point, the ship will disappear off either edge of the screen if you hold down an arrow key long enough. Let's correct this so the ship stops moving when it reaches the screen's edge. We do this by modifying the `update()` method in `Ship`:

ship.py

```
    def update(self):
        """Update the ship's position based on movement flags."""
        # Update the ship's x value, not the rect.
❶       if self.moving_right and self.rect.right < self.screen_rect.right:
            self.x += self.settings.ship_speed
❷       if self.moving_left and self.rect.left > 0:
            self.x -= self.settings.ship_speed

        # Update rect object from self.x.
        self.rect.x = self.x
```

This code checks the position of the ship before changing the value of `self.x`. The code `self.rect.right` returns the x-coordinate of the right edge of the ship's `rect`. If this value is less than the value returned by `self.screen_rect.right`, the ship hasn't reached the right edge of the screen ❶. The same goes for the left edge: if the value of the left side of the `rect` is greater than zero, the ship hasn't reached the left edge of the screen ❷. This ensures the ship is within these bounds before adjusting the value of `self.x`.

When you run *alien_invasion.py* now, the ship should stop moving at either edge of the screen. This is pretty cool; all we've done is add a conditional test in an `if` statement, but it feels like the ship hits a wall or a force field at either edge of the screen!

Refactoring _check_events()

The `_check_events()` method will increase in length as we continue to develop the game, so let's break `_check_events()` into two more methods: one that handles `KEYDOWN` events and another that handles `KEYUP` events:

alien_invasion.py

```
    def _check_events(self):
        """Respond to keypresses and mouse events."""
        for event in pygame.event.get():
```

```
            if event.type == pygame.QUIT:
                sys.exit()
            elif event.type == pygame.KEYDOWN:
                self._check_keydown_events(event)
            elif event.type == pygame.KEYUP:
                self._check_keyup_events(event)

    def _check_keydown_events(self, event):
        """Respond to keypresses."""
        if event.key == pygame.K_RIGHT:
            self.ship.moving_right = True
        elif event.key == pygame.K_LEFT:
            self.ship.moving_left = True

    def _check_keyup_events(self, event):
        """Respond to key releases."""
        if event.key == pygame.K_RIGHT:
            self.ship.moving_right = False
        elif event.key == pygame.K_LEFT:
            self.ship.moving_left = False
```

We make two new helper methods: `_check_keydown_events()` and `_check_keyup_events()`. Each needs a `self` parameter and an `event` parameter. The bodies of these two methods are copied from `_check_events()`, and we've replaced the old code with calls to the new methods. The `_check_events()` method is simpler now with this cleaner code structure, which will make it easier to develop further responses to player input.

Pressing Q to Quit

Now that we're responding to keypresses efficiently, we can add another way to quit the game. It gets tedious to click the X at the top of the game window to end the game every time you test a new feature, so we'll add a keyboard shortcut to end the game when the player presses Q:

alien_invasion.py

```
    def _check_keydown_events(self, event):
        --snip--
        elif event.key == pygame.K_LEFT:
            self.ship.moving_left = True
        elif event.key == pygame.K_q:
            sys.exit()
```

In `_check_keydown_events()`, we add a new block that ends the game when the player presses Q. Now, when testing, you can press Q to close the game rather than using your cursor to close the window.

Running the Game in Fullscreen Mode

Pygame has a fullscreen mode that you might like better than running the game in a regular window. Some games look better in fullscreen mode, and macOS users might see better performance in fullscreen mode.

To run the game in fullscreen mode, make the following changes in `__init__()`:

alien_invasion.py

```
    def __init__(self):
        """Initialize the game, and create game resources."""
        pygame.init()
        self.settings = Settings()

❶       self.screen = pygame.display.set_mode((0, 0), pygame.FULLSCREEN)
❷       self.settings.screen_width = self.screen.get_rect().width
        self.settings.screen_height = self.screen.get_rect().height
        pygame.display.set_caption("Alien Invasion")
```

When creating the screen surface, we pass a size of `(0, 0)` and the parameter `pygame.FULLSCREEN` ❶. This tells Pygame to figure out a window size that will fill the screen. Because we don't know the width and height of the screen ahead of time, we update these settings after the screen is created ❷. We use the `width` and `height` attributes of the screen's rect to update the `settings` object.

If you like how the game looks or behaves in fullscreen mode, keep these settings. If you liked the game better in its own window, you can revert back to the original approach where we set a specific screen size for the game.

NOTE *Make sure you can quit by pressing Q before running the game in fullscreen mode; Pygame offers no default way to quit a game while in fullscreen mode.*

A Quick Recap

In the next section, we'll add the ability to shoot bullets, which involves adding a new file called *bullet.py* and making some modifications to some of the files we're already using. Right now, we have three files containing a number of classes and methods. To be clear about how the project is organized, let's review each of these files before adding more functionality.

alien_invasion.py

The main file, *alien_invasion.py*, contains the `AlienInvasion` class. This class creates a number of important attributes used throughout the game: the settings are assigned to `settings`, the main display surface is assigned to `screen`, and a `ship` instance is created in this file as well. The main loop of the game, a `while` loop, is also stored in this module. The `while` loop calls `_check_events()`, `ship.update()`, and `_update_screen()`.

The `_check_events()` method detects relevant events, such as keypresses and releases, and processes each of these types of events through the methods `_check_keydown_events()` and `_check_keyup_events()`. For now,

these methods manage the ship's movement. The `AlienInvasion` class also contains `_update_screen()`, which redraws the screen on each pass through the main loop.

The *alien_invasion.py* file is the only file you need to run when you want to play *Alien Invasion*. The other files—*settings.py* and *ship.py*—contain code that is imported into this file.

settings.py

The *settings.py* file contains the `Settings` class. This class only has an `__init__()` method, which initializes attributes controlling the game's appearance and the ship's speed.

ship.py

The *ship.py* file contains the `Ship` class. The `Ship` class has an `__init__()` method, an `update()` method to manage the ship's position, and a `blitme()` method to draw the ship to the screen. The image of the ship is stored in *ship.bmp*, which is in the *images* folder.

TRY IT YOURSELF

12-3. Pygame Documentation: We're far enough into the game now that you might want to look at some of the Pygame documentation. The Pygame home page is at *https://www.pygame.org/*, and the home page for the documentation is at *https://www.pygame.org/docs/*. Just skim the documentation for now. You won't need it to complete this project, but it will help if you want to modify *Alien Invasion* or make your own game afterward.

12-4. Rocket: Make a game that begins with a rocket in the center of the screen. Allow the player to move the rocket up, down, left, or right using the four arrow keys. Make sure the rocket never moves beyond any edge of the screen.

12-5. Keys: Make a Pygame file that creates an empty screen. In the event loop, print the `event.key` attribute whenever a `pygame.KEYDOWN` event is detected. Run the program and press various keys to see how Pygame responds.

Shooting Bullets

Now let's add the ability to shoot bullets. We'll write code that fires a bullet, which is represented by a small rectangle, when the player presses the spacebar. Bullets will then travel straight up the screen until they disappear off the top of the screen.

Adding the Bullet Settings

At the end of the `__init__()` method, we'll update *settings.py* to include the values we'll need for a new `Bullet` class:

settings.py

```
    def __init__(self):
        --snip--
        # Bullet settings
        self.bullet_speed = 1.0
        self.bullet_width = 3
        self.bullet_height = 15
        self.bullet_color = (60, 60, 60)
```

These settings create dark gray bullets with a width of 3 pixels and a height of 15 pixels. The bullets will travel slightly slower than the ship.

Creating the Bullet Class

Now create a *bullet.py* file to store our `Bullet` class. Here's the first part of *bullet.py*:

bullet.py

```
import pygame
from pygame.sprite import Sprite

class Bullet(Sprite):
    """A class to manage bullets fired from the ship"""

    def __init__(self, ai_game):
        """Create a bullet object at the ship's current position."""
        super().__init__()
        self.screen = ai_game.screen
        self.settings = ai_game.settings
        self.color = self.settings.bullet_color

        # Create a bullet rect at (0, 0) and then set correct position.
❶       self.rect = pygame.Rect(0, 0, self.settings.bullet_width,
            self.settings.bullet_height)
❷       self.rect.midtop = ai_game.ship.rect.midtop

        # Store the bullet's position as a decimal value.
❸       self.y = float(self.rect.y)
```

The `Bullet` class inherits from `Sprite`, which we import from the `pygame.sprite` module. When you use sprites, you can group related elements in your game and act on all the grouped elements at once. To create a bullet instance, `__init__()` needs the current instance of `AlienInvasion`, and we call `super()` to inherit properly from `Sprite`. We also set attributes for the screen and settings objects, and for the bullet's color.

At ❶, we create the bullet's `rect` attribute. The bullet isn't based on an image, so we have to build a rect from scratch using the `pygame.Rect()` class. This class requires the x- and y-coordinates of the top-left corner of the

rect, and the width and height of the rect. We initialize the rect at (0, 0), but we'll move it to the correct location in the next line, because the bullet's position depends on the ship's position. We get the width and height of the bullet from the values stored in self.settings.

At ❷, we set the bullet's midtop attribute to match the ship's midtop attribute. This will make the bullet emerge from the top of the ship, making it look like the bullet is fired from the ship. We store a decimal value for the bullet's y-coordinate so we can make fine adjustments to the bullet's speed ❸.

Here's the second part of *bullet.py*, update() and draw_bullet():

bullet.py

```
    def update(self):
        """Move the bullet up the screen."""
        # Update the decimal position of the bullet.
❶       self.y -= self.settings.bullet_speed
        # Update the rect position.
❷       self.rect.y = self.y

    def draw_bullet(self):
        """Draw the bullet to the screen."""
❸       pygame.draw.rect(self.screen, self.color, self.rect)
```

The update() method manages the bullet's position. When a bullet is fired, it moves up the screen, which corresponds to a decreasing y-coordinate value. To update the position, we subtract the amount stored in settings .bullet_speed from self.y ❶. We then use the value of self.y to set the value of self.rect.y ❷.

The bullet_speed setting allows us to increase the speed of the bullets as the game progresses or as needed to refine the game's behavior. Once a bullet is fired, we never change the value of its x-coordinate, so it will travel vertically in a straight line even if the ship moves.

When we want to draw a bullet, we call draw_bullet(). The draw.rect() function fills the part of the screen defined by the bullet's rect with the color stored in self.color ❸.

Storing Bullets in a Group

Now that we have a Bullet class and the necessary settings defined, we can write code to fire a bullet each time the player presses the spacebar. We'll create a group in AlienInvasion to store all the live bullets so we can manage the bullets that have already been fired. This group will be an instance of the pygame.sprite.Group class, which behaves like a list with some extra functionality that's helpful when building games. We'll use this group to draw bullets to the screen on each pass through the main loop and to update each bullet's position.

We'll create the group in __init__():

alien_invasion.py

```
    def __init__(self):
        --snip--
        self.ship = Ship(self)
        self.bullets = pygame.sprite.Group()
```

Then we need to update the position of the bullets on each pass through the `while` loop:

alien_invasion.py

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
            self._check_events()
            self.ship.update()
❶           self.bullets.update()
            self._update_screen()
```

When you call `update()` on a group ❶, the group automatically calls `update()` for each sprite in the group. The line `self.bullets.update()` calls `bullet.update()` for each bullet we place in the group `bullets`.

Firing Bullets

In `AlienInvasion`, we need to modify `_check_keydown_events()` to fire a bullet when the player presses the spacebar. We don't need to change `_check_keyup_events()` because nothing happens when the spacebar is released. We also need to modify `_update_screen()` to make sure each bullet is drawn to the screen before we call `flip()`.

We know there will be a bit of work to do when we fire a bullet, so let's write a new method, `_fire_bullet()`, to handle this work:

alien_invasion.py

```
--snip--
from ship import Ship
❶ from bullet import Bullet

class AlienInvasion:
    --snip--
    def _check_keydown_events(self, event):
        --snip--
        elif event.key == pygame.K_q:
            sys.exit()
❷       elif event.key == pygame.K_SPACE:
            self._fire_bullet()

    def _check_keyup_events(self, event):
        --snip--

    def _fire_bullet(self):
        """Create a new bullet and add it to the bullets group."""
❸       new_bullet = Bullet(self)
❹       self.bullets.add(new_bullet)

    def _update_screen(self):
        """Update images on the screen, and flip to the new screen."""
        self.screen.fill(self.settings.bg_color)
        self.ship.blitme()
❺       for bullet in self.bullets.sprites():
            bullet.draw_bullet()
```

```
        pygame.display.flip()
--snip--
```

First, we import `Bullet` ❶. Then we call `_fire_bullet()` when the spacebar is pressed ❷. In `_fire_bullet()`, we make an instance of `Bullet` and call it `new_bullet` ❸. We then add it to the group `bullets` using the `add()` method ❹. The `add()` method is similar to `append()`, but it's a method that's written specifically for Pygame groups.

The `bullets.sprites()` method returns a list of all sprites in the group `bullets`. To draw all fired bullets to the screen, we loop through the sprites in `bullets` and call `draw_bullet()` on each one ❺.

When you run *alien_invasion.py* now, you should be able to move the ship right and left, and fire as many bullets as you want. The bullets travel up the screen and disappear when they reach the top, as shown in Figure 12-3. You can alter the size, color, and speed of the bullets in *settings.py*.

Figure 12-3: The ship after firing a series of bullets

Deleting Old Bullets

At the moment, the bullets disappear when they reach the top, but only because Pygame can't draw them above the top of the screen. The bullets actually continue to exist; their y-coordinate values just grow increasingly negative. This is a problem, because they continue to consume memory and processing power.

We need to get rid of these old bullets, or the game will slow down from doing so much unnecessary work. To do this, we need to detect when the `bottom` value of a bullet's `rect` has a value of 0, which indicates the bullet has passed off the top of the screen:

alien_invasion.py

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
            self._check_events()
            self.ship.update()
            self.bullets.update()

            # Get rid of bullets that have disappeared.
❶           for bullet in self.bullets.copy():
❷               if bullet.rect.bottom <= 0:
❸                   self.bullets.remove(bullet)
❹           print(len(self.bullets))

            self._update_screen()
```

When you use a `for` loop with a list (or a group in Pygame), Python expects that the list will stay the same length as long as the loop is running. Because we can't remove items from a list or group within a `for` loop, we have to loop over a copy of the group. We use the `copy()` method to set up the `for` loop ❶, which enables us to modify `bullets` inside the loop. We check each bullet to see whether it has disappeared off the top of the screen at ❷. If it has, we remove it from `bullets` ❸. At ❹ we insert a `print()` call to show how many bullets currently exist in the game and verify that they're being deleted when they reach the top of the screen.

If this code works correctly, we can watch the terminal output while firing bullets and see that the number of bullets decreases to zero after each series of bullets has cleared the top of the screen. After you run the game and verify that bullets are being deleted properly, remove the `print()` call. If you leave it in, the game will slow down significantly because it takes more time to write output to the terminal than it does to draw graphics to the game window.

Limiting the Number of Bullets

Many shooting games limit the number of bullets a player can have on the screen at one time; doing so encourages players to shoot accurately. We'll do the same in *Alien Invasion*.

First, store the number of bullets allowed in *settings.py*:

settings.py

```
        # Bullet settings
        --snip--
        self.bullet_color = (60, 60, 60)
        self.bullets_allowed = 3
```

This limits the player to three bullets at a time. We'll use this setting in `AlienInvasion` to check how many bullets exist before creating a new bullet in `_fire_bullet()`:

alien_invasion.py

```
    def _fire_bullet(self):
        """Create a new bullet and add it to the bullets group."""
        if len(self.bullets) < self.settings.bullets_allowed:
            new_bullet = Bullet(self)
            self.bullets.add(new_bullet)
```

When the player presses the spacebar, we check the length of `bullets`. If `len(self.bullets)` is less than three, we create a new bullet. But if three bullets are already active, nothing happens when the spacebar is pressed. When you run the game now, you should be able to fire bullets only in groups of three.

Creating the _update_bullets() Method

We want to keep the `AlienInvasion` class reasonably well organized, so now that we've written and checked the bullet management code, we can move it to a separate method. We'll create a new method called `_update_bullets()` and add it just before `_update_screen()`:

alien_invasion.py

```
    def _update_bullets(self):
        """Update position of bullets and get rid of old bullets."""
        # Update bullet positions.
        self.bullets.update()

        # Get rid of bullets that have disappeared.
        for bullet in self.bullets.copy():
            if bullet.rect.bottom <= 0:
                self.bullets.remove(bullet)
```

The code for `_update_bullets()` is cut and pasted from `run_game()`; all we've done here is clarify the comments.

The `while` loop in `run_game()` looks simple again:

alien_invasion.py

```
        while True:
            self._check_events()
            self.ship.update()
            self._update_bullets()
            self._update_screen()
```

Now our main loop contains only minimal code, so we can quickly read the method names and understand what's happening in the game. The main loop checks for player input, and then updates the position of the ship and any bullets that have been fired. We then use the updated positions to draw a new screen.

Run *alien_invasion.py* one more time, and make sure you can still fire bullets without errors.

TRY IT YOURSELF

12-6. Sideways Shooter: Write a game that places a ship on the left side of the screen and allows the player to move the ship up and down. Make the ship fire a bullet that travels right across the screen when the player presses the spacebar. Make sure bullets are deleted once they disappear off the screen.

Summary

In this chapter, you learned to make a plan for a game and learned the basic structure of a game written in Pygame. You learned to set a background color and store settings in a separate class where you can adjust them more easily. You saw how to draw an image to the screen and give the player control over the movement of game elements. You created elements that move on their own, like bullets flying up a screen, and deleted objects that are no longer needed. You also learned to refactor code in a project on a regular basis to facilitate ongoing development.

In Chapter 13, we'll add aliens to *Alien Invasion*. By the end of the chapter, you'll be able to shoot down aliens, hopefully before they reach your ship!

13

ALIENS!

In this chapter, we'll add aliens to *Alien Invasion*. We'll add one alien near the top of the screen and then generate a whole fleet of aliens. We'll make the fleet advance sideways and down, and we'll get rid of any aliens hit by a bullet. Finally, we'll limit the number of ships a player has and end the game when the player runs out of ships.

As you work through this chapter, you'll learn more about Pygame and about managing a large project. You'll also learn to detect collisions between game objects, like bullets and aliens. Detecting collisions helps you define interactions between elements in your games: for example, you can

confine a character inside the walls of a maze or pass a ball between two characters. We'll continue to work from a plan that we revisit occasionally to maintain the focus of our code-writing sessions.

Before we start writing new code to add a fleet of aliens to the screen, let's look at the project and update our plan.

Reviewing the Project

When you're beginning a new phase of development on a large project, it's always a good idea to revisit your plan and clarify what you want to accomplish with the code you're about to write. In this chapter, we'll:

- Examine our code and determine if we need to refactor before implementing new features.
- Add a single alien to the top-left corner of the screen with appropriate spacing around it.
- Use the spacing around the first alien and the overall screen size to determine how many aliens can fit on the screen. We'll write a loop to create aliens to fill the upper portion of the screen.
- Make the fleet move sideways and down until the entire fleet is shot down, an alien hits the ship, or an alien reaches the ground. If the entire fleet is shot down, we'll create a new fleet. If an alien hits the ship or the ground, we'll destroy the ship and create a new fleet.
- Limit the number of ships the player can use, and end the game when the player has used up the allotted number of ships.

We'll refine this plan as we implement features, but this is sufficient to start with.

You should also review your existing code when you begin working on a new series of features in a project. Because each new phase typically makes a project more complex, it's best to clean up any cluttered or inefficient code. We've been refactoring as we go, so there isn't any code that we need to refactor at this point.

Creating the First Alien

Placing one alien on the screen is like placing a ship on the screen. Each alien's behavior is controlled by a class called `Alien`, which we'll structure like the `Ship` class. We'll continue using bitmap images for simplicity. You can find your own image for an alien or use the one shown in Figure 13-1, which is available in the book's resources at *https://nostarch.com/pythoncrashcourse2e/*. This image has a gray background, which matches the screen's background color. Make sure you save the image file you choose in the *images* folder.

Figure 13-1: The alien we'll use to build the fleet

Creating the Alien Class

Now we'll write the `Alien` class and save it as *alien.py*:

alien.py

```
import pygame
from pygame.sprite import Sprite

class Alien(Sprite):
    """A class to represent a single alien in the fleet."""

    def __init__(self, ai_game):
        """Initialize the alien and set its starting position."""
        super().__init__()
        self.screen = ai_game.screen

        # Load the alien image and set its rect attribute.
        self.image = pygame.image.load('images/alien.bmp')
        self.rect = self.image.get_rect()

        # Start each new alien near the top left of the screen.
❶       self.rect.x = self.rect.width
        self.rect.y = self.rect.height

        # Store the alien's exact horizontal position.
❷       self.x = float(self.rect.x)
```

Most of this class is like the `Ship` class except for the aliens' placement on the screen. We initially place each alien near the top-left corner of the screen; we add a space to the left of it that's equal to the alien's width and a space above it equal to its height ❶ so it's easy to see. We're mainly

concerned with the aliens' horizontal speed, so we'll track the horizontal position of each alien precisely ❷.

This `Alien` class doesn't need a method for drawing it to the screen; instead, we'll use a Pygame group method that automatically draws all the elements of a group to the screen.

Creating an Instance of the Alien

We want to create an instance of `Alien` so we can see the first alien on the screen. Because it's part of our setup work, we'll add the code for this instance at the end of the `__init__()` method in `AlienInvasion`. Eventually, we'll create an entire fleet of aliens, which will be quite a bit of work, so we'll make a new helper method called `_create_fleet()`.

The order of methods in a class doesn't matter, as long as there's some consistency to how they're placed. I'll place `_create_fleet()` just before the `_update_screen()` method, but anywhere in `AlienInvasion` will work. First, we'll import the `Alien` class.

Here are the updated `import` statements for *alien_invasion.py*:

alien_invasion.py

```
--snip--
from bullet import Bullet
from alien import Alien
```

And here's the updated `__init__()` method:

alien_invasion.py

```
    def __init__(self):
        --snip--
        self.ship = Ship(self)
        self.bullets = pygame.sprite.Group()
        self.aliens = pygame.sprite.Group()

        self._create_fleet()
```

We create a group to hold the fleet of aliens, and we call `_create_fleet()`, which we're about to write.

Here's the new `_create_fleet()` method:

alien_invasion.py

```
    def _create_fleet(self):
        """Create the fleet of aliens."""
        # Make an alien.
        alien = Alien(self)
        self.aliens.add(alien)
```

In this method, we're creating one instance of `Alien`, and then adding it to the group that will hold the fleet. The alien will be placed in the default upper-left area of the screen, which is perfect for the first alien.

To make the alien appear, we need to call the group's `draw()` method in `_update_screen()`:

alien_invasion.py

```
    def _update_screen(self):
        --snip--
        for bullet in self.bullets.sprites():
            bullet.draw_bullet()
        self.aliens.draw(self.screen)

        pygame.display.flip()
```

When you call `draw()` on a group, Pygame draws each element in the group at the position defined by its `rect` attribute. The `draw()` method requires one argument: a surface on which to draw the elements from the group. Figure 13-2 shows the first alien on the screen.

Figure 13-2: The first alien appears.

Now that the first alien appears correctly, we'll write the code to draw an entire fleet.

Building the Alien Fleet

To draw a fleet, we need to figure out how many aliens can fit across the screen and how many rows of aliens can fit down the screen. We'll first figure out the horizontal spacing between aliens and create a row; then we'll determine the vertical spacing and create an entire fleet.

Determining How Many Aliens Fit in a Row

To figure out how many aliens fit in a row, let's look at how much horizontal space we have. The screen width is stored in `settings.screen_width`, but we need an empty margin on either side of the screen. We'll make this margin the width of one alien. Because we have two margins, the available space for aliens is the screen width minus two alien widths:

```
available_space_x = settings.screen_width - (2 * alien_width)
```

We also need to set the spacing between aliens; we'll make it one alien width. The space needed to display one alien is twice its width: one width for the alien and one width for the empty space to its right. To find the number of aliens that fit across the screen, we divide the available space by two times the width of an alien. We use *floor division* (`//`), which divides two numbers and drops any remainder, so we'll get an integer number of aliens:

```
number_aliens_x = available_space_x // (2 * alien_width)
```

We'll use these calculations when we create the fleet.

NOTE *One great aspect of calculations in programming is that you don't have to be sure your formulas are correct when you first write them. You can try them out and see if they work. At worst, you'll have a screen that's overcrowded with aliens or has too few aliens. You can then revise your calculations based on what you see on the screen.*

Creating a Row of Aliens

We're ready to generate a full row of aliens. Because our code for making a single alien works, we'll rewrite `_create_fleet()` to make a whole row of aliens:

alien_invasion.py

```
    def _create_fleet(self):
        """Create the fleet of aliens."""
        # Create an alien and find the number of aliens in a row.
        # Spacing between each alien is equal to one alien width.
❶       alien = Alien(self)
❷       alien_width = alien.rect.width
❸       available_space_x = self.settings.screen_width - (2 * alien_width)
        number_aliens_x = available_space_x // (2 * alien_width)

        # Create the first row of aliens.
❹       for alien_number in range(number_aliens_x):
            # Create an alien and place it in the row.
            alien = Alien(self)
❺           alien.x = alien_width + 2 * alien_width * alien_number
            alien.rect.x = alien.x
            self.aliens.add(alien)
```

We've already thought through most of this code. We need to know the alien's width and height to place aliens, so we create an alien at ❶ before we perform calculations. This alien won't be part of the fleet, so don't add it to the group `aliens`. At ❷ we get the alien's width from its `rect` attribute and store this value in `alien_width` so we don't have to keep working through the `rect` attribute. At ❸ we calculate the horizontal space available for aliens and the number of aliens that can fit into that space.

Next, we set up a loop that counts from 0 to the number of aliens we need to make ❹. In the main body of the loop, we create a new alien and then set its x-coordinate value to place it in the row ❺. Each alien is pushed to the right one alien width from the left margin. Next, we multiply the alien width by 2 to account for the space each alien takes up, including the empty space to its right, and we multiply this amount by the alien's position in the row. We use the alien's `x` attribute to set the position of its rect. Then we add each new alien to the group `aliens`.

When you run *Alien Invasion* now, you should see the first row of aliens appear, as in Figure 13-3.

Figure 13-3: The first row of aliens

The first row is offset to the left, which is actually good for gameplay. The reason is that we want the fleet to move right until it hits the edge of the screen, then drop down a bit, then move left, and so forth. Like the classic game *Space Invaders*, this movement is more interesting than having the fleet drop straight down. We'll continue this motion until all aliens are shot down or until an alien hits the ship or the bottom of the screen.

NOTE *Depending on the screen width you've chosen, the alignment of the first row of aliens might look slightly different on your system.*

Refactoring _create_fleet()

If the code we've written so far was all we need to create a fleet, we'd probably leave `_create_fleet()` as is. But we have more work to do, so let's clean up the method a bit. We'll add a new helper method, `_create_alien()`, and call it from `_create_fleet()`:

alien_invasion.py

```
    def _create_fleet(self):
        --snip--
        # Create the first row of aliens.
        for alien_number in range(number_aliens_x):
            self._create_alien(alien_number)

    def _create_alien(self, alien_number):
        """Create an alien and place it in the row."""
        alien = Alien(self)
        alien_width = alien.rect.width
        alien.x = alien_width + 2 * alien_width * alien_number
        alien.rect.x = alien.x
        self.aliens.add(alien)
```

The method `_create_alien()` requires one parameter in addition to `self`: it needs the alien number that's currently being created. We use the same body we made for `_create_fleet()` except that we get the width of an alien inside the method instead of passing it as an argument. This refactoring will make it easier to add new rows and create an entire fleet.

Adding Rows

To finish the fleet, we'll determine the number of rows that fit on the screen and then repeat the loop for creating the aliens in one row until we have the correct number of rows. To determine the number of rows, we find the available vertical space by subtracting the alien height from the top, the ship height from the bottom, and two alien heights from the bottom of the screen:

```
available_space_y = settings.screen_height - (3 * alien_height) - ship_height
```

The result will create some empty space above the ship, so the player has some time to start shooting aliens at the beginning of each level.

Each row needs some empty space below it, which we'll make equal to the height of one alien. To find the number of rows, we divide the available space by two times the height of an alien. We use floor division because we can only make an integer number of rows. (Again, if these calculations are off, we'll see it right away and adjust our approach until we have reasonable spacing.)

```
number_rows = available_height_y // (2 * alien_height)
```

Now that we know how many rows fit in a fleet, we can repeat the code for creating a row:

alien_invasion.py

```
    def _create_fleet(self):
        --snip--
        alien = Alien(self)
❶       alien_width, alien_height = alien.rect.size
        available_space_x = self.settings.screen_width - (2 * alien_width)
        number_aliens_x = available_space_x // (2 * alien_width)

        # Determine the number of rows of aliens that fit on the screen.
        ship_height = self.ship.rect.height
❷       available_space_y = (self.settings.screen_height -
                                (3 * alien_height) - ship_height)
        number_rows = available_space_y // (2 * alien_height)

        # Create the full fleet of aliens.
❸       for row_number in range(number_rows):
            for alien_number in range(number_aliens_x):
                self._create_alien(alien_number, row_number)

    def _create_alien(self, alien_number, row_number):
        """Create an alien and place it in the row."""
        alien = Alien(self)
        alien_width, alien_height = alien.rect.size
        alien.x = alien_width + 2 * alien_width * alien_number
        alien.rect.x = alien.x
❹       alien.rect.y = alien.rect.height + 2 * alien.rect.height * row_number
        self.aliens.add(alien)
```

We need the width and height of an alien, so at ❶ we use the attribute `size`, which contains a tuple with the width and height of a `rect` object. To calculate the number of rows we can fit on the screen, we write our `available_space_y` calculation right after the calculation for `available_space_x` ❷. The calculation is wrapped in parentheses so the outcome can be split over two lines, which results in lines of 79 characters or less, as is recommended.

To create multiple rows, we use two nested loops: one outer and one inner loop ❸. The inner loop creates the aliens in one row. The outer loop counts from 0 to the number of rows we want; Python uses the code for making a single row and repeats it `number_rows` times.

To nest the loops, write the new `for` loop and indent the code you want to repeat. (Most text editors make it easy to indent and unindent blocks of code, but for help see Appendix B.) Now when we call `_create_alien()`, we include an argument for the row number so each row can be placed farther down the screen.

The definition of `_create_alien()` needs a parameter to hold the row number. Within `_create_alien()`, we change an alien's y-coordinate value when it's not in the first row ❹ by starting with one alien's height to create empty space at the top of the screen. Each row starts two alien heights below

the previous row, so we multiply the alien height by two and then by the row number. The first row number is 0, so the vertical placement of the first row is unchanged. All subsequent rows are placed farther down the screen.

When you run the game now, you should see a full fleet of aliens, as shown in Figure 13-4.

Figure 13-4: The full fleet appears.

In the next section, we'll make the fleet move!

TRY IT YOURSELF

13-1. Stars: Find an image of a star. Make a grid of stars appear on the screen.

13-2. Better Stars: You can make a more realistic star pattern by introducing randomness when you place each star. Recall that you can get a random number like this:

```
from random import randint
random_number = randint(-10, 10)
```

This code returns a random integer between –10 and 10. Using your code in Exercise 13-1, adjust each star's position by a random amount.

Making the Fleet Move

Now let's make the fleet of aliens move to the right across the screen until it hits the edge, and then make it drop a set amount and move in the other direction. We'll continue this movement until all aliens have been shot down, one collides with the ship, or one reaches the bottom of the screen. Let's begin by making the fleet move to the right.

Moving the Aliens Right

To move the aliens, we'll use an `update()` method in *alien.py*, which we'll call for each alien in the group of aliens. First, add a setting to control the speed of each alien:

settings.py

```
    def __init__(self):
        --snip--
        # Alien settings
        self.alien_speed = 1.0
```

Then use this setting to implement `update()`:

alien.py

```
    def __init__(self, ai_game):
        """Initialize the alien and set its starting position."""
        super().__init__()
        self.screen = ai_game.screen
        self.settings = ai_game.settings
        --snip--

    def update(self):
        """Move the alien to the right."""
❶       self.x += self.settings.alien_speed
❷       self.rect.x = self.x
```

We create a settings parameter in `__init__()` so we can access the alien's speed in `update()`. Each time we update an alien's position, we move it to the right by the amount stored in `alien_speed`. We track the alien's exact position with the `self.x` attribute, which can hold decimal values ❶. We then use the value of `self.x` to update the position of the alien's `rect` ❷.

In the main `while` loop, we already have calls to update the ship and bullet positions. Now we'll add a call to update the position of each alien as well:

alien_invasion.py

```
        while True:
            self._check_events()
            self.ship.update()
            self._update_bullets()
            self._update_aliens()
            self._update_screen()
```

We're about to write some code to manage the movement of the fleet, so we create a new method called `_update_aliens()`. We set the aliens' positions to update after the bullets have been updated, because we'll soon be checking to see whether any bullets hit any aliens.

Where you place this method in the module is not critical. But to keep the code organized, I'll place it just after `_update_bullets()` to match the order of method calls in the `while` loop. Here's the first version of `_update_aliens()`:

alien_invasion.py

```
    def _update_aliens(self):
        """Update the positions of all aliens in the fleet."""
        self.aliens.update()
```

We use the `update()` method on the `aliens` group, which calls each alien's `update()` method. When you run *Alien Invasion* now, you should see the fleet move right and disappear off the side of the screen.

Creating Settings for Fleet Direction

Now we'll create the settings that will make the fleet move down the screen and to the left when it hits the right edge of the screen. Here's how to implement this behavior:

settings.py

```
        # Alien settings
        self.alien_speed = 1.0
        self.fleet_drop_speed = 10
        # fleet_direction of 1 represents right; -1 represents left.
        self.fleet_direction = 1
```

The setting `fleet_drop_speed` controls how quickly the fleet drops down the screen each time an alien reaches either edge. It's helpful to separate this speed from the aliens' horizontal speed so you can adjust the two speeds independently.

To implement the setting `fleet_direction`, we could use a text value, such as `'left'` or `'right'`, but we'd end up with `if-elif` statements testing for the fleet direction. Instead, because we have only two directions to deal with, let's use the values 1 and –1, and switch between them each time the fleet changes direction. (Using numbers also makes sense because moving right involves adding to each alien's x-coordinate value, and moving left involves subtracting from each alien's x-coordinate value.)

Checking Whether an Alien Has Hit the Edge

We need a method to check whether an alien is at either edge, and we need to modify `update()` to allow each alien to move in the appropriate direction. This code is part of the `Alien` class:

alien.py

```
    def check_edges(self):
        """Return True if alien is at edge of screen."""
        screen_rect = self.screen.get_rect()
```

```
❶         if self.rect.right >= screen_rect.right or self.rect.left <= 0:
              return True

    def update(self):
        """Move the alien right or left."""
❷       self.x += (self.settings.alien_speed *
                       self.settings.fleet_direction)
        self.rect.x = self.x
```

We can call the new method `check_edges()` on any alien to see whether it's at the left or right edge. The alien is at the right edge if the `right` attribute of its `rect` is greater than or equal to the `right` attribute of the screen's `rect`. It's at the left edge if its `left` value is less than or equal to 0 ❶.

We modify the method `update()` to allow motion to the left or right by multiplying the alien's speed by the value of `fleet_direction` ❷. If `fleet_direction` is 1, the value of `alien_speed` will be added to the alien's current position, moving the alien to the right; if `fleet_direction` is –1, the value will be subtracted from the alien's position, moving the alien to the left.

Dropping the Fleet and Changing Direction

When an alien reaches the edge, the entire fleet needs to drop down and change direction. Therefore, we need to add some code to `AlienInvasion` because that's where we'll check whether any aliens are at the left or right edge. We'll make this happen by writing the methods `_check_fleet_edges()` and `_change_fleet_direction()`, and then modifying `_update_aliens()`. I'll put these new methods after `_create_alien()`, but again the placement of these methods in the class isn't critical.

alien_invasion.py

```
    def _check_fleet_edges(self):
        """Respond appropriately if any aliens have reached an edge."""
❶       for alien in self.aliens.sprites():
            if alien.check_edges():
❷               self._change_fleet_direction()
                break

    def _change_fleet_direction(self):
        """Drop the entire fleet and change the fleet's direction."""
        for alien in self.aliens.sprites():
❸           alien.rect.y += self.settings.fleet_drop_speed
        self.settings.fleet_direction *= -1
```

In `_check_fleet_edges()`, we loop through the fleet and call `check_edges()` on each alien ❶. If `check_edges()` returns `True`, we know an alien is at an edge and the whole fleet needs to change direction; so we call `_change_fleet_direction()` and break out of the loop ❷. In `_change_fleet_direction()`, we loop through all the aliens and drop each one using the setting `fleet_drop_speed` ❸; then we change the value of `fleet_direction` by multiplying its current value by –1. The line that changes the fleet's direction isn't part of the `for` loop. We want to change each alien's vertical position, but we only want to change the direction of the fleet once.

Here are the changes to `_update_aliens()`:

alien_invasion.py

```
def _update_aliens(self):
    """
    Check if the fleet is at an edge,
      then update the positions of all aliens in the fleet.
    """
    self._check_fleet_edges()
    self.aliens.update()
```

We've modified the method by calling `_check_fleet_edges()` before updating each alien's position.

When you run the game now, the fleet should move back and forth between the edges of the screen and drop down every time it hits an edge. Now we can start shooting down aliens and watch for any aliens that hit the ship or reach the bottom of the screen.

TRY IT YOURSELF

13-3. Raindrops: Find an image of a raindrop and create a grid of raindrops. Make the raindrops fall toward the bottom of the screen until they disappear.

13-4. Steady Rain: Modify your code in Exercise 13-3 so when a row of raindrops disappears off the bottom of the screen, a new row appears at the top of the screen and begins to fall.

Shooting Aliens

We've built our ship and a fleet of aliens, but when the bullets reach the aliens, they simply pass through because we aren't checking for collisions. In game programming, *collisions* happen when game elements overlap. To make the bullets shoot down aliens, we'll use the method `sprite.groupcollide()` to look for collisions between members of two groups.

Detecting Bullet Collisions

We want to know right away when a bullet hits an alien so we can make an alien disappear as soon as it's hit. To do this, we'll look for collisions immediately after updating the position of all the bullets.

The `sprite.groupcollide()` function compares the rects of each element in one group with the rects of each element in another group. In this case, it compares each bullet's `rect` with each alien's `rect` and returns a dictionary containing the bullets and aliens that have collided. Each key in the

dictionary will be a bullet, and the corresponding value will be the alien that was hit. (We'll also use this dictionary when we implement a scoring system in Chapter 14.)

Add the following code to the end of `_update_bullets()` to check for collisions between bullets and aliens:

alien_invasion.py

```
    def _update_bullets(self):
        """Update position of bullets and get rid of old bullets."""
        --snip--

        # Check for any bullets that have hit aliens.
        #   If so, get rid of the bullet and the alien.
        collisions = pygame.sprite.groupcollide(
                self.bullets, self.aliens, True, True)
```

The new code we added compares the positions of all the bullets in `self.bullets` and all the aliens in `self.aliens`, and identifies any that overlap. Whenever the rects of a bullet and alien overlap, `groupcollide()` adds a key-value pair to the dictionary it returns. The two `True` arguments tell Pygame to delete the bullets and aliens that have collided. (To make a high-powered bullet that can travel to the top of the screen, destroying every alien in its path, you could set the first Boolean argument to `False` and keep the second Boolean argument set to `True`. The aliens hit would disappear, but all bullets would stay active until they disappeared off the top of the screen.)

When you run *Alien Invasion* now, aliens you hit should disappear. Figure 13-5 shows a fleet that has been partially shot down.

Figure 13-5: We can shoot aliens!

Making Larger Bullets for Testing

You can test many features of the game simply by running the game. But some features are tedious to test in the normal version of the game. For example, it's a lot of work to shoot down every alien on the screen multiple times to test whether your code responds to an empty fleet correctly.

To test particular features, you can change certain game settings to focus on a particular area. For example, you might shrink the screen so there are fewer aliens to shoot down or increase the bullet speed and give yourself lots of bullets at once.

My favorite change for testing *Alien Invasion* is to use really wide bullets that remain active even after they've hit an alien (see Figure 13-6). Try setting `bullet_width` to 300, or even 3000, to see how quickly you can shoot down the fleet!

Figure 13-6: Extra-powerful bullets make some aspects of the game easier to test.

Changes like these will help you test the game more efficiently and possibly spark ideas for giving players bonus powers. Just remember to restore the settings to normal when you're finished testing a feature.

Repopulating the Fleet

One key feature of *Alien Invasion* is that the aliens are relentless: every time the fleet is destroyed, a new fleet should appear.

To make a new fleet of aliens appear after a fleet has been destroyed, we first check whether the `aliens` group is empty. If it is, we make a call to `_create_fleet()`. We'll perform this check at the end of `_update_bullets()`, because that's where individual aliens are destroyed.

alien_invasion.py

```
    def _update_bullets(self):
        --snip--
❶       if not self.aliens:
            # Destroy existing bullets and create new fleet.
❷           self.bullets.empty()
            self._create_fleet()
```

At ❶, we check whether the `aliens` group is empty. An empty group evaluates to `False`, so this is a simple way to check whether the group is empty. If it is, we get rid of any existing bullets by using the `empty()` method, which removes all the remaining sprites from a group ❷. We also call `_create_fleet()`, which fills the screen with aliens again.

Now a new fleet appears as soon as you destroy the current fleet.

Speeding Up the Bullets

If you've tried firing at the aliens in the game's current state, you might find that the bullets aren't traveling at the best speed for gameplay. They might be a little slow on your system or way too fast. At this point, you can modify the settings to make the gameplay interesting and enjoyable on your system.

We modify the speed of the bullets by adjusting the value of `bullet_speed` in *settings.py*. On my system, I'll adjust the value of `bullet_speed` to 1.5, so the bullets travel a little faster:

settings.py

```
        # Bullet settings
        self.bullet_speed = 1.5
        self.bullet_width = 3
        --snip--
```

The best value for this setting depends on your system's speed, so find a value that works for you. You can adjust other settings as well.

Refactoring _update_bullets()

Let's refactor `_update_bullets()` so it's not doing so many different tasks. We'll move the code for dealing with bullet and alien collisions to a separate method:

alien_invasion.py

```
    def _update_bullets(self):
        --snip--
        # Get rid of bullets that have disappeared.
        for bullet in self.bullets.copy():
            if bullet.rect.bottom <= 0:
                 self.bullets.remove(bullet)

        self._check_bullet_alien_collisions()

    def _check_bullet_alien_collisions(self):
        """Respond to bullet-alien collisions."""
        # Remove any bullets and aliens that have collided.
```

```
        collisions = pygame.sprite.groupcollide(
                self.bullets, self.aliens, True, True)

        if not self.aliens:
            # Destroy existing bullets and create new fleet.
            self.bullets.empty()
            self._create_fleet()
```

We've created a new method, `_check_bullet_alien_collisions()`, to look for collisions between bullets and aliens, and to respond appropriately if the entire fleet has been destroyed. Doing so keeps `_update_bullets()` from growing too long and simplifies further development.

TRY IT YOURSELF

13-5. Sideways Shooter Part 2: We've come a long way since Exercise 12-6, Sideways Shooter. For this exercise, try to develop Sideways Shooter to the same point we've brought *Alien Invasion* to. Add a fleet of aliens, and make them move sideways toward the ship. Or, write code that places aliens at random positions along the right side of the screen and then sends them toward the ship. Also, write code that makes the aliens disappear when they're hit.

Ending the Game

What's the fun and challenge in a game if you can't lose? If the player doesn't shoot down the fleet quickly enough, we'll have the aliens destroy the ship when they make contact. At the same time, we'll limit the number of ships a player can use, and we'll destroy the ship when an alien reaches the bottom of the screen. The game will end when the player has used up all their ships.

Detecting Alien and Ship Collisions

We'll start by checking for collisions between aliens and the ship so we can respond appropriately when an alien hits it. We'll check for alien and ship collisions immediately after updating the position of each alien in `AlienInvasion`:

alien_invasion.py

```
    def _update_aliens(self):
        --snip--
        self.aliens.update()

        # Look for alien-ship collisions.
❶       if pygame.sprite.spritecollideany(self.ship, self.aliens):
❷           print("Ship hit!!!")
```

The spritecollideany() function takes two arguments: a sprite and a group. The function looks for any member of the group that has collided with the sprite and stops looping through the group as soon as it finds one member that has collided with the sprite. Here, it loops through the group aliens and returns the first alien it finds that has collided with ship.

If no collisions occur, spritecollideany() returns None and the if block at ❶ won't execute. If it finds an alien that has collided with the ship, it returns that alien and the if block executes: it prints *Ship hit!!!* ❷. When an alien hits the ship, we'll need to do a number of tasks: we'll need to delete all remaining aliens and bullets, recenter the ship, and create a new fleet. Before we write code to do all this, we need to know that our approach for detecting alien and ship collisions works correctly. Writing a print() call is a simple way to ensure we're detecting these collisions properly.

Now when you run *Alien Invasion,* the message *Ship hit!!!* should appear in the terminal whenever an alien runs into the ship. When you're testing this feature, set alien_drop_speed to a higher value, such as 50 or 100, so the aliens reach your ship faster.

Responding to Alien and Ship Collisions

Now we need to figure out exactly what will happen when an alien collides with the ship. Instead of destroying the ship instance and creating a new one, we'll count how many times the ship has been hit by tracking statistics for the game. Tracking statistics will also be useful for scoring.

Let's write a new class, GameStats, to track game statistics, and save it as *game_stats.py*:

game_stats.py

```
class GameStats:
    """Track statistics for Alien Invasion."""

    def __init__(self, ai_game):
        """Initialize statistics."""
        self.settings = ai_game.settings
        self.reset_stats()

    def reset_stats(self):
        """Initialize statistics that can change during the game."""
        self.ships_left = self.settings.ship_limit
```

We'll make one GameStats instance for the entire time *Alien Invasion* is running. But we'll need to reset some statistics each time the player starts a new game. To do this, we'll initialize most of the statistics in the reset_stats() method instead of directly in __init__(). We'll call this method from __init__() so the statistics are set properly when the GameStats instance is first created ❶. But we'll also be able to call reset_stats() any time the player starts a new game.

Right now we have only one statistic, `ships_left`, the value of which will change throughout the game. The number of ships the player starts with should be stored in *settings.py* as `ship_limit`:

settings.py

```
        # Ship settings
        self.ship_speed = 1.5
        self.ship_limit = 3
```

We also need to make a few changes in *alien_invasion.py* to create an instance of `GameStats`. First, we'll update the `import` statements at the top of the file:

alien_invasion.py

```
import sys
from time import sleep

import pygame

from settings import Settings
from game_stats import GameStats
from ship import Ship
--snip--
```

We import the `sleep()` function from the `time` module in the Python standard library so we can pause the game for a moment when the ship is hit. We also import `GameStats`.

We'll create an instance of `GameStats` in `__init__()`:

alien_invasion.py

```
    def __init__(self):
        --snip--
        self.screen = pygame.display.set_mode(
            (self.settings.screen_width, self.settings.screen_height))
        pygame.display.set_caption("Alien Invasion")

        # Create an instance to store game statistics.
        self.stats = GameStats(self)

        self.ship = Ship(self)
        --snip--
```

We make the instance after creating the game window but before defining other game elements, such as the ship.

When an alien hits the ship, we'll subtract one from the number of ships left, destroy all existing aliens and bullets, create a new fleet, and reposition the ship in the middle of the screen. We'll also pause the game for a moment so the player can notice the collision and regroup before a new fleet appears.

Let's put most of this code in a new method called `_ship_hit()`. We'll call this method from `_update_aliens()` when an alien hits the ship:

alien_invasion.py

```
    def _ship_hit(self):
        """Respond to the ship being hit by an alien."""
```

```
        # Decrement ships_left.
❶       self.stats.ships_left -= 1

        # Get rid of any remaining aliens and bullets.
❷       self.aliens.empty()
        self.bullets.empty()

        # Create a new fleet and center the ship.
❸       self._create_fleet()
        self.ship.center_ship()

        # Pause.
❹       sleep(0.5)
```

The new method `_ship_hit()` coordinates the response when an alien hits a ship. Inside `_ship_hit()`, the number of ships left is reduced by 1 at ❶, after which we empty the groups `aliens` and `bullets` ❷.

Next, we create a new fleet and center the ship ❸. (We'll add the method `center_ship()` to `Ship` in a moment.) Then we add a pause after the updates have been made to all the game elements but before any changes have been drawn to the screen, so the player can see that their ship has been hit ❹. The `sleep()` call pauses program execution for half a second, long enough for the player to see that the alien has hit the ship. When the `sleep()` function ends, code execution moves on to the `_update_screen()` method, which draws the new fleet to the screen.

In `_update_aliens()`, we replace the `print()` call with a call to `_ship_hit()` when an alien hits the ship:

alien_invasion.py

```
def _update_aliens(self):
    --snip--
    if pygame.sprite.spritecollideany(self.ship, self.aliens):
        self._ship_hit()
```

Here's the new method `center_ship()`; add it to the end of *ship.py*:

ship.py

```
def center_ship(self):
    """Center the ship on the screen."""
    self.rect.midbottom = self.screen_rect.midbottom
    self.x = float(self.rect.x)
```

We center the ship the same way we did in `__init__()`. After centering it, we reset the `self.x` attribute, which allows us to track the ship's exact position.

NOTE *Notice that we never make more than one ship; we make only one ship instance for the whole game and recenter it whenever the ship has been hit. The statistic* `ships_left` *will tell us when the player has run out of ships.*

Run the game, shoot a few aliens, and let an alien hit the ship. The game should pause, and a new fleet should appear with the ship centered at the bottom of the screen again.

Aliens that Reach the Bottom of the Screen

If an alien reaches the bottom of the screen, we'll have the game respond the same way it does when an alien hits the ship. To check when this happens, add a new method in *alien_invasion.py*:

alien_invasion.py

```
    def _check_aliens_bottom(self):
        """Check if any aliens have reached the bottom of the screen."""
        screen_rect = self.screen.get_rect()
        for alien in self.aliens.sprites():
❶           if alien.rect.bottom >= screen_rect.bottom:
                # Treat this the same as if the ship got hit.
                self._ship_hit()
                break
```

The method `_check_aliens_bottom()` checks whether any aliens have reached the bottom of the screen. An alien reaches the bottom when its `rect.bottom` value is greater than or equal to the screen's `rect.bottom` attribute ❶. If an alien reaches the bottom, we call `_ship_hit()`. If one alien hits the bottom, there's no need to check the rest, so we break out of the loop after calling `_ship_hit()`.

We'll call this method from `_update_aliens()`:

alien_invasion.py

```
    def _update_aliens(self):
        --snip--
        # Look for alien-ship collisions.
        if pygame.sprite.spritecollideany(self.ship, self.aliens):
            self._ship_hit()

        # Look for aliens hitting the bottom of the screen.
        self._check_aliens_bottom()
```

We call `_check_aliens_bottom()` after updating the positions of all the aliens and after looking for alien and ship collisions ❷. Now a new fleet will appear every time the ship is hit by an alien or an alien reaches the bottom of the screen.

Game Over!

Alien Invasion feels more complete now, but the game never ends. The value of `ships_left` just grows increasingly negative. Let's add a `game_active` flag as an attribute to `GameStats` to end the game when the player runs out of ships. We'll set this flag at the end of the `__init__()` method in `GameStats`:

game_stats.py

```
    def __init__(self, ai_game):
        --snip--
        # Start Alien Invasion in an active state.
        self.game_active = True
```

Now we add code to `_ship_hit()` that sets `game_active` to `False` when the player has used up all their ships:

alien_invasion.py

```
    def _ship_hit(self):
        """Respond to ship being hit by alien."""
        if self.stats.ships_left > 0:
            # Decrement ships_left.
            self.stats.ships_left -= 1
           --snip--
            # Pause.
            sleep(0.5)
        else:
            self.stats.game_active = False
```

Most of `_ship_hit()` is unchanged. We've moved all the existing code into an `if` block, which tests to make sure the player has at least one ship remaining. If so, we create a new fleet, pause, and move on. If the player has no ships left, we set `game_active` to `False`.

Identifying When Parts of the Game Should Run

We need to identify the parts of the game that should always run and the parts that should run only when the game is active:

alien_invasion.py

```
    def run_game(self):
        """Start the main loop for the game."""
        while True:
            self._check_events()

            if self.stats.game_active:
                self.ship.update()
                self._update_bullets()
                self._update_aliens()

            self._update_screen()
```

In the main loop, we always need to call `_check_events()`, even if the game is inactive. For example, we still need to know if the user presses Q to quit the game or clicks the button to close the window. We also continue updating the screen so we can make changes to the screen while waiting to see whether the player chooses to start a new game. The rest of the function calls only need to happen when the game is active, because when the game is inactive, we don't need to update the positions of game elements.

Now when you play *Alien Invasion*, the game should freeze when you've used up all your ships.

TRY IT YOURSELF

13-6. Game Over: In Sideways Shooter, keep track of the number of times the ship is hit and the number of times an alien is hit by the ship. Decide on an appropriate condition for ending the game, and stop the game when this situation occurs.

Summary

In this chapter, you learned how to add a large number of identical elements to a game by creating a fleet of aliens. You used nested loops to create a grid of elements, and you made a large set of game elements move by calling each element's `update()` method. You learned to control the direction of objects on the screen and to respond to specific situations, such as when the fleet reaches the edge of the screen. You detected and responded to collisions when bullets hit aliens and aliens hit the ship. You also learned how to track statistics in a game and use a `game_active` flag to determine when the game is over.

In the next and final chapter of this project, we'll add a Play button so the player can choose when to start their first game and whether to play again when the game ends. We'll speed up the game each time the player shoots down the entire fleet, and we'll add a scoring system. The final result will be a fully playable game!

14

SCORING

In this chapter, we'll finish the *Alien Invasion* game. We'll add a Play button to start a game on demand or to restart a game once it ends. We'll also change the game so it speeds up when the player moves up a level, and implement a scoring system. By the end of the chapter, you'll know enough to start writing games that increase in difficulty as a player progresses and show scores.

Adding the Play Button

In this section, we'll add a Play button that appears before a game begins and reappears when the game ends so the player can play again.

Right now the game begins as soon as you run *alien_invasion.py*. Let's start the game in an inactive state and then prompt the player to click a Play button to begin. To do this, modify the `__init__()` method of `GameStats`:

game_stats.py

```
    def __init__(self, ai_game):
        """Initialize statistics."""
        self.settings = ai_game.settings
        self.reset_stats()

        # Start game in an inactive state.
        self.game_active = False
```

Now the game should start in an inactive state with no way for the player to start it until we make a Play button.

Creating a Button Class

Because Pygame doesn't have a built-in method for making buttons, we'll write a `Button` class to create a filled rectangle with a label. You can use this code to make any button in a game. Here's the first part of the `Button` class; save it as *button.py*:

button.py

```
import pygame.font

class Button:

❶   def __init__(self, ai_game, msg):
        """Initialize button attributes."""
        self.screen = ai_game.screen
        self.screen_rect = self.screen.get_rect()

        # Set the dimensions and properties of the button.
❷       self.width, self.height = 200, 50
        self.button_color = (0, 255, 0)
        self.text_color = (255, 255, 255)
❸       self.font = pygame.font.SysFont(None, 48)

        # Build the button's rect object and center it.
❹       self.rect = pygame.Rect(0, 0, self.width, self.height)
        self.rect.center = self.screen_rect.center

        # The button message needs to be prepped only once.
❺       self._prep_msg(msg)
```

First, we import the `pygame.font` module, which lets Pygame render text to the screen. The `__init__()` method takes the parameters `self`, the `ai_game`

object, and msg, which contains the button's text ❶. We set the button dimensions at ❷, and then set button_color to color the button's rect object bright green and set text_color to render the text in white.

At ❸, we prepare a font attribute for rendering text. The None argument tells Pygame to use the default font, and 48 specifies the size of the text. To center the button on the screen, we create a rect for the button ❹ and set its center attribute to match that of the screen.

Pygame works with text by rendering the string you want to display as an image. At ❺, we call _prep_msg() to handle this rendering.

Here's the code for _prep_msg():

button.py

```
    def _prep_msg(self, msg):
        """Turn msg into a rendered image and center text on the button."""
❶       self.msg_image = self.font.render(msg, True, self.text_color,
                self.button_color)
❷       self.msg_image_rect = self.msg_image.get_rect()
        self.msg_image_rect.center = self.rect.center
```

The _prep_msg() method needs a self parameter and the text to be rendered as an image (msg). The call to font.render() turns the text stored in msg into an image, which we then store in self.msg_image ❶. The font.render() method also takes a Boolean value to turn antialiasing on or off (antialiasing makes the edges of the text smoother). The remaining arguments are the specified font color and background color. We set antialiasing to True and set the text background to the same color as the button. (If you don't include a background color, Pygame will try to render the font with a transparent background.)

At ❷, we center the text image on the button by creating a rect from the image and setting its center attribute to match that of the button.

Finally, we create a draw_button() method that we can call to display the button onscreen:

button.py

```
    def draw_button(self):
        # Draw blank button and then draw message.
        self.screen.fill(self.button_color, self.rect)
        self.screen.blit(self.msg_image, self.msg_image_rect)
```

We call screen.fill() to draw the rectangular portion of the button. Then we call screen.blit() to draw the text image to the screen, passing it an image and the rect object associated with the image. This completes the Button class.

Drawing the Button to the Screen

We'll use the Button class to create a Play button in AlienInvasion. First, we'll update the import statements:

alien_invasion.py

```
--snip--
from game_stats import GameStats
from button import Button
```

Because we need only one Play button, we'll create the button in the `__init__()` method of `AlienInvasion`. We can place this code at the very end of `__init__()`:

alien_invasion.py

```
    def __init__(self):
        --snip--
        self._create_fleet()

        # Make the Play button.
        self.play_button = Button(self, "Play")
```

This code creates an instance of `Button` with the label *Play*, but it doesn't draw the button to the screen. We'll call the button's `draw_button()` method in `_update_screen()`:

alien_invasion.py

```
    def _update_screen(self):
        --snip--
        self.aliens.draw(self.screen)

        # Draw the play button if the game is inactive.
        if not self.stats.game_active:
            self.play_button.draw_button()

        pygame.display.flip()
```

To make the Play button visible above all other elements on the screen, we draw it after all the other elements have been drawn but before flipping to a new screen. We include it in an `if` block, so the button only appears when the game is inactive.

Now when you run *Alien Invasion*, you should see a Play button in the center of the screen, as shown in Figure 14-1.

Figure 14-1: A Play button appears when the game is inactive.

Starting the Game

To start a new game when the player clicks Play, add the following `elif` block to the end of `_check_events()` to monitor mouse events over the button:

alien_invasion.py

```
    def _check_events(self):
        """Respond to keypresses and mouse events."""
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                --snip--
❶           elif event.type == pygame.MOUSEBUTTONDOWN:
❷               mouse_pos = pygame.mouse.get_pos()
❸               self._check_play_button(mouse_pos)
```

Pygame detects a `MOUSEBUTTONDOWN` event when the player clicks anywhere on the screen ❶, but we want to restrict our game to respond to mouse clicks only on the Play button. To accomplish this, we use `pygame.mouse.get_pos()`, which returns a tuple containing the mouse cursor's x- and y-coordinates when the mouse button is clicked ❷. We send these values to the new method `_check_play_button()` ❸.

Here's `_check_play_button()`, which I chose to place after `_check_events()`:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
❶       if self.play_button.rect.collidepoint(mouse_pos):
            self.stats.game_active = True
```

We use the `rect` method `collidepoint()` to check whether the point of the mouse click overlaps the region defined by the Play button's `rect` ❶. If so, we set `game_active` to `True`, and the game begins!

At this point, you should be able to start and play a full game. When the game ends, the value of `game_active` should become `False` and the Play button should reappear.

Resetting the Game

The Play button code we just wrote works the first time the player clicks Play. But it doesn't work after the first game ends, because the conditions that caused the game to end haven't been reset.

To reset the game each time the player clicks Play, we need to reset the game statistics, clear out the old aliens and bullets, build a new fleet, and center the ship, as shown here:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
        if self.play_button.rect.collidepoint(mouse_pos):
            # Reset the game statistics.
❶           self.stats.reset_stats()
            self.stats.game_active = True

            # Get rid of any remaining aliens and bullets.
❷           self.aliens.empty()
```

```
            self.bullets.empty()

            # Create a new fleet and center the ship.
❸           self._create_fleet()
            self.ship.center_ship()
```

At ❶, we reset the game statistics, which gives the player three new ships. Then we set `game_active` to `True` so the game will begin as soon as the code in this function finishes running. We empty the `aliens` and `bullets` groups ❷, and then create a new fleet and center the ship ❸.

Now the game will reset properly each time you click Play, allowing you to play it as many times as you want!

Deactivating the Play Button

One issue with our Play button is that the button region on the screen will continue to respond to clicks even when the Play button isn't visible. If you click the Play button area by accident after a game begins, the game will restart!

To fix this, set the game to start only when `game_active` is `False`:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
❶       button_clicked = self.play_button.rect.collidepoint(mouse_pos)
❷       if button_clicked and not self.stats.game_active:
            # Reset the game statistics.
            self.stats.reset_stats()
            --snip--
```

The flag `button_clicked` stores a `True` or `False` value ❶, and the game will restart only if Play is clicked *and* the game is not currently active ❷. To test this behavior, start a new game and repeatedly click where the Play button should be. If everything works as expected, clicking the Play button area should have no effect on the gameplay.

Hiding the Mouse Cursor

We want the mouse cursor to be visible to begin play, but once play begins, it just gets in the way. To fix this, we'll make it invisible when the game becomes active. We can do this at the end of the `if` block in `_check_play_button()`:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
        button_clicked = self.play_button.rect.collidepoint(mouse_pos)
        if button_clicked and not self.stats.game_active:
            --snip--
            # Hide the mouse cursor.
            pygame.mouse.set_visible(False)
```

Passing `False` to `set_visible()` tells Pygame to hide the cursor when the mouse is over the game window.

We'll make the cursor reappear once the game ends so the player can click Play again to begin a new game. Here's the code to do that:

alien_invasion.py

```
    def _ship_hit(self):
        """Respond to ship being hit by alien."""
        if self.stats.ships_left > 0:
            --snip--
        else:
            self.stats.game_active = False
            pygame.mouse.set_visible(True)
```

We make the cursor visible again as soon as the game becomes inactive, which happens in `_ship_hit()`. Attention to details like this makes your game more professional looking and allows the player to focus on playing rather than figuring out the user interface.

TRY IT YOURSELF

14-1. Press P to Play: Because *Alien Invasion* uses keyboard input to control the ship, it would be useful to start the game with a keypress. Add code that lets the player press P to start. It might help to move some code from `_check_play_button()` to a `_start_game()` method that can be called from `_check_play_button()` *and* `_check_keydown_events()`.

14-2. Target Practice: Create a rectangle at the right edge of the screen that moves up and down at a steady rate. Then have a ship appear on the left side of the screen that the player can move up and down while firing bullets at the moving, rectangular target. Add a Play button that starts the game, and when the player misses the target three times, end the game and make the Play button reappear. Let the player restart the game with this Play button.

Leveling Up

In our current game, once a player shoots down the entire alien fleet, the player reaches a new level, but the game difficulty doesn't change. Let's liven things up a bit and make the game more challenging by increasing the game's speed each time a player clears the screen.

Modifying the Speed Settings

We'll first reorganize the `Settings` class to group the game settings into static and changing ones. We'll also make sure that settings that change

during the game reset when we start a new game. Here's the `__init__()` method for *settings.py*:

settings.py

```
    def __init__(self):
        """Initialize the game's static settings."""
        # Screen settings
        self.screen_width = 1200
        self.screen_height = 800
        self.bg_color = (230, 230, 230)

        # Ship settings
        self.ship_limit = 3

        # Bullet settings
        self.bullet_width = 3
        self.bullet_height = 15
        self.bullet_color = 60, 60, 60
        self.bullets_allowed = 3

        # Alien settings
        self.fleet_drop_speed = 10

        # How quickly the game speeds up
❶       self.speedup_scale = 1.1

❷       self.initialize_dynamic_settings()
```

We continue to initialize those settings that stay constant in the `__init__()` method. At ❶, we add a `speedup_scale` setting to control how quickly the game speeds up: a value of 2 will double the game speed every time the player reaches a new level; a value of 1 will keep the speed constant. A value like 1.1 should increase the speed enough to make the game challenging but not impossible. Finally, we call the `initialize_dynamic_settings()` method to initialize the values for attributes that need to change throughout the game ❷.

Here's the code for `initialize_dynamic_settings()`:

settings.py

```
    def initialize_dynamic_settings(self):
        """Initialize settings that change throughout the game."""
        self.ship_speed = 1.5
        self.bullet_speed = 3.0
        self.alien_speed = 1.0

        # fleet_direction of 1 represents right; -1 represents left.
        self.fleet_direction = 1
```

This method sets the initial values for the ship, bullet, and alien speeds. We'll increase these speeds as the player progresses in the game and reset them each time the player starts a new game. We include `fleet_direction` in this method so the aliens always move right at the beginning of a new game. We don't need to increase the value of `fleet_drop_speed`, because when the aliens move faster across the screen, they'll also come down the screen faster.

To increase the speeds of the ship, bullets, and aliens each time the player reaches a new level, we'll write a new method called `increase_speed()`:

settings.py

```
    def increase_speed(self):
        """Increase speed settings."""
        self.ship_speed *= self.speedup_scale
        self.bullet_speed *= self.speedup_scale
        self.alien_speed *= self.speedup_scale
```

To increase the speed of these game elements, we multiply each speed setting by the value of `speedup_scale`.

We increase the game's tempo by calling `increase_speed()` in `_check_bullet_alien_collisions()` when the last alien in a fleet has been shot down:

alien_invasion.py

```
    def _check_bullet_alien_collisions(self):
        --snip--
        if not self.aliens:
            # Destroy existing bullets and create new fleet.
            self.bullets.empty()
            self._create_fleet()
            self.settings.increase_speed()
```

Changing the values of the speed settings `ship_speed`, `alien_speed`, and `bullet_speed` is enough to speed up the entire game!

Resetting the Speed

Now we need to return any changed settings to their initial values each time the player starts a new game; otherwise, each new game would start with the increased speed settings of the previous game:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        """Start a new game when the player clicks Play."""
        button_clicked = self.play_button.rect.collidepoint(mouse_pos)
        if button_clicked and not self.stats.game_active:
            # Reset the game settings.
            self.settings.initialize_dynamic_settings()
            --snip--
```

Playing *Alien Invasion* should be more fun and challenging now. Each time you clear the screen, the game should speed up and become slightly more difficult. If the game becomes too difficult too quickly, decrease the value of `settings.speedup_scale`. Or if the game isn't challenging enough, increase the value slightly. Find a sweet spot by ramping up the difficulty in a reasonable amount of time. The first couple of screens should be easy, the next few challenging but doable, and subsequent screens almost impossibly difficult.

TRY IT YOURSELF

14-3. Challenging Target Practice: Start with your work from Exercise 14-2 (page 285). Make the target move faster as the game progresses, and restart the target at the original speed when the player clicks Play.

14-4. Difficulty Levels: Make a set of buttons for *Alien Invasion* that allows the player to select an appropriate starting difficulty level for the game. Each button should assign the appropriate values for the attributes in `Settings` needed to create different difficulty levels.

Scoring

Let's implement a scoring system to track the game's score in real time and display the high score, level, and number of ships remaining.

The score is a game statistic, so we'll add a `score` attribute to `GameStats`:

game_stats.py

```
class GameStats:
    --snip--
    def reset_stats(self):
        """Initialize statistics that can change during the game."""
        self.ships_left = self.ai_settings.ship_limit
        self.score = 0
```

To reset the score each time a new game starts, we initialize `score` in `reset_stats()` rather than `__init__()`.

Displaying the Score

To display the score on the screen, we first create a new class, `Scoreboard`. For now, this class will just display the current score, but eventually we'll use it to report the high score, level, and number of ships remaining as well. Here's the first part of the class; save it as *scoreboard.py*:

scoreboard.py

```
import pygame.font

class Scoreboard:
    """A class to report scoring information."""

❶   def __init__(self, ai_game):
        """Initialize scorekeeping attributes."""
        self.screen = ai_game.screen
        self.screen_rect = self.screen.get_rect()
        self.settings = ai_game.settings
        self.stats = ai_game.stats

        # Font settings for scoring information.
❷       self.text_color = (30, 30, 30)
❸       self.font = pygame.font.SysFont(None, 48)
```

```
        # Prepare the initial score image.
❹       self.prep_score()
```

Because Scoreboard writes text to the screen, we begin by importing the pygame.font module. Next, we give __init__() the ai_game parameter so it can access the settings, screen, and stats objects, which it will need to report the values we're tracking ❶. Then we set a text color ❷ and instantiate a font object ❸.

To turn the text to be displayed into an image, we call prep_score() ❹, which we define here:

scoreboard.py

```
    def prep_score(self):
        """Turn the score into a rendered image."""
❶       score_str = str(self.stats.score)
❷       self.score_image = self.font.render(score_str, True,
                self.text_color, self.settings.bg_color)

        # Display the score at the top right of the screen.
❸       self.score_rect = self.score_image.get_rect()
❹       self.score_rect.right = self.screen_rect.right - 20
❺       self.score_rect.top = 20
```

In prep_score(), we turn the numerical value stats.score into a string ❶, and then pass this string to render(), which creates the image ❷. To display the score clearly onscreen, we pass the screen's background color and the text color to render().

We'll position the score in the upper-right corner of the screen and have it expand to the left as the score increases and the width of the number grows. To make sure the score always lines up with the right side of the screen, we create a rect called score_rect ❸ and set its right edge 20 pixels from the right edge of the screen ❹. We then place the top edge 20 pixels down from the top of the screen ❺.

Then we create a show_score() method to display the rendered score image:

scoreboard.py

```
    def show_score(self):
        """Draw score to the screen."""
        self.screen.blit(self.score_image, self.score_rect)
```

This method draws the score image onscreen at the location score_rect specifies.

Making a Scoreboard

To display the score, we'll create a Scoreboard instance in AlienInvasion. First, let's update the import statements:

alien_invasion.py

```
--snip--
from game_stats import GameStats
from scoreboard import Scoreboard
--snip--
```

Next, we make an instance of `Scoreboard` in `__init__()`:

alien_invasion.py

```
def __init__(self):
    --snip--
    pygame.display.set_caption("Alien Invasion")

    # Create an instance to store game statistics,
    #   and create a scoreboard.
    self.stats = GameStats(self)
    self.sb = Scoreboard(self)
    --snip--
```

Then we draw the scoreboard onscreen in `_update_screen()`:

alien_invasion.py

```
def _update_screen(self):
    --snip--
    self.aliens.draw(self.screen)

    # Draw the score information.
    self.sb.show_score()

    # Draw the play button if the game is inactive.
    --snip--
```

We call `show_score()` just before we draw the Play button.

When you run *Alien Invasion* now, a 0 should appear at the top right of the screen. (At this point, we just want to make sure the score appears in the right place before developing the scoring system further.) Figure 14-2 shows the score as it appears before the game starts.

Figure 14-2: The score appears at the top-right corner of the screen.

Next, we'll assign point values to each alien!

Updating the Score as Aliens Are Shot Down

To write a live score onscreen, we update the value of `stats.score` whenever an alien is hit, and then call `prep_score()` to update the score image. But first, let's determine how many points a player gets each time they shoot down an alien:

settings.py

```
    def initialize_dynamic_settings(self):
        --snip--

        # Scoring
        self.alien_points = 50
```

We'll increase each alien's point value as the game progresses. To make sure this point value is reset each time a new game starts, we set the value in `initialize_dynamic_settings()`.

Let's update the score each time an alien is shot down in `_check_bullet_alien_collisions()`:

alien_invasion.py

```
    def _check_bullet_alien_collisions(self):
        """Respond to bullet-alien collisions."""
        # Remove any bullets and aliens that have collided.
        collisions = pygame.sprite.groupcollide(
                self.bullets, self.aliens, True, True)

        if collisions:
            self.stats.score += self.settings.alien_points
            self.sb.prep_score()
        --snip--
```

When a bullet hits an alien, Pygame returns a `collisions` dictionary. We check whether the dictionary exists, and if it does, the alien's value is added to the score. We then call `prep_score()` to create a new image for the updated score.

Now when you play *Alien Invasion*, you should be able to rack up points!

Resetting the Score

Right now, we're only prepping a new score *after* an alien has been hit, which works for most of the game. But we still see the old score when a new game starts until the first alien is hit in the new game.

We can fix this by prepping the score when starting a new game:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        --snip--
        if button_clicked and not self.stats.game_active:
            --snip--
            # Reset the game statistics.
            self.stats.reset_stats()
```

```
        self.stats.game_active = True
        self.sb.prep_score()
        --snip--
```

We call `prep_score()` after resetting the game stats when starting a new game. This preps the scoreboard with a 0 score.

Making Sure to Score All Hits

As currently written, our code could miss scoring for some aliens. For example, if two bullets collide with aliens during the same pass through the loop or if we make an extra-wide bullet to hit multiple aliens, the player will only receive points for hitting one of the aliens. To fix this, let's refine the way that bullet and alien collisions are detected.

In `_check_bullet_alien_collisions()`, any bullet that collides with an alien becomes a key in the `collisions` dictionary. The value associated with each bullet is a list of aliens it has collided with. We loop through the values in the `collisions` dictionary to make sure we award points for each alien hit:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    --snip--
    if collisions:
❶       for aliens in collisions.values():
            self.stats.score += self.settings.alien_points * len(aliens)
        self.sb.prep_score()
    --snip--
```

If the `collisions` dictionary has been defined, we loop through all values in the dictionary. Remember that each value is a list of aliens hit by a single bullet. We multiply the value of each alien by the number of aliens in each list and add this amount to the current score. To test this, change the width of a bullet to 300 pixels and verify that you receive points for each alien you hit with your extra-wide bullets; then return the bullet width to its normal value.

Increasing Point Values

Because the game gets more difficult each time a player reaches a new level, aliens in later levels should be worth more points. To implement this functionality, we'll add code to increase the point value when the game's speed increases:

settings.py

```
class Settings:
    """A class to store all settings for Alien Invasion."""

    def __init__(self):
        --snip--
        # How quickly the game speeds up
        self.speedup_scale = 1.1
```

```
        # How quickly the alien point values increase
❶       self.score_scale = 1.5

        self.initialize_dynamic_settings()

    def initialize_dynamic_settings(self):
        --snip--

    def increase_speed(self):
        """Increase speed settings and alien point values."""
        self.ship_speed *= self.speedup_scale
        self.bullet_speed *= self.speedup_scale
        self.alien_speed *= self.speedup_scale

❷       self.alien_points = int(self.alien_points * self.score_scale)
```

We define a rate at which points increase, which we call `score_scale` ❶. A small increase in speed (1.1) makes the game more challenging quickly. But to see a more notable difference in scoring, we need to change the alien point value by a larger amount (1.5). Now when we increase the game's speed, we also increase the point value of each hit ❷. We use the `int()` function to increase the point value by whole integers.

To see the value of each alien, add a `print()` call to the `increase_speed()` method in `Settings`:

settings.py

```
    def increase_speed(self):
        --snip--
        self.alien_points = int(self.alien_points * self.score_scale)
        print(self.alien_points)
```

The new point value should appear in the terminal every time you reach a new level.

NOTE *Be sure to remove the `print()` call after verifying that the point value is increasing, or it might affect your game's performance and distract the player.*

Rounding the Score

Most arcade-style shooting games report scores as multiples of 10, so let's follow that lead with our scores. Also, let's format the score to include comma separators in large numbers. We'll make this change in `Scoreboard`:

scoreboard.py

```
    def prep_score(self):
        """Turn the score into a rendered image."""
❶       rounded_score = round(self.stats.score, -1)
❷       score_str = "{:,}".format(rounded_score)
        self.score_image = self.font.render(score_str, True,
                self.text_color, self.settings.bg_color)
        --snip--
```

The `round()` function normally rounds a decimal number to a set number of decimal places given as the second argument. However, when you pass a negative number as the second argument, `round()` will round the value to the nearest 10, 100, 1000, and so on. The code at ❶ tells Python to round the value of `stats.score` to the nearest 10 and store it in `rounded_score`.

At ❷, a string formatting directive tells Python to insert commas into numbers when converting a numerical value to a string: for example, to output `1,000,000` instead of `1000000`. Now when you run the game, you should see a neatly formatted, rounded score even when you rack up lots of points, as shown in Figure 14-3.

Figure 14-3: A rounded score with comma separators

High Scores

Every player wants to beat a game's high score, so let's track and report high scores to give players something to work toward. We'll store high scores in `GameStats`:

game_stats.py

```
    def __init__(self, ai_game):
        --snip--
        # High score should never be reset.
        self.high_score = 0
```

Because the high score should never be reset, we initialize `high_score` in `__init__()` rather than in `reset_stats()`.

Next, we'll modify Scoreboard to display the high score. Let's start with the __init__() method:

scoreboard.py

```
    def __init__(self, ai_game):
        --snip--
        # Prepare the initial score images.
        self.prep_score()
❶       self.prep_high_score()
```

The high score will be displayed separately from the score, so we need a new method, prep_high_score(), to prepare the high score image ❶.

Here's the prep_high_score() method:

scoreboard.py

```
    def prep_high_score(self):
        """Turn the high score into a rendered image."""
❶       high_score = round(self.stats.high_score, -1)
        high_score_str = "{:,}".format(high_score)
❷       self.high_score_image = self.font.render(high_score_str, True,
                self.text_color, self.settings.bg_color)

        # Center the high score at the top of the screen.
        self.high_score_rect = self.high_score_image.get_rect()
❸       self.high_score_rect.centerx = self.screen_rect.centerx
❹       self.high_score_rect.top = self.score_rect.top
```

We round the high score to the nearest 10 and format it with commas ❶. We then generate an image from the high score ❷, center the high score rect horizontally ❸, and set its top attribute to match the top of the score image ❹.

The show_score() method now draws the current score at the top right and the high score at the top center of the screen:

scoreboard.py

```
    def show_score(self):
        """Draw score to the screen."""
        self.screen.blit(self.score_image, self.score_rect)
        self.screen.blit(self.high_score_image, self.high_score_rect)
```

To check for high scores, we'll write a new method, check_high_score(), in Scoreboard:

scoreboard.py

```
    def check_high_score(self):
        """Check to see if there's a new high score."""
        if self.stats.score > self.stats.high_score:
            self.stats.high_score = self.stats.score
            self.prep_high_score()
```

The method check_high_score() checks the current score against the high score. If the current score is greater, we update the value of high_score and call prep_high_score() to update the high score's image.

We need to call `check_high_score()` each time an alien is hit after updating the score in `_check_bullet_alien_collisions()`:

alien_invasion.py

```
def _check_bullet_alien_collisions(self):
    --snip--
    if collisions:
        for aliens in collisions.values():
            self.stats.score += self.settings.alien_points * len(aliens)
        self.sb.prep_score()
        self.sb.check_high_score()
    --snip--
```

We call `check_high_score()` when the `collisions` dictionary is present, and we do so after updating the score for all the aliens that have been hit.

The first time you play *Alien Invasion*, your score will be the high score, so it will be displayed as the current score and the high score. But when you start a second game, your high score should appear in the middle and your current score at the right, as shown in Figure 14-4.

Figure 14-4: The high score is shown at the top center of the screen.

Displaying the Level

To display the player's level in the game, we first need an attribute in `GameStats` representing the current level. To reset the level at the start of each new game, initialize it in `reset_stats()`:

game_stats.py

```
def reset_stats(self):
    """Initialize statistics that can change during the game."""
    self.ships_left = self.settings.ship_limit
```

```
        self.score = 0
        self.level = 1
```

To have Scoreboard display the current level, we call a new method, prep_level(), from __init__():

scoreboard.py

```
    def __init__(self, ai_game):
        --snip--
        self.prep_high_score()
        self.prep_level()
```

Here's prep_level():

scoreboard.py

```
    def prep_level(self):
        """Turn the level into a rendered image."""
        level_str = str(self.stats.level)
❶       self.level_image = self.font.render(level_str, True,
                self.text_color, self.settings.bg_color)

        # Position the level below the score.
        self.level_rect = self.level_image.get_rect()
❷       self.level_rect.right = self.score_rect.right
❸       self.level_rect.top = self.score_rect.bottom + 10
```

The prep_level() method creates an image from the value stored in stats.level ❶ and sets the image's right attribute to match the score's right attribute ❷. It then sets the top attribute 10 pixels beneath the bottom of the score image to leave space between the score and the level ❸.

We also need to update show_score():

scoreboard.py

```
    def show_score(self):
        """Draw scores and level to the screen."""
        self.screen.blit(self.score_image, self.score_rect)
        self.screen.blit(self.high_score_image, self.high_score_rect)
        self.screen.blit(self.level_image, self.level_rect)
```

This new line draws the level image to the screen.

We'll increment stats.level and update the level image in _check_bullet_alien_collisions():

alien_invasion.py

```
    def _check_bullet_alien_collisions(self):
        --snip--
        if not self.aliens:
            # Destroy existing bullets and create new fleet.
            self.bullets.empty()
            self._create_fleet()
            self.settings.increase_speed()

            # Increase level.
            self.stats.level += 1
            self.sb.prep_level()
```

If a fleet is destroyed, we increment the value of `stats.level` and call `prep_level()` to make sure the new level displays correctly.

To ensure the level image updates properly at the start of a new game, we also call `prep_level()` when the player clicks the Play button:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        --snip--
        if button_clicked and not self.stats.game_active:
            --snip--
            self.sb.prep_score()
            self.sb.prep_level()
            --snip--
```

We call `prep_level()` right after calling `prep_score()`.

Now you'll see how many levels you've completed, as shown in Figure 14-5.

Figure 14-5: The current level appears just below the current score.

NOTE *In some classic games, the scores have labels, such as Score, High Score, and Level. We've omitted these labels because the meaning of each number becomes clear once you've played the game. To include these labels, add them to the score strings just before the calls to `font.render()` in `Scoreboard`.*

Displaying the Number of Ships

Finally, let's display the number of ships the player has left, but this time, let's use a graphic. To do so, we'll draw ships in the upper-left corner of

the screen to represent how many ships are left, just as many classic arcade games do.

First, we need to make `Ship` inherit from `Sprite` so we can create a group of ships:

ship.py

```
import pygame
from pygame.sprite import Sprite

❶ class Ship(Sprite):
    """A class to manage the ship."""

    def __init__(self, ai_game):
        """Initialize the ship and set its starting position."""
❷       super().__init__()
        --snip--
```

Here we import `Sprite`, make sure `Ship` inherits from `Sprite` ❶, and call `super()` at the beginning of `__init__()` ❷.

Next, we need to modify `Scoreboard` to create a group of ships we can display. Here are the `import` statements for `Scoreboard`:

scoreboard.py

```
import pygame.font
from pygame.sprite import Group

from ship import Ship
```

Because we're making a group of ships, we import the `Group` and `Ship` classes.

Here's `__init__()`:

scoreboard.py

```
    def __init__(self, ai_game):
        """Initialize scorekeeping attributes."""
        self.ai_game = ai_game
        self.screen = ai_game.screen
        --snip--
        self.prep_level()
        self.prep_ships()
```

We assign the game instance to an attribute, because we'll need it to create some ships. We call `prep_ships()` after the call to `prep_level()`.

Here's `prep_ships()`:

scoreboard.py

```
    def prep_ships(self):
        """Show how many ships are left."""
❶       self.ships = Group()
❷       for ship_number in range(self.stats.ships_left):
            ship = Ship(self.ai_game)
❸           ship.rect.x = 10 + ship_number * ship.rect.width
❹           ship.rect.y = 10
❺           self.ships.add(ship)
```

The prep_ships() method creates an empty group, self.ships, to hold the ship instances ❶. To fill this group, a loop runs once for every ship the player has left ❷. Inside the loop, we create a new ship and set each ship's x-coordinate value so the ships appear next to each other with a 10-pixel margin on the left side of the group of ships ❸. We set the y-coordinate value 10 pixels down from the top of the screen so the ships appear in the upper-left corner of the screen ❹. Then we add each new ship to the group ships ❺.

Now we need to draw the ships to the screen:

scoreboard.py

```
    def show_score(self):
        """Draw scores, level, and ships to the screen."""
        self.screen.blit(self.score_image, self.score_rect)
        self.screen.blit(self.high_score_image, self.high_score_rect)
        self.screen.blit(self.level_image, self.level_rect)
        self.ships.draw(self.screen)
```

To display the ships on the screen, we call draw() on the group, and Pygame draws each ship.

To show the player how many ships they have to start with, we call prep_ships() when a new game starts. We do this in _check_play_button() in AlienInvasion:

alien_invasion.py

```
    def _check_play_button(self, mouse_pos):
        --snip--
        if button_clicked and not self.stats.game_active:
            --snip--
            self.sb.prep_score()
            self.sb.prep_level()
            self.sb.prep_ships()
            --snip--
```

We also call prep_ships() when a ship is hit to update the display of ship images when the player loses a ship:

alien_invasion.py

```
    def _ship_hit(self):
        """Respond to ship being hit by alien."""
        if self.stats.ships_left > 0:
            # Decrement ships_left, and update scoreboard.
            self.stats.ships_left -= 1
            self.sb.prep_ships()
            --snip--
```

We call prep_ships() after decreasing the value of ships_left, so the correct number of ships displays each time a ship is destroyed.

Figure 14-6 shows the complete scoring system with the remaining ships displayed at the top left of the screen.

Figure 14-6: The complete scoring system for Alien Invasion

TRY IT YOURSELF

14-5. All-Time High Score: The high score is reset every time a player closes and restarts *Alien Invasion*. Fix this by writing the high score to a file before calling `sys.exit()` and reading in the high score when initializing its value in `GameStats`.

14-6. Refactoring: Look for methods that are doing more than one task, and refactor them to organize your code and make it efficient. For example, move some of the code in `_check_bullet_alien_collisions()`, which starts a new level when the fleet of aliens has been destroyed, to a function called `start_new_level()`. Also, move the four separate method calls in the `__init__()` method in `Scoreboard` to a method called `prep_images()` to shorten `__init__()`. The `prep_images()` method could also help simplify `_check_play_button()` or `start_game()` if you've already refactored `_check_play_button()`.

NOTE *Before attempting to refactor the project, see Appendix D to learn how to restore the project to a working state if you introduce bugs while refactoring.*

(continued)

14-7. Expanding the Game: Think of a way to expand *Alien Invasion*. For example, you could program the aliens to shoot bullets down at the ship or add shields for your ship to hide behind, which can be destroyed by bullets from either side. Or use something like the `pygame.mixer` module to add sound effects, such as explosions and shooting sounds.

14-8. Sideways Shooter, Final Version: Continue developing Sideways Shooter, using everything we've done in this project. Add a Play button, make the game speed up at appropriate points, and develop a scoring system. Be sure to refactor your code as you work, and look for opportunities to customize the game beyond what was shown in this chapter.

Summary

In this chapter, you learned how to implement a Play button to start a new game, detect mouse events, and hide the cursor in active games. You can use what you've learned to create other buttons in your games, like a Help button to display instructions on how to play. You also learned how to modify the speed of a game as it progresses, implement a progressive scoring system, and display information in textual and nontextual ways.

PROJECT 2

DATA VISUALIZATION

15

GENERATING DATA

Data visualization involves exploring data through visual representations. It's closely associated with *data analysis*, which uses code to explore the patterns and connections in a data set. A data set can be made up of a small list of numbers that fits in one line of code or it can be many gigabytes of data.

Making beautiful data representations is about more than pretty pictures. When a representation of a data set is simple and visually appealing, its meaning becomes clear to viewers. People will see patterns and significance in your data sets that they never knew existed.

Fortunately, you don't need a supercomputer to visualize complex data. With Python's efficiency, you can quickly explore data sets made of millions of individual data points on just a laptop. Also, the data points don't have to be numbers. With the basics you learned in the first part of this book, you can analyze nonnumerical data as well.

People use Python for data-intensive work in genetics, climate research, political and economic analysis, and much more. Data scientists have written

an impressive array of visualization and analysis tools in Python, many of which are available to you as well. One of the most popular tools is Matplotlib, a mathematical plotting library. We'll use Matplotlib to make simple plots, such as line graphs and scatter plots. Then we'll create a more interesting data set based on the concept of a random walk—a visualization generated from a series of random decisions.

We'll also use a package called Plotly, which creates visualizations that work well on digital devices. Plotly generates visualizations that automatically resize to fit a variety of display devices. These visualizations can also include a number of interactive features, such as emphasizing particular aspects of the data set when users hover over different parts of the visualization. We'll use Plotly to analyze the results of rolling dice.

Installing Matplotlib

To use Matplotlib for your initial set of visualizations, you'll need to install it using `pip`, a module that downloads and installs Python packages. Enter the following command at a terminal prompt:

```
$ python -m pip install --user matplotlib
```

This command tells Python to run the `pip` module and install the `matplotlib` package to the current user's Python installation. If you use a command other than `python` on your system to run programs or start a terminal session, such as `python3`, your command will look like this:

```
$ python3 -m pip install --user matplotlib
```

NOTE *If this command doesn't work on macOS, try running the command again without the `--user` flag.*

To see the kinds of visualizations you can make with Matplotlib, visit the sample gallery at *https://matplotlib.org/gallery/*. When you click a visualization in the gallery, you'll see the code used to generate the plot.

Plotting a Simple Line Graph

Let's plot a simple line graph using Matplotlib, and then customize it to create a more informative data visualization. We'll use the square number sequence 1, 4, 9, 16, 25 as the data for the graph.

Just provide Matplotlib with the numbers, as shown here, and Matplotlib should do the rest:

mpl_squares.py

```
import matplotlib.pyplot as plt

squares = [1, 4, 9, 16, 25]
```

```
❶ fig, ax = plt.subplots()
  ax.plot(squares)

  plt.show()
```

We first import the `pyplot` module using the alias `plt` so we don't have to type `pyplot` repeatedly. (You'll see this convention often in online examples, so we'll do the same here.) The `pyplot` module contains a number of functions that generate charts and plots.

We create a list called `squares` to hold the data that we'll plot. Then we follow another common Matplotlib convention by calling the `subplots()` function ❶. This function can generate one or more plots in the same figure. The variable `fig` represents the entire figure or collection of plots that are generated. The variable `ax` represents a single plot in the figure and is the variable we'll use most of the time.

We then use the `plot()` method, which will try to plot the data it's given in a meaningful way. The function `plt.show()` opens Matplotlib's viewer and displays the plot, as shown in Figure 15-1. The viewer allows you to zoom and navigate the plot, and when you click the disk icon, you can save any plot images you like.

Figure 15-1: One of the simplest plots you can make in Matplotlib

Changing the Label Type and Line Thickness

Although the plot in Figure 15-1 shows that the numbers are increasing, the label type is too small and the line is a little thin to read easily. Fortunately, Matplotlib allows you to adjust every feature of a visualization.

We'll use a few of the available customizations to improve this plot's readability, as shown here:

mpl_squares.py

```
import matplotlib.pyplot as plt

squares = [1, 4, 9, 16, 25]

fig, ax = plt.subplots()
❶ ax.plot(squares, linewidth=3)

# Set chart title and label axes.
❷ ax.set_title("Square Numbers", fontsize=24)
❸ ax.set_xlabel("Value", fontsize=14)
ax.set_ylabel("Square of Value", fontsize=14)

# Set size of tick labels.
❹ ax.tick_params(axis='both', labelsize=14)

plt.show()
```

The `linewidth` parameter at ❶ controls the thickness of the line that `plot()` generates. The `set_title()` method at ❷ sets a title for the chart. The `fontsize` parameters, which appear repeatedly throughout the code, control the size of the text in various elements on the chart.

The `set_xlabel()` and `set_ylabel()` methods allow you to set a title for each of the axes ❸, and the method `tick_params()` styles the tick marks ❹. The arguments shown here affect the tick marks on both the x- and y-axes (`axis='both'`) and set the font size of the tick mark labels to 14 (`labelsize=14`).

As you can see in Figure 15-2, the resulting chart is much easier to read. The label type is bigger, and the line graph is thicker. It's often worth experimenting with these values to get an idea of what will look best in the resulting graph.

Figure 15-2: The chart is much easier to read now.

Correcting the Plot

But now that we can read the chart better, we see that the data is not plotted correctly. Notice at the end of the graph that the square of 4.0 is shown as 25! Let's fix that.

When you give `plot()` a sequence of numbers, it assumes the first data point corresponds to an x-coordinate value of 0, but our first point corresponds to an x-value of 1. We can override the default behavior by giving `plot()` the input and output values used to calculate the squares:

mpl_squares.py

```
import matplotlib.pyplot as plt

input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]

fig, ax = plt.subplots()
ax.plot(input_values, squares, linewidth=3)

# Set chart title and label axes.
--snip--
```

Now `plot()` will graph the data correctly because we've provided the input and output values, so it doesn't have to assume how the output numbers were generated. The resulting plot, shown in Figure 15-3, is correct.

Figure 15-3: The data is now plotted correctly.

You can specify numerous arguments when using `plot()` and use a number of functions to customize your plots. We'll continue to explore these customization functions as we work with more interesting data sets throughout this chapter.

Using Built-in Styles

Matplotlib has a number of predefined styles available, with good starting settings for background colors, gridlines, line widths, fonts, font sizes, and more that will make your visualizations appealing without requiring much customization. To see the styles available on your system, run the following lines in a terminal session:

```
>>> import matplotlib.pyplot as plt
>>> plt.style.available
['seaborn-dark', 'seaborn-darkgrid', 'seaborn-ticks', 'fivethirtyeight',
--snip--
```

To use any of these styles, add one line of code before starting to generate the plot:

mpl_squares.py

```
import matplotlib.pyplot as plt

input_values = [1, 2, 3, 4, 5]
squares = [1, 4, 9, 16, 25]

plt.style.use('seaborn')
fig, ax = plt.subplots()
--snip--
```

This code generates the plot shown in Figure 15-4. A wide variety of styles is available; play around with these styles to find some that you like.

Figure 15-4: The built-in seaborn style

Plotting and Styling Individual Points with scatter()

Sometimes, it's useful to plot and style individual points based on certain characteristics. For example, you might plot small values in one color and larger values in a different color. You could also plot a large data set with

one set of styling options and then emphasize individual points by replotting them with different options.

To plot a single point, use the `scatter()` method. Pass the single (x, y) values of the point of interest to `scatter()` to plot those values:

scatter_squares.py

```
import matplotlib.pyplot as plt

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(2, 4)

plt.show()
```

Let's style the output to make it more interesting. We'll add a title, label the axes, and make sure all the text is large enough to read:

```
import matplotlib.pyplot as plt

plt.style.use('seaborn')
fig, ax = plt.subplots()
❶ ax.scatter(2, 4, s=200)

# Set chart title and label axes.
ax.set_title("Square Numbers", fontsize=24)
ax.set_xlabel("Value", fontsize=14)
ax.set_ylabel("Square of Value", fontsize=14)

# Set size of tick labels.
ax.tick_params(axis='both', which='major', labelsize=14)

plt.show()
```

At ❶ we call `scatter()` and use the `s` argument to set the size of the dots used to draw the graph. When you run *scatter_squares.py* now, you should see a single point in the middle of the chart, as shown in Figure 15-5.

Figure 15-5: Plotting a single point

Plotting a Series of Points with scatter()

To plot a series of points, we can pass `scatter()` separate lists of x- and y-values, like this:

scatter_squares.py

```
import matplotlib.pyplot as plt

x_values = [1, 2, 3, 4, 5]
y_values = [1, 4, 9, 16, 25]

plt.style.use('seaborn')
fig, ax = plt.subplots()
ax.scatter(x_values, y_values, s=100)

# Set chart title and label axes.
--snip--
```

The `x_values` list contains the numbers to be squared, and `y_values` contains the square of each number. When these lists are passed to `scatter()`, Matplotlib reads one value from each list as it plots each point. The points to be plotted are (1, 1), (2, 4), (3, 9), (4, 16), and (5, 25); Figure 15-6 shows the result.

Figure 15-6: A scatter plot with multiple points

Calculating Data Automatically

Writing lists by hand can be inefficient, especially when we have many points. Rather than passing our points in a list, let's use a loop in Python to do the calculations for us.

Here's how this would look with 1000 points:

scatter_squares.py

```
import matplotlib.pyplot as plt

❶ x_values = range(1, 1001)
  y_values = [x**2 for x in x_values]

  plt.style.use('seaborn')
  fig, ax = plt.subplots()
❷ ax.scatter(x_values, y_values, s=10)

  # Set chart title and label axes.
  --snip--

  # Set the range for each axis.
❸ ax.axis([0, 1100, 0, 1100000])

  plt.show()
```

We start with a range of x-values containing the numbers 1 through 1000 ❶. Next, a list comprehension generates the y-values by looping through the x-values (`for x in x_values`), squaring each number (`x**2`) and storing the results in `y_values`. We then pass the input and output lists to `scatter()` ❷. Because this is a large data set, we use a smaller point size.

At ❸ we use the `axis()` method to specify the range of each axis. The `axis()` method requires four values: the minimum and maximum values for the x-axis and the y-axis. Here, we run the x-axis from 0 to 1100 and the y-axis from 0 to 1,100,000. Figure 15-7 shows the result.

Figure 15-7: Python can plot 1000 points as easily as it plots 5 points.

Defining Custom Colors

To change the color of the points, pass `c` to `scatter()` with the name of a color to use in quotation marks, as shown here:

```
ax.scatter(x_values, y_values, c='red', s=10)
```

You can also define custom colors using the RGB color model. To define a color, pass the `c` argument a tuple with three decimal values (one each for red, green, and blue in that order), using values between 0 and 1. For example, the following line creates a plot with light-green dots:

```
ax.scatter(x_values, y_values, c=(0, 0.8, 0), s=10)
```

Values closer to 0 produce dark colors, and values closer to 1 produce lighter colors.

Using a Colormap

A *colormap* is a series of colors in a gradient that moves from a starting to an ending color. You use colormaps in visualizations to emphasize a pattern in the data. For example, you might make low values a light color and high values a darker color.

The `pyplot` module includes a set of built-in colormaps. To use one of these colormaps, you need to specify how `pyplot` should assign a color to each point in the data set. Here's how to assign each point a color based on its y-value:

scatter_squares.py

```
import matplotlib.pyplot as plt

x_values = range(1, 1001)
y_values = [x**2 for x in x_values]

ax.scatter(x_values, y_values, c=y_values, cmap=plt.cm.Blues, s=10)

# Set chart title and label axes.
--snip--
```

We pass the list of y-values to `c`, and then tell `pyplot` which colormap to use using the `cmap` argument. This code colors the points with lower y-values light blue and colors the points with higher y-values dark blue. Figure 15-8 shows the resulting plot.

NOTE *You can see all the colormaps available in* `pyplot` *at* https://matplotlib.org/*; go to* ***Examples****, scroll down to* ***Color****, and click* ***Colormap reference****.*

Figure 15-8: A plot using the `Blues` colormap

Saving Your Plots Automatically

If you want your program to automatically save the plot to a file, you can replace the call to `plt.show()` with a call to `plt.savefig()`:

```
plt.savefig('squares_plot.png', bbox_inches='tight')
```

The first argument is a filename for the plot image, which will be saved in the same directory as *scatter_squares.py*. The second argument trims extra whitespace from the plot. If you want the extra whitespace around the plot, just omit this argument.

TRY IT YOURSELF

15-1. Cubes: A number raised to the third power is a *cube*. Plot the first five cubic numbers, and then plot the first 5000 cubic numbers.

15-2. Colored Cubes: Apply a colormap to your cubes plot.

Random Walks

In this section, we'll use Python to generate data for a random walk, and then use Matplotlib to create a visually appealing representation of that data. A *random walk* is a path that has no clear direction but is determined by a series of random decisions, each of which is left entirely to chance. You might imagine a random walk as the path a confused ant would take if it took every step in a random direction.

Random walks have practical applications in nature, physics, biology, chemistry, and economics. For example, a pollen grain floating on a drop of water moves across the surface of the water because it's constantly pushed around by water molecules. Molecular motion in a water drop is random, so the path a pollen grain traces on the surface is a random walk. The code we'll write next models many real-world situations.

Creating the RandomWalk() Class

To create a random walk, we'll create a `RandomWalk` class, which will make random decisions about which direction the walk should take. The class needs three attributes: one variable to store the number of points in the walk and two lists to store the x- and y-coordinate values of each point in the walk.

We'll only need two methods for the `RandomWalk` class: the `__init__()` method and `fill_walk()`, which will calculate the points in the walk. Let's start with `__init__()` as shown here:

random_walk.py

```
❶ from random import choice

  class RandomWalk:
      """A class to generate random walks."""

❷     def __init__(self, num_points=5000):
          """Initialize attributes of a walk."""
          self.num_points = num_points

          # All walks start at (0, 0).
❸         self.x_values = [0]
          self.y_values = [0]
```

To make random decisions, we'll store possible moves in a list and use the `choice()` function, from the `random` module, to decide which move to make each time a step is taken ❶. We then set the default number of points in a walk to 5000, which is large enough to generate some interesting patterns but small enough to generate walks quickly ❷. Then at ❸ we make two lists to hold the x- and y-values, and we start each walk at the point (0, 0).

Choosing Directions

We'll use the `fill_walk()` method, as shown here, to fill our walk with points and determine the direction of each step. Add this method to *random_walk.py*:

random_walk.py

```
      def fill_walk(self):
          """Calculate all the points in the walk."""

          # Keep taking steps until the walk reaches the desired length.
❶         while len(self.x_values) < self.num_points:

              # Decide which direction to go and how far to go in that direction.
❷             x_direction = choice([1, -1])
```

```
            x_distance = choice([0, 1, 2, 3, 4])
❸           x_step = x_direction * x_distance

            y_direction = choice([1, -1])
            y_distance = choice([0, 1, 2, 3, 4])
❹           y_step = y_direction * y_distance

            # Reject moves that go nowhere.
❺           if x_step == 0 and y_step == 0:
                continue

            # Calculate the new position.
❻           x = self.x_values[-1] + x_step
            y = self.y_values[-1] + y_step

            self.x_values.append(x)
            self.y_values.append(y)
```

At ❶ we set up a loop that runs until the walk is filled with the correct number of points. The main part of the `fill_walk()` method tells Python how to simulate four random decisions: will the walk go right or left? How far will it go in that direction? Will it go up or down? How far will it go in that direction?

We use `choice([1, -1])` to choose a value for `x_direction`, which returns either 1 for right movement or –1 for left ❷. Next, `choice([0, 1, 2, 3, 4])` tells Python how far to move in that direction (`x_distance`) by randomly selecting an integer between 0 and 4. (The inclusion of a 0 allows us to take steps along the y-axis as well as steps that have movement along both axes.)

At ❸ and ❹ we determine the length of each step in the *x* and *y* directions by multiplying the direction of movement by the distance chosen. A positive result for `x_step` means move right, a negative result means move left, and 0 means move vertically. A positive result for `y_step` means move up, negative means move down, and 0 means move horizontally. If the value of both `x_step` and `y_step` are 0, the walk doesn't go anywhere, so we continue the loop to ignore this move ❺.

To get the next x-value for the walk, we add the value in `x_step` to the last value stored in `x_values` ❻ and do the same for the y-values. When we have these values, we append them to `x_values` and `y_values`.

Plotting the Random Walk

Here's the code to plot all the points in the walk:

rw_visual.py

```
import matplotlib.pyplot as plt

from random_walk import RandomWalk

# Make a random walk.
❶ rw = RandomWalk()
rw.fill_walk()
```

```
# Plot the points in the walk.
plt.style.use('classic')
fig, ax = plt.subplots()
❷ ax.scatter(rw.x_values, rw.y_values, s=15)
plt.show()
```

We begin by importing `pyplot` and `RandomWalk`. We then create a random walk and store it in `rw` ❶, making sure to call `fill_walk()`. At ❷ we feed the walk's x- and y-values to `scatter()` and choose an appropriate dot size. Figure 15-9 shows the resulting plot with 5000 points. (The images in this section omit Matplotlib's viewer, but you'll continue to see it when you run *rw_visual.py*.)

Figure 15-9: A random walk with 5000 points

Generating Multiple Random Walks

Every random walk is different, and it's fun to explore the various patterns that can be generated. One way to use the preceding code to make multiple walks without having to run the program several times is to wrap it in a `while` loop, like this:

rw_visual.py

```
import matplotlib.pyplot as plt

from random_walk import RandomWalk

# Keep making new walks, as long as the program is active.
while True:
    # Make a random walk.
    rw = RandomWalk()
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
```

```
    fig, ax = plt.subplots()
    ax.scatter(rw.x_values, rw.y_values, s=15)
    plt.show()

    keep_running = input("Make another walk? (y/n): ")
    if keep_running == 'n':
        break
```

This code generates a random walk, displays it in Matplotlib's viewer, and pauses with the viewer open. When you close the viewer, you'll be asked whether you want to generate another walk. Press **y** to generate walks that stay near the starting point, that wander off mostly in one direction, that have thin sections connecting larger groups of points, and so on. When you want to end the program, press **n**.

Styling the Walk

In this section, we'll customize our plots to emphasize the important characteristics of each walk and deemphasize distracting elements. To do so, we identify the characteristics we want to emphasize, such as where the walk began, where it ended, and the path taken. Next, we identify the characteristics to deemphasize, such as tick marks and labels. The result should be a simple visual representation that clearly communicates the path taken in each random walk.

Coloring the Points

We'll use a colormap to show the order of the points in the walk, and then remove the black outline from each dot so the color of the dots will be clearer. To color the points according to their position in the walk, we pass the `c` argument a list containing the position of each point. Because the points are plotted in order, the list just contains the numbers from 0 to 4999, as shown here:

rw_visual.py

```
--snip--
while True:
    # Make a random walk.
    rw = RandomWalk()
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
    fig, ax = plt.subplots()
❶   point_numbers = range(rw.num_points)
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
        edgecolors='none', s=15)
    plt.show()

    keep_running = input("Make another walk? (y/n): ")
    --snip--
```

At ❶ we use `range()` to generate a list of numbers equal to the number of points in the walk. Then we store them in the list `point_numbers`, which we'll use to set the color of each point in the walk. We pass `point_numbers` to the `c` argument, use the `Blues` colormap, and then pass `edgecolors='none'` to get rid of the black outline around each point. The result is a plot of the walk that varies from light to dark blue along a gradient, as shown in Figure 15-10.

Figure 15-10: A random walk colored with the `Blues` colormap

Plotting the Starting and Ending Points

In addition to coloring points to show their position along the walk, it would be useful to see where each walk begins and ends. To do so, we can plot the first and last points individually after the main series has been plotted. We'll make the end points larger and color them differently to make them stand out, as shown here:

rw_visual.py

```
--snip--
while True:
    --snip--
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
        edgecolors='none', s=15)

    # Emphasize the first and last points.
    ax.scatter(0, 0, c='green', edgecolors='none', s=100)
    ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none',
        s=100)

    plt.show()
    --snip--
```

To show the starting point, we plot point (0, 0) in green in a larger size (`s=100`) than the rest of the points. To mark the end point, we plot the last x- and y-value in the walk in red with a size of 100. Make sure you insert this code just before the call to `plt.show()` so the starting and ending points are drawn on top of all the other points.

When you run this code, you should be able to spot exactly where each walk begins and ends. (If these end points don't stand out clearly, adjust their color and size until they do.)

Cleaning Up the Axes

Let's remove the axes in this plot so they don't distract from the path of each walk. To turn off the axes, use this code:

rw_visual.py

```
--snip--
while True:
    --snip--
    ax.scatter(rw.x_values[-1], rw.y_values[-1], c='red', edgecolors='none',
        s=100)

    # Remove the axes.
❶   ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    plt.show()
    --snip--
```

To modify the axes, we use the `ax.get_xaxis()` and `ax.get_yaxis()` methods ❶ to set the visibility of each axis to `False`. As you continue to work with visualizations, you'll frequently see this chaining of methods.

Run *rw_visual.py* now; you should see a series of plots with no axes.

Adding Plot Points

Let's increase the number of points to give us more data to work with. To do so, we increase the value of `num_points` when we make a `RandomWalk` instance and adjust the size of each dot when drawing the plot, as shown here:

rw_visual.py

```
--snip--
while True:
    # Make a random walk.
    rw = RandomWalk(50_000)
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
    fig, ax = plt.subplots()
    point_numbers = range(rw.num_points)
    ax.scatter(rw.x_values, rw.y_values, c=point_numbers, cmap=plt.cm.Blues,
        edgecolor='none', s=1)
    --snip--
```

This example creates a random walk with 50,000 points (to mirror real-world data) and plots each point at size `s=1`. The resulting walk is wispy and cloud-like, as shown in Figure 15-11. As you can see, we've created a piece of art from a simple scatter plot!

Experiment with this code to see how much you can increase the number of points in a walk before your system starts to slow down significantly or the plot loses its visual appeal.

Figure 15-11: A walk with 50,000 points

Altering the Size to Fill the Screen

A visualization is much more effective at communicating patterns in data if it fits nicely on the screen. To make the plotting window better fit your screen, adjust the size of Matplotlib's output, like this:

rw_visual.py

```
--snip--
while True:
    # Make a random walk.
    rw = RandomWalk(50_000)
    rw.fill_walk()

    # Plot the points in the walk.
    plt.style.use('classic')
    fig, ax = plt.subplots(figsize=(15, 9))
    --snip--
```

When creating the plot, you can pass a `figsize` argument to set the size of the figure. The `figsize` parameter takes a tuple, which tells Matplotlib the dimensions of the plotting window in inches.

Matplotlib assumes that your screen resolution is 100 pixels per inch; if this code doesn't give you an accurate plot size, adjust the numbers as

necessary. Or, if you know your system's resolution, pass `plt.subplots()` the resolution using the `dpi` parameter to set a plot size that makes effective use of the space available on your screen, as shown here:

```
fig, ax = plt.subplots(figsize=(10, 6), dpi=128)
```

TRY IT YOURSELF

15-3. Molecular Motion: Modify *rw_visual.py* by replacing `plt.scatter()` with `plt.plot()`. To simulate the path of a pollen grain on the surface of a drop of water, pass in the `rw.x_values` and `rw.y_values`, and include a `linewidth` argument. Use 5000 instead of 50,000 points.

15-4. Modified Random Walks: In the `RandomWalk` class, `x_step` and `y_step` are generated from the same set of conditions. The direction is chosen randomly from the list `[1, -1]` and the distance from the list `[0, 1, 2, 3, 4]`. Modify the values in these lists to see what happens to the overall shape of your walks. Try a longer list of choices for the distance, such as 0 through 8, or remove the –1 from the *x* or *y* direction list.

15-5. Refactoring: The `fill_walk()` method is lengthy. Create a new method called `get_step()` to determine the direction and distance for each step, and then calculate the step. You should end up with two calls to `get_step()` in `fill_walk()`:

```
x_step = self.get_step()
y_step = self.get_step()
```

This refactoring should reduce the size of `fill_walk()` and make the method easier to read and understand.

Rolling Dice with Plotly

In this section, we'll use the Python package Plotly to produce interactive visualizations. Plotly is particularly useful when you're creating visualizations that will be displayed in a browser, because the visualizations will scale automatically to fit the viewer's screen. Visualizations that Plotly generates are also interactive; when the user hovers over certain elements on the screen, information about that element is highlighted.

In this project, we'll analyze the results of rolling dice. When you roll one regular, six-sided die, you have an equal chance of rolling any of the numbers from 1 through 6. However, when you use two dice, you're more likely to roll certain numbers rather than others. We'll try to determine

which numbers are most likely to occur by generating a data set that represents rolling dice. Then we'll plot the results of a large number of rolls to determine which results are more likely than others.

The study of rolling dice is often used in mathematics to explain various types of data analysis. But it also has real-world applications in casinos and other gambling scenarios, as well as in the way games like Monopoly and many role-playing games are played.

Installing Plotly

Install Plotly using `pip`, just as you did for Matplotlib:

```
$ python -m pip install --user plotly
```

If you used **`python3`** or something else when installing Matplotlib, make sure you use the same command here.

To see what kind of visualizations are possible with Plotly, visit the gallery of chart types at *https://plot.ly/python/*. Each example includes source code, so you can see how Plotly generates the visualizations.

Creating the Die Class

We'll create the following `Die` class to simulate the roll of one die:

die.py

```
from random import randint

class Die:
    """A class representing a single die."""

❶   def __init__(self, num_sides=6):
        """Assume a six-sided die."""
        self.num_sides = num_sides

    def roll(self):
        """"Return a random value between 1 and number of sides."""
❷       return randint(1, self.num_sides)
```

The `__init__()` method takes one optional argument. With the `Die` class, when an instance of our die is created, the number of sides will always be six if no argument is included. If an argument *is* included, that value will set the number of sides on the die ❶. (Dice are named for their number of sides: a six-sided die is a D6, an eight-sided die is a D8, and so on.)

The `roll()` method uses the `randint()` function to return a random number between 1 and the number of sides ❷. This function can return the starting value (1), the ending value (`num_sides`), or any integer between the two.

Rolling the Die

Before creating a visualization based on the `Die` class, let's roll a D6, print the results, and check that the results look reasonable:

die_visual.py

```
from die import Die

# Create a D6.
❶ die = Die()

# Make some rolls, and store results in a list.
results = []
❷ for roll_num in range(100):
    result = die.roll()
    results.append(result)

print(results)
```

At ❶ we create an instance of `Die` with the default six sides. At ❷ we roll the die 100 times and store the results of each roll in the list `results`. Here's a sample set of results:

```
[4, 6, 5, 6, 1, 5, 6, 3, 5, 3, 5, 3, 2, 2, 1, 3, 1, 5, 3, 6, 3, 6, 5, 4,
 1, 1, 4, 2, 3, 6, 4, 2, 6, 4, 1, 3, 2, 5, 6, 3, 6, 2, 1, 1, 3, 4, 1, 4,
 3, 5, 1, 4, 5, 5, 2, 3, 3, 1, 2, 3, 5, 6, 2, 5, 6, 1, 3, 2, 1, 1, 1, 6,
 5, 5, 2, 2, 6, 4, 1, 4, 5, 1, 1, 1, 4, 5, 3, 3, 1, 3, 5, 4, 5, 6, 5, 4,
 1, 5, 1, 2]
```

A quick scan of these results shows that the `Die` class seems to be working. We see the values 1 and 6, so we know the smallest and largest possible values are being returned, and because we don't see 0 or 7, we know all the results are in the appropriate range. We also see each number from 1 through 6, which indicates that all possible outcomes are represented. Let's determine exactly how many times each number appears.

Analyzing the Results

We'll analyze the results of rolling one D6 by counting how many times we roll each number:

die_visual.py

```
--snip--
# Make some rolls, and store results in a list.
results = []
❶ for roll_num in range(1000):
    result = die.roll()
    results.append(result)

# Analyze the results.
frequencies = []
❷ for value in range(1, die.num_sides+1):
❸     frequency = results.count(value)
```

```
❹     frequencies.append(frequency)

print(frequencies)
```

Because we're no longer printing the results, we can increase the number of simulated rolls to 1000 ❶. To analyze the rolls, we create the empty list `frequencies` to store the number of times each value is rolled. We loop through the possible values (1 through 6 in this case) at ❷, count how many times each number appears in `results` ❸, and then append this value to the `frequencies` list ❹. We then print this list before making a visualization:

```
[155, 167, 168, 170, 159, 181]
```

These results look reasonable: we see six frequencies, one for each possible number when you roll a D6, and we see that no frequency is significantly higher than any other. Now let's visualize these results.

Making a Histogram

With a list of frequencies, we can make a *histogram* of the results. A histogram is a bar chart showing how often certain results occur. Here's the code to create the histogram:

die_visual.py

```
from plotly.graph_objs import Bar, Layout
from plotly import offline

from die import Die
--snip--

# Analyze the results.
frequencies = []
for value in range(1, die.num_sides+1):
    frequency = results.count(value)
    frequencies.append(frequency)

# Visualize the results.
❶ x_values = list(range(1, die.num_sides+1))
❷ data = [Bar(x=x_values, y=frequencies)]

❸ x_axis_config = {'title': 'Result'}
y_axis_config = {'title': 'Frequency of Result'}
❹ my_layout = Layout(title='Results of rolling one D6 1000 times',
        xaxis=x_axis_config, yaxis=y_axis_config)
❺ offline.plot({'data': data, 'layout': my_layout}, filename='d6.html')
```

To make a histogram, we need a bar for each of the possible results. We store these in a list called `x_values`, which starts at 1 and ends at the number of sides on the die ❶. Plotly doesn't accept the results of the `range()` function directly, so we need to convert the range to a list explicitly using

the `list()` function. The Plotly class `Bar()` represents a data set that will be formatted as a bar chart ❷. This class needs a list of x-values, and a list of y-values. The class must be wrapped in square brackets, because a data set can have multiple elements.

Each axis can be configured in a number of ways, and each configuration option is stored as an entry in a dictionary. At this point, we're just setting the title of each axis ❸. The `Layout()` class returns an object that specifies the layout and configuration of the graph as a whole ❹. Here we set the title of the graph and pass the x- and y-axis configuration dictionaries as well.

To generate the plot, we call the `offline.plot()` function ❺. This function needs a dictionary containing the data and layout objects, and it also accepts a name for the file where the graph will be saved. We store the output in a file called *d6.html*.

When you run the program *die_visual.py*, a browser will probably open showing the file *d6.html*. If this doesn't happen automatically, open a new tab in any web browser, and then open the file *d6.html* (in the folder where you saved *die_visual.py*). You should see a chart that looks like the one in Figure 15-12. (I've modified this chart slightly for printing; by default, Plotly generates charts with smaller text than what you see here.)

Figure 15-12: A simple bar chart created with Plotly

Notice that Plotly has made the chart interactive: hover your cursor over any bar in the chart, and you'll see the associated data. This feature is particularly useful when you're plotting multiple data sets on the same chart. Also notice the icons in the upper right, which allow you to pan and zoom the visualization, and save your visualization as an image.

Rolling Two Dice

Rolling two dice results in larger numbers and a different distribution of results. Let's modify our code to create two D6 dice to simulate the way we roll a pair of dice. Each time we roll the pair, we'll add the two numbers (one from each die) and store the sum in `results`. Save a copy of *die_visual.py* as *dice_visual.py*, and make the following changes:

dice_visual.py

```
from plotly.graph_objs import Bar, Layout
from plotly import offline

from die import Die

# Create two D6 dice.
die_1 = Die()
die_2 = Die()

# Make some rolls, and store results in a list.
results = []
for roll_num in range(1000):
❶    result = die_1.roll() + die_2.roll()
    results.append(result)

# Analyze the results.
frequencies = []
❷ max_result = die_1.num_sides + die_2.num_sides
❸ for value in range(2, max_result+1):
    frequency = results.count(value)
    frequencies.append(frequency)

# Visualize the results.
x_values = list(range(2, max_result+1))
data = [Bar(x=x_values, y=frequencies)]

❹ x_axis_config = {'title': 'Result', 'dtick': 1}
y_axis_config = {'title': 'Frequency of Result'}
my_layout = Layout(title='Results of rolling two D6 dice 1000 times',
        xaxis=x_axis_config, yaxis=y_axis_config)
offline.plot({'data': data, 'layout': my_layout}, filename='d6_d6.html')
```

After creating two instances of `Die`, we roll the dice and calculate the sum of the two dice for each roll ❶. The largest possible result (12) is the sum of the largest number on both dice, which we store in `max_result` ❷. The smallest possible result (2) is the sum of the smallest number on both dice. When we analyze the results, we count the number of results for each value between 2 and `max_result` ❸. (We could have used `range(2, 13)`, but this would work only for two D6 dice. When modeling real-world situations, it's best to write code that can easily model a variety of situations. This code allows us to simulate rolling a pair of dice with any number of sides.)

When creating the chart, we include the `dtick` key in the `x_axis_config` dictionary ❹. This setting controls the spacing between tick marks on the x-axis. Now that we have more bars on the histogram, Plotly's default

settings will only label some of the bars. The `'dtick': 1` setting tells Plotly to label every tick mark. We also update the title of the chart and change the output filename as well.

After running this code, you should see a chart that looks like the one in Figure 15-13.

Figure 15-13: Simulated results of rolling two six-sided dice 1000 times

This graph shows the approximate results you're likely to get when you roll a pair of D6 dice. As you can see, you're least likely to roll a 2 or a 12 and most likely to roll a 7. This happens because there are six ways to roll a 7, namely: 1 and 6, 2 and 5, 3 and 4, 4 and 3, 5 and 2, or 6 and 1.

Rolling Dice of Different Sizes

Let's create a six-sided die and a ten-sided die, and see what happens when we roll them 50,000 times:

dice_visual.py

```
from plotly.graph_objs import Bar, Layout
from plotly import offline

from die import Die

# Create a D6 and a D10.
die_1 = Die()
❶ die_2 = Die(10)

# Make some rolls, and store results in a list.
results = []
for roll_num in range(50_000):
    result = die_1.roll() + die_2.roll()
    results.append(result)
```

```
# Analyze the results.
--snip--

# Visualize the results.
x_values = list(range(2, max_result+1))
data = [Bar(x=x_values, y=frequencies)]

x_axis_config = {'title': 'Result', 'dtick': 1}
y_axis_config = {'title': 'Frequency of Result'}
❷ my_layout = Layout(title='Results of rolling a D6 and a D10 50000 times',
        xaxis=x_axis_config, yaxis=y_axis_config)
offline.plot({'data': data, 'layout': my_layout}, filename='d6_d10.html')
```

To make a D10, we pass the argument `10` when creating the second `Die` instance ❶ and change the first loop to simulate 50,000 rolls instead of 1000. We change the title of the graph and update the output filename as well ❷.

Figure 15-14 shows the resulting chart. Instead of one most likely result, there are five. This happens because there's still only one way to roll the smallest value (1 and 1) and the largest value (6 and 10), but the smaller die limits the number of ways you can generate the middle numbers: there are six ways to roll a 7, 8, 9, 10, and 11. Therefore, these are the most common results, and you're equally likely to roll any one of these numbers.

Figure 15-14: The results of rolling a six-sided die and a ten-sided die 50,000 times

Our ability to use Plotly to model the rolling of dice gives us considerable freedom in exploring this phenomenon. In just minutes you can simulate a tremendous number of rolls using a large variety of dice.

TRY IT YOURSELF

15-6. Two D8s: Create a simulation showing what happens when you roll two eight-sided dice 1000 times. Try to picture what you think the visualization will look like before you run the simulation; then see if your intuition was correct. Gradually increase the number of rolls until you start to see the limits of your system's capabilities.

15-7. Three Dice: When you roll three D6 dice, the smallest number you can roll is 3 and the largest number is 18. Create a visualization that shows what happens when you roll three D6 dice.

15-8. Multiplication: When you roll two dice, you usually add the two numbers together to get the result. Create a visualization that shows what happens if you multiply these numbers instead.

15-9. Die Comprehensions: For clarity, the listings in this section use the long form of `for` loops. If you're comfortable using list comprehensions, try writing a comprehension for one or both of the loops in each of these programs.

15-10. Practicing with Both Libraries: Try using Matplotlib to make a die-rolling visualization, and use Plotly to make the visualization for a random walk. (You'll need to consult the documentation for each library to complete this exercise.)

Summary

In this chapter, you learned to generate data sets and create visualizations of that data. You created simple plots with Matplotlib and used a scatter plot to explore random walks. You also created a histogram with Plotly and used a histogram to explore the results of rolling dice of different sizes.

Generating your own data sets with code is an interesting and powerful way to model and explore a wide variety of real-world situations. As you continue to work through the data visualization projects that follow, keep an eye out for situations you might be able to model with code. Look at the visualizations you see in news media, and see if you can identify those that were generated using methods similar to the ones you're learning in these projects.

In Chapter 16, you'll download data from online sources and continue to use Matplotlib and Plotly to explore that data.

16

DOWNLOADING DATA

In this chapter, you'll download data sets from online sources and create working visualizations of that data. You can find an incredible variety of data online, much of which hasn't been examined thoroughly. The ability to analyze this data allows you to discover patterns and connections that no one else has found.

We'll access and visualize data stored in two common data formats, CSV and JSON. We'll use Python's `csv` module to process weather data stored in the CSV (comma-separated values) format and analyze high and low temperatures over time in two different locations. We'll then use Matplotlib to generate a chart based on our downloaded data to display variations in temperature in two dissimilar environments: Sitka, Alaska, and Death Valley, California. Later in the chapter, we'll use the `json` module to access earthquake data stored in the JSON format and use Plotly to draw a world map showing the locations and magnitudes of recent earthquakes.

By the end of this chapter, you'll be prepared to work with different types and data set formats, and you'll have a deeper understanding of how to build complex visualizations. Being able to access and visualize online data of different types and formats is essential to working with a wide variety of real-world data sets.

The CSV File Format

One simple way to store data in a text file is to write the data as a series of values separated by commas, which is called *comma-separated values*. The resulting files are called *CSV* files. For example, here's a chunk of weather data in CSV format:

```
"USW00025333","SITKA AIRPORT, AK US","2018-01-01","0.45",,"48","38"
```

This is an excerpt of some weather data from January 1, 2018 in Sitka, Alaska. It includes the day's high and low temperatures, as well as a number of other measurements from that day. CSV files can be tricky for humans to read, but they're easy for programs to process and extract values from, which speeds up the data analysis process.

We'll begin with a small set of CSV-formatted weather data recorded in Sitka, which is available in the book's resources at *https://nostarch.com/pythoncrashcourse2e/*. Make a folder called *data* inside the folder where you're saving this chapter's programs. Copy the file *sitka_weather_07-2018_simple.csv* into this new folder. (After you download the book's resources, you'll have all the files you need for this project.)

NOTE *The weather data in this project was originally downloaded from* https://ncdc.noaa.gov/cdo-web/.

Parsing the CSV File Headers

Python's `csv` module in the standard library parses the lines in a CSV file and allows us to quickly extract the values we're interested in. Let's start by examining the first line of the file, which contains a series of headers for the data. These headers tell us what kind of information the data holds:

sitka_highs.py

```
import csv

filename = 'data/sitka_weather_07-2018_simple.csv'
❶ with open(filename) as f:
❷     reader = csv.reader(f)
❸     header_row = next(reader)
      print(header_row)
```

After importing the `csv` module, we assign the name of the file we're working with to `filename`. We then open the file and assign the resulting file

object to `f` ❶. Next, we call `csv.reader()` and pass it the file object as an argument to create a reader object associated with that file ❷. We assign the reader object to `reader`.

The `csv` module contains a `next()` function, which returns the next line in the file when passed the reader object. In the preceding listing, we call `next()` only once so we get the first line of the file, which contains the file headers ❸. We store the data that's returned in `header_row`. As you can see, `header_row` contains meaningful, weather-related headers that tell us what information each line of data holds:

```
['STATION', 'NAME', 'DATE', 'PRCP', 'TAVG', 'TMAX', 'TMIN']
```

The `reader` object processes the first line of comma-separated values in the file and stores each as an item in a list. The header `STATION` represents the code for the weather station that recorded this data. The position of this header tells us that the first value in each line will be the weather station code. The `NAME` header indicates that the second value in each line is the name of the weather station that made the recording. The rest of the headers specify what kinds of information were recorded in each reading. The data we're most interested in for now are the date, the high temperature (`TMAX`), and the low temperature (`TMIN`). This is a simple data set that contains only precipitation and temperature-related data. When you download your own weather data, you can choose to include a number of other measurements relating to wind speed, direction, and more detailed precipitation data.

Printing the Headers and Their Positions

To make it easier to understand the file header data, we print each header and its position in the list:

sitka_highs.py

```
--snip--
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

❶   for index, column_header in enumerate(header_row):
        print(index, column_header)
```

The `enumerate()` function returns both the index of each item and the value of each item as you loop through a list ❶. (Note that we've removed the line `print(header_row)` in favor of this more detailed version.)

Here's the output showing the index of each header:

```
0 STATION
1 NAME
2 DATE
3 PRCP
4 TAVG
5 TMAX
6 TMIN
```

Here we see that the dates and their high temperatures are stored in columns 2 and 5. To explore this data, we'll process each row of data in *sitka_weather_07-2018_simple.csv* and extract the values with the indexes 2 and 5.

Extracting and Reading Data

Now that we know which columns of data we need, let's read in some of that data. First, we'll read in the high temperature for each day:

sitka_highs.py

```
--snip--
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

    # Get high temperatures from this file.
❶   highs = []
❷   for row in reader:
❸       high = int(row[5])
        highs.append(high)

print(highs)
```

We make an empty list called `highs` ❶ and then loop through the remaining rows in the file ❷. The `reader` object continues from where it left off in the CSV file and automatically returns each line following its current position. Because we've already read the header row, the loop will begin at the second line where the actual data begins. On each pass through the loop, we pull the data from index 5, which corresponds to the header `TMAX`, and assign it to the variable `high` ❸. We use the `int()` function to convert the data, which is stored as a string, to a numerical format so we can use it. We then append this value to `highs`.

The following listing shows the data now stored in `highs`:

```
[62, 58, 70, 70, 67, 59, 58, 62, 66, 59, 56, 63, 65, 58, 56, 59, 64, 60, 60,
 61, 65, 65, 63, 59, 64, 65, 68, 66, 64, 67, 65]
```

We've extracted the high temperature for each date and stored each value in a list. Now let's create a visualization of this data.

Plotting Data in a Temperature Chart

To visualize the temperature data we have, we'll first create a simple plot of the daily highs using Matplotlib, as shown here:

sitka_highs.py

```
import csv

import matplotlib.pyplot as plt

filename = 'data/sitka_weather_07-2018_simple.csv'
with open(filename) as f:
    --snip--
```

```
# Plot the high temperatures.
plt.style.use('seaborn')
fig, ax = plt.subplots()
❶ ax.plot(highs, c='red')

# Format plot.
❷ plt.title("Daily high temperatures, July 2018", fontsize=24)
❸ plt.xlabel('', fontsize=16)
plt.ylabel("Temperature (F)", fontsize=16)
plt.tick_params(axis='both', which='major', labelsize=16)

plt.show()
```

We pass the list of highs to `plot()` and pass `c='red'` to plot the points in red ❶. (We'll plot the highs in red and the lows in blue.) We then specify a few other formatting details, such as the title, font size, and labels ❷, which you should recognize from Chapter 15. Because we have yet to add the dates, we won't label the x-axis, but `plt.xlabel()` does modify the font size to make the default labels more readable ❸. Figure 16-1 shows the resulting plot: a simple line graph of the high temperatures for July 2018 in Sitka, Alaska.

Figure 16-1: A line graph showing daily high temperatures for July 2018 in Sitka, Alaska

The datetime Module

Let's add dates to our graph to make it more useful. The first date from the weather data file is in the second row of the file:

```
"USW00025333","SITKA AIRPORT, AK US","2018-07-01","0.25",,"62","50"
```

The data will be read in as a string, so we need a way to convert the string "2018-07-01" to an object representing this date. We can construct an object representing July 1, 2018 using the strptime() method from the datetime module. Let's see how strptime() works in a terminal session:

```
>>> from datetime import datetime
>>> first_date = datetime.strptime('2018-07-01', '%Y-%m-%d')
>>> print(first_date)
2018-07-01 00:00:00
```

We first import the datetime class from the datetime module. Then we call the method strptime() using the string containing the date we want to work with as its first argument. The second argument tells Python how the date is formatted. In this example, Python interprets '%Y-' to mean the part of the string before the first dash is a four-digit year; '%m-' means the part of the string before the second dash is a number representing the month; and '%d' means the last part of the string is the day of the month, from 1 to 31.

The strptime() method can take a variety of arguments to determine how to interpret the date. Table 16-1 shows some of these arguments.

Table 16-1: Date and Time Formatting Arguments from the datetime Module

Argument	Meaning
%A	Weekday name, such as *Monday*
%B	Month name, such as *January*
%m	Month, as a number (01 to 12)
%d	Day of the month, as a number (01 to 31)
%Y	Four-digit year, such as 2019
%y	Two-digit year, such as 19
%H	Hour, in 24-hour format (00 to 23)
%I	Hour, in 12-hour format (01 to 12)
%p	AM or PM
%M	Minutes (00 to 59)
%S	Seconds (00 to 61)

Plotting Dates

Now we can improve our temperature data plot by extracting dates for the daily highs and passing those highs and dates to plot(), as shown here:

sitka_highs.py

```
import csv
from datetime import datetime

import matplotlib.pyplot as plt

filename = 'data/sitka_weather_07-2018_simple.csv'
```

```
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

    # Get dates and high temperatures from this file.
❶   dates, highs = [], []
    for row in reader:
❷       current_date = datetime.strptime(row[2], '%Y-%m-%d')
        high = int(row[5])
        dates.append(current_date)
        highs.append(high)

# Plot the high temperatures.
plt.style.use('seaborn')
fig, ax = plt.subplots()
❸ ax.plot(dates, highs, c='red')

# Format plot.
plt.title("Daily high temperatures, July 2018", fontsize=24)
plt.xlabel('', fontsize=16)
❹ fig.autofmt_xdate()
plt.ylabel("Temperature (F)", fontsize=16)
plt.tick_params(axis='both', which='major', labelsize=16)

plt.show()
```

We create two empty lists to store the dates and high temperatures from the file ❶. We then convert the data containing the date information (`row[2]`) to a `datetime` object ❷ and append it to `dates`. We pass the dates and the high temperature values to `plot()` ❸. The call to `fig.autofmt_xdate()` ❹ draws the date labels diagonally to prevent them from overlapping. Figure 16-2 shows the improved graph.

Figure 16-2: The graph is more meaningful now that it has dates on the x-axis.

Plotting a Longer Timeframe

With our graph set up, let's add more data to get a more complete picture of the weather in Sitka. Copy the file *sitka_weather_2018_simple.csv*, which contains a full year's worth of weather data for Sitka, to the folder where you're storing the data for this chapter's programs.

Now we can generate a graph for the entire year's weather:

sitka_highs.py

```
--snip--
❶ filename = 'data/sitka_weather_2018_simple.csv'
with open(filename) as f:
--snip--
# Format plot.
❷ plt.title("Daily high temperatures - 2018", fontsize=24)
plt.xlabel('', fontsize=16)
--snip--
```

We modify the filename to use the new data file *sitka_weather_2018_simple.csv* ❶, and we update the title of our plot to reflect the change in its content ❷. Figure 16-3 shows the resulting plot.

Figure 16-3: A year's worth of data

Plotting a Second Data Series

We can make our informative graph even more useful by including the low temperatures. We need to extract the low temperatures from the data file and then add them to our graph, as shown here:

sitka_highs_lows.py

```
--snip--
filename = 'sitka_weather_2018_simple.csv'
```

```
    with open(filename) as f:
        reader = csv.reader(f)
        header_row = next(reader)

        # Get dates, and high and low temperatures from this file.
❶       dates, highs, lows = [], [], []
        for row in reader:
            current_date = datetime.strptime(row[2], '%Y-%m-%d')
            high = int(row[5])
❷           low = int(row[6])
            dates.append(current_date)
            highs.append(high)
            lows.append(low)

    # Plot the high and low temperatures.
    plt.style.use('seaborn')
    fig, ax = plt.subplots()
    ax.plot(dates, highs, c='red')
❸   ax.plot(dates, lows, c='blue')

    # Format plot.
❹   plt.title("Daily high and low temperatures - 2018", fontsize=24)
    --snip--
```

At ❶ we add the empty list `lows` to hold low temperatures, and then extract and store the low temperature for each date from the seventh position in each row (`row[6]`) ❷. At ❸ we add a call to `plot()` for the low temperatures and color these values blue. Finally, we update the title ❹. Figure 16-4 shows the resulting chart.

Figure 16-4: Two data series on the same plot

Shading an Area in the Chart

Having added two data series, we can now examine the range of temperatures for each day. Let's add a finishing touch to the graph by using shading to show the range between each day's high and low temperatures. To do so, we'll use the `fill_between()` method, which takes a series of x-values and two series of y-values, and fills the space between the two y-value series:

sitka_highs_lows.py

```
--snip--
# Plot the high and low temperatures.
plt.style.use('seaborn')
fig, ax = plt.subplots()
❶ ax.plot(dates, highs, c='red', alpha=0.5)
  ax.plot(dates, lows, c='blue', alpha=0.5)
❷ plt.fill_between(dates, highs, lows, facecolor='blue', alpha=0.1)
--snip--
```

The `alpha` argument at ❶ controls a color's transparency. An `alpha` value of 0 is completely transparent, and 1 (the default) is completely opaque. By setting `alpha` to 0.5, we make the red and blue plot lines appear lighter.

At ❷ we pass `fill_between()` the list `dates` for the x-values and then the two y-value series `highs` and `lows`. The `facecolor` argument determines the color of the shaded region; we give it a low `alpha` value of 0.1 so the filled region connects the two data series without distracting from the information they represent. Figure 16-5 shows the plot with the shaded region between the highs and lows.

Figure 16-5: The region between the two data sets is shaded.

The shading helps make the range between the two data sets immediately apparent.

Error Checking

We should be able to run the *sitka_highs_lows.py* code using data for any location. But some weather stations collect different data than others, and some occasionally malfunction and fail to collect some of the data they're supposed to. Missing data can result in exceptions that crash our programs unless we handle them properly.

For example, let's see what happens when we attempt to generate a temperature plot for Death Valley, California. Copy the file *death_valley_2018_simple.csv* to the folder where you're storing the data for this chapter's programs.

First, let's run the code to see the headers that are included in this data file:

death_valley_highs_lows.py

```
import csv

filename = 'data/death_valley_2018_simple.csv'
with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

    for index, column_header in enumerate(header_row):
        print(index, column_header)
```

Here's the output:

```
0 STATION
1 NAME
2 DATE
3 PRCP
4 TMAX
5 TMIN
6 TOBS
```

The date is in the same position at index 2. But the high and low temperatures are at indexes 4 and 5, so we'd need to change the indexes in our code to reflect these new positions. Instead of including an average temperature reading for the day, this station includes `TOBS`, a reading for a specific observation time.

I removed one of the temperature readings from this file to show what happens when some data is missing from a file. Change *sitka_highs_lows.py* to generate a graph for Death Valley using the indexes we just noted, and see what happens:

death_valley_highs_lows.py

```
--snip--
filename = 'data/death_valley_2018_simple.csv'
with open(filename) as f:
    --snip--
    # Get dates, and high and low temperatures from this file.
    dates, highs, lows = [], [], []
    for row in reader:
        current_date = datetime.strptime(row[2], '%Y-%m-%d')
```

```
❶         high = int(row[4])
          low = int(row[5])
          dates.append(current_date)
--snip--
```

At ❶ we update the indexes to correspond to this file's `TMAX` and `TMIN` positions.

When we run the program, we get an error, as shown in the last line in the following output:

```
Traceback (most recent call last):
  File "death_valley_highs_lows.py", line 15, in <module>
    high = int(row[4])
ValueError: invalid literal for int() with base 10: ''
```

The traceback tells us that Python can't process the high temperature for one of the dates because it can't turn an empty string (`''`) into an integer. Rather than look through the data and finding out which reading is missing, we'll just handle cases of missing data directly.

We'll run error-checking code when the values are being read from the CSV file to handle exceptions that might arise. Here's how that works:

death_valley_highs_lows.py

```
--snip--
filename = 'data/death_valley_2018_simple.csv'
with open(filename) as f:
    --snip--
    for row in reader:
            current_date = datetime.strptime(row[2], '%Y-%m-%d')
❶           try:
                high = int(row[4])
                low = int(row[5])
            except ValueError:
❷               print(f"Missing data for {current_date}")
❸           else:
                dates.append(current_date)
                highs.append(high)
                lows.append(low)

# Plot the high and low temperatures.
--snip--

# Format plot.
❹ title = "Daily high and low temperatures - 2018\nDeath Valley, CA"
plt.title(title, fontsize=20)
plt.xlabel('', fontsize=16)
--snip--
```

Each time we examine a row, we try to extract the date and the high and low temperature ❶. If any data is missing, Python will raise a `ValueError` and we handle it by printing an error message that includes the date of the missing data ❷. After printing the error, the loop will continue processing the next row. If all data for a date is retrieved without error, the `else` block

will run and the data will be appended to the appropriate lists ❸. Because we're plotting information for a new location, we update the title to include the location on the plot, and we use a smaller font size to accommodate the longer title ❹.

When you run *death_valley_highs_lows.py* now, you'll see that only one date had missing data:

```
Missing data for 2018-02-18 00:00:00
```

Because the error is handled appropriately, our code is able to generate a plot, which skips over the missing data. Figure 16-6 shows the resulting plot.

Figure 16-6: Daily high and low temperatures for Death Valley

Comparing this graph to the Sitka graph, we can see that Death Valley is warmer overall than southeast Alaska, as we expect. Also, the range of temperatures each day is greater in the desert. The height of the shaded region makes this clear.

Many data sets you work with will have missing, improperly formatted, or incorrect data. You can use the tools you learned in the first half of this book to handle these situations. Here we used a `try-except-else` block to handle missing data. Sometimes you'll use `continue` to skip over some data or use `remove()` or `del` to eliminate some data after it's been extracted. Use any approach that works, as long as the result is a meaningful, accurate visualization.

Downloading Your Own Data

If you want to download your own weather data, follow these steps:

1. Visit the NOAA Climate Data Online site at *https://www.ncdc.noaa.gov/cdo-web/*. In the *Discover Data By* section, click **Search Tool**. In the *Select a Dataset* box, choose **Daily Summaries**.

2. Select a date range, and in the *Search For* section, choose **ZIP Codes**. Enter the ZIP Code you're interested in, and click **Search**.
3. On the next page, you'll see a map and some information about the area you're focusing on. Below the location name, click **View Full Details**, or click the map and then click **Full Details**.
4. Scroll down and click **Station List** to see the weather stations that are available in this area. Choose one of the stations, and click **Add to Cart**. This data is free, even though the site uses a shopping cart icon. In the upper-right corner, click the cart.
5. In *Select the Output*, choose **Custom GHCN-Daily CSV**. Make sure the date range is correct, and click **Continue**.
6. On the next page, you can select the kinds of data you want. You can download one kind of data, for example, focusing on air temperature, or you can download all the data available from this station. Make your choices, and then click **Continue**.
7. On the last page, you'll see a summary of your order. Enter your email address, and click **Submit Order**. You'll receive a confirmation that your order was received, and in a few minutes you should receive another email with a link to download your data.

The data you download will be structured just like the data we worked with in this section. It might have different headers than those you saw in this section. But if you follow the same steps we used here, you should be able to generate visualizations of the data you're interested in.

TRY IT YOURSELF

16-1. Sitka Rainfall: Sitka is in a temperate rainforest, so it gets a fair amount of rainfall. In the data file *sitka_weather_2018_simple.csv* is a header called `PRCP`, which represents daily rainfall amounts. Make a visualization focusing on the data in this column. You can repeat the exercise for Death Valley if you're curious how little rainfall occurs in a desert.

16-2. Sitka–Death Valley Comparison: The temperature scales on the Sitka and Death Valley graphs reflect the different data ranges. To accurately compare the temperature range in Sitka to that of Death Valley, you need identical scales on the y-axis. Change the settings for the y-axis on one or both of the charts in Figures 16-5 and 16-6. Then make a direct comparison between temperature ranges in Sitka and Death Valley (or any two places you want to compare).

16-3. San Francisco: Are temperatures in San Francisco more like temperatures in Sitka or temperatures in Death Valley? Download some data for San Francisco, and generate a high-low temperature plot for San Francisco to make a comparison.

16-4. Automatic Indexes: In this section, we hardcoded the indexes corresponding to the `TMIN` and `TMAX` columns. Use the header row to determine the indexes for these values, so your program can work for Sitka or Death Valley. Use the station name to automatically generate an appropriate title for your graph as well.

16-5. Explore: Generate a few more visualizations that examine any other weather aspect you're interested in for any locations you're curious about.

Mapping Global Data Sets: JSON Format

In this section, you'll download a data set representing all the earthquakes that have occurred in the world during the previous month. Then you'll make a map showing the location of these earthquakes and how significant each one was. Because the data is stored in the JSON format, we'll work with it using the `json` module. Using Plotly's beginner-friendly mapping tool for location-based data, you'll create visualizations that clearly show the global distribution of earthquakes.

Downloading Earthquake Data

Copy the file *eq_1_day_m1.json* to the folder where you're storing the data for this chapter's programs. Earthquakes are categorized by their magnitude on the Richter scale. This file includes data for all earthquakes with a magnitude M1 or greater that took place in the last 24 hours (at the time of this writing). This data comes from one of the United States Geological Survey's earthquake data feeds, which you can find at *https://earthquake.usgs.gov/earthquakes/feed/*.

Examining JSON Data

When you open *eq_1_day_m1.json*, you'll see that it's very dense and hard to read:

```
{"type":"FeatureCollection","metadata":{"generated":1550361461000,...
{"type":"Feature","properties":{"mag":1.2,"place":"11km NNE of Nor...
{"type":"Feature","properties":{"mag":4.3,"place":"69km NNW of Ayn...
{"type":"Feature","properties":{"mag":3.6,"place":"126km SSE of Co...
{"type":"Feature","properties":{"mag":2.1,"place":"21km NNW of Teh...
{"type":"Feature","properties":{"mag":4,"place":"57km SSW of Kakto...
--snip--
```

This file is formatted more for machines than it is for humans. But we can see that the file contains some dictionaries, as well as information that we're interested in, such as earthquake magnitudes and locations.

The `json` module provides a variety of tools for exploring and working with JSON data. Some of these tools will help us reformat the file so we can look at the raw data more easily before we begin to work with it programmatically.

Let's start by loading the data and displaying it in a format that's easier to read. This is a long data file, so instead of printing it, we'll rewrite the data to a new file. Then we can open that file and scroll back and forth easily through the data:

eq_explore_data.py

```
import json

# Explore the structure of the data.
filename = 'data/eq_data_1_day_m1.json'
with open(filename) as f:
❶    all_eq_data = json.load(f)

❷ readable_file = 'data/readable_eq_data.json'
with open(readable_file, 'w') as f:
❸    json.dump(all_eq_data, f, indent=4)
```

We first import the `json` module to load the data properly from the file, and then store the entire set of data in `all_eq_data` ❶. The `json.load()` function converts the data into a format Python can work with: in this case, a giant dictionary. At ❷ we create a file to write this same data into a more readable format. The `json.dump()` function takes a JSON data object and a file object, and writes the data to that file ❸. The `indent=4` argument tells `dump()` to format the data using indentation that matches the data's structure.

When you look in your *data* directory and open the file *readable_eq_data.json*, here's the first part of what you'll see:

readable_eq_data.json

```
{
    "type": "FeatureCollection",
❶   "metadata": {
        "generated": 1550361461000,
        "url": "https://earthquake.usgs.gov/earthquakes/.../1.0_day.geojson",
        "title": "USGS Magnitude 1.0+ Earthquakes, Past Day",
        "status": 200,
        "api": "1.7.0",
        "count": 158
    },
❷   "features": [
    --snip--
```

The first part of the file includes a section with the key `"metadata"`. This tells us when the data file was generated and where we can find the data online. It also gives us a human-readable title and the number of earthquakes included in this file. In this 24-hour period, 158 earthquakes were recorded.

This *geoJSON* file has a structure that's helpful for location-based data. The information is stored in a list associated with the key `"features"` ❷. Because this file contains earthquake data, the data is in list form where every item in the list corresponds to a single earthquake. This structure might look confusing, but it's quite powerful. It allows geologists to store as much information as they need to in a dictionary about each earthquake, and then stuff all those dictionaries into one big list.

Let's look at a dictionary representing a single earthquake:

readable_eq_data.json

```
    --snip--
        {
            "type": "Feature",
❶           "properties": {
                "mag": 0.96,
                --snip--
❷               "title": "M 1.0 - 8km NE of Aguanga, CA"
            },
❸           "geometry": {
                "type": "Point",
                "coordinates": [
❹                   -116.7941667,
❺                   33.4863333,
                    3.22
                ]
            },
            "id": "ci37532978"
        },
```

The key `"properties"` contains a lot of information about each earthquake ❶. We're mainly interested in the magnitude of each quake, which is associated with the key `"mag"`. We're also interested in the title of each earthquake, which provides a nice summary of its magnitude and location ❷.

The key `"geometry"` helps us understand where the earthquake occurred ❸. We'll need this information to map each event. We can find the longitude ❹ and the latitude ❺ for each earthquake in a list associated with the key `"coordinates"`.

This file contains way more nesting than we'd use in the code we write, so if it looks confusing, don't worry: Python will handle most of the complexity. We'll only be working with one or two nesting levels at a time. We'll start by pulling out a dictionary for each earthquake that was recorded in the 24-hour time period.

NOTE *When we talk about locations, we often say the location's latitude first, followed by the longitude. This convention probably arose because humans discovered latitude long before we developed the concept of longitude. However, many geospatial frameworks list the longitude first and then the latitude, because this corresponds to the* (x, y) *convention we use in mathematical representations. The geoJSON format follows the (longitude, latitude) convention, and if you use a different framework it's important to learn what convention that framework follows.*

Making a List of All Earthquakes

First, we'll make a list that contains all the information about every earthquake that occurred.

eq_explore_data.py

```
import json

# Explore the structure of the data.
filename = 'data/eq_data_1_day_m1.json'
with open(filename) as f:
    all_eq_data = json.load(f)

all_eq_dicts = all_eq_data['features']
print(len(all_eq_dicts))
```

We take the data associated with the key `'features'` and store it in `all_eq_dicts`. We know this file contains records about 158 earthquakes, and the output verifies that we've captured all of the earthquakes in the file:

```
158
```

Notice how short this code is. The neatly formatted file *readable_eq_data.json* has over 6,000 lines. But in just a few lines, we can read through all that data and store it in a Python list. Next, we'll pull the magnitudes from each earthquake.

Extracting Magnitudes

Using the list containing data about each earthquake, we can loop through that list and extract any information we want. Now we'll pull the magnitude of each earthquake:

eq_explore_data.py

```
--snip--
all_eq_dicts = all_eq_data['features']

❶ mags = []
for eq_dict in all_eq_dicts:
❷     mag = eq_dict['properties']['mag']
    mags.append(mag)

print(mags[:10])
```

We make an empty list to store the magnitudes, and then loop through the dictionary `all_eq_dicts` ❶. Inside this loop, each earthquake is represented by the dictionary `eq_dict`. Each earthquake's magnitude is stored in the `'properties'` section of this dictionary under the key `'mag'` ❷. We store each magnitude in the variable `mag`, and then append it to the list `mags`.

We print the first 10 magnitudes, so we can see whether we're getting the correct data:

```
[0.96, 1.2, 4.3, 3.6, 2.1, 4, 1.06, 2.3, 4.9, 1.8]
```

Next, we'll pull the location data for each earthquake, and then we can make a map of the earthquakes.

Extracting Location Data

The location data is stored under the key `"geometry"`. Inside the geometry dictionary is a `"coordinates"` key, and the first two values in this list are the longitude and latitude. Here's how we'll pull this data:

eq_explore_data.py

```
--snip--
all_eq_dicts = all_eq_data['features']

mags, lons, lats = [], [], []
for eq_dict in all_eq_dicts:
    mag = eq_dict['properties']['mag']
❶   lon = eq_dict['geometry']['coordinates'][0]
    lat = eq_dict['geometry']['coordinates'][1]
    mags.append(mag)
    lons.append(lon)
    lats.append(lat)

print(mags[:10])
print(lons[:5])
print(lats[:5])
```

We make empty lists for the longitudes and latitudes. The code `eq_dict['geometry']` accesses the dictionary representing the geometry element of the earthquake ❶. The second key, `'coordinates'`, pulls the list of values associated with `'coordinates'`. Finally, the 0 index asks for the first value in the list of coordinates, which corresponds to an earthquake's longitude.

When we print the first five longitudes and latitudes, the output shows that we're pulling the correct data:

```
[0.96, 1.2, 4.3, 3.6, 2.1, 4, 1.06, 2.3, 4.9, 1.8]
[-116.7941667, -148.9865, -74.2343, -161.6801, -118.5316667]
[33.4863333, 64.6673, -12.1025, 54.2232, 35.3098333]
```

With this data, we can move on to mapping each earthquake.

Building a World Map

With the information we've pulled so far, we can build a simple world map. Although it won't look presentable yet, we want to make sure the information is displayed correctly before focusing on style and presentation issues. Here's the initial map:

eq_world_map.py

```
import json

❶ from plotly.graph_objs import Scattergeo, Layout
from plotly import offline

--snip--
```

```
for eq_dict in all_eq_dicts:
    --snip--

# Map the earthquakes.
❷ data = [Scattergeo(lon=lons, lat=lats)]
❸ my_layout = Layout(title='Global Earthquakes')

❹ fig = {'data': data, 'layout': my_layout}
offline.plot(fig, filename='global_earthquakes.html')
```

We import the `Scattergeo` chart type and the `Layout` class, and then import the `offline` module to render the map ❶. As we did when making a bar chart, we define a list called `data`. We create the `Scattergeo` object inside this list ❷, because you can plot more than one data set on any visualization you make. A `Scattergeo` chart type allows you to overlay a scatter plot of geographic data on a map. In the simplest use of this chart type, you only need to provide a list of longitudes and a list of latitudes.

We give the chart an appropriate title ❸ and create a dictionary called `fig` that contains the data and the layout ❹. Finally, we pass `fig` to the `plot()` function along with a descriptive filename for the output. When you run this file, you should see a map that looks like the one in Figure 16-7. Earthquakes usually occur near plate boundaries, which matches what we see in the chart.

Figure 16-7: A simple map showing where all the earthquakes in the last 24 hours occurred

We can do a lot of modifications to make this map more meaningful and easier to read, so let's make some of these changes.

A Different Way of Specifying Chart Data

Before we configure the chart, let's look at a slightly different way to specify the data for a Plotly chart. In the current chart, the `data` list is defined in one line:

```
data = [Scattergeo(lon=lons, lat=lats)]
```

This is one of the simplest ways to define the data for a chart in Plotly. But it's not the best way when you want to customize the presentation. Here's an equivalent way to define the data for the current chart:

```
data = [{
    'type': 'scattergeo',
    'lon': lons,
    'lat': lats,
}]
```

In this approach, all the information about the data is structured as key-value pairs in a dictionary. If you put this code into *eq_plot.py*, you'll see the same chart we just generated. This format allows us to specify customizations more easily than the previous format.

Customizing Marker Size

When we're figuring out how to improve the map's styling, we should focus on aspects of the data that we want to communicate more clearly. The current map shows the location of each earthquake, but it doesn't communicate the severity of any earthquake. We want viewers to immediately see where the most significant earthquakes occur in the world.

To do this, we'll change the size of markers depending on the magnitude of each earthquake:

eq_world_map.py

```
import json
--snip--
# Map the earthquakes.
data = [{
    'type': 'scattergeo',
    'lon': lons,
    'lat': lats,
❶   'marker': {
❷       'size': [5*mag for mag in mags],
    },
}]
my_layout = Layout(title='Global Earthquakes')
--snip--
```

Plotly offers a huge variety of customizations you can make to a data series, each of which can be expressed as a key-value pair. Here we're using the key `'marker'` to specify how big each marker on the map should be ❶. We use a nested dictionary as the value associated with `'marker'`, because you can specify a number of settings for all the markers in a series.

We want the size to correspond to the magnitude of each earthquake. But if we just pass in the `mags` list, the markers would be too small to easily see the size differences. We need to multiply the magnitude by a scale factor to get an appropriate marker size. On my screen, a value of 5 works well; a slightly smaller or larger value might work better for your map. We use a list comprehension, which generates an appropriate marker size for each value in the `mags` list ❷.

When you run this code, you should see a map that looks like the one in Figure 16-8. This is a much better map, but we can still do more.

Figure 16-8: The map now shows the magnitude of each earthquake.

Customizing Marker Colors

We can also customize each marker's color to provide some classification to the severity of each earthquake. We'll use Plotly's colorscales to do this. Before you make these changes, copy the file *eq_data_30_day_m1.json* to your data directory. This file includes earthquake data for a 30-day period, and the map will be much more interesting to look at using this larger data set.

Here's how to use a colorscale to represent the magnitude of each earthquake:

eq_world_map.py

```
--snip--
❶ filename = 'data/eq_data_30_day_m1.json'
--snip--
# Map the earthquakes.
data = [{
    --snip--
```

```
    'marker': {
        'size': [5*mag for mag in mags],
❷       'color': mags,
❸       'colorscale': 'Viridis',
❹       'reversescale': True,
❺       'colorbar': {'title': 'Magnitude'},
    },
}]
--snip--
```

Be sure to update the filename so you're using the 30-day data set ❶. All the significant changes here occur in the `'marker'` dictionary, because we're only modifying the markers' appearance. The `'color'` setting tells Plotly what values it should use to determine where each marker falls on the colorscale ❷. We use the `mags` list to determine the color that's used. The `'colorscale'` setting tells Plotly which range of colors to use: `'Viridis'` is a colorscale that ranges from dark blue to bright yellow and works well for this data set ❸. We set `'reversescale'` to `True`, because we want to use bright yellow for the lowest values and dark blue for the most severe earthquakes ❹. The `'colorbar'` setting allows us to control the appearance of the colorscale shown on the side of the map. Here we title the colorscale `'Magnitude'` to make it clear what the colors represent ❹.

When you run the program now, you'll see a much nicer-looking map. In Figure 16-9, the colorscale shows the severity of individual earthquakes. Plotting this many earthquakes really makes it clear where the tectonic plate boundaries are!

Figure 16-9: In 30 days' worth of earthquakes, color and size are used to represent the magnitude of each earthquake.

Other Colorscales

You can also choose from a number of other colorscales. To see the available colorscales, save the following short program as *show_color_scales.py*:

show_color_scales.py

```
from plotly import colors

for key in colors.PLOTLY_SCALES.keys():
    print(key)
```

Plotly stores the colorscales in the `colors` module. The colorscales are defined in the dictionary `PLOTLY_SCALES`, and the names of the colorscales serve as the keys in the dictionary. Here's the output showing all the available colorscales:

```
Greys
YlGnBu
Greens
--snip--
Viridis
```

Feel free to try out these colorscales; remember that you can reverse any of these scales using the `reversescale` setting.

NOTE *If you print the `PLOTLY_SCALES` dictionary, you can see how colorscales are defined. Every scale has a beginning color and an end color, and some scales have one or more intermediate colors defined as well. Plotly interpolates shades between each of these defined colors.*

Adding Hover Text

To finish this map, we'll add some informative text that appears when you hover over the marker representing an earthquake. In addition to showing the longitude and latitude, which appear by default, we'll show the magnitude and provide a description of the approximate location as well.

To make this change, we need to pull a little more data from the file and add it to the dictionary in `data` as well:

eq_world_map.py

```
--snip--
❶ mags, lons, lats, hover_texts = [], [], [], []
for eq_dict in all_eq_dicts:
    --snip--
    lat = eq_dict['geometry']['coordinates'][1]
❷   title = eq_dict['properties']['title']
    mags.append(mag)
    lons.append(lon)
    lats.append(lat)
    hover_texts.append(title)
--snip--
```

```
# Map the earthquakes.
data = [{
    'type': 'scattergeo',
    'lon': lons,
    'lat': lats,
❸   'text': hover_texts,
    'marker': {
        --snip--
    },
}]
--snip--
```

We first make a list called `hover_texts` to store the label we'll use for each marker ❶. The "title" section of the earthquake data contains a descriptive name of the magnitude and location of each earthquake in addition to its longitude and latitude. At ❷ we pull this information and assign it to the variable `title`, and then append it to the list `hover_texts`.

When we include the key `'text'` in the `data` object, Plotly uses this value as a label for each marker when the viewer hovers over the marker. When we pass a list that matches the number of markers, Plotly pulls an individual label for each marker it generates ❸. When you run this program, you should be able to hover over any marker, see a description of where that earthquake took place, and read its exact magnitude.

This is impressive! In approximately 40 lines of code, we've created a visually appealing and meaningful map of global earthquake activity that also illustrates the geological structure of the planet. Plotly offers a wide range of ways you can customize the appearance and behavior of your visualizations. Using Plotly's many options, you can make charts and maps that show exactly what you want them to.

TRY IT YOURSELF

16-6. Refactoring: The loop that pulls data from `all_eq_dicts` uses variables for the magnitude, longitude, latitude, and title of each earthquake before appending these values to their appropriate lists. This approach was chosen for clarity in how to pull data from a JSON file, but it's not necessary in your code. Instead of using these temporary variables, pull each value from `eq_dict` and append it to the appropriate list in one line. Doing so should shorten the body of this loop to just four lines.

16-7. Automated Title: In this section, we specified the title manually when defining `my_layout`, which means we have to remember to update the title every time the source file changes. Instead, you can use the title for the data set in the metadata part of the JSON file. Pull this value, assign it to a variable, and use this for the title of the map when you're defining `my_layout`.

(continued)

16-8. Recent Earthquakes: You can find data files containing information about the most recent earthquakes over 1-hour, 1-day, 7-day, and 30-day periods online. Go to *https://earthquake.usgs.gov/earthquakes/feed/v1.0/geojson.php* and you'll see a list of links to data sets for various time periods, focusing on earthquakes of different magnitudes. Download one of these data sets, and create a visualization of the most recent earthquake activity.

16-9. World Fires: In the resources for this chapter, you'll find a file called *world_fires_1_day.csv*. This file contains information about fires burning in different locations around the globe, including the latitude and longitude, and the brightness of each fire. Using the data processing work from the first part of this chapter and the mapping work from this section, make a map that shows which parts of the world are affected by fires.

You can download more recent versions of this data at *https://earthdata.nasa.gov/earth-observation-data/near-real-time/firms/active-fire-data/*. You can find links to the data in CSV format in the *TXT* section.

Summary

In this chapter, you learned to work with real-world data sets. You processed CSV and JSON files, and extracted the data you want to focus on. Using historical weather data, you learned more about working with Matplotlib, including how to use the `datetime` module and how to plot multiple data series on one chart. You plotted geographical data on a world map in Plotly and styled Plotly maps and charts as well.

As you gain experience working with CSV and JSON files, you'll be able to process almost any data you want to analyze. You can download most online data sets in either or both of these formats. By working with these formats, you'll be able to learn how to work with other data formats more easily as well.

In the next chapter, you'll write programs that automatically gather their own data from online sources, and then you'll create visualizations of that data. These are fun skills to have if you want to program as a hobby and critical skills if you're interested in programming professionally.

17

WORKING WITH APIs

In this chapter, you'll learn how to write a self-contained program that generates a visualization based on data that it retrieves. Your program will use a web *application programming interface (API)* to automatically request specific information from a website—rather than entire pages—and then use that information to generate a visualization. Because programs written like this will always use current data to generate a visualization, even when that data might be rapidly changing, it will always be up to date.

Using a Web API

A web API is a part of a website designed to interact with programs. Those programs use very specific URLs to request certain information. This kind of request is called an *API call*. The requested data will be returned in an

easily processed format, such as JSON or CSV. Most apps that rely on external data sources, such as apps that integrate with social media sites, rely on API calls.

Git and GitHub

We'll base our visualization on information from GitHub, a site that allows programmers to collaborate on coding projects. We'll use GitHub's API to request information about Python projects on the site, and then generate an interactive visualization of the relative popularity of these projects using Plotly.

GitHub (*https://github.com/*) takes its name from Git, a distributed version control system. Git helps people manage their work on a project, so changes made by one person won't interfere with changes other people are making. When you implement a new feature in a project, Git tracks the changes you make to each file. When your new code works, you *commit* the changes you've made, and Git records the new state of your project. If you make a mistake and want to revert your changes, you can easily return to any previously working state. (To learn more about version control using Git, see Appendix D.) Projects on GitHub are stored in *repositories*, which contain everything associated with the project: its code, information on its collaborators, any issues or bug reports, and so on.

When users on GitHub like a project, they can "star" it to show their support and keep track of projects they might want to use. In this chapter, we'll write a program to automatically download information about the most-starred Python projects on GitHub, and then we'll create an informative visualization of these projects.

Requesting Data Using an API Call

GitHub's API lets you request a wide range of information through API calls. To see what an API call looks like, enter the following into your browser's address bar and press ENTER:

```
https://api.github.com/search/repositories?q=language:python&sort=stars
```

This call returns the number of Python projects currently hosted on GitHub, as well as information about the most popular Python repositories. Let's examine the call. The first part, `https://api.github.com/`, directs the request to the part of GitHub that responds to API calls. The next part, `search/repositories`, tells the API to conduct a search through all repositories on GitHub.

The question mark after `repositories` signals that we're about to pass an argument. The `q` stands for *query*, and the equal sign (`=`) lets us begin specifying a query (`q=`). By using `language:python`, we indicate that we want information only on repositories that have Python as the primary language. The final part, `&sort=stars`, sorts the projects by the number of stars they've been given.

The following snippet shows the first few lines of the response.

```
{
❶  "total_count": 3494012,
❷  "incomplete_results": false,
❸  "items": [
     {
       "id": 21289110,
       "node_id": "MDEwOlJlcG9zaXRvcnkyMTI4OTExMA==",
       "name": "awesome-python",
       "full_name": "vinta/awesome-python",
       --snip--
```

You can see from the response that this URL is not primarily intended to be entered by humans, because it's in a format that's meant to be processed by a program. GitHub found 3,494,012 Python projects as of this writing ❶. Because the value for `"incomplete_results"` is `false`, we know that the request was successful (it's not incomplete) ❷. If GitHub had been unable to fully process the API request, it would have returned `true` here. The `"items"` returned are displayed in the list that follows, which contains details about the most popular Python projects on GitHub ❸.

Installing Requests

The Requests package allows a Python program to easily request information from a website and examine the response. Use `pip` to install Requests:

```
$ python -m pip install --user requests
```

This line tells Python to run the `pip` module and install the Requests package to the current user's Python installation. If you use **`python3`** or a different command when running programs or installing packages, make sure you use the same command here.

NOTE *If this command doesn't work on macOS, try running the command again without the `--user` flag.*

Processing an API Response

Now we'll begin to write a program to automatically issue an API call and process the results by identifying the most starred Python projects on GitHub:

python_repos.py

```
❶ import requests

  # Make an API call and store the response.
❷ url = 'https://api.github.com/search/repositories?q=language:python&sort=stars'
❸ headers = {'Accept': 'application/vnd.github.v3+json'}
❹ r = requests.get(url, headers=headers)
❺ print(f"Status code: {r.status_code}")
```

```
# Store API response in a variable.
❻ response_dict = r.json()

# Process results.
print(response_dict.keys())
```

At ❶ we import the `requests` module. At ❷ we store the URL of the API call in the `url` variable. GitHub is currently on the third version of its API, so we define headers for the API call ❸ that ask explicitly to use this version of the API. Then we use `requests` to make the call to the API ❹.

We call `get()` and pass it the URL and the header that we defined, and we assign the response object to the variable `r`. The response object has an attribute called `status_code`, which tells us whether the request was successful. (A status code of 200 indicates a successful response.) At ❺ we print the value of `status_code` so we can make sure the call went through successfully.

The API returns the information in JSON format, so we use the `json()` method to convert the information to a Python dictionary ❻. We store the resulting dictionary in `response_dict`.

Finally, we print the keys from `response_dict` and see this output:

```
Status code: 200
dict_keys(['total_count', 'incomplete_results', 'items'])
```

Because the status code is 200, we know that the request was successful. The response dictionary contains only three keys: `'total_count'`, `'incomplete_results'`, and `'items'`. Let's take a look inside the response dictionary.

NOTE *Simple calls like this should return a complete set of results, so it's safe to ignore the value associated with `'incomplete_results'`. But when you're making more complex API calls, your program should check this value.*

Working with the Response Dictionary

With the information from the API call stored as a dictionary, we can work with the data stored there. Let's generate some output that summarizes the information. This is a good way to make sure we received the information we expected and to start examining the information we're interested in:

python_repos.py

```
import requests

# Make an API call and store the response.
--snip--

# Store API response in a variable.
response_dict = r.json()
❶ print(f"Total repositories: {response_dict['total_count']}")

# Explore information about the repositories.
❷ repo_dicts = response_dict['items']
print(f"Repositories returned: {len(repo_dicts)}")
```

```
# Examine the first repository.
❸ repo_dict = repo_dicts[0]
❹ print(f"\nKeys: {len(repo_dict)}")
❺ for key in sorted(repo_dict.keys()):
    print(key)
```

At ❶ we print the value associated with `'total_count'`, which represents the total number of Python repositories on GitHub.

The value associated with `'items'` is a list containing a number of dictionaries, each of which contains data about an individual Python repository. At ❷ we store this list of dictionaries in `repo_dicts`. We then print the length of `repo_dicts` to see how many repositories we have information for.

To look closer at the information returned about each repository, we pull out the first item from `repo_dicts` and store it in `repo_dict` ❸. We then print the number of keys in the dictionary to see how much information we have ❹. At ❺ we print all the dictionary's keys to see what kind of information is included.

The results give us a clearer picture of the actual data:

```
Status code: 200
Total repositories: 3494030
Repositories returned: 30

❶ Keys: 73
archive_url
archived
assignees_url
--snip--
url
watchers
watchers_count
```

GitHub's API returns a lot of information about each repository: there are 73 keys in `repo_dict` ❶. When you look through these keys, you'll get a sense of the kind of information you can extract about a project. (The only way to know what information is available through an API is to read the documentation or to examine the information through code, as we're doing here.)

Let's pull out the values for some of the keys in `repo_dict`:

python_repos.py

```
--snip--
# Explore information about the repositories.
repo_dicts = response_dict['items']
print(f"Repositories returned: {len(repo_dicts)}")

# Examine the first repository.
repo_dict = repo_dicts[0]

print("\nSelected information about first repository:")
❶ print(f"Name: {repo_dict['name']}")
❷ print(f"Owner: {repo_dict['owner']['login']}")
❸ print(f"Stars: {repo_dict['stargazers_count']}")
print(f"Repository: {repo_dict['html_url']}")
```

```
❹ print(f"Created: {repo_dict['created_at']}")
❺ print(f"Updated: {repo_dict['updated_at']}")
  print(f"Description: {repo_dict['description']}")
```

Here, we print the values for a number of keys from the first repository's dictionary. At ❶ we print the name of the project. An entire dictionary represents the project's owner, so at ❷ we use the key `owner` to access the dictionary representing the owner, and then use the key `login` to get the owner's login name. At ❸ we print how many stars the project has earned and the URL for the project's GitHub repository. We then show when it was created ❹ and when it was last updated ❺. Finally, we print the repository's description; the output should look something like this:

```
Status code: 200
Total repositories: 3494032
Repositories returned: 30

Selected information about first repository:
Name: awesome-python
Owner: vinta
Stars: 61549
Repository: https://github.com/vinta/awesome-python
Created: 2014-06-27T21:00:06Z
Updated: 2019-02-17T04:30:00Z
Description: A curated list of awesome Python frameworks, libraries, software
    and resources
```

We can see that the most-starred Python project on GitHub as of this writing is *awesome-python*, its owner is user *vinta*, and it has been starred by more than 60,000 GitHub users. We can see the URL for the project's repository, its creation date of June 2014, and that it was updated recently. Additionally, the description tells us that awesome-python contains a list of popular Python resources.

Summarizing the Top Repositories

When we make a visualization for this data, we'll want to include more than one repository. Let's write a loop to print selected information about each repository the API call returns so we can include them all in the visualization:

python_repos.py

```
--snip--
# Explore information about the repositories.
repo_dicts = response_dict['items']
print(f"Repositories returned: {len(repo_dicts)}")

❶ print("\nSelected information about each repository:")
❷ for repo_dict in repo_dicts:
    print(f"\nName: {repo_dict['name']}")
    print(f"Owner: {repo_dict['owner']['login']}")
    print(f"Stars: {repo_dict['stargazers_count']}")
    print(f"Repository: {repo_dict['html_url']}")
    print(f"Description: {repo_dict['description']}")
```

We print an introductory message at ❶. At ❷ we loop through all the dictionaries in `repo_dicts`. Inside the loop, we print the name of each project, its owner, how many stars it has, its URL on GitHub, and the project's description, as shown here:

```
Status code: 200
Total repositories: 3494040
Repositories returned: 30

Selected information about each repository:

Name: awesome-python
Owner: vinta
Stars: 61549
Repository: https://github.com/vinta/awesome-python
Description: A curated list of awesome Python frameworks, libraries, software
  and resources

Name: system-design-primer
Owner: donnemartin
Stars: 57256
Repository: https://github.com/donnemartin/system-design-primer
Description: Learn how to design large-scale systems. Prep for the system
  design interview.  Includes Anki flashcards.
--snip--

Name: python-patterns
Owner: faif
Stars: 19058
Repository: https://github.com/faif/python-patterns
Description: A collection of design patterns/idioms in Python
```

Some interesting projects appear in these results, and it might be worth looking at a few. But don't spend too much time, because shortly we'll create a visualization that will make the results much easier to read.

Monitoring API Rate Limits

Most APIs are rate limited, which means there's a limit to how many requests you can make in a certain amount of time. To see if you're approaching GitHub's limits, enter *https://api.github.com/rate_limit* into a web browser. You should see a response that begins like this:

```
{
  "resources": {
    "core": {
      "limit": 60,
      "remaining": 58,
      "reset": 1550385312
    },
❶   "search": {
❷     "limit": 10,
❸     "remaining": 8,
```

```
❹       "reset": 1550381772
    },
    --snip--
```

The information we're interested in is the rate limit for the search API ❶. We see at ❷ that the limit is 10 requests per minute and that we have 8 requests remaining for the current minute ❸. The reset value represents the time in *Unix* or *epoch time* (the number of seconds since midnight on January 1, 1970) when our quota will reset ❹. If you reach your quota, you'll get a short response that lets you know you've reached the API limit. If you reach the limit, just wait until your quota resets.

NOTE *Many APIs require you to register and obtain an API key to make API calls. As of this writing, GitHub has no such requirement, but if you obtain an API key, your limits will be much higher.*

Visualizing Repositories Using Plotly

Let's make a visualization using the data we have now to show the relative popularity of Python projects on GitHub. We'll make an interactive bar chart: the height of each bar will represent the number of stars the project has acquired, and you can click the bar's label to go to that project's home on GitHub. Save a copy of the program we've been working on as *python_repos_visual.py*, and then modify it so it reads as follows:

python_repos_visual.py

```
    import requests

❶ from plotly.graph_objs import Bar
  from plotly import offline

❷ # Make an API call and store the response.
  url = 'https://api.github.com/search/repositories?q=language:python&sort=stars'
  headers = {'Accept': 'application/vnd.github.v3+json'}
  r = requests.get(url, headers=headers)
  print(f"Status code: {r.status_code}")

  # Process results.
  response_dict = r.json()
  repo_dicts = response_dict['items']
❸ repo_names, stars = [], []
  for repo_dict in repo_dicts:
      repo_names.append(repo_dict['name'])
      stars.append(repo_dict['stargazers_count'])

  # Make visualization.
❹ data = [{
      'type': 'bar',
      'x': repo_names,
      'y': stars,
  }]
```

```
❺ my_layout = {
      'title': 'Most-Starred Python Projects on GitHub',
      'xaxis': {'title': 'Repository'},
      'yaxis': {'title': 'Stars'},
  }

  fig = {'data': data, 'layout': my_layout}
  offline.plot(fig, filename='python_repos.html')
```

We import the `Bar` class and the `offline` module from `plotly` ❶. We don't need to import the `Layout` class because we'll use the dictionary approach to define the layout, just as we did for the `data` list in the earthquake mapping project in Chapter 16. We continue to print the status of the API call response so we'll know if there is a problem ❷. We also remove some of the code that processes the API response, because we're no longer in the exploratory phase; we know we have the data we want.

We then create two empty lists ❸ to store the data we'll include in the initial chart. We'll need the name of each project to label the bars, and the number of stars to determine the height of the bars. In the loop, we append the name of each project and the number of stars it has to these lists.

Next, we define the `data` list ❹. This contains a dictionary, like we used in Chapter 16, which defines the type of the plot and provides the data for the x- and y-values. The x-values are the names of the projects, and the y-values are the number of stars each project has been given.

At ❺ we define the layout for this chart using the dictionary approach. Instead of making an instance of the `Layout` class, we build a dictionary with the layout specifications we want to use. We set a title for the overall chart, and we define a label for each axis.

Figure 17-1 shows the resulting chart. We can see that the first few projects are significantly more popular than the rest, but all of them are important projects in the Python ecosystem.

Figure 17-1: The most-starred Python projects on GitHub

Refining Plotly Charts

Let's refine the chart's styling. As you saw in Chapter 16, you can include all the styling directives as key-value pairs in the `data` and `my_layout` dictionaries.

Changes to the `data` object affect the bars. Here's a modified version of the `data` object for our chart that gives us a specific color and a clear border for each bar:

python_repos_visual.py

```
--snip--
data = [{
    'type': 'bar',
    'x': repo_names,
    'y': stars,
    'marker': {
        'color': 'rgb(60, 100, 150)',
        'line': {'width': 1.5, 'color': 'rgb(25, 25, 25)'}
    },
    'opacity': 0.6,
}]
--snip--
```

The `marker` settings shown here affect the design of the bars. We set a custom blue color for the bars and specify that they'll be outlined with a dark gray line that's 1.5 pixels wide. We also set the opacity of the bars to 0.6 to soften the appearance of the chart a little.

Next, we'll modify `my_layout`:

python_repos_visual.py

```
--snip--
my_layout = {
    'title': 'Most-Starred Python Projects on GitHub',
❶   'titlefont': {'size': 28},
❷   'xaxis': {
        'title': 'Repository',
        'titlefont': {'size': 24},
        'tickfont': {'size': 14},
    },
❸   'yaxis': {
        'title': 'Stars',
        'titlefont': {'size': 24},
        'tickfont': {'size': 14},
    },
}
--snip--
```

We use the `'titlefont'` key to define the font size of the overall chart title ❶. Within the `'xaxis'` dictionary, we add settings to control the font size of the x-axis title (`'titlefont'`) and also of the tick labels (`'tickfont'`) ❷. Because these are individual nested dictionaries, you can include keys for the color and font family of the axis titles and tick labels. At ❸ we define similar settings for the y-axis.

Figure 17-2 shows the restyled chart.

Figure 17-2: The styling for the chart has been refined.

Adding Custom Tooltips

In Plotly, you can hover the cursor over an individual bar to show the information that the bar represents. This is commonly called a *tooltip*, and in this case, it currently shows the number of stars a project has. Let's create a custom tooltip to show each project's description as well as the project's owner.

We need to pull some additional data to generate the tooltips and modify the `data` object:

python_repos_visual.py

```
--snip--
# Process results.
response_dict = r.json()
repo_dicts = response_dict['items']
❶ repo_names, stars, labels = [], [], []
for repo_dict in repo_dicts:
    repo_names.append(repo_dict['name'])
    stars.append(repo_dict['stargazers_count'])

❷     owner = repo_dict['owner']['login']
    description = repo_dict['description']
❸     label = f"{owner}<br />{description}"
    labels.append(label)


# Make visualization.
data = [{
    'type': 'bar',
    'x': repo_names,
    'y': stars,
❹     'hovertext': labels,
    'marker': {
```

```
        'color': 'rgb(60, 100, 150)',
        'line': {'width': 1.5, 'color': 'rgb(25, 25, 25)'}
    },
    'opacity': 0.6,
}]
--snip--
```

We first define a new empty list, `labels`, to hold the text we want to display for each project ❶. In the loop where we process the data, we pull the owner and the description for each project ❷. Plotly allows you to use HTML code within text elements, so we generate a string for the label with a line break (`<br />`) between the project owner's username and the description ❸. We then store this label in the list `labels`.

In the `data` dictionary, we add an entry with the key `'hovertext'` and assign it the list we just created ❹. As Plotly creates each bar, it will pull labels from this list and only display them when the viewer hovers over a bar.

Figure 17-3 shows the resulting chart.

Figure 17-3: Hovering over a bar shows the project's owner and description.

Adding Clickable Links to Our Graph

Because Plotly allows you to use HTML on text elements, we can easily add links to a chart. Let's use the x-axis labels as a way to let the viewer visit any project's home page on GitHub. We need to pull the URLs from the data and use them when generating the x-axis labels:

python_repos_visual.py

```
--snip--
# Process results.
response_dict = r.json()
repo_dicts = response_dict['items']
```

```
❶ repo_links, stars, labels = [], [], []
  for repo_dict in repo_dicts:
      repo_name = repo_dict['name']
❷     repo_url = repo_dict['html_url']
❸     repo_link = f"<a href='{repo_url}'>{repo_name}</a>"
      repo_links.append(repo_link)

      stars.append(repo_dict['stargazers_count'])
      --snip--

  # Make visualization.
  data = [{
      'type': 'bar',
❹     'x': repo_links,
      'y': stars,
      --snip--
  }]
  --snip--
```

We update the name of the list we're creating from `repo_names` to `repo_links` to more accurately communicate the kind of information we're putting together for the chart ❶. We then pull the URL for the project from `repo_dict` and assign it to the temporary variable `repo_url` ❷. At ❸ we generate a link to the project. We use the HTML anchor tag, which has the form `<a href='URL'>link text</a>`, to generate the link. We then append this link to the list `repo_links`.

At ❹ we use this list for the x-values in the chart. The result looks the same as before, but now the viewer can click any of the project names at the bottom of the chart to visit that project's home page on GitHub. Now we have an interactive, informative visualization of data retrieved through an API!

More About Plotly and the GitHub API

To read more about working with Plotly charts, there are two good places to start. You can find the *Plotly User Guide in Python* at *https://plot.ly/python/user-guide/*. This resource gives you a better understanding of how Plotly uses your data to construct a visualization and why it approaches defining data visualizations in this way.

The *python figure reference* at *https://plot.ly/python/reference/* lists all the settings you can use to configure Plotly visualizations. All the possible chart types are listed as well as all the attributes you can set for every configuration option.

For more about the GitHub API, refer to its documentation at *https://developer.github.com/v3/*. Here you'll learn how to pull a wide variety of specific information from GitHub. If you have a GitHub account, you can work with your own data as well as the publicly available data for other users' repositories.

The Hacker News API

To explore how to use API calls on other sites, let's take a quick look at Hacker News (*http://news.ycombinator.com/*). On Hacker News, people share articles about programming and technology, and engage in lively discussions about those articles. The Hacker News API provides access to data about all submissions and comments on the site, and you can use the API without having to register for a key.

The following call returns information about the current top article as of this writing:

```
https://hacker-news.firebaseio.com/v0/item/19155826.json
```

When you enter this URL in a browser, you'll see that the text on the page is enclosed by braces, meaning it's a dictionary. But the response is difficult to examine without some better formatting. Let's run this URL through the `json.dump()` method, like we did in the earthquake project in Chapter 16, so we can explore the kind of information that's returned about an article:

hn_article.py

```
import requests
import json

# Make an API call, and store the response.
url = 'https://hacker-news.firebaseio.com/v0/item/19155826.json'
r = requests.get(url)
print(f"Status code: {r.status_code}")

# Explore the structure of the data.
response_dict = r.json()
readable_file = 'data/readable_hn_data.json'
with open(readable_file, 'w') as f:
    json.dump(response_dict, f, indent=4)
```

Everything in this program should look familiar, because we've used it all in the previous two chapters. The output is a dictionary of information about the article with the ID 19155826:

readable_hn_data.json

```
{
    "by": "jimktrains2",
❶   "descendants": 220,
    "id": 19155826,
❷   "kids": [
        19156572,
        19158857,
        --snip--
    ],
    "score": 722,
    "time": 1550085414,
❸   "title": "Nasa's Mars Rover Opportunity Concludes a 15-Year Mission",
    "type": "story",
❹   "url": "https://www.nytimes.com/.../mars-opportunity-rover-dead.html"
}
```

The dictionary contains a number of keys we can work with. The key `'descendants'` tells us the number of comments the article has received ❶. The key `'kids'` provides the IDs of all comments made directly in response to this submission ❷. Each of these comments might have comments of their own as well, so the number of descendants a submission has is usually greater than its number of kids. We can see the title of the article being discussed ❸, and a URL for the article that's being discussed as well ❹.

The following URL returns a simple list of all the IDs of the current top articles on Hacker News:

```
https://hacker-news.firebaseio.com/v0/topstories.json
```

We can use this call to find out which articles are on the home page right now, and then generate a series of API calls similar to the one we just examined. With this approach, we can print a summary of all the articles on the front page of Hacker News at the moment:

hn_submissions.py

```
  from operator import itemgetter

  import requests

  # Make an API call and store the response.
❶ url = 'https://hacker-news.firebaseio.com/v0/topstories.json'
  r = requests.get(url)
  print(f"Status code: {r.status_code}")

  # Process information about each submission.
❷ submission_ids = r.json()
❸ submission_dicts = []
  for submission_id in submission_ids[:30]:
      # Make a separate API call for each submission.
❹     url = f"https://hacker-news.firebaseio.com/v0/item/{submission_id}.json"
      r = requests.get(url)
      print(f"id: {submission_id}\tstatus: {r.status_code}")
      response_dict = r.json()

      # Build a dictionary for each article.
❺     submission_dict = {
          'title': response_dict['title'],
          'hn_link': f"http://news.ycombinator.com/item?id={submission_id}",
          'comments': response_dict['descendants'],
      }
❻     submission_dicts.append(submission_dict)

❼ submission_dicts = sorted(submission_dicts, key=itemgetter('comments'),
                            reverse=True)

❽ for submission_dict in submission_dicts:
      print(f"\nTitle: {submission_dict['title']}")
      print(f"Discussion link: {submission_dict['hn_link']}")
      print(f"Comments: {submission_dict['comments']}")
```

First, we make an API call, and then print the status of the response ❶. This API call returns a list containing the IDs of up to the 500 most popular articles on Hacker News at the time the call is issued. We then convert the response object to a Python list at ❷, which we store in `submission_ids`. We'll use these IDs to build a set of dictionaries that each store information about one of the current submissions.

We set up an empty list called `submission_dicts` at ❸ to store these dictionaries. We then loop through the IDs of the top 30 submissions. We make a new API call for each submission by generating a URL that includes the current value of `submission_id` ❹. We print the status of each request along with its ID, so we can see whether it's successful.

At ❺ we create a dictionary for the submission currently being processed, where we store the title of the submission, a link to the discussion page for that item, and the number of comments the article has received so far. Then we append each `submission_dict` to the list `submission_dicts` ❻.

Each submission on Hacker News is ranked according to an overall score based on a number of factors including how many times it's been voted up, how many comments it's received, and how recent the submission is. We want to sort the list of dictionaries by the number of comments. To do this, we use a function called `itemgetter()` ❼, which comes from the `operator` module. We pass this function the key `'comments'`, and it pulls the value associated with that key from each dictionary in the list. The `sorted()` function then uses this value as its basis for sorting the list. We sort the list in reverse order to place the most-commented stories first.

Once the list is sorted, we loop through the list at ❽ and print out three pieces of information about each of the top submissions: the title, a link to the discussion page, and the number of comments the submission currently has:

```
Status code: 200
id: 19155826    status: 200
id: 19180181    status: 200
id: 19181473    status: 200
--snip--

Title: Nasa's Mars Rover Opportunity Concludes a 15-Year Mission
Discussion link: http://news.ycombinator.com/item?id=19155826
Comments: 220

Title: Ask HN: Is it practical to create a software-controlled model rocket?
Discussion link: http://news.ycombinator.com/item?id=19180181
Comments: 72

Title: Making My Own USB Keyboard from Scratch
Discussion link: http://news.ycombinator.com/item?id=19181473
Comments: 62
--snip--
```

You would use a similar process to access and analyze information with any API. With this data, you could make a visualization showing which submissions have inspired the most active recent discussions. This is also the basis for apps that provide a customized reading experience for sites like Hacker News. To learn more about what kind of information you can access through the Hacker News API, visit the documentation page at *https://github.com/HackerNews/API/*.

TRY IT YOURSELF

17-1. Other Languages: Modify the API call in *python_repos.py* so it generates a chart showing the most popular projects in other languages. Try languages such as *JavaScript*, *Ruby*, *C*, *Java*, *Perl*, *Haskell*, and *Go*.

17-2. Active Discussions: Using the data from *hn_submissions.py*, make a bar chart showing the most active discussions currently happening on Hacker News. The height of each bar should correspond to the number of comments each submission has. The label for each bar should include the submission's title and should act as a link to the discussion page for that submission.

17-3. Testing python_repos.py: In *python_repos.py*, we printed the value of `status_code` to make sure the API call was successful. Write a program called *test_python_repos.py* that uses `unittest` to assert that the value of `status_code` is 200. Figure out some other assertions you can make—for example, that the number of items returned is expected and that the total number of repositories is greater than a certain amount.

17-4. Further Exploration: Visit the documentation for Plotly and either the GitHub API or the Hacker News API. Use some of the information you find there to either customize the style of the plots we've already made or pull some different information and create your own visualizations.

Summary

In this chapter, you learned how to use APIs to write self-contained programs that automatically gather the data they need and use that data to create a visualization. You used the GitHub API to explore the most-starred Python projects on GitHub, and you also looked briefly at the Hacker News API. You learned how to use the Requests package to automatically issue an API call to GitHub and how to process the results of that call. Some Plotly settings were also introduced that further customize the appearance of the charts you generate.

In the next chapter, you'll use Django to build a web application as your final project.

PROJECT 3

WEB APPLICATIONS

18

GETTING STARTED WITH DJANGO

Behind the scenes, today's websites are rich applications that act like fully developed desktop applications. Python has a great set of tools called Django for building web applications. Django is a *web framework*—a set of tools designed to help you build interactive websites. In this chapter, you'll learn how to use Django (*https://djangoproject.com/*) to build a project called Learning Log—an online journal system that lets you keep track of information you've learned about particular topics.

We'll write a specification for this project, and then we'll define models for the data the app will work with. We'll use Django's admin system to enter some initial data, and then you'll learn to write views and templates so Django can build the site's pages.

Django can respond to page requests and make it easier to read and write to a database, manage users, and much more. In Chapters 19 and 20, you'll refine the Learning Log project and then deploy it to a live server so you (and your friends) can use it.

Setting Up a Project

When beginning a project, you first need to describe the project in a specification, or *spec*. Then you'll set up a virtual environment in which to build the project.

Writing a Spec

A full spec details the project goals, describes the project's functionality, and discusses its appearance and user interface. Like any good project or business plan, a spec should keep you focused and help keep your project on track. We won't write a full project spec here, but we'll lay out a few clear goals to keep the development process focused. Here's the spec we'll use:

> We'll write a web app called Learning Log that allows users to log the topics they're interested in and to make journal entries as they learn about each topic. The Learning Log home page will describe the site and invite users to either register or log in. Once logged in, a user can create new topics, add new entries, and read and edit existing entries.

When you learn about a new topic, keeping a journal of what you've learned can be helpful in tracking and revisiting information. A good app makes this process efficient.

Creating a Virtual Environment

To work with Django, we'll first set up a virtual environment. A *virtual environment* is a place on your system where you can install packages and isolate them from all other Python packages. Separating one project's libraries from other projects is beneficial and will be necessary when we deploy Learning Log to a server in Chapter 20.

Create a new directory for your project called *learning_log*, switch to that directory in a terminal, and enter the following code to create a virtual environment:

```
learning_log$ python -m venv ll_env
learning_log$
```

Here we're running the `venv` virtual environment module and using it to create a virtual environment named *ll_env* (note that this is *ll_env* with two lowercase *L*s, not two ones). If you use a command such as **`python3`** when running programs or installing packages, make sure to use that command here.

Activating the Virtual Environment

Now we need to activate the virtual environment using the following command:

```
learning_log$ source ll_env/bin/activate
❶ (ll_env)learning_log$
```

This command runs the script *activate* in *ll_env/bin*. When the environment is active, you'll see the name of the environment in parentheses, as shown at ❶; then you can install packages to the environment and use packages that have already been installed. Packages you install in *ll_env* will be available only while the environment is active.

NOTE *If you're using Windows, use the command* `ll_env\Scripts\activate` *(without the word* `source`*) to activate the virtual environment. If you're using PowerShell, you might need to capitalize* `Activate`*.*

To stop using a virtual environment, enter **`deactivate`**:

```
(ll_env)learning_log$ deactivate
learning_log$
```

The environment will also become inactive when you close the terminal it's running in.

Installing Django

Once the virtual environment is activated, enter the following to install Django:

```
(ll_env)learning_log$ pip install django
Collecting django
--snip--
Installing collected packages: pytz, django
Successfully installed django-2.2.0 pytz-2018.9 sqlparse-0.2.4
(ll_env)learning_log$
```

Because we're working in a virtual environment, which is its own self-contained environment, this command is the same on all systems. There's no need to use the `--user` flag, and there's no need to use longer commands, such as `python -m pip install` *`package_name`*.

Keep in mind that Django will be available only when the *ll_env* environment is active.

NOTE *Django releases a new version about every eight months, so you may see a newer version when you install Django. This project will most likely work as it's written here, even on newer versions of Django. If you want to make sure to use the same version of Django you see here, use the command* `pip install django==2.2.*`*. This will install the latest release of Django 2.2. If you have any issues related to the version you're using, see the online resources for the book at* https://nostarch.com/pythoncrashcourse2e/.

Creating a Project in Django

Without leaving the active virtual environment (remember to look for *ll_env* in parentheses in the terminal prompt), enter the following commands to create a new project:

```
❶ (ll_env)learning_log$ django-admin startproject learning_log .
❷ (ll_env)learning_log$ ls
  learning_log ll_env manage.py
❸ (ll_env)learning_log$ ls learning_log
  __init__.py  settings.py  urls.py  wsgi.py
```

The command at ❶ tells Django to set up a new project called *learning_log*. The dot at the end of the command creates the new project with a directory structure that will make it easy to deploy the app to a server when we're finished developing it.

NOTE *Don't forget this dot, or you might run into some configuration issues when you deploy the app. If you forget the dot, delete the files and folders that were created (except* ll_env*), and run the command again.*

Running the `ls` command (`dir` on Windows) ❷ shows that Django has created a new directory called *learning_log*. It also created a *manage.py* file, which is a short program that takes in commands and feeds them to the relevant part of Django to run them. We'll use these commands to manage tasks, such as working with databases and running servers.

The *learning_log* directory contains four files ❸; the most important are *settings.py*, *urls.py*, and *wsgi.py*. The *settings.py* file controls how Django interacts with your system and manages your project. We'll modify a few of these settings and add some settings of our own as the project evolves. The *urls.py* file tells Django which pages to build in response to browser requests. The *wsgi.py* file helps Django serve the files it creates. The filename is an acronym for *web server gateway interface*.

Creating the Database

Django stores most of the information for a project in a database, so next we need to create a database that Django can work with. Enter the following command (still in an active environment):

```
  (ll_env)learning_log$ python manage.py migrate
❶ Operations to perform:
    Apply all migrations: admin, auth, contenttypes, sessions
  Running migrations:
    Applying contenttypes.0001_initial... OK
    Applying auth.0001_initial... OK
    --snip--
    Applying sessions.0001_initial... OK
❷ (ll_env)learning_log$ ls
  db.sqlite3  learning_log  ll_env  manage.py
```

Any time we modify a database, we say we're *migrating* the database. Issuing the `migrate` command for the first time tells Django to make sure the database matches the current state of the project. The first time we run this command in a new project using SQLite (more about SQLite in a moment), Django will create a new database for us. At ❶, Django reports that it will prepare the database to store information it needs to handle administrative and authentication tasks.

Running the `ls` command shows that Django created another file called *db.sqlite3* ❷. SQLite is a database that runs off a single file; it's ideal for writing simple apps because you won't have to pay much attention to managing the database.

NOTE *In an active virtual environment, use the command* `python` *to run* `manage.py` *commands, even if you use something different, like* `python3`*, to run other programs. In a virtual environment, the command* `python` *refers to the version of Python that created the virtual environment.*

Viewing the Project

Let's make sure that Django has set up the project properly. Enter the `runserver` command as follows to view the project in its current state:

```
(ll_env)learning_log$ python manage.py runserver
Watchman unavailable: pywatchman not installed.
Watching for file changes with StatReloader
Performing system checks...

❶ System check identified no issues (0 silenced).
  February 18, 2019 - 16:26:07
❷ Django version 2.2.0, using settings 'learning_log.settings'
❸ Starting development server at http://127.0.0.1:8000/
  Quit the server with CONTROL-C.
```

Django should start a server called the *development server*, so you can view the project on your system to see how well it works. When you request a page by entering a URL in a browser, the Django server responds to that request by building the appropriate page and sending it to the browser.

At ❶, Django checks to make sure the project is set up properly; at ❷ it reports the version of Django in use and the name of the settings file in use; and at ❸ it reports the URL where the project is being served. The URL *http://127.0.0.1:8000/* indicates that the project is listening for requests on port 8000 on your computer, which is called a localhost. The term *localhost* refers to a server that only processes requests on your system; it doesn't allow anyone else to see the pages you're developing.

Open a web browser and enter the URL *http://localhost:8000/*, or *http://127.0.0.1:8000/* if the first one doesn't work. You should see something like Figure 18-1, a page that Django creates to let you know all is working properly so far. Keep the server running for now, but when you want to stop the server, press CTRL-C in the terminal where the `runserver` command was issued.

Figure 18-1: Everything is working so far.

NOTE *If you receive the error message* That port is already in use, *tell Django to use a different port by entering* `python manage.py runserver 8001`, *and then cycle through higher numbers until you find an open port.*

TRY IT YOURSELF

18-1. New Projects: To get a better idea of what Django does, build a couple of empty projects and look at what Django creates. Make a new folder with a simple name, like *snap_gram* or *insta_chat* (outside of your *learning_log* directory), navigate to that folder in a terminal, and create a virtual environment. Install Django and run the command `django-admin.py startproject snap_gram .` (make sure you include the dot at the end of the command).

Look at the files and folders this command creates, and compare them to Learning Log. Do this a few times until you're familiar with what Django creates when starting a new project. Then delete the project directories if you wish.

Starting an App

A Django *project* is organized as a group of individual *apps* that work together to make the project work as a whole. For now, we'll create just one app to do most of our project's work. We'll add another app in Chapter 19 to manage user accounts.

You should leave the development server running in the terminal window you opened earlier. Open a new terminal window (or tab), and navigate to the directory that contains *manage.py*. Activate the virtual environment, and then run the `startapp` command:

```
learning_log$ source ll_env/bin/activate
(ll_env)learning_log$ python manage.py startapp learning_logs
❶ (ll_env)learning_log$ ls
db.sqlite3  learning_log  learning_logs  ll_env  manage.py
❷ (ll_env)learning_log$ ls learning_logs/
__init__.py  admin.py  apps.py  migrations  models.py  tests.py  views.py
```

The command `startapp` *appname* tells Django to create the infrastructure needed to build an app. When you look in the project directory now, you'll see a new folder called *learning_logs* ❶. Open that folder to see what Django has created ❷. The most important files are *models.py*, *admin.py*, and *views.py*. We'll use *models.py* to define the data we want to manage in our app. We'll look at *admin.py* and *views.py* a little later.

Defining Models

Let's think about our data for a moment. Each user will need to create a number of topics in their learning log. Each entry they make will be tied to a topic, and these entries will be displayed as text. We'll also need to store the timestamp of each entry, so we can show users when they made each entry.

Open the file *models.py*, and look at its existing content:

models.py

```
from django.db import models

# Create your models here.
```

A module called `models` is being imported for us, and we're being invited to create models of our own. A *model* tells Django how to work with the data that will be stored in the app. Code-wise, a model is just a class; it has attributes and methods, just like every class we've discussed. Here's the model for the topics users will store:

```
from django.db import models

class Topic(models.Model):
    """A topic the user is learning about."""
❶   text = models.CharField(max_length=200)
❷   date_added = models.DateTimeField(auto_now_add=True)

❸   def __str__(self):
        """Return a string representation of the model."""
        return self.text
```

We've created a class called `Topic`, which inherits from `Model`—a parent class included in Django that defines a model's basic functionality. We add two attributes to the `Topic` class: `text` and `date_added`.

The `text` attribute is a `CharField`—a piece of data that's made up of characters, or text ❶. You use `CharField` when you want to store a small amount of text, such as a name, a title, or a city. When we define a `CharField` attribute, we have to tell Django how much space it should reserve in the database. Here we give it a `max_length` of 200 characters, which should be enough to hold most topic names.

The `date_added` attribute is a `DateTimeField`—a piece of data that will record a date and time ❷. We pass the argument `auto_now_add=True`, which tells Django to automatically set this attribute to the current date and time whenever the user creates a new topic.

NOTE *To see the different kinds of fields you can use in a model, see the* Django Model Field Reference *at* https://docs.djangoproject.com/en/2.2/ref/models/fields/. *You won't need all the information right now, but it will be extremely useful when you're developing your own apps.*

We tell Django which attribute to use by default when it displays information about a topic. Django calls a `__str__()` method to display a simple representation of a model. Here we've written a `__str__()` method that returns the string stored in the `text` attribute ❸.

Activating Models

To use our models, we have to tell Django to include our app in the overall project. Open *settings.py* (in the *learning_log/learning_log* directory); you'll see a section that tells Django which apps are installed and work together in the project:

settings.py

```
--snip--
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
--snip--
```

Add our app to this list by modifying `INSTALLED_APPS` so it looks like this:

```
--snip--
INSTALLED_APPS = [
    # My apps
    'learning_logs',

   # Default django apps.
    'django.contrib.admin',
```

```
    --snip--
]
--snip--
```

Grouping apps together in a project helps to keep track of them as the project grows to include more apps. Here we start a section called *My apps*, which includes only `learning_logs` for now. It's important to place your own apps before the default apps in case you need to override any behavior of the default apps with your own custom behavior.

Next, we need to tell Django to modify the database so it can store information related to the model `Topic`. From the terminal, run the following command:

```
(ll_env)learning_log$ python manage.py makemigrations learning_logs
Migrations for 'learning_logs':
  learning_logs/migrations/0001_initial.py
    - Create model Topic
(ll_env)learning_log$
```

The command `makemigrations` tells Django to figure out how to modify the database so it can store the data associated with any new models we've defined. The output here shows that Django has created a migration file called *0001_initial.py*. This migration will create a table for the model `Topic` in the database.

Now we'll apply this migration and have Django modify the database for us:

```
(ll_env)learning_log$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, learning_logs, sessions
Running migrations:
❶ Applying learning_logs.0001_initial... OK
```

Most of the output from this command is identical to the first time we issued the `migrate` command. The line we need to check appears at ❶, where Django confirms that the migration for `learning_logs` worked `OK`.

Whenever we want to modify the data that Learning Log manages, we'll follow these three steps: modify *models.py*, call `makemigrations` on `learning_logs`, and tell Django to `migrate` the project.

The Django Admin Site

Django makes it easy to work with your models through the *admin site*. Only the site's administrators use the admin site, not general users. In this section, we'll set up the admin site and use it to add some topics through the `Topic` model.

Setting Up a Superuser

Django allows you to create a *superuser*, a user who has all privileges available on the site. A user's *privileges* control the actions that user can take.

The most restrictive privilege settings allow a user to only read public information on the site. Registered users typically have the privilege of reading their own private data and some selected information available only to members. To effectively administer a web application, the site owner usually needs access to all information stored on the site. A good administrator is careful with their users' sensitive information, because users put a lot of trust into the apps they access.

To create a superuser in Django, enter the following command and respond to the prompts:

```
(ll_env)learning_log$ python manage.py createsuperuser
❶ Username (leave blank to use 'eric'): ll_admin
❷ Email address:
❸ Password:
Password (again):
Superuser created successfully.
(ll_env)learning_log$
```

When you issue the command `createsuperuser`, Django prompts you to enter a username for the superuser ❶. Here I'm using *ll_admin*, but you can enter any username you want. You can enter an email address if you want or just leave this field blank ❷. You'll need to enter your password twice ❸.

NOTE *Some sensitive information can be hidden from a site's administrators. For example, Django doesn't store the password you enter; instead, it stores a string derived from the password, called a* hash. *Each time you enter your password, Django hashes your entry and compares it to the stored hash. If the two hashes match, you're authenticated. By requiring hashes to match, if an attacker gains access to a site's database, they'll be able to read its stored hashes but not the passwords. When a site is set up properly, it's almost impossible to get the original passwords from the hashes.*

Registering a Model with the Admin Site

Django includes some models in the admin site automatically, such as `User` and `Group`, but the models we create need to be added manually.

When we started the `learning_logs` app, Django created an *admin.py* file in the same directory as *models.py*. Open the *admin.py* file:

admin.py

```
from django.contrib import admin

# Register your models here.
```

To register `Topic` with the admin site, enter the following:

```
from django.contrib import admin

❶ from .models import Topic

❷ admin.site.register(Topic)
```

This code first imports the model we want to register, `Topic` ❶. The dot in front of `models` tells Django to look for *models.py* in the same directory as *admin.py*. The code `admin.site.register()` tells Django to manage our model through the admin site ❷.

Now use the superuser account to access the admin site. Go to *http://localhost:8000/admin/*, and enter the username and password for the superuser you just created. You should see a screen like the one in Figure 18-2. This page allows you to add new users and groups, and change existing ones. You can also work with data related to the `Topic` model that we just defined.

Figure 18-2: The admin site with `Topic` included

NOTE *If you see a message in your browser that the web page is not available, make sure you still have the Django server running in a terminal window. If you don't, activate a virtual environment and reissue the command `python manage.py runserver`. If you're having trouble viewing your project at any point in the development process, closing any open terminals and reissuing the `runserver` command is a good first troubleshooting step.*

Adding Topics

Now that `Topic` has been registered with the admin site, let's add our first topic. Click **Topics** to go to the Topics page, which is mostly empty, because we have no topics to manage yet. Click **Add Topic**, and a form for adding a new topic appears. Enter **`Chess`** in the first box and click **Save**. You'll be sent back to the Topics admin page, and you'll see the topic you just created.

Let's create a second topic so we'll have more data to work with. Click **Add Topic** again, and enter **`Rock Climbing`**. Click **Save**, and you'll be sent back to the main Topics page again. Now you'll see Chess and Rock Climbing listed.

Defining the Entry Model

For a user to record what they've been learning about chess and rock climbing, we need to define a model for the kinds of entries users can make in their learning logs. Each entry needs to be associated with a particular topic. This relationship is called a *many-to-one relationship*, meaning many entries can be associated with one topic.

Here's the code for the `Entry` model. Place it in your *models.py* file:

models.py

```
from django.db import models

class Topic(models.Model):
    --snip--

❶ class Entry(models.Model):
      """Something specific learned about a topic."""
❷     topic = models.ForeignKey(Topic, on_delete=models.CASCADE)
❸     text = models.TextField()
      date_added = models.DateTimeField(auto_now_add=True)

❹     class Meta:
          verbose_name_plural = 'entries'

      def __str__(self):
          """Return a string representation of the model."""
❺         return f"{self.text[:50]}..."
```

The `Entry` class inherits from Django's base `Model` class, just as `Topic` did ❶. The first attribute, `topic`, is a `ForeignKey` instance ❷. A *foreign key* is a database term; it's a reference to another record in the database. This is the code that connects each entry to a specific topic. Each topic is assigned a key, or ID, when it's created. When Django needs to establish a connection between two pieces of data, it uses the key associated with each piece of information. We'll use these connections shortly to retrieve all the entries associated with a certain topic. The `on_delete=models.CASCADE` argument tells Django that when a topic is deleted, all the entries associated with that topic should be deleted as well. This is known as a *cascading delete*.

Next is an attribute called `text`, which is an instance of `TextField` ❸. This kind of field doesn't need a size limit, because we don't want to limit the size of individual entries. The `date_added` attribute allows us to present entries in the order they were created and to place a timestamp next to each entry.

At ❹ we nest the `Meta` class inside our `Entry` class. The `Meta` class holds extra information for managing a model; here, it allows us to set a special attribute telling Django to use *Entries* when it needs to refer to more than one entry. Without this, Django would refer to multiple entries as *Entrys*.

The `__str__()` method tells Django which information to show when it refers to individual entries. Because an entry can be a long body of text, we tell Django to show just the first 50 characters of `text` ❺. We also add an ellipsis to clarify that we're not always displaying the entire entry.

Migrating the Entry Model

Because we've added a new model, we need to migrate the database again. This process will become quite familiar: you modify *models.py*, run the command `python manage.py makemigrations` *`app_name`*, and then run the command `python manage.py migrate`.

Migrate the database and check the output by entering the following commands:

```
(ll_env)learning_log$ python manage.py makemigrations learning_logs
Migrations for 'learning_logs':
❶  learning_logs/migrations/0002_entry.py
    - Create model Entry
(ll_env)learning_log$ python manage.py migrate
Operations to perform:
  --snip--
❷  Applying learning_logs.0002_entry... OK
```

A new migration called *0002_entry.py* is generated, which tells Django how to modify the database to store information related to the model `Entry` ❶. When we issue the `migrate` command, we see that Django applied this migration, and everything was okay ❷.

Registering Entry with the Admin Site

We also need to register the `Entry` model. Here's what *admin.py* should look like now:

admin.py

```
from django.contrib import admin

from .models import Topic, Entry

admin.site.register(Topic)
admin.site.register(Entry)
```

Go back to *http://localhost/admin/*, and you should see *Entries* listed under *Learning_Logs*. Click the **Add** link for Entries, or click **Entries**, and then choose **Add entry**. You should see a drop-down list to select the topic you're creating an entry for and a text box for adding an entry. Select **Chess** from the drop-down list, and add an entry. Here's the first entry I made:

> The opening is the first part of the game, roughly the first ten moves or so. In the opening, it's a good idea to do three things—bring out your bishops and knights, try to control the center of the board, and castle your king.
>
> Of course, these are just guidelines. It will be important to learn when to follow these guidelines and when to disregard these suggestions.

When you click **Save**, you'll be brought back to the main admin page for entries. Here, you'll see the benefit of using `text[:50]` as the string representation for each entry; it's much easier to work with multiple entries in the admin interface if you see only the first part of an entry rather than the entire text of each entry.

Make a second entry for Chess and one entry for Rock Climbing so we have some initial data. Here's a second entry for Chess:

> In the opening phase of the game, it's important to bring out your bishops and knights. These pieces are powerful and maneuverable enough to play a significant role in the beginning moves of a game.

And here's a first entry for Rock Climbing:

> One of the most important concepts in climbing is to keep your weight on your feet as much as possible. There's a myth that climbers can hang all day on their arms. In reality, good climbers have practiced specific ways of keeping their weight over their feet whenever possible.

These three entries will give us something to work with as we continue to develop Learning Log.

The Django Shell

With some data entered, we can examine that data programmatically through an interactive terminal session. This interactive environment is called the Django *shell*, and it's a great environment for testing and troubleshooting your project. Here's an example of an interactive shell session:

```
(ll_env)learning_log$ python manage.py shell
❶ >>> from learning_logs.models import Topic
>>> Topic.objects.all()
<QuerySet [<Topic: Chess>, <Topic: Rock Climbing>]>
```

The command `python manage.py shell`, run in an active virtual environment, launches a Python interpreter that you can use to explore the data stored in your project's database. Here, we import the model `Topic` from the `learning_logs.models` module ❶. We then use the method `Topic.objects.all()` to get all the instances of the model `Topic`; the list that's returned is called a *queryset*.

We can loop over a queryset just as we'd loop over a list. Here's how you can see the ID that's been assigned to each topic object:

```
>>> topics = Topic.objects.all()
>>> for topic in topics:
...   print(topic.id, topic)
```

```
...
1 Chess
2 Rock Climbing
```

We store the queryset in `topics`, and then print each topic's `id` attribute and the string representation of each topic. We can see that Chess has an ID of 1, and Rock Climbing has an ID of 2.

If you know the ID of a particular object, you can use the method `Topic.objects.get()` to retrieve that object and examine any attribute the object has. Let's look at the `text` and `date_added` values for Chess:

```
>>> t = Topic.objects.get(id=1)
>>> t.text
'Chess'
>>> t.date_added
datetime.datetime(2019, 2, 19, 1, 55, 31, 98500, tzinfo=<UTC>)
```

We can also look at the entries related to a certain topic. Earlier we defined the `topic` attribute for the `Entry` model. This was a `ForeignKey`, a connection between each entry and a topic. Django can use this connection to get every entry related to a certain topic, like this:

```
❶ >>> t.entry_set.all()
<QuerySet [<Entry: The opening is the first part of the game, roughly...>,
<Entry:
In the opening phase of the game, it's important t...>]>
```

To get data through a foreign key relationship, you use the lowercase name of the related model followed by an underscore and the word `set` ❶. For example, say you have the models `Pizza` and `Topping`, and `Topping` is related to `Pizza` through a foreign key. If your object is called `my_pizza`, representing a single pizza, you can get all of the pizza's toppings using the code `my_pizza.topping_set.all()`.

We'll use this kind of syntax when we begin to code the pages users can request. The shell is very useful for making sure your code retrieves the data you want it to. If your code works as you expect it to in the shell, you can expect it to work properly in the files within your project. If your code generates errors or doesn't retrieve the data you expect it to, it's much easier to troubleshoot your code in the simple shell environment than within the files that generate web pages. We won't refer to the shell much, but you should continue using it to practice working with Django's syntax for accessing the data stored in the project.

NOTE *Each time you modify your models, you'll need to restart the shell to see the effects of those changes. To exit a shell session, press CTRL-D; on Windows, press CTRL-Z and then press ENTER.*

TRY IT YOURSELF

18-2. Short Entries: The `__str__()` method in the `Entry` model currently appends an ellipsis to every instance of `Entry` when Django shows it in the admin site or the shell. Add an `if` statement to the `__str__()` method that adds an ellipsis only if the entry is longer than 50 characters. Use the admin site to add an entry that's fewer than 50 characters in length, and check that it doesn't have an ellipsis when viewed.

18-3. The Django API: When you write code to access the data in your project, you're writing a *query*. Skim through the documentation for querying your data at *https://docs.djangoproject.com/en/2.2/topics/db/queries/*. Much of what you see will look new to you, but it will be very useful as you start to work on your own projects.

18-4. Pizzeria: Start a new project called `pizzeria` with an app called `pizzas`. Define a model `Pizza` with a field called `name`, which will hold name values, such as `Hawaiian` and `Meat Lovers`. Define a model called `Topping` with fields called `pizza` and `name`. The `pizza` field should be a foreign key to `Pizza`, and `name` should be able to hold values such as `pineapple`, `Canadian bacon`, and `sausage`.

Register both models with the admin site, and use the site to enter some pizza names and toppings. Use the shell to explore the data you entered.

Making Pages: The Learning Log Home Page

Making web pages with Django consists of three stages: defining URLs, writing views, and writing templates. You can do these in any order, but in this project we'll always start by defining the URL pattern. A URL pattern describes the way the URL is laid out. It also tells Django what to look for when matching a browser request with a site URL so it knows which page to return.

Each URL then maps to a particular *view*—the view function retrieves and processes the data needed for that page. The view function often renders the page using a *template*, which contains the overall structure of the page. To see how this works, let's make the home page for Learning Log. We'll define the URL for the home page, write its view function, and create a simple template.

Because all we're doing is making sure Learning Log works as it's supposed to, we'll make a simple page for now. A functioning web app is fun to style when it's complete; an app that looks good but doesn't work well is pointless. For now, the home page will display only a title and a brief description.

Mapping a URL

Users request pages by entering URLs into a browser and clicking links, so we'll need to decide what URLs are needed. The home page URL is first: it's the base URL people use to access the project. At the moment the base URL, *http://localhost:8000/*, returns the default Django site that lets us know the project was set up correctly. We'll change this by mapping the base URL to Learning Log's home page.

In the main *learning_log* project folder, open the file *urls.py*. Here's the code you should see:

urls.py

```
❶ from django.contrib import admin
  from django.urls import path

❷ urlpatterns = [
❸     path('admin/', admin.site.urls),
  ]
```

The first two lines import a module and a function to manage URLs for the admin site ❶. The body of the file defines the `urlpatterns` variable ❷. In this *urls.py* file, which represents the project as a whole, the `urlpatterns` variable includes sets of URLs from the apps in the project. The code at ❸ includes the module `admin.site.urls`, which defines all the URLs that can be requested from the admin site.

We need to include the URLs for `learning_logs`, so add the following:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
❶   path('', include('learning_logs.urls')),
]
```

We've added a line to include the module `learning_logs.urls` at ❶.

The default *urls.py* is in the *learning_log* folder; now we need to make a second *urls.py* file in the *learning_logs* folder. Create a new Python file and save it as *urls.py* in *learning_logs*, and enter this code into it:

urls.py

```
❶ """Defines URL patterns for learning_logs."""

❷ from django.urls import path

❸ from . import views

❹ app_name = 'learning_logs'
❺ urlpatterns = [
      # Home page
❻     path('', views.index, name='index'),
  ]
```

To make it clear which *urls.py* we're working in, we add a docstring at the beginning of the file ❶. We then import the `path` function, which is needed when mapping URLs to views ❷. We also import the `views` module ❸; the dot tells Python to import the *views.py* module from the same directory as the current *urls.py* module. The variable `app_name` helps Django distinguish this *urls.py* file from files of the same name in other apps within the project ❹. The variable `urlpatterns` in this module is a list of individual pages that can be requested from the `learning_logs` app ❺.

The actual URL pattern is a call to the `path()` function, which takes three arguments ❻. The first argument is a string that helps Django route the current request properly. Django receives the requested URL and tries to route the request to a view. It does this by searching all the URL patterns we've defined to find one that matches the current request. Django ignores the base URL for the project (*http://localhost:8000/*), so the empty string (`''`) matches the base URL. Any other URL won't match this pattern, and Django will return an error page if the URL requested doesn't match any existing URL patterns.

The second argument in `path()` ❻ specifies which function to call in *views.py*. When a requested URL matches the pattern we're defining, Django calls the `index()` function from *views.py* (we'll write this view function in the next section). The third argument provides the name `index` for this URL pattern so we can refer to it in other code sections. Whenever we want to provide a link to the home page, we'll use this name instead of writing out a URL.

Writing a View

A view function takes in information from a request, prepares the data needed to generate a page, and then sends the data back to the browser, often by using a template that defines what the page will look like.

The file *views.py* in *learning_logs* was generated automatically when we ran the command `python manage.py startapp`. Here's what's in *views.py* right now:

views.py

```
from django.shortcuts import render

# Create your views here.
```

Currently, this file just imports the `render()` function, which renders the response based on the data provided by views. Open the views file and add the following code for the home page:

```
from django.shortcuts import render

def index(request):
    """The home page for Learning Log."""
    return render(request, 'learning_logs/index.html')
```

When a URL request matches the pattern we just defined, Django looks for a function called `index()` in the *views.py* file. Django then passes the

request object to this view function. In this case, we don't need to process any data for the page, so the only code in the function is a call to render(). The render() function here passes two arguments—the original request object and a template it can use to build the page. Let's write this template.

Writing a Template

The template defines what the page should look like, and Django fills in the relevant data each time the page is requested. A template allows you to access any data provided by the view. Because our view for the home page provided no data, this template is fairly simple.

Inside the *learning_logs* folder, make a new folder called *templates*. Inside the *templates* folder, make another folder called *learning_logs*. This might seem a little redundant (we have a folder named *learning_logs* inside a folder named *templates* inside a folder named *learning_logs*), but it sets up a structure that Django can interpret unambiguously, even in the context of a large project containing many individual apps. Inside the inner *learning_logs* folder, make a new file called *index.html*. The path to the file will be *learning_log/learning_logs/templates/learning_logs/index.html*. Enter the following code into that file:

index.html

```
<p>Learning Log</p>

<p>Learning Log helps you keep track of your learning, for any topic you're
learning about.</p>
```

This is a very simple file. If you're not familiar with HTML, the tags signify paragraphs. The tag opens a paragraph, and the tag closes a paragraph. We have two paragraphs: the first acts as a title, and the second describes what users can do with Learning Log.

Now when you request the project's base URL, *http://localhost:8000/*, you should see the page we just built instead of the default Django page. Django will take the requested URL, and that URL will match the pattern ''; then Django will call the function views.index(), which will render the page using the template contained in *index.html*. Figure 18-3 shows the resulting page.

Figure 18-3: The home page for Learning Log

Although it might seem like a complicated process for creating one page, this separation between URLs, views, and templates works quite well. It allows you to think about each aspect of a project separately. In larger projects, it allows individuals working on the project to focus on the areas in which they're strongest. For example, a database specialist can focus on the models, a programmer can focus on the view code, and a web designer can focus on the templates.

NOTE *You might see the following error message:*

```
ModuleNotFoundError: No module named 'learning_logs.urls'
```

If you do, stop the development server by pressing CTRL-C *in the terminal window where you issued the* `runserver` *command. Then reissue the command* `python manage.py runserver`*. You should be able to see the home page. Any time you run into an error like this, try stopping and restarting the server.*

TRY IT YOURSELF

18-5. Meal Planner: Consider an app that helps people plan their meals throughout the week. Make a new folder called *meal_planner*, and start a new Django project inside this folder. Then make a new app called `meal_plans`. Make a simple home page for this project.

18-6. Pizzeria Home Page: Add a home page to the Pizzeria project you started in Exercise 18-4 (page 394).

Building Additional Pages

Now that we've established a routine for building a page, we can start to build out the Learning Log project. We'll build two pages that display data: a page that lists all topics and a page that shows all the entries for a particular topic. For each page, we'll specify a URL pattern, write a view function, and write a template. But before we do this, we'll create a base template that all templates in the project can inherit from.

Template Inheritance

When building a website, some elements will always need to be repeated on each page. Rather than writing these elements directly into each page, you can write a base template containing the repeated elements and then have each page inherit from the base. This approach lets you focus on developing the unique aspects of each page and makes it much easier to change the overall look and feel of the project.

The Parent Template

We'll create a template called *base.html* in the same directory as *index.html*. This file will contain elements common to all pages; every other template will inherit from *base.html*. The only element we want to repeat on each page right now is the title at the top. Because we'll include this template on every page, let's make the title a link to the home page:

base.html

```
<p>
❶   <a href="{% url 'learning_logs:index' %}">Learning Log</a>
</p>

❷ {% block content %}{% endblock content %}
```

The first part of this file creates a paragraph containing the name of the project, which also acts as a home page link. To generate a link, we use a *template tag*, which is indicated by braces and percent signs `{% %}`. A template tag generates information to be displayed on a page. Our template tag `{% url 'learning_logs:index' %}` generates a URL matching the URL pattern defined in *learning_logs/urls.py* with the name `'index'` ❶. In this example, `learning_logs` is the *namespace* and `index` is a uniquely named URL pattern in that namespace. The namespace comes from the value we assigned to `app_name` in the *learning_logs/urls.py* file.

In a simple HTML page, a link is surrounded by the *anchor* tag `<a>`:

```
<a href="link_url">link text</a>
```

Having the template tag generate the URL for us makes it much easier to keep our links up to date. We only need to change the URL pattern in *urls.py*, and Django will automatically insert the updated URL the next time the page is requested. Every page in our project will inherit from *base.html*, so from now on, every page will have a link back to the home page.

At ❷ we insert a pair of `block` tags. This block, named `content`, is a placeholder; the child template will define the kind of information that goes in the `content` block.

A child template doesn't have to define every block from its parent, so you can reserve space in parent templates for as many blocks as you like; the child template uses only as many as it requires.

NOTE *In Python code, we almost always use four spaces when we indent. Template files tend to have more levels of nesting than Python files, so it's common to use only two spaces for each indentation level. You just need to ensure that you're consistent.*

The Child Template

Now we need to rewrite *index.html* to inherit from *base.html*. Add the following code to *index.html*:

index.html

```
❶ {% extends "learning_logs/base.html" %}

❷ {% block content %}
```

```
    <p>Learning Log helps you keep track of your learning, for any topic you're
    learning about.</p>
❸ {% endblock content %}
```

If you compare this to the original *index.html*, you can see that we've replaced the Learning Log title with the code for inheriting from a parent template ❶. A child template must have an `{% extends %}` tag on the first line to tell Django which parent template to inherit from. The file *base.html* is part of `learning_logs`, so we include *learning_logs* in the path to the parent template. This line pulls in everything contained in the *base.html* template and allows *index.html* to define what goes in the space reserved by the `content` block.

We define the content block at ❷ by inserting a `{% block %}` tag with the name `content`. Everything that we aren't inheriting from the parent template goes inside the `content` block. Here, that's the paragraph describing the Learning Log project. At ❸ we indicate that we're finished defining the content by using an `{% endblock content %}` tag. The `{% endblock %}` tag doesn't require a name, but if a template grows to contain multiple blocks, it can be helpful to know exactly which block is ending.

You can start to see the benefit of template inheritance: in a child template, we only need to include content that's unique to that page. This not only simplifies each template, but also makes it much easier to modify the site. To modify an element common to many pages, you only need to modify the parent template. Your changes are then carried over to every page that inherits from that template. In a project that includes tens or hundreds of pages, this structure can make it much easier and faster to improve your site.

NOTE *In a large project, it's common to have one parent template called* base.html *for the entire site and parent templates for each major section of the site. All the section templates inherit from* base.html*, and each page in the site inherits from a section template. This way you can easily modify the look and feel of the site as a whole, any section in the site, or any individual page. This configuration provides a very efficient way to work, and it encourages you to steadily update your site over time.*

The Topics Page

Now that we have an efficient approach to building pages, we can focus on our next two pages: the general topics page and the page to display entries for a single topic. The topics page will show all topics that users have created, and it's the first page that will involve working with data.

The Topics URL Pattern

First, we define the URL for the topics page. It's common to choose a simple URL fragment that reflects the kind of information presented on the page.

We'll use the word *topics*, so the URL *http://localhost:8000/topics/* will return this page. Here's how we modify *learning_logs/urls.py*:

urls.py

```
"""Defines URL patterns for learning_logs."""
--snip--
urlpatterns = [
    # Home page.
    path('', views.index, name='index'),
    # Page that shows all topics.
❶   path('topics/', views.topics, name='topics'),
]
```

We've simply added `topics/` into the string argument used for the home page URL ❶. When Django examines a requested URL, this pattern will match any URL that has the base URL followed by *topics*. You can include or omit a forward slash at the end, but there can't be anything else after the word *topics*, or the pattern won't match. Any request with a URL that matches this pattern will then be passed to the function `topics()` in *views.py*.

The Topics View

The `topics()` function needs to retrieve some data from the database and send it to the template. Here's what we need to add to *views.py*:

views.py

```
from django.shortcuts import render

❶ from .models import Topic

def index(request):
    --snip--

❷ def topics(request):
    """Show all topics."""
❸   topics = Topic.objects.order_by('date_added')
❹   context = {'topics': topics}
❺   return render(request, 'learning_logs/topics.html', context)
```

We first import the model associated with the data we need ❶. The `topics()` function needs one parameter: the `request` object Django received from the server ❷. At ❸ we query the database by asking for the `Topic` objects, sorted by the `date_added` attribute. We store the resulting queryset in `topics`.

At ❹ we define a context that we'll send to the template. A *context* is a dictionary in which the keys are names we'll use in the template to access the data, and the values are the data we need to send to the template. In this case, there's one key-value pair, which contains the set of topics we'll display on the page. When building a page that uses data, we pass the `context` variable to `render()` as well as the `request` object and the path to the template ❺.

The Topics Template

The template for the topics page receives the `context` dictionary, so the template can use the data that `topics()` provides. Make a file called *topics.html* in the same directory as *index.html*. Here's how we can display the topics in the template:

topics.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <p>Topics</p>

❶ <ul>
❷   {% for topic in topics %}
❸     <li>{{ topic }}</li>
❹   {% empty %}
      <li>No topics have been added yet.</li>
❺   {% endfor %}
❻ </ul>

{% endblock content %}
```

We use the `{% extends %}` tag to inherit from *base.html*, just as the index template does, and then open a `content` block. The body of this page contains a bulleted list of the topics that have been entered. In standard HTML, a bulleted list is called an *unordered list* and is indicated by the tags `<ul></ul>`. We begin the bulleted list of topics at ❶.

At ❷ we have another template tag equivalent to a `for` loop, which loops through the list `topics` from the `context` dictionary. The code used in templates differs from Python in some important ways. Python uses indentation to indicate which lines of a `for` statement are part of a loop. In a template, every `for` loop needs an explicit `{% endfor %}` tag indicating where the end of the loop occurs. So in a template, you'll see loops written like this:

```
{% for item in list %}
    do something with each item
{% endfor %}
```

Inside the loop, we want to turn each topic into an item in the bulleted list. To print a variable in a template, wrap the variable name in double braces. The braces won't appear on the page; they just indicate to Django that we're using a template variable. So the code `{{ topic }}` at ❸ will be replaced by the value of `topic` on each pass through the loop. The HTML tag `<li></li>` indicates a *list item*. Anything between these tags, inside a pair of `<ul></ul>` tags, will appear as a bulleted item in the list.

At ❹ we use the `{% empty %}` template tag, which tells Django what to do if there are no items in the list. In this case, we print a message informing the user that no topics have been added yet. The last two lines close out the `for` loop ❺ and then close out the bulleted list ❻.

Now we need to modify the base template to include a link to the topics page. Add the following code to *base.html*:

base.html

```
<p>
❶   <a href="{% url 'learning_logs:index' %}">Learning Log</a> -
❷   <a href="{% url 'learning_logs:topics' %}">Topics</a>
</p>

{% block content %}{% endblock content %}
```

We add a dash after the link to the home page ❶, and then add a link to the topics page using the `{% url %}` template tag again ❷. This line tells Django to generate a link matching the URL pattern with the name `'topics'` in *learning_logs/urls.py*.

Now when you refresh the home page in your browser, you'll see a *Topics* link. When you click the link, you'll see a page that looks similar to Figure 18-4.

Figure 18-4: The topics page

Individual Topic Pages

Next, we need to create a page that can focus on a single topic, showing the topic name and all the entries for that topic. We'll again define a new URL pattern, write a view, and create a template. We'll also modify the topics page so each item in the bulleted list links to its corresponding topic page.

The Topic URL Pattern

The URL pattern for the topic page is a little different than the prior URL patterns because it will use the topic's `id` attribute to indicate which topic was requested. For example, if the user wants to see the detail page for the Chess topic, where the `id` is 1, the URL will be *http://localhost:8000/topics/1/*.

Here's a pattern to match this URL, which you should place in *learning_logs/urls.py*:

urls.py

```
--snip--
urlpatterns = [
    --snip--
    # Detail page for a single topic.
    path('topics/<int:topic_id>/', views.topic, name='topic'),
]
```

Let's examine the string `'topics/<int:topic_id>/'` in this URL pattern. The first part of the string tells Django to look for URLs that have the word *topics* after the base URL. The second part of the string, `/<int:topic_id>/`, matches an integer between two forward slashes and stores the integer value in an argument called `topic_id`.

When Django finds a URL that matches this pattern, it calls the view function `topic()` with the value stored in `topic_id` as an argument. We'll use the value of `topic_id` to get the correct topic inside the function.

The Topic View

The `topic()` function needs to get the topic and all associated entries from the database, as shown here:

views.py

```
--snip--
❶ def topic(request, topic_id):
    """Show a single topic and all its entries."""
❷     topic = Topic.objects.get(id=topic_id)
❸     entries = topic.entry_set.order_by('-date_added')
❹     context = {'topic': topic, 'entries': entries}
❺     return render(request, 'learning_logs/topic.html', context)
```

This is the first view function that requires a parameter other than the `request` object. The function accepts the value captured by the expression `/<int:topic_id>/` and stores it in `topic_id` ❶. At ❷ we use `get()` to retrieve the topic, just as we did in the Django shell. At ❸ we get the entries associated with this topic, and we order them according to `date_added`. The minus sign in front of `date_added` sorts the results in reverse order, which will display the most recent entries first. We store the topic and entries in the context dictionary ❹ and send `context` to the template *topic.html* ❺.

NOTE *The code phrases at ❷ and ❸ are called* queries, *because they query the database for specific information. When you're writing queries like these in your own projects, it's helpful to try them out in the Django shell first. You'll get much quicker feedback in the shell than you will by writing a view and template, and then checking the results in a browser.*

The Topic Template

The template needs to display the name of the topic and the entries. We also need to inform the user if no entries have been made yet for this topic.

topic.html

```
{% extends 'learning_logs/base.html' %}

{% block content %}

❶   <p>Topic: {{ topic }}</p>

    <p>Entries:</p>
❷   <ul>
❸   {% for entry in entries %}
      <li>
❹       <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
❺       <p>{{ entry.text|linebreaks }}</p>
      </li>
❻   {% empty %}
      <li>There are no entries for this topic yet.</li>
    {% endfor %}
    </ul>

{% endblock content %}
```

We extend *base.html*, as we do for all pages in the project. Next, we show the topic that's currently being displayed ❶, which is stored in the template variable `{{ topic }}`. The variable `topic` is available because it's included in the `context` dictionary. We then start a bulleted list to show each of the entries ❷ and loop through them as we did the topics earlier ❸.

Each bullet lists two pieces of information: the timestamp and the full text of each entry. For the timestamp ❹, we display the value of the attribute `date_added`. In Django templates, a vertical line (`|`) represents a template *filter*—a function that modifies the value in a template variable. The filter `date:'M d, Y H:i'` displays timestamps in the format *January 1, 2018 23:00*. The next line displays the full value of `text` rather than just the first 50 characters from `entry`. The filter `linebreaks` ❺ ensures that long text entries include line breaks in a format understood by browsers rather than showing a block of uninterrupted text. At ❻ we use the `{% empty %}` template tag to print a message informing the user that no entries have been made.

Links from the Topics Page

Before we look at the topic page in a browser, we need to modify the topics template so each topic links to the appropriate page. Here's the change you need to make to *topics.html*:

topics.html

```
--snip--
    {% for topic in topics %}
      <li>
        <a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a>
      </li>
    {% empty %}
--snip--
```

We use the URL template tag to generate the proper link, based on the URL pattern in `learning_logs` with the name `'topic'`. This URL pattern requires a `topic_id` argument, so we add the attribute `topic.id` to the URL template tag. Now each topic in the list of topics is a link to a topic page, such as *http://localhost:8000/topics/1/*.

When you refresh the topics page and click a topic, you should see a page that looks like Figure 18-5.

NOTE *There's a subtle but important difference between* `topic.id` *and* `topic_id`*. The expression* `topic.id` *examines a topic and retrieves the value of the corresponding ID. The variable* `topic_id` *is a reference to that ID in the code. If you run into errors when working with IDs, make sure you're using these expressions in the appropriate ways.*

Figure 18-5: The detail page for a single topic, showing all entries for a topic

TRY IT YOURSELF

18-7. Template Documentation: Skim the Django template documentation at *https://docs.djangoproject.com/en/2.2/ref/templates/*. You can refer back to it when you're working on your own projects.

18-8. Pizzeria Pages: Add a page to the Pizzeria project from Exercise 18-6 (page 398) that shows the names of available pizzas. Then link each pizza name to a page displaying the pizza's toppings. Make sure you use template inheritance to build your pages efficiently.

Summary

In this chapter, you learned how to build simple web applications using the Django framework. You wrote a brief project specification, installed Django to a virtual environment, set up a project, and checked that the project was set up correctly. You set up an app and defined models to represent the data for your app. You learned about databases and how Django helps you migrate your database after you make a change to your models. You created a superuser for the admin site, and you used the admin site to enter some initial data.

You also explored the Django shell, which allows you to work with your project's data in a terminal session. You learned to define URLs, create view functions, and write templates to make pages for your site. You also used template inheritance to simplify the structure of individual templates and make it easier to modify the site as the project evolves.

In Chapter 19, you'll make intuitive, user-friendly pages that allow users to add new topics and entries and edit existing entries without going through the admin site. You'll also add a user registration system, allowing users to create an account and make their own learning log. This is the heart of a web app—the ability to create something that any number of users can interact with.

19

USER ACCOUNTS

At the heart of a web application is the ability for any user, anywhere in the world, to register an account with your app and start using it. In this chapter, you'll build forms so users can add their own topics and entries, and edit existing entries. You'll also learn how Django guards against common attacks to form-based pages so you don't have to spend much time thinking about securing your apps.

You'll also implement a user authentication system. You'll build a registration page for users to create accounts, and then restrict access to certain pages to logged-in users only. Then you'll modify some of the view functions so users can only see their own data. You'll learn to keep your users' data safe and secure.

Allowing Users to Enter Data

Before we build an authentication system for creating accounts, we'll first add some pages that allow users to enter their own data. We'll give users the ability to add a new topic, add a new entry, and edit their previous entries.

Currently, only a superuser can enter data through the admin site. We don't want users to interact with the admin site, so we'll use Django's form-building tools to build pages that allow users to enter data.

Adding New Topics

Let's start by allowing users to add a new topic. Adding a form-based page works in much the same way as the pages we've already built: we define a URL, write a view function, and write a template. The one major difference is the addition of a new module called *forms.py*, which will contain the forms.

The Topic ModelForm

Any page that lets a user enter and submit information on a web page is a *form*, even if it doesn't look like one. When users enter information, we need to *validate* that the information provided is the right kind of data and is not malicious, such as code to interrupt our server. We then need to process and save valid information to the appropriate place in the database. Django automates much of this work.

The simplest way to build a form in Django is to use a *ModelForm*, which uses the information from the models we defined in Chapter 18 to automatically build a form. Write your first form in the file *forms.py*, which you should create in the same directory as *models.py*:

forms.py

```
from django import forms

from .models import Topic

❶ class TopicForm(forms.ModelForm):
      class Meta:
❷         model = Topic
❸         fields = ['text']
❹         labels = {'text': ''}
```

We first import the `forms` module and the model we'll work with, called `Topic`. At ❶ we define a class called `TopicForm`, which inherits from `forms.ModelForm`.

The simplest version of a `ModelForm` consists of a nested `Meta` class telling Django which model to base the form on and which fields to include in the form. At ❷ we build a form from the `Topic` model and include only the `text` field ❸. The code at ❹ tells Django not to generate a label for the `text` field.

The new_topic URL

The URL for a new page should be short and descriptive. When the user wants to add a new topic, we'll send them to *http://localhost:8000/new_topic/*. Here's the URL pattern for the `new_topic` page, which you add to *learning_logs/urls.py*:

urls.py

```
--snip--
urlpatterns = [
    --snip--
    # Page for adding a new topic
    path('new_topic/', views.new_topic, name='new_topic'),
]
```

This URL pattern sends requests to the view function `new_topic()`, which we'll write next.

The new_topic() View Function

The `new_topic()` function needs to handle two different situations: initial requests for the `new_topic` page (in which case it should show a blank form) and the processing of any data submitted in the form. After data from a submitted form is processed, it needs to redirect the user back to the `topics` page:

views.py

```
from django.shortcuts import render, redirect

from .models import Topic
from .forms import TopicForm

--snip--
def new_topic(request):
    """Add a new topic."""
❶   if request.method != 'POST':
        # No data submitted; create a blank form.
❷       form = TopicForm()
    else:
        # POST data submitted; process data.
❸       form = TopicForm(data=request.POST)
❹       if form.is_valid():
❺           form.save()
❻           return redirect('learning_logs:topics')

    # Display a blank or invalid form.
❼   context = {'form': form}
    return render(request, 'learning_logs/new_topic.html', context)
```

We import the function `redirect`, which we'll use to redirect the user back to the `topics` page after they submit their topic. The `redirect()` function takes in the name of a view and redirects the user to that view. We also import the form we just wrote, `TopicForm`.

GET and POST Requests

The two main types of request you'll use when building web apps are GET requests and POST requests. You use *GET* requests for pages that only read data from the server. You usually use *POST* requests when the user needs to submit information through a form. We'll be specifying the POST method for processing all of our forms. (A few other kinds of requests exist, but we won't use them in this project.)

The `new_topic()` function takes in the request object as a parameter. When the user initially requests this page, their browser will send a GET request. Once the user has filled out and submitted the form, their browser will submit a POST request. Depending on the request, we'll know whether the user is requesting a blank form (a GET request) or asking us to process a completed form (a POST request).

The test at ❶ determines whether the request method is GET or POST. If the request method isn't POST, the request is probably GET, so we need to return a blank form (if it's another kind of request, it's still safe to return a blank form). We make an instance of `TopicForm` ❷, assign it to the variable `form`, and send the form to the template in the context dictionary ❼. Because we included no arguments when instantiating `TopicForm`, Django creates a blank form that the user can fill out.

If the request method is POST, the `else` block runs and processes the data submitted in the form. We make an instance of `TopicForm` ❸ and pass it the data entered by the user, stored in `request.POST`. The `form` object that's returned contains the information submitted by the user.

We can't save the submitted information in the database until we've checked that it's valid ❹. The `is_valid()` method checks that all required fields have been filled in (all fields in a form are required by default) and that the data entered matches the field types expected—for example, that the length of `text` is less than 200 characters, as we specified in *models.py* in Chapter 18. This automatic validation saves us a lot of work. If everything is valid, we can call `save()` ❺, which writes the data from the form to the database.

Once we've saved the data, we can leave this page. We use `redirect()` to redirect the user's browser to the `topics` page, where the user should see the topic they just entered in the list of topics.

The `context` variable is defined at the end of the view function, and the page is rendered using the template *new_topic.html*, which we'll create next. This code is placed outside of any `if` block; it will run if a blank form was created, and it will run if a submitted form is determined to be invalid. An invalid form will include some default error messages to help the user submit acceptable data.

The new_topic Template

Now we'll make a new template called *new_topic.html* to display the form we just created.

new_topic.html

```
{% extends "learning_logs/base.html" %}

{% block content %}
  <p>Add a new topic:</p>

❶ <form action="{% url 'learning_logs:new_topic' %}" method='post'>
❷   {% csrf_token %}
❸   {{ form.as_p }}
❹   <button name="submit">Add topic</button>
  </form>

{% endblock content %}
```

This template extends *base.html*, so it has the same base structure as the rest of the pages in Learning Log. At ❶ we define an HTML form. The `action` argument tells the browser where to send the data submitted in the form; in this case, we send it back to the view function `new_topic()`. The `method` argument tells the browser to submit the data as a POST request.

Django uses the template tag `{% csrf_token %}` ❷ to prevent attackers from using the form to gain unauthorized access to the server (this kind of attack is called a *cross-site request forgery*). At ❸ we display the form; here you see how simple Django can make certain tasks, such as displaying a form. We only need to include the template variable `{{ form.as_p }}` for Django to create all the fields necessary to display the form automatically. The `as_p` modifier tells Django to render all the form elements in paragraph format, as a simple way to display the form neatly.

Django doesn't create a submit button for forms, so we define one at ❹.

Linking to the new_topic Page

Next, we include a link to the `new_topic` page on the `topics` page:

topics.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <p>Topics</p>

  <ul>
    --snip--
  </ul>

  <a href="{% url 'learning_logs:new_topic' %}">Add a new topic</a>

{% endblock content %}
```

Place the link after the list of existing topics. Figure 19-1 shows the resulting form. Use the form to add a few new topics of your own.

Figure 19-1: The page for adding a new topic

Adding New Entries

Now that the user can add a new topic, they'll want to add new entries too. We'll again define a URL, write a view function and a template, and link to the page. But first, we'll add another class to *forms.py*.

The Entry ModelForm

We need to create a form associated with the `Entry` model but this time with a bit more customization than `TopicForm`:

forms.py

```
from django import forms

from .models import Topic, Entry

class TopicForm(forms.ModelForm):
    --snip--

class EntryForm(forms.ModelForm):
    class Meta:
        model = Entry
        fields = ['text']
❶       labels = {'text': 'Entry:'}
❷       widgets = {'text': forms.Textarea(attrs={'cols': 80})}
```

We update the `import` statement to include `Entry` as well as `Topic`. We make a new class called `EntryForm` that inherits from `forms.ModelForm`. The `EntryForm` class has a nested `Meta` class listing the model it's based on and the field to include in the form. We again give the field `'text'` a blank label ❶.

At ❷ we include the `widgets` attribute. A *widget* is an HTML form element, such as a single-line text box, multi-line text area, or drop-down list. By including the `widgets` attribute, you can override Django's default widget choices. By telling Django to use a `forms.Textarea` element, we're customizing

the input widget for the field `'text'` so the text area will be 80 columns wide instead of the default 40. This gives users enough room to write a meaningful entry.

The new_entry URL

New entries must be associated with a particular topic, so we need to include a `topic_id` argument in the URL for adding a new entry. Here's the URL, which you add to *learning_logs/urls.py*:

urls.py

```
--snip--
urlpatterns = [
    --snip--
    # Page for adding a new entry
    path('new_entry/<int:topic_id>/', views.new_entry, name='new_entry'),
]
```

This URL pattern matches any URL with the form `http://localhost:8000/new_entry/id/`, where *id* is a number matching the topic ID. The code `<int:topic_id>` captures a numerical value and assigns it to the variable `topic_id`. When a URL matching this pattern is requested, Django sends the request and the topic's ID to the `new_entry()` view function.

The new_entry() View Function

The view function for `new_entry` is much like the function for adding a new topic. Add the following code to your *views.py* file:

views.py

```
from django.shortcuts import render, redirect

from .models import Topic
from .forms import TopicForm, EntryForm

--snip--
def new_entry(request, topic_id):
    """Add a new entry for a particular topic."""
❶   topic = Topic.objects.get(id=topic_id)

❷   if request.method != 'POST':
        # No data submitted; create a blank form.
❸       form = EntryForm()
    else:
        # POST data submitted; process data.
❹       form = EntryForm(data=request.POST)
        if form.is_valid():
❺           new_entry = form.save(commit=False)
❻           new_entry.topic = topic
            new_entry.save()
❼           return redirect('learning_logs:topic', topic_id=topic_id)

    # Display a blank or invalid form.
    context = {'topic': topic, 'form': form}
    return render(request, 'learning_logs/new_entry.html', context)
```

We update the import statement to include the EntryForm we just made. The definition of new_entry() has a topic_id parameter to store the value it receives from the URL. We'll need the topic to render the page and process the form's data, so we use topic_id to get the correct topic object at ❶.

At ❷ we check whether the request method is POST or GET. The if block executes if it's a GET request, and we create a blank instance of EntryForm ❸.

If the request method is POST, we process the data by making an instance of EntryForm, populated with the POST data from the request object ❹. We then check whether the form is valid. If it is, we need to set the entry object's topic attribute before saving it to the database. When we call save(), we include the argument commit=False ❺ to tell Django to create a new entry object and assign it to new_entry without saving it to the database yet. We set the topic attribute of new_entry to the topic we pulled from the database at the beginning of the function ❻. Then we call save() with no arguments, saving the entry to the database with the correct associated topic.

The redirect() call at ❼ requires two arguments—the name of the view we want to redirect to and the argument that view function requires. Here, we're redirecting to topic(), which needs the argument topic_id. This view then renders the topic page that the user made an entry for, and they should see their new entry in the list of entries.

At the end of the function, we create a context dictionary and render the page using the *new_entry.html* template. This code will execute for a blank form or for a submitted form that is evaluated as invalid.

The new_entry Template

As you can see in the following code, the template for new_entry is similar to the template for new_topic:

new_entry.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

❶   <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

    <p>Add a new entry:</p>
❷   <form action="{% url 'learning_logs:new_entry' topic.id %}" method='post'>
      {% csrf_token %}
      {{ form.as_p }}
      <button name='submit'>Add entry</button>
    </form>

{% endblock content %}
```

We show the topic at the top of the page ❶, so the user can see which topic they're adding an entry to. The topic also acts as a link back to the main page for that topic.

The form's `action` argument includes the `topic_id` value in the URL, so the view function can associate the new entry with the correct topic ❷. Other than that, this template looks just like *new_topic.html*.

Linking to the new_entry Page

Next, we need to include a link to the `new_entry` page from each topic page in the topic template:

topic.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <p>Topic: {{ topic }}</p>

  <p>Entries:</p>
  <p>
    <a href="{% url 'learning_logs:new_entry' topic.id %}">Add new entry</a>
  </p>

  <ul>
  --snip--
  </ul>

{% endblock content %}
```

We place the link to add entries just before showing the entries, because adding a new entry will be the most common action on this page. Figure 19-2 shows the `new_entry` page. Now users can add new topics and as many entries as they want for each topic. Try out the `new_entry` page by adding a few entries to some of the topics you've created.

Figure 19-2: The `new_entry` page

Editing Entries

Now we'll make a page so users can edit the entries they've added.

The edit_entry URL

The URL for the page needs to pass the ID of the entry to be edited. Here's *learning_logs/urls.py*:

urls.py

```
--snip--
urlpatterns = [
    --snip--
    # Page for editing an entry.
    path('edit_entry/<int:entry_id>/', views.edit_entry, name='edit_entry'),
]
```

The ID passed in the URL (for example, *http://localhost:8000/edit_entry/1/)* is stored in the parameter `entry_id`. The URL pattern sends requests that match this format to the view function `edit_entry()`.

The edit_entry() View Function

When the `edit_entry` page receives a GET request, the `edit_entry()` function returns a form for editing the entry. When the page receives a POST request with revised entry text, it saves the modified text into the database:

views.py

```
from django.shortcuts import render, redirect

from .models import Topic, Entry
from .forms import TopicForm, EntryForm
--snip--

def edit_entry(request, entry_id):
    """Edit an existing entry."""
❶   entry = Entry.objects.get(id=entry_id)
    topic = entry.topic

    if request.method != 'POST':
        # Initial request; pre-fill form with the current entry.
❷       form = EntryForm(instance=entry)
    else:
        # POST data submitted; process data.
❸       form = EntryForm(instance=entry, data=request.POST)
        if form.is_valid():
❹           form.save()
❺           return redirect('learning_logs:topic', topic_id=topic.id)

    context = {'entry': entry, 'topic': topic, 'form': form}
    return render(request, 'learning_logs/edit_entry.html', context)
```

We first import the `Entry` model. At ❶ we get the entry object that the user wants to edit and the topic associated with this entry. In the `if` block, which runs for a GET request, we make an instance of `EntryForm` with the argument `instance=entry` ❷. This argument tells Django to create the form prefilled with information from the existing entry object. The user will see their existing data and be able to edit that data.

When processing a POST request, we pass the `instance=entry` argument and the `data=request.POST` argument ❸. These arguments tell Django to create a form instance based on the information associated with the existing entry object, updated with any relevant data from `request.POST`. We then check whether the form is valid; if it is, we call `save()` with no arguments because the entry is already associated with the correct topic ❹. We then redirect to the `topic` page, where the user should see the updated version of the entry they edited ❺.

If we're showing an initial form for editing the entry or if the submitted form is invalid, we create the context dictionary and render the page using the *edit_entry.html* template.

The edit_entry Template

Next, we create an *edit_entry.html* template, which is similar to *new_entry.html*:

edit_entry.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

  <p>Edit entry:</p>

❶ <form action="{% url 'learning_logs:edit_entry' entry.id %}" method='post'>
    {% csrf_token %}
    {{ form.as_p }}
❷   <button name="submit">Save changes</button>
  </form>

{% endblock content %}
```

At ❶ the `action` argument sends the form back to the `edit_entry()` function for processing. We include the entry ID as an argument in the `{% url %}` tag, so the view function can modify the correct entry object. We label the submit button as *Save changes* to remind the user they're saving edits, not creating a new entry ❷.

Linking to the edit_entry Page

Now we need to include a link to the `edit_entry` page for each entry on the topic page:

topic.html

```
--snip--
  {% for entry in entries %}
    <li>
      <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
      <p>{{ entry.text|linebreaks }}</p>
      <p>
        <a href="{% url 'learning_logs:edit_entry' entry.id %}">Edit entry</a>
      </p>
    </li>
--snip--
```

We include the edit link after each entry's date and text has been displayed. We use the `{% url %}` template tag to determine the URL for the named URL pattern `edit_entry`, along with the ID attribute of the current entry in the loop (`entry.id`). The link text *Edit entry* appears after each entry on the page. Figure 19-3 shows what the topic page looks like with these links.

Figure 19-3: Each entry now has a link for editing that entry.

Learning Log now has most of the functionality it needs. Users can add topics and entries, and read through any set of entries they want. In the next section, we'll implement a user registration system so anyone can make an account with Learning Log and create their own set of topics and entries.

TRY IT YOURSELF

19-1. Blog: Start a new Django project called *Blog*. Create an app called *blogs* in the project and a model called `BlogPost`. The model should have fields like `title`, `text`, and `date_added`. Create a superuser for the project, and use the admin site to make a couple of short posts. Make a home page that shows all posts in chronological order.

Create a form for making new posts and another for editing existing posts. Fill in your forms to make sure they work.

Setting Up User Accounts

In this section, we'll set up a user registration and authorization system so people can register an account and log in and out. We'll create a new app to contain all the functionality related to working with users. We'll use the default user authentication system included with Django to do as much of the work as possible. We'll also modify the `Topic` model slightly so every topic belongs to a certain user.

The users App

We'll start by creating a new app called `users`, using the `startapp` command:

```
(ll_env)learning_log$ python manage.py startapp users
(ll_env)learning_log$ ls
❶ db.sqlite3  learning_log  learning_logs  ll_env  manage.py  users
(ll_env)learning_log$ ls users
❷ __init__.py admin.py apps.py migrations models.py tests.py views.py
```

This command makes a new directory called *users* ❶ with a structure identical to the `learning_logs` app ❷.

Adding users to settings.py

We need to add our new app to `INSTALLED_APPS` in *settings.py*, like so:

settings.py

```
--snip--
INSTALLED_APPS = [
    # My apps
    'learning_logs',
    'users',

    # Default django apps.
    --snip--
]
--snip--
```

Now Django will include the users app in the overall project.

Including the URLs from users

Next, we need to modify the root *urls.py* so it includes the URLs we'll write for the users app:

urls.py

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('users/', include('users.urls')),
    path('', include('learning_logs.urls')),
]
```

We add a line to include the file *urls.py* from users. This line will match any URL that starts with the word *users*, such as *http://localhost:8000/users/login/*.

The Login Page

We'll first implement a login page. We'll use the default login view Django provides, so the URL pattern for this app looks a little different. Make a new *urls.py* file in the directory *learning_log/users/*, and add the following to it:

urls.py

```
"""Defines URL patterns for users"""

from django.urls import path, include

❶ app_name = 'users'
urlpatterns = [
    # Include default auth urls.
❷   path('', include('django.contrib.auth.urls')),
]
```

We import the path function, and then import the include function so we can include some default authentication URLs that Django has defined. These default URLs include named URL patterns, such as 'login' and 'logout'. We set the variable app_name to 'users' so Django can distinguish these URLs from URLs belonging to other apps ❶. Even default URLs provided by Django, when included in the users app's *urls.py* file, will be accessible through the users namespace.

The login page's pattern matches the URL *http://localhost:8000/users/login/* ❷. When Django reads this URL, the word *users* tells Django to look in *users/urls.py*, and *login* tells it to send requests to Django's default login view.

The login Template

When the user requests the login page, Django will use a default view function, but we still need to provide a template for the page. The default

authentication views look for templates inside a folder called *registration*, so we'll need to make that folder. Inside the *learning_log/users/* directory, make a directory called *templates*; inside that, make another directory called *registration*. Here's the *login.html* template, which you should save in *learning_log/users/templates/registration*:

login.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

❶  {% if form.errors %}
     <p>Your username and password didn't match. Please try again.</p>
   {% endif %}

❷  <form method="post" action="{% url 'users:login' %}">
     {% csrf_token %}
❸    {{ form.as_p }}

❹    <button name="submit">Log in</button>
❺    <input type="hidden" name="next"
       value="{% url 'learning_logs:index' %}" />
   </form>

{% endblock content %}
```

This template extends *base.html* to ensure that the login page will have the same look and feel as the rest of the site. Note that a template in one app can inherit from a template in another app.

If the form's `errors` attribute is set, we display an error message ❶, reporting that the username and password combination don't match anything stored in the database.

We want the login view to process the form, so we set the `action` argument as the URL of the login page ❷. The login view sends a form to the template, and it's up to us to display the form ❸ and add a submit button ❹. At ❺ we include a hidden form element, `'next'`; the `value` argument tells Django where to redirect the user after they've logged in successfully. In this case, we send the user back to the home page.

Linking to the Login Page

Let's add the login link to *base.html* so it appears on every page. We don't want the link to display when the user is already logged in, so we nest it inside an `{% if %}` tag:

base.html

```
<p>
  <a href="{% url 'learning_logs:index' %}">Learning Log</a> -
  <a href="{% url 'learning_logs:topics' %}">Topics</a> -
❶ {% if user.is_authenticated %}
❷   Hello, {{ user.username }}.
  {% else %}
❸   <a href="{% url 'users:login' %}">Log in</a>
  {% endif %}
```

```
</p>

{% block content %}{% endblock content %}
```

In Django's authentication system, every template has a `user` variable available, which always has an `is_authenticated` attribute set: the attribute is `True` if the user is logged in and `False` if they aren't. This attribute allows you to display one message to authenticated users and another to unauthenticated users.

Here we display a greeting to users currently logged in ❶. Authenticated users have an additional `username` attribute set, which we use to personalize the greeting and remind the user they're logged in ❷. At ❸ we display a link to the login page for users who haven't been authenticated.

Using the Login Page

We've already set up a user account, so let's log in to see if the page works. Go to *http://localhost:8000/admin/*. If you're still logged in as an admin, look for a logout link in the header and click it.

When you're logged out, go to *http://localhost:8000/users/login/*. You should see a login page similar to the one shown in Figure 19-4. Enter the username and password you set up earlier, and you should be brought back to the index page. The header on the home page should display a greeting personalized with your username.

Figure 19-4: The login page

Logging Out

Now we need to provide a way for users to log out. We'll put a link in *base.html* that logs out users; when they click this link, they'll go to a page confirming that they've been logged out.

Adding a Logout Link to base.html

We'll add the link for logging out to *base.html* so it's available on every page. We'll include it in the `{% if user.is_authenticated %}` portion so only users who are already logged in can see it:

base.html

```
--snip–
  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
    <a href="{% url 'users:logout' %}">Log out</a>
  {% else %}
    --snip--
```

The default named URL pattern for logging out is simply `'logout'`.

The Logout Confirmation Page

Users will want to know that they've successfully logged out, so the default logout view renders the page using the template *logged_out.html*, which we'll create now. Here's a simple page confirming that the user has been logged out. Save this file in *templates/registration*, the same place where you saved *login.html*:

logged_out.html

```
{% extends "learning_logs/base.html" %}

{% block content %}
  <p>You have been logged out. Thank you for visiting!</p>
{% endblock content %}
```

We don't need anything else on this page, because *base.html* provides links back to the home page and the login page if the user wants to go back to either page.

Figure 19-5 shows the logged out page as it appears to a user who has just clicked the *Log out* link. The styling is minimal because we're focusing on building a site that works properly. When the required set of features works, we'll style the site to look more professional.

Figure 19-5: The logged out page confirms that a user has successfully logged out.

The Registration Page

Next, we'll build a page so new users can register. We'll use Django's default `UserCreationForm` but write our own view function and template.

The register URL

The following code provides the URL pattern for the registration page, again in *users/urls.py*:

urls.py

```
"""Defines URL patterns for users"""

from django.urls import path, include

from . import views

app_name = 'users'
urlpatterns = [
    # Include default auth urls.
    path('', include('django.contrib.auth.urls')),
    # Registration page.
    path('register/', views.register, name='register'),
]
```

We import the `views` module from `users`, which we need because we're writing our own view for the registration page. The pattern for the registration page matches the URL *http://localhost:8000/users/register/* and sends requests to the `register()` function we're about to write.

The register() View Function

The `register()` view function needs to display a blank registration form when the registration page is first requested and then process completed registration forms when they're submitted. When a registration is successful, the function also needs to log in the new user. Add the following code to *users/views.py*:

views.py

```
from django.shortcuts import render, redirect
from django.contrib.auth import login
from django.contrib.auth.forms import UserCreationForm

def register(request):
    """Register a new user."""
    if request.method != 'POST':
        # Display blank registration form.
❶       form = UserCreationForm()
    else:
        # Process completed form.
❷       form = UserCreationForm(data=request.POST)

❸       if form.is_valid():
❹           new_user = form.save()
            # Log the user in and then redirect to home page.
```

```
❺            login(request, new_user)
❻            return redirect('learning_logs:index')

    # Display a blank or invalid form.
    context = {'form': form}
    return render(request, 'registration/register.html', context)
```

We import the `render()` and `redirect()` functions. Then we import the `login()` function to log the user in if their registration information is correct. We also import the default `UserCreationForm`. In the `register()` function, we check whether or not we're responding to a POST request. If we're not, we make an instance of `UserCreationForm` with no initial data ❶.

If we're responding to a POST request, we make an instance of `UserCreationForm` based on the submitted data ❷. We check that the data is valid ❸—in this case, that the username has the appropriate characters, the passwords match, and the user isn't trying to do anything malicious in their submission.

If the submitted data is valid, we call the form's `save()` method to save the username and the hash of the password to the database ❹. The `save()` method returns the newly created user object, which we assign to `new_user`. When the user's information is saved, we log them in by calling the `login()` function with the `request` and `new_user` objects ❺, which creates a valid session for the new user. Finally, we redirect the user to the home page ❻, where a personalized greeting in the header tells them their registration was successful.

At the end of the function we render the page, which will either be a blank form or a submitted form that is invalid.

The register Template

Now create a template for the registration page, which will be similar to the login page. Be sure to save it in the same directory as *login.html*:

register.html

```
{% extends "learning_logs/base.html" %}

{% block content %}

  <form method="post" action="{% url 'users:register' %}">
    {% csrf_token %}
    {{ form.as_p }}

    <button name="submit">Register</button>
    <input type="hidden" name="next" value="{% url 'learning_logs:index' %}" />
  </form>

{% endblock content %}
```

We use the `as_p` method again so Django will display all the fields in the form appropriately, including any error messages if the form isn't filled out correctly.

Linking to the Registration Page

Next, we'll add the code to show the registration page link to any user who isn't currently logged in:

base.html

```
--snip--
  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
    <a href="{% url 'users:logout' %}">Log out</a>
  {% else %}
    <a href="{% url 'users:register' %}">Register</a> -
    <a href="{% url 'users:login' %}">Log in</a>
  {% endif %}
--snip--
```

Now users who are logged in see a personalized greeting and a logout link. Users who aren't logged in see a registration page link and a login link. Try out the registration page by making several user accounts with different usernames.

In the next section, we'll restrict some of the pages so they're available only to registered users, and we'll make sure every topic belongs to a specific user.

NOTE *The registration system we've set up allows anyone to make any number of accounts for Learning Log. But some systems require users to confirm their identity by sending a confirmation email the user must reply to. By doing so, the system generates fewer spam accounts than the simple system we're using here. However, when you're learning to build apps, it's perfectly appropriate to practice with a simple user registration system like the one we're using.*

TRY IT YOURSELF

19-2. Blog Accounts: Add a user authentication and registration system to the Blog project you started in Exercise 19-1 (page 421). Make sure logged-in users see their username somewhere on the screen and unregistered users see a link to the registration page.

Allowing Users to Own Their Data

Users should be able to enter data exclusive to them, so we'll create a system to figure out which data belongs to which user. Then we'll restrict access to certain pages so users can work with only their own data.

We'll modify the `Topic` model so every topic belongs to a specific user. This will also take care of entries, because every entry belongs to a specific topic. We'll start by restricting access to certain pages.

Restricting Access with @login_required

Django makes it easy to restrict access to certain pages to logged-in users through the `@login_required` decorator. A *decorator* is a directive placed just before a function definition that Python applies to the function before it runs, to alter how the function code behaves. Let's look at an example.

Restricting Access to the Topics Page

Each topic will be owned by a user, so only registered users can request the topics page. Add the following code to *learning_logs/views.py*:

views.py

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required

from .models import Topic, Entry
--snip--

@login_required
def topics(request):
    """Show all topics."""
    --snip--
```

We first import the `login_required()` function. We apply `login_required()` as a decorator to the `topics()` view function by prepending `login_required` with the `@` symbol. As a result, Python knows to run the code in `login_required()` before the code in `topics()`.

The code in `login_required()` checks whether a user is logged in, and Django runs the code in `topics()` only if they are. If the user isn't logged in, they're redirected to the login page.

To make this redirect work, we need to modify *settings.py* so Django knows where to find the login page. Add the following at the very end of *settings.py*:

settings.py

```
--snip--

# My settings
LOGIN_URL = 'users:login'
```

Now when an unauthenticated user requests a page protected by the `@login_required` decorator, Django will send the user to the URL defined by `LOGIN_URL` in *settings.py*.

You can test this setting by logging out of any user accounts and going to the home page. Click the **Topics** link, which should redirect you to the login page. Then log in to any of your accounts, and from the home page click the **Topics** link again. You should be able to access the topics page.

Restricting Access Throughout Learning Log

Django makes it easy to restrict access to pages, but you have to decide which pages to protect. It's best to think about which pages need to be unrestricted first, and then restrict all the other pages in the project. You can easily correct overrestricting access, and it's less dangerous than leaving sensitive pages unrestricted.

In Learning Log, we'll keep the home page and the registration page unrestricted. We'll restrict access to every other page.

Here's *learning_logs/views.py* with `@login_required` decorators applied to every view except `index()`:

views.py

```
--snip--
@login_required
def topics(request):
    --snip--

@login_required
def topic(request, topic_id):
    --snip--

@login_required
def new_topic(request):
    --snip--

@login_required
def new_entry(request, topic_id):
    --snip--

@login_required
def edit_entry(request, entry_id):
    --snip--
```

Try accessing each of these pages while logged out: you'll be redirected back to the login page. You'll also be unable to click links to pages such as `new_topic`. But if you enter the URL *http://localhost:8000/new_topic/*, you'll be redirected to the login page. You should restrict access to any URL that's publicly accessible and relates to private user data.

Connecting Data to Certain Users

Next, we need to connect the data to the user who submitted it. We need to connect only the data highest in the hierarchy to a user, and the lower-level data will follow. For example, in Learning Log, topics are the highest level of data in the app, and all entries are connected to a topic. As long as each topic belongs to a specific user, we can trace the ownership of each entry in the database.

We'll modify the `Topic` model by adding a foreign key relationship to a user. We'll then have to migrate the database. Finally, we'll modify some of the views so they only show the data associated with the currently logged in user.

Modifying the Topic Model

The modification to *models.py* is just two lines:

models.py

```
from django.db import models
from django.contrib.auth.models import User

class Topic(models.Model):
   """A topic the user is learning about."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(auto_now_add=True)
    owner = models.ForeignKey(User, on_delete=models.CASCADE)

    def __str__(self):
        """Return a string representation of the model."""
        return self.text

class Entry(models.Model):
    --snip--
```

We import the `User` model from `django.contrib.auth`. Then we add an `owner` field to `Topic`, which establishes a foreign key relationship to the `User` model. If a user is deleted, all the topics associated with that user will be deleted as well.

Identifying Existing Users

When we migrate the database, Django will modify the database so it can store a connection between each topic and a user. To make the migration, Django needs to know which user to associate with each existing topic. The simplest approach is to start by giving all existing topics to one user—for example, the superuser. But first we need to know that user's ID.

Let's look at the IDs of all users created so far. Start a Django shell session and issue the following commands:

```
(ll_env)learning_log$ python manage.py shell
❶ >>> from django.contrib.auth.models import User
❷ >>> User.objects.all()
<QuerySet [<User: ll_admin>, <User: eric>, <User: willie>]>
❸ >>> for user in User.objects.all():
...     print(user.username, user.id)
...
ll_admin 1
eric 2
willie 3
>>>
```

At ❶ we import the `User` model into the shell session. We then look at all the users that have been created so far ❷. The output shows three users: *ll_admin*, *eric*, and *willie*.

At ❸ we loop through the list of users and print each user's username and ID. When Django asks which user to associate the existing topics with, we'll use one of these ID values.

Migrating the Database

Now that we know the IDs, we can migrate the database. When we do this, Python will ask us to connect the `Topic` model to a particular owner temporarily or to add a default to our *models.py* file to tell it what to do. Choose option 1:

```
❶ (ll_env)learning_log$ python manage.py makemigrations learning_logs
❷ You are trying to add a non-nullable field 'owner' to topic without a default;
  we can't do that (the database needs something to populate existing rows).
❸ Please select a fix:
   1) Provide a one-off default now (will be set on all existing rows with a
      null value for this column)
   2) Quit, and let me add a default in models.py
❹ Select an option: 1
❺ Please enter the default value now, as valid Python
  The datetime and django.utils.timezone modules are available, so you can do
  e.g. timezone.now
  Type 'exit' to exit this prompt
❻ >>> 1
  Migrations for 'learning_logs':
    learning_logs/migrations/0003_topic_owner.py
  - Add field owner to topic
  (ll_env)learning_log$
```

We start by issuing the `makemigrations` command ❶. In the output at ❷, Django indicates that we're trying to add a required (`non-nullable`) field to an existing model (`topic`) with no default value specified. Django gives us two options at ❸: we can provide a default right now, or we can quit and add a default value in *models.py*. At ❹ we've chosen the first option. Django then asks us to enter the default value ❺.

To associate all existing topics with the original admin user, *ll_admin*, I entered the user ID of 1 at ❻. You can use the ID of any user you've created; it doesn't have to be a superuser. Django then migrates the database using this value and generates the migration file *0003_topic_owner.py*, which adds the field `owner` to the `Topic` model.

Now we can execute the migration. Enter the following in an active virtual environment:

```
(ll_env)learning_log$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, learning_logs, sessions
Running migrations:
❶   Applying learning_logs.0003_topic_owner... OK
(ll_env)learning_log$
```

Django applies the new migration, and the result is `OK` ❶.

We can verify that the migration worked as expected in the shell session, like this:

```
❶ >>> from learning_logs.models import Topic
❷ >>> for topic in Topic.objects.all():
...     print(topic, topic.owner)
...
Chess ll_admin
Rock Climbing ll_admin
>>>
```

We import `Topic` from `learning_logs.models` ❶, and then loop through all existing topics, printing each topic and the user it belongs to ❷. You can see that each topic now belongs to the user *ll_admin*. (If you get an error when you run this code, try exiting the shell and starting a new shell.)

NOTE *You can simply reset the database instead of migrating, but that will lose all existing data. It's good practice to learn how to migrate a database while maintaining the integrity of users' data. If you do want to start with a fresh database, issue the command* `python manage.py flush` *to rebuild the database structure. You'll have to create a new superuser, and all of your data will be gone.*

Restricting Topics Access to Appropriate Users

Currently, if you're logged in, you'll be able to see all the topics, no matter which user you're logged in as. We'll change that by showing users only the topics that belong to them.

Make the following change to the `topics()` function in *views.py*:

views.py

```
--snip--
@login_required
def topics(request):
    """Show all topics."""
    topics = Topic.objects.filter(owner=request.user).order_by('date_added')
    context = {'topics': topics}
    return render(request, 'learning_logs/topics.html', context)
--snip–
```

When a user is logged in, the request object has a `request.user` attribute set that stores information about the user. The query `Topic.objects.filter(owner=request.user)` tells Django to retrieve only the `Topic` objects from the database whose `owner` attribute matches the current user. Because we're not changing how the topics are displayed, we don't need to change the template for the topics page at all.

To see if this works, log in as the user you connected all existing topics to, and go to the topics page. You should see all the topics. Now log out, and log back in as a different user. The topics page should list no topics.

Protecting a User's Topics

We haven't restricted access to the topic pages yet, so any registered user could try a bunch of URLs, like *http://localhost:8000/topics/1/*, and retrieve topic pages that happen to match.

Try it yourself. While logged in as the user that owns all topics, copy the URL or note the ID in the URL of a topic, and then log out and log back in as a different user. Enter that topic's URL. You should be able to read the entries, even though you're logged in as a different user.

We'll fix this now by performing a check before retrieving the requested entries in the `topic()` view function:

views.py

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required
❶ from django.http import Http404

--snip--
@login_required
def topic(request, topic_id):
    """Show a single topic and all its entries."""
    topic = Topic.objects.get(id=topic_id)
    # Make sure the topic belongs to the current user.
❷   if topic.owner != request.user:
        raise Http404

    entries = topic.entry_set.order_by('-date_added')
    context = {'topic': topic, 'entries': entries}
    return render(request, 'learning_logs/topic.html', context)
--snip--
```

A 404 response is a standard error response that's returned when a requested resource doesn't exist on a server. Here we import the `Http404` exception ❶, which we'll raise if the user requests a topic they shouldn't see. After receiving a topic request, we make sure the topic's user matches the currently logged in user before rendering the page. If the current user doesn't own the requested topic, we raise the `Http404` exception ❷, and Django returns a 404 error page.

Now if you try to view another user's topic entries, you'll see a *Page Not Found* message from Django. In Chapter 20, we'll configure the project so users will see a proper error page.

Protecting the edit_entry Page

The `edit_entry` pages have URLs in the form `http://localhost:8000/edit_entry/entry_id/`, where the *`entry_id`* is a number. Let's protect this page so no one can use the URL to gain access to someone else's entries:

views.py

```
--snip--
@login_required
def edit_entry(request, entry_id):
    """Edit an existing entry."""
    entry = Entry.objects.get(id=entry_id)
```

```
    topic = entry.topic
    if topic.owner != request.user:
        raise Http404

    if request.method != 'POST':
        --snip--
```

We retrieve the entry and the topic associated with this entry. We then check whether the owner of the topic matches the currently logged in user; if they don't match, we raise an `Http404` exception.

Associating New Topics with the Current User

Currently, our page for adding new topics is broken, because it doesn't associate new topics with any particular user. If you try adding a new topic, you'll see the error message `IntegrityError` along with `NOT NULL constraint failed: learning_logs_topic.owner_id`. Django's saying you can't create a new topic without specifying a value for the topic's `owner` field.

There's a straightforward fix for this problem, because we have access to the current user through the `request` object. Add the following code, which associates the new topic with the current user:

views.py

```
--snip--
@login_required
def new_topic(request):
    """Add a new topic."""
    if request.method != 'POST':
        # No data submitted; create a blank form.
        form = TopicForm()
    else:
        # POST data submitted; process data.
        form = TopicForm(data=request.POST)
        if form.is_valid():
❶           new_topic = form.save(commit=False)
❷           new_topic.owner = request.user
❸           new_topic.save()
            return redirect('learning_logs:topics')

    # Display a blank or invalid form.
    context = {'form': form}
    return render(request, 'learning_logs/new_topic.html', context)
 --snip--
```

When we first call `form.save()`, we pass the `commit=False` argument because we need to modify the new topic before saving it to the database ❶. We then set the new topic's `owner` attribute to the current user ❷. Finally, we call `save()` on the topic instance just defined ❸. Now the topic has all the required data and will save successfully.

You should be able to add as many new topics as you want for as many different users as you want. Each user will have access only to their own data, whether they're viewing data, entering new data, or modifying old data.

TRY IT YOURSELF

19-3. Refactoring: There are two places in *views.py* where we make sure the user associated with a topic matches the currently logged in user. Put the code for this check in a function called `check_topic_owner()`, and call this function where appropriate.

19-4. Protecting new_entry: Currently, a user can add a new entry to another user's learning log by entering a URL with the ID of a topic belonging to another user. Prevent this attack by checking that the current user owns the entry's topic before saving the new entry.

19-5. Protected Blog: In your Blog project, make sure each blog post is connected to a particular user. Make sure all posts are publicly accessible but only registered users can add posts and edit existing posts. In the view that allows users to edit their posts, make sure the user is editing their own post before processing the form.

Summary

In this chapter, you learned to use forms to allow users to add new topics and entries, and edit existing entries. You then learned how to implement user accounts. You allowed existing users to log in and out, and used Django's default `UserCreationForm` to let people create new accounts.

After building a simple user authentication and registration system, you restricted access to logged-in users for certain pages using the `@login_required` decorator. You then attributed data to specific users through a foreign key relationship. You also learned to migrate the database when the migration requires you to specify some default data.

Finally, you learned how to make sure a user can only see data that belongs to them by modifying the view functions. You retrieved appropriate data using the `filter()` method and compared the owner of the requested data to the currently logged in user.

It might not always be immediately obvious what data you should make available and what data you should protect, but this skill will come with practice. The decisions we've made in this chapter to secure our users' data also illustrate why working with others is a good idea when building a project: having someone else look over your project makes it more likely that you'll spot vulnerable areas.

You now have a fully functioning project running on your local machine. In the final chapter, you'll style Learning Log to make it visually appealing, and you'll deploy the project to a server so anyone with internet access can register and make an account.

20

STYLING AND DEPLOYING AN APP

Learning Log is fully functional now, but it has no styling and runs only on your local machine. In this chapter, you'll style the project in a simple but professional manner and then deploy it to a live server so anyone in the world can make an account and use it.

For the styling we'll use the Bootstrap library, a collection of tools for styling web applications so they look professional on all modern devices, from a large flat-screen monitor to a smartphone. To do this, we'll use the django-bootstrap4 app, which will also give you practice using apps made by other Django developers.

We'll deploy Learning Log using Heroku, a site that lets you push your project to one of its servers, making it available to anyone with an internet connection. We'll also start using a version control system called Git to track changes to the project.

When you're finished with Learning Log, you'll be able to develop simple web applications, make them look good, and deploy them to a live server. You'll also be able to use more advanced learning resources as you develop your skills.

Styling Learning Log

We've purposely ignored styling until now to focus on Learning Log's functionality first. This is a good way to approach development, because an app is useful only if it works. Of course, once it's working, appearance is critical so people will want to use it.

In this section, I'll introduce the django-bootstrap4 app and show you how to integrate it into a project to make it ready for live deployment.

The django-bootstrap4 App

We'll use django-bootstrap4 to integrate Bootstrap into our project. This app downloads the required Bootstrap files, places them in an appropriate location in your project, and makes the styling directives available in your project's templates.

To install django-bootstrap4, issue the following command in an active virtual environment:

```
(ll_env)learning_log$ pip install django-bootstrap4
--snip--
Successfully installed django-bootstrap4-0.0.7
```

Next, we need to add the following code to include django-bootstrap4 in `INSTALLED_APPS` in *settings.py*:

settings.py

```
--snip--
INSTALLED_APPS = [
    # My apps.
    'learning_logs',
    'users',

    # Third party apps.
    'bootstrap4',

    # Default django apps.
    'django.contrib.admin',
    --snip--
```

Start a new section called *Third party apps* for apps created by other developers and add `'bootstrap4'` to this section. Make sure you place this section after `# My apps` but before the section containing Django's default apps.

Using Bootstrap to Style Learning Log

Bootstrap is a large collection of styling tools. It also has a number of templates you can apply to your project to create an overall style. It's much easier to use these templates than it is to use individual styling tools. To see the templates Bootstrap offers, go to *https://getbootstrap.com/*, click **Examples**, and look for the *Navbars* section. We'll use the *Navbar static* template, which provides a simple top navigation bar and a container for the page's content.

Figure 20-1 shows what the home page will look like after we apply Bootstrap's template to *base.html* and modify *index.html* slightly.

Figure 20-1: The Learning Log home page using Bootstrap

Modifying base.html

We need to modify the *base.html* template to accommodate the Bootstrap template. I'll introduce the new *base.html* in parts.

Defining the HTML Headers

The first change we'll make to *base.html* defines the HTML headers in the file, so whenever a Learning Log page is open, the browser title bar displays the site name. We'll also add some requirements for using Bootstrap in our templates. Delete everything in *base.html* and replace it with the following code:

base.html

```
❶ {% load bootstrap4 %}

❷ <!doctype html>
❸ <html lang="en">
❹ <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1,
      shrink-to-fit=no">
❺   <title>Learning Log</title>

❻   {% bootstrap_css %}
    {% bootstrap_javascript jquery='full' %}

❼ </head>
```

At ❶ we load the collection of template tags available in django-bootstrap4. Next, we declare this file as an HTML document ❷ written in English ❸. An HTML file is divided into two main parts, the *head* and the *body*—the head of the file begins at ❹. The head of an HTML file doesn't contain any content: it just tells the browser what it needs to know to display the page correctly. At ❺ we include a `title` element for the page, which will display in the browser's title bar whenever Learning Log is open.

At ❻ we use one of django-bootstrap4's custom template tags, which tells Django to include all the Bootstrap style files. The tag that follows enables all the interactive behavior you might use on a page, such as collapsible navigation bars. At ❼ is the closing `</head>` tag.

Defining the Navigation Bar

The code that defines the navigation bar at the top of the page is fairly long, because it has to work well on narrow phone screens and wide desktop monitors. We'll work through the navigation bar in sections.

Here's the first part of the navigation bar:

base.html

```
   --snip--
   </head>
❶ <body>

❷   <nav class="navbar navbar-expand-md navbar-light bg-light mb-4 border">

❸     <a class="navbar-brand" href="{% url 'learning_logs:index'%}">
          Learning Log</a>

❹     <button class="navbar-toggler" type="button" data-toggle="collapse"
          data-target="#navbarCollapse" aria-controls="navbarCollapse"
          aria-expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span></button>
```

The first element is the opening `<body>` tag ❶. The *body* of an HTML file contains the content users will see on a page. At ❷ is a `<nav>` element that indicates the page's navigation links section. Everything contained in this element is styled according to the Bootstrap style rules defined by the selectors `navbar`, `navbar-expand-md`, and the rest that you see here. A *selector* determines which elements on a page a certain style rule applies to. The `navbar-light` and `bg-light` selectors style the navigation bar with a light-themed background. The `mb` in `mb-4` is short for *margin-bottom*; this selector ensures that a little space appears between the navigation bar and the rest of the page. The `border` selector provides a thin border around the light background to set it off a little from the rest of the page.

At ❸ we set the project's name to appear at the far left of the navigation bar and make it a link to the home page; it will appear on every page in the project. The `navbar-brand` selector styles this link so it stands out from the rest of the links and is a way of branding the site.

At ❹ the template defines a button that appears if the browser window is too narrow to display the whole navigation bar horizontally. When the

user clicks the button, the navigation elements will appear in a drop-down list. The `collapse` reference causes the navigation bar to collapse when the user shrinks the browser window or when the site is displayed on mobile devices with small screens.

Here's the next section of code that defines the navigation bar:

base.html

```
        --snip--
      <span class="navbar-toggler-icon"></span></button>
❶   <div class="collapse navbar-collapse" id="navbarCollapse">
❷     <ul class="navbar-nav mr-auto">
❸       <li class="nav-item">
          <a class="nav-link" href="{% url 'learning_logs:topics'%}">
              Topics</a></li>
      </ul>
```

At ❶ we open a new section of the navigation bar. The term *div* is short for division; you build a web page by dividing it into sections and defining style and behavior rules that apply to that section. Any styling or behavior rules that are defined in an opening div tag affect everything you see until the next closing div tag, which is written as `</div>`. This is the beginning of the part of the navigation bar that will be collapsed on narrow screens and windows.

At ❷ we define a new set of links. Bootstrap defines navigation elements as items in an unordered list with style rules that make it look nothing like a list. Every link or element you need on the bar can be included as an item in one of these lists. Here, the only item in the list is our link to the Topics page ❸.

Here's the next part of the navigation bar:

base.html

```
            --snip--
      </ul>
❶     <ul class="navbar-nav ml-auto">
❷       {% if user.is_authenticated %}
          <li class="nav-item">
❸           <span class="navbar-text"}">Hello, {{ user.username }}.</span>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="{% url 'users:logout' %}">Log out</a>
          </li>
        {% else %}
          <li class="nav-item">
            <a class="nav-link" href="{% url 'users:register' %}">Register</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="{% url 'users:login' %}">Log in</a></li>
        {% endif %}
      </ul>
❹   </div>

  </nav>
```

At ❶ we begin a new set of links by using another opening `<ul>` tag. You can have as many groups of links as you need on a page. This will be the group of links related to login and registration that appears on the right side of the navigation bar. The selector `ml-auto` is short for *margin-left-automatic*: this selector examines the other elements in the navigation bar and works out a left margin that pushes this group of links to the right side of the screen.

The `if` block at ❷ is the same conditional block we used earlier to display appropriate messages to users depending on whether or not they're logged in. The block is a little longer now because some styling rules are inside the conditional tags. At ❸ is a `<span>` element. The *span element* styles pieces of text, or elements of a page, that are part of a longer line. Whereas div elements create their own division in a page, span elements are continuous within a larger section. This can be confusing at first, because many pages have deeply nested div elements. Here, we're using the span element to style informational text on the navigation bar, such as the logged-in user's name. We want this information to appear different from a link, so users aren't tempted to click these elements.

At ❹ we close the div element that contains the parts of the navigation bar that will collapse on narrow screens, and at the end of this section we close the navigation bar overall. If you wanted to add more links to the navigation bar, you'd add another `<li>` item to any of the `<ul>` groups that we've defined in the navigation bar by using identical styling directives as what you've seen here.

There's still a bit more we need to add to *base.html*. We need to define two blocks that the individual pages can use to place the content specific to those pages.

Defining the Main Part of the Page

The rest of *base.html* contains the main part of the page:

base.html

```
    --snip--
  </nav>

❶ <main role="main" class="container">
❷   <div class="pb-2 mb-2 border-bottom">
      {% block page_header %}{% endblock page_header %}
    </div>
❸   <div>
      {% block content %}{% endblock content %}
    </div>
  </main>

</body>

</html>
```

At ❶ we open a `<main>` tag. The *main* element is used for the most significant part of the body of a page. Here we assign the bootstrap selector

container, which is a simple way to group elements on a page. We'll place two div elements in this container.

The first div element ❷ contains a page_header block. We'll use this block to title most pages. To make this section stand out from the rest of the page, we place some padding below the header. *Padding* refers to space between an element's content and its border. The selector pb-2 is a bootstrap directive that provides a moderate amount of padding at the bottom of the styled element. A *margin* is the space between an element's border and other elements on the page. We want a border only on the bottom of the page, so we use the selector border-bottom, which provides a thin border at the bottom of the page_header block.

At ❸ we define one more div element, which contains the block content. We don't apply any specific style to this block, so we can style the content of any page as we see fit for that page. We end the *base.html* file with closing tags for the main, body, and html elements.

When you load Learning Log's home page in a browser, you should see a professional-looking navigation bar that matches the one shown in Figure 20-1. Try resizing the window so it's very narrow; a button should replace the navigation bar. Click the button, and all the links should appear in a drop-down list.

Styling the Home Page Using a Jumbotron

To update the home page, we'll use a Bootstrap element called a *jumbotron*, which is a large box that stands out from the rest of the page and can contain anything you want. Typically, it's used on home pages to hold a brief description of the overall project and a call to action that invites the viewer to get involved.

Here's the revised *index.html* file:

index.html

```
{% extends "learning_logs/base.html" %}

❶ {% block page_header %}
❷   <div class="jumbotron">
❸     <h1 class="display-3">Track your learning.</h1>

❹     <p class="lead">Make your own Learning Log, and keep a list of the
          topics you're learning about. Whenever you learn something new
          about a topic, make an entry summarizing what you've learned.</p>

❺     <a class="btn btn-lg btn-primary" href="{% url 'users:register' %}"
          role="button">Register &raquo;</a>
    </div>
❻ {% endblock page_header %}
```

At ❶ we tell Django that we're about to define what goes in the page_header block. A jumbotron is just a div element with a set of styling directives applied to it ❷. The jumbotron selector applies this group of styling directives from the Bootstrap library to this element.

Inside the jumbotron are three elements. The first is a short message, *Track your learning*, that gives first-time visitors a sense of what Learning Log does. The `h1` class is a first-level header, and the `display-3` selector adds a thinner and taller look to this particular header ❸. At ❹ we include a longer message that provides more information about what the user can do with their learning log.

Rather than just using a text link, we create a button at ❺ that invites users to register their Learning Log account. This is the same link as in the header, but the button stands out on the page and shows the viewer what they need to do to start using the project. The selectors you see here style this as a large button that represents a call to action. The code `»` is an *HTML entity* that looks like two right angle brackets combined (>>). At ❻ we close the `page_header` block. We aren't adding any more content to this page, so we don't need to define the `content` block on this page.

The index page now looks like Figure 20-1 and is a significant improvement over our unstyled project.

Styling the Login Page

We've refined the overall appearance of the login page but not the login form yet. Let's make the form look consistent with the rest of the page by modifying the *login.html* file:

login.html

```
  {% extends "learning_logs/base.html" %}
❶ {% load bootstrap4 %}

❷ {% block page_header %}
    <h2>Log in to your account.</h2>
  {% endblock page_header %}

  {% block content %}
❸   <form method="post" action="{% url 'users:login' %}" class="form">
      {% csrf_token %}
❹     {% bootstrap_form form %}
❺     {% buttons %}
        <button name="submit" class="btn btn-primary">Log in</button>
      {% endbuttons %}

      <input type="hidden" name="next"
        value="{% url 'learning_logs:index' %}" />
    </form>

  {% endblock content %}
```

At ❶ we load the bootstrap4 template tags into this template. At ❷ we define the `page_header` block, which tells the user what the page is for. Notice that we've removed the `{% if form.errors %}` block from the template; django-bootstrap4 manages form errors automatically.

At ❸ we add a `class="form"` attribute, and then we use the template tag `{% bootstrap_form %}` when we display the form ❹; this replaces the `{{ form .as_p }}` tag we were using in Chapter 19. The `{% booostrap_form %}` template

tag inserts Bootstrap style rules into the form's individual elements as the form is rendered. At ❺ we open a bootstrap4 template tag `{% buttons %}`, which adds Bootstrap styling to buttons.

Figure 20-2 shows the login form now. The page is much cleaner and has consistent styling and a clear purpose. Try logging in with an incorrect username or password; you'll see that even the error messages are styled consistently and integrate well with the overall site.

Figure 20-2: The login page styled with Bootstrap

Styling the Topics Page

Let's make sure the pages for viewing information are styled appropriately as well, starting with the topics page:

topics.html

```
{% extends "learning_logs/base.html" %}

❶ {% block page_header %}
    <h1>Topics</h1>
  {% endblock page_header %}

  {% block content %}
    <ul>
      {% for topic in topics %}
❷       <li><h3>
          <a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a>
        </h3></li>
      {% empty %}
        <li><h3>No topics have been added yet.</h3></li>
      {% endfor %}
    </ul>

❸   <h3><a href="{% url 'learning_logs:new_topic' %}">Add a new topic</a></h3>
  {% endblock content %}
```

We don't need the {% load bootstrap4 %} tag, because we're not using any custom bootstrap4 template tags in this file. We move the heading *Topics* into the page_header block and give it a header styling instead of using the simple paragraph tag ❶. We style each topic as an <h3> element to make them a little larger on the page ❷ and do the same for the link to add a new topic ❸.

Styling the Entries on the Topic Page

The topic page has more content than most pages, so it needs a bit more work. We'll use Bootstrap's card component to make each entry stand out. A *card* is a div with a set of flexible, predefined styles that's perfect for displaying a topic's entries:

topic.html

```
{% extends 'learning_logs/base.html' %}

❶ {% block page_header %}
    <h3>{{ topic }}</h3>
  {% endblock page_header %}

  {% block content %}
    <p>
      <a href="{% url 'learning_logs:new_entry' topic.id %}">Add new entry</a>
    </p>

    {% for entry in entries %}
❷     <div class="card mb-3">
❸       <h4 class="card-header">
          {{ entry.date_added|date:'M d, Y H:i' }}
❹         <small><a href="{% url 'learning_logs:edit_entry' entry.id %}">
              edit entry</a></small>
        </h4>
❺       <div class="card-body">
          {{ entry.text|linebreaks }}
        </div>
      </div>
    {% empty %}
      <p>There are no entries for this topic yet.</p>
    {% endfor %}

  {% endblock content %}
```

We first place the topic in the page_header block ❶. Then we delete the unordered list structure previously used in this template. Instead of making each entry a list item, we create a div element with the selector card at ❷. This card has two nested elements: one to hold the timestamp and the link to edit the entry, and another to hold the body of the entry.

The first element in the card is a header, which is an <h4> element with the selector card-header ❸. This card header contains the date the entry was made and a link to edit the entry. The <small> tag around the edit_entry link

makes it appear a little smaller than the timestamp ❹. The second element is a div with the selector `card-body` ❺, which places the text of the entry in a simple box on the card. Notice that the Django code for including the information on the page hasn't changed; only the elements that affect the appearance of the page have changed.

Figure 20-3 shows the topic page with its new look. Learning Log's functionality hasn't changed, but it looks more professional and inviting to users now.

Figure 20-3: The topic page with Bootstrap styling

NOTE *If you want to use a different Bootstrap template, follow a similar process to what we've done so far in this chapter. Copy the template you want to use into* base.html, *and modify the elements that contain actual content so the template displays your project's information. Then use Bootstrap's individual styling tools to style the content on each page.*

TRY IT YOURSELF

20-1. Other Forms: We applied Bootstrap's styles to the `login` page. Make similar changes to the rest of the form-based pages including `new_topic`, `new_entry`, `edit_entry`, and `register`.

20-2. Stylish Blog: Use Bootstrap to style the Blog project you created in Chapter 19.

Deploying Learning Log

Now that we have a professional-looking project, let's deploy it to a live server so anyone with an internet connection can use it. We'll use Heroku, a web-based platform that allows you to manage the deployment of web applications. We'll get Learning Log up and running on Heroku.

Making a Heroku Account

To make an account, go to *https://heroku.com/* and click one of the signup links. It's free to make an account, and Heroku has a free tier that allows you to test your projects in live deployment before properly deploying them.

NOTE *Heroku's free tier has limits, such as the number of apps you can deploy and how often people can visit your app. But these limits are generous enough to let you practice deploying apps without any cost.*

Installing the Heroku CLI

To deploy and manage a project on Heroku's servers, you'll need the tools available in the Heroku Command Line Interface (CLI). To install the latest version of the Heroku CLI, visit *https://devcenter.heroku.com/articles/heroku-cli/* and follow the instructions for your operating system. The instructions will include either a one-line terminal command or an installer you can download and run.

Installing Required Packages

You'll also need to install three packages that help serve Django projects on a live server. In an active virtual environment, issue the following commands:

```
(ll_env)learning_log$ pip install psycopg2==2.7.*
(ll_env)learning_log$ pip install django-heroku
(ll_env)learning_log$ pip install gunicorn
```

The `psycopg2` package is required to manage the database that Heroku uses. The `django-heroku` package handles almost the entire configuration our app needs to run properly on Heroku servers. This includes managing the database and storing static files in a place where they can be served properly. *Static files* contain style rules and JavaScript files. The `gunicorn` package provides a server capable of serving apps in a live environment.

Creating a requirements.txt File

Heroku needs to know which packages our project depends on, so we'll use pip to generate a file listing them. Again, from an active virtual environment, issue the following command:

```
(ll_env)learning_log$ pip freeze > requirements.txt
```

The `freeze` command tells pip to write the names of all the packages currently installed in the project into the file *requirements.txt*. Open this file to see the packages and version numbers installed in your project:

requirements.txt

```
dj-database-url==0.5.0
Django==2.2.0
django-bootstrap4==0.0.7
django-heroku==0.3.1
gunicorn==19.9.0
psycopg2==2.7.7
pytz==2018.9
sqlparse==0.2.4
whitenoise==4.1.2
```

Learning Log already depends on eight different packages with specific version numbers, so it requires a specific environment to run properly. (We installed four of these packages manually, and four of them were installed automatically as dependencies of these packages.)

When we deploy Learning Log, Heroku will install all the packages listed in *requirements.txt*, creating an environment with the same packages we're using locally. For this reason, we can be confident the deployed project will behave the same as it does on our local system. This is a huge advantage as you start to build and maintain various projects on your system.

NOTE *If a package is listed on your system but the version number differs from what's shown here, keep the version you have on your system.*

Specifying the Python Runtime

Unless you specify a Python version, Heroku will use its current default version of Python. Let's make sure Heroku uses the same version of Python we're using. In an active virtual environment, issue the command `python --version`:

```
(ll_env)learning_log$ python --version
Python 3.7.2
```

In this example I'm running Python 3.7.2. Make a new file called *runtime.txt* in the same directory as *manage.py*, and enter the following:

runtime.txt

```
python-3.7.2
```

This file should contain one line with your Python version specified in the format shown; make sure you enter `python` in lowercase, followed by a hyphen, followed by the three-part version number.

NOTE *If you get an error reporting that the Python runtime you requested isn't available, go to* https://devcenter.heroku.com/categories/language-support/ *and look for a link to* Specifying a Python Runtime. *Scan through the article to find the available runtimes, and use the one that most closely matches your Python version.*

Modifying settings.py for Heroku

Now we need to add a section at the end of *settings.py* to define some specific settings for the Heroku environment:

settings.py

```
--snip--
# My settings
LOGIN_URL = 'users:login'

# Heroku settings.
import django_heroku
django_heroku.settings(locals())
```

Here we import the `django_heroku` module and call the `settings()` function. This function modifies some settings that need specific values for the Heroku environment.

Making a Procfile to Start Processes

A *Procfile* tells Heroku which processes to start to properly serve the project. Save this one-line file as *Procfile*, with an uppercase *P* and no file extension, in the same directory as *manage.py*.

Here's the line that goes in *Procfile*:

Procfile

```
web: gunicorn learning_log.wsgi --log-file -
```

This line tells Heroku to use gunicorn as a server and to use the settings in *learning_log/wsgi.py* to launch the app. The `log-file` flag tells Heroku the kinds of events to log.

Using Git to Track the Project's Files

As discussed in Chapter 17, Git is a version control program that allows you to take a snapshot of the code in your project each time you implement a new feature successfully. If anything goes wrong, you can easily return to the last working snapshot of your project; for example, if you accidentally introduce a bug while working on a new feature. Each snapshot is called a *commit*.

Using Git, you can try implementing new features without worrying about breaking your project. When you're deploying to a live server, you need to make sure you're deploying a working version of your project. To read more about Git and version control, see Appendix D.

Installing Git

Git may already be installed on your system. To find out if Git is already installed, open a new terminal window and issue the command `git --version`:

```
(ll_env)learning_log$ git --version
git version 2.17.0
```

If you get an error message for some reason, see the installation instructions for Git in Appendix D.

Configuring Git

Git keeps track of who makes changes to a project, even when only one person is working on the project. To do this, Git needs to know your username and email. You must provide your username, but you can make up an email for your practice projects:

```
(ll_env)learning_log$ git config --global user.name "ehmatthes"
(ll_env)learning_log$ git config --global user.email "eric@example.com"
```

If you forget this step, Git will prompt you for this information when you make your first commit.

Ignoring Files

We don't need Git to track every file in the project, so we'll tell it to ignore some files. Create a file called *.gitignore* in the folder that contains *manage.py*. Notice that this filename begins with a dot and has no file extension. Here's the code that goes in *.gitignore*:

.gitignore

```
ll_env/
__pycache__/
*.sqlite3
```

We tell Git to ignore the entire *ll_env* directory, because we can re-create it automatically at any time. We also don't track the *__pycache__* directory, which contains the *.pyc* files that are created automatically when Django runs the *.py* files. We don't track changes to the local database, because it's a bad habit: if you're ever using SQLite on a server, you might accidentally overwrite the live database with your local test database when you push the project to the server. The asterisk in `*.sqlite3` tells Git to ignore any file that ends with the extension *.sqlite3*.

NOTE *If you're using macOS, add* `.DS_Store` *to your* .gitignore *file. This is a file that stores information about folder settings on macOS, and it has nothing to do with this project.*

Making Hidden Files Visible

Most operating systems hide files and folders that begin with a dot, such as *.gitignore*. When you open a file browser or try to open a file from an application such as Sublime Text, you won't see these kinds of files by default. But as a programmer, you'll need to see them. Here's how to view hidden files, depending on your operating system:

- On Windows, open Windows Explorer, and then open a folder such as *Desktop*. Click the **View** tab, and make sure **File name extensions** and **Hidden items** are checked.

- On macOS, you can press ⌘-SHIFT-. (dot) in any file browser window to see hidden files and folders.
- On Linux systems such as Ubuntu, you can press CTRL-H in any file browser to display hidden files and folders. To make this setting permanent, open a file browser such as Nautilus and click the options tab (indicated by three lines). Select the **Show Hidden Files** checkbox.

Committing the Project

We need to initialize a Git repository for Learning Log, add all the necessary files to the repository, and commit the initial state of the project. Here's how to do that:

```
❶ (ll_env)learning_log$ git init
  Initialized empty Git repository in /home/ehmatthes/pcc/learning_log/.git/
❷ (ll_env)learning_log$ git add .
❸ (ll_env)learning_log$ git commit -am "Ready for deployment to heroku."
  [master (root-commit) 79fef72] Ready for deployment to heroku.
   45 files changed, 712 insertions(+)
   create mode 100644 .gitignore
   create mode 100644 Procfile
   --snip--
   create mode 100644 users/views.py
❹ (ll_env)learning_log$ git status
  On branch master
  nothing to commit, working tree clean
  (ll_env)learning_log$
```

At ❶ we issue the `git init` command to initialize an empty repository in the directory containing Learning Log. At ❷ we use the `git add .` command, which adds all the files that aren't being ignored to the repository. (Don't forget the dot.) At ❸ we issue the command `git commit -am` *`commit message`*: the `-a` flag tells Git to include all changed files in this commit, and the `-m` flag tells Git to record a log message.

Issuing the `git status` command ❹ indicates that we're on the *master* branch and that our working tree is *clean*. This is the status you'll want to see any time you push your project to Heroku.

Pushing to Heroku

We're finally ready to push the project to Heroku. In an active virtual environment, issue the following commands:

```
❶ (ll_env)learning_log$ heroku login
  heroku: Press any key to open up the browser to login or q to exit:
  Logging in... done
  Logged in as eric@example.com
❷ (ll_env)learning_log$ heroku create
  Creating app... done, ⬢ secret-lowlands-82594
  https://secret-lowlands-82594.herokuapp.com/ |
      https://git.heroku.com/secret-lowlands-82594.git
```

```
❸ (ll_env)learning_log$ git push heroku master
  --snip--
  remote: -----> Launching...
  remote:        Released v5
❹ remote:        https://secret-lowlands-82594.herokuapp.com/ deployed to Heroku
  remote: Verifying deploy... done.
  To https://git.heroku.com/secret-lowlands-82594.git
   * [new branch]      master -> master
  (ll_env)learning_log$
```

First you issue the `heroku login` command, which will take you to a page in your browser where you can log in to your Heroku account ❶. Then you tell Heroku to build an empty project ❷. Heroku generates a name made up of two words and a number; you can change this later on. Next, we issue the command `git push heroku master` ❸, which tells Git to push the master branch of the project to the repository Heroku just created. Then Heroku builds the project on its servers using these files. At ❹ is the URL we'll use to access the live project, which we can change along with the project name.

When you've issued these commands, the project is deployed but not fully configured. To check that the server process started correctly, use the `heroku ps` command:

```
  (ll_env)learning_log$ heroku ps
❶ Free dyno hours quota remaining this month: 450h 44m (81%)
  Free dyno usage for this app: 0h 0m (0%)
  For more information on dyno sleeping and how to upgrade, see:
  https://devcenter.heroku.com/articles/dyno-sleeping
❷ === web (Free): gunicorn learning_log.wsgi --log-file - (1)
  web.1: up 2019/02/19 23:40:12 -0900 (~ 10m ago)
  (ll_env)learning_log$
```

The output shows how much more time the project can be active in the next month ❶. At the time of this writing, Heroku allows free deployments to be active for up to 550 hours in a month. If a project exceeds this limit, a standard server error page will display; we'll customize this error page shortly. At ❷ we see that the process defined in *Procfile* has been started.

Now we can open the app in a browser using the command `heroku open`:

```
(ll_env)learning_log$ heroku open
(ll_env)learning_log$
```

This command spares you from opening a browser and entering the URL Heroku showed you, but that's another way to open the site. You should see the home page for Learning Log, styled correctly. However, you can't use the app yet because we haven't set up the database.

NOTE *Heroku's deployment process changes from time to time. If you have any unresolvable issues, look at Heroku's documentation for help. Go to* https://devcenter.heroku.com/, *click* ***Python****, and look for a link to* Get Started with Python *or* Deploying Python and Django Apps on Heroku. *If you don't understand what you see there, check out the suggestions in Appendix C.*

Setting Up the Database on Heroku

We need to run `migrate` once to set up the live database and apply all the migrations we generated during development. You can run Django and Python commands on a Heroku project using the command `heroku run`. Here's how to run `migrate` on the Heroku deployment:

```
❶ (ll_env)learning_log$ heroku run python manage.py migrate
❷ Running 'python manage.py migrate' on ⬢ secret-lowlands-82594... up, run.3060
    --snip--
❸ Running migrations:
    --snip--
    Applying learning_logs.0001_initial... OK
    Applying learning_logs.0002_entry... OK
    Applying learning_logs.0003_topic_owner... OK
    Applying sessions.0001_initial... OK
  (ll_env)learning_log$
```

We first issue the command `heroku run python manage.py migrate` ❶. Heroku then creates a terminal session to run the `migrate` command ❷. At ❸ Django applies the default migrations and the migrations we generated during the development of Learning Log.

Now when you visit your deployed app, you should be able to use it just as you did on your local system. But you won't see any of the data you entered on your local deployment, including your superuser account, because we didn't copy the data to the live server. This is normal practice: you don't usually copy local data to a live deployment because the local data is usually test data.

You can share your Heroku link to let anyone use your version of Learning Log. In the next section, we'll complete a few more tasks to finish the deployment process and set you up to continue developing Learning Log.

Refining the Heroku Deployment

Now we'll refine the deployment by creating a superuser, just as we did locally. We'll also make the project more secure by changing the setting `DEBUG` to `False`, so users won't see any extra information in error messages that they could use to attack the server.

Creating a Superuser on Heroku

You've already seen that we can run one-off commands using the `heroku run` command. But you can also run commands by opening a Bash terminal session while connected to the Heroku server using the command `heroku run bash`. Bash is the language that runs in many Linux terminals. We'll use the Bash terminal session to create a superuser so we can access the admin site on the live app:

```
  (ll_env)learning_log$ heroku run bash
  Running 'bash' on ⬢ secret-lowlands-82594... up, run.9858
❶ ~ $ ls
  learning_log learning_logs manage.py Procfile requirements.txt runtime.txt
```

```
  staticfiles users
❷ ~ $ python manage.py createsuperuser
  Username (leave blank to use ' u47318'): ll_admin
  Email address:
  Password:
  Password (again):
  Superuser created successfully.
❸ ~ $ exit
  exit
  (ll_env)learning_log$
```

At ❶ we run `ls` to see which files and directories exist on the server, which should be the same files we have on our local system. You can navigate this filesystem like any other.

NOTE *Windows users will use the same commands shown here (such as `ls` instead of `dir`), because you're running a Linux terminal through a remote connection.*

At ❷ we run the command to create a superuser, which outputs the same prompts we saw on our local system when we created a superuser in Chapter 18. When you're finished creating the superuser in this terminal session, run the **`exit`** command to return to your local system's terminal session ❸.

Now you can add */admin/* to the end of the URL for the live app and log in to the admin site. For me, the URL is *https://secret-lowlands-82594.herokuapp.com/admin/*.

If others have already started using your project, be aware that you'll have access to all of their data! Don't take this lightly, and users will continue to trust you with their data.

Creating a User-Friendly URL on Heroku

Most likely, you'll want your URL to be friendlier and more memorable than *https://secret-lowlands-82594.herokuapp.com/*. You can rename the app using a single command:

```
(ll_env)learning_log$ heroku apps:rename learning-log
Renaming secret-lowlands-82594 to learning-log-2e... done
https://learning-log.herokuapp.com/ | https://git.heroku.com/learning-log.git
Git remote heroku updated
 ⬢  Don't forget to update git remotes for all other local checkouts of the app.
(ll_env)learning_log$
```

You can use letters, numbers, and dashes when naming your app, and call it whatever you want, as long as no one else has claimed the name. This deployment now lives at *https://learning-log.herokuapp.com/*. The project is no longer available at the previous URL; the `apps:rename` command completely moves the project to the new URL.

NOTE *When you deploy your project using Heroku's free service, Heroku puts your deployment to sleep if it hasn't received any requests after a certain amount of time or if it's been too active for the free tier. The first time a user accesses the site after it's been sleeping, it will take longer to load, but the server will respond to subsequent requests more quickly. This is how Heroku can afford to offer free deployments.*

Securing the Live Project

One glaring security issue exists in the way our project is currently deployed: the setting `DEBUG=True` in *settings.py*, which provides debug messages when errors occur. Django's error pages give you vital debugging information when you're developing a project; however, they give way too much information to attackers if you leave them enabled on a live server.

We'll control whether debugging information is shown on the live site by setting an environment variable. *Environment variables* are values set in a specific environment. This is one of the ways sensitive information is stored on a server, keeping it separate from the rest of the project's code.

Let's modify *settings.py* so it looks for an environment variable when the project is running on Heroku:

settings.py

```
--snip--
# Heroku settings.
import django_heroku
django_heroku.settings(locals())

if os.environ.get('DEBUG') == 'TRUE':
    DEBUG = True
elif os.environ.get('DEBUG') == 'FALSE':
    DEBUG = False
```

The method `os.environ.get()` reads the value associated with a specific environment variable in any environment where the project is running. If the variable we're asking for is set, the method returns its value; if it's not set, the method returns `None`. Using environment variables to store Boolean values can be confusing. In most cases, environment variables are stored as strings, and you have to be careful about this. Consider this snippet from a simple Python terminal session:

```
>>> bool('False')
True
```

The Boolean value of the string `'False'` is `True`, because any non-empty string evaluates to `True`. So we'll use the strings `'TRUE'` and `'FALSE'`, in all capitals, to be clear that we're not storing Python's actual `True` and `False` Boolean values. When Django reads in the environment variable with the key `'DEBUG'` on Heroku, we'll set `DEBUG` to `True` if the value is `'TRUE'` and `False` if the value is `'FALSE'`.

Committing and Pushing Changes

Now we need to commit the changes made to *settings.py* to the Git repository, and then push the changes to Heroku. Here's a terminal session showing this process:

```
❶ (ll_env)learning_log$ git commit -am "Set DEBUG based on environment variables."
[master 3427244] Set DEBUG based on environment variables.
 1 file changed, 4 insertions(+)
❷ (ll_env)learning_log$ git status
On branch master
nothing to commit, working tree clean
(ll_env)learning_log$
```

We issue the `git commit` command with a short but descriptive commit message ❶. Remember that the `-am` flag makes sure Git commits all the files that have changed and records the log message. Git recognizes that one file has changed and commits this change to the repository.

At ❷ the status shows that we're working on the master branch of the repository and that there are now no new changes to commit. It's essential that you check the status for this message before pushing to Heroku. If you don't see this message, some changes haven't been committed, and those changes won't be pushed to the server. You can try issuing the `commit` command again, but if you're not sure how to resolve the issue, read through Appendix D to better understand how to work with Git.

Now let's push the updated repository to Heroku:

```
(ll_env)learning_log$ git push heroku master
remote: Building source:
remote:
remote: -----> Python app detected
remote: -----> Installing requirements with pip
--snip--
remote: -----> Launching...
remote:        Released v6
remote:        https://learning-log.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/learning-log.git
   144f020..d5075a1  master -> master
(ll_env)learning_log$
```

Heroku recognizes that the repository has been updated, and it rebuilds the project to make sure all the changes have been taken into account. It doesn't rebuild the database, so we won't have to run `migrate` for this update.

Setting Environment Variables on Heroku

Now we can set the value we want for `DEBUG` in *settings.py* through Heroku. The command `heroku config:set` sets an environment variable for us:

```
(ll_env)learning_log$ heroku config:set DEBUG='FALSE'
Setting DEBUG and restarting ⬢ learning-log... done, v7
DEBUG: FALSE
(ll_env)learning_log$
```

Whenever you set an environment variable on Heroku, it automatically restarts the project so the environment variable can take effect.

To check that the deployment is more secure now, enter your project's URL with a path we haven't defined. For example, try to visit *http://learning-log.herokuapp.com/letmein/*. You should see a generic error page on your live deployment that doesn't give away any specific information about the project. If you try the same request on the local version of Learning Log at *http://localhost:8000/letmein/*, you should see the full Django error page. The result is perfect: you'll see informative error messages when you're developing the project further on your own system. But users on the live site won't see critical information about the project's code.

If you're just deploying an app and you're troubleshooting the initial deployment, you can run `heroku config:set DEBUG='TRUE'` and temporarily see a full error report on the live site. Just make sure you reset the value to `'FALSE'` once you've finished troubleshooting. Also, be careful not to do this once users are regularly accessing your site.

Creating Custom Error Pages

In Chapter 19, we configured Learning Log to return a 404 error if the user requests a topic or entry that doesn't belong to them. You've probably seen some 500 server errors (internal errors) by this point as well. A 404 error usually means your Django code is correct, but the object being requested doesn't exist; a 500 error usually means there's an error in the code you've written, such as an error in a function in *views.py*. Currently, Django returns the same generic error page in both situations. But we can write our own 404 and 500 error page templates that match Learning Log's overall appearance. These templates must go in the root template directory.

Making Custom Templates

In the outermost *learning_log* folder, make a new folder called *templates*. Then make a new file called *404.html*; the path to this file should be *learning_log/templates/404.html*. Here's the code for this file:

404.html

```
{% extends "learning_logs/base.html" %}

{% block page_header %}
  <h2>The item you requested is not available. (404)</h2>
{% endblock page_header %}
```

This simple template provides the generic 404 error page information but is styled to match the rest of the site.

Make another file called *500.html* using the following code:

500.html

```
{% extends "learning_logs/base.html" %}

{% block page_header %}
  <h2>There has been an internal error. (500)</h2>
{% endblock page_header %}
```

These new files require a slight change to *settings.py*.

settings.py

```
--snip--
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        --snip--
    },
]
--snip--
```

This change tells Django to look in the root template directory for the error page templates.

Viewing the Error Pages Locally

If you want to see what the error pages look like on your system before pushing them to Heroku, you'll first need to set `Debug=False` on your local settings to suppress the default Django debug pages. To do so, make the following change to *settings.py* (make sure you're working in the part of *settings.py* that applies to the local environment, not the part that applies to Heroku):

settings.py

```
--snip--
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = False
--snip--
```

Now request a topic or entry that doesn't belong to you to see the 404 error page. To test the 500 error page, request a topic or entry that doesn't exist. For example, the URL *http://localhost:8000/topics/999/* should generate a 500 error unless you've generated 999 example topics already!

When you're finished checking the error pages, set the local value of `DEBUG` back to `True` to further develop Learning Log. (Make sure you don't change the way `DEBUG` is handled in the section that manages settings in the Heroku environment.)

NOTE *The 500 error page won't show any information about the user who's logged in, because Django doesn't send any context information in the response when there's a server error.*

Pushing the Changes to Heroku

Now we need to commit the error page changes we just made, and push them live to Heroku:

```
❶ (ll_env)learning_log$ git add .
❷ (ll_env)learning_log$ git commit -am "Added custom 404 and 500 error pages."
 3 files changed, 15 insertions(+), 10 deletions(-)
 create mode 100644 templates/404.html
 create mode 100644 templates/500.html
❸ (ll_env)learning_log$ git push heroku master
--snip--
remote: Verifying deploy.... done.
To https://git.heroku.com/learning-log.git
   d5075a1..4bd3b1c  master -> master
(ll_env)learning_log$
```

We issue the `git add .` command at ❶ because we created some new files in the project, so we need to tell Git to start tracking these files. Then we commit the changes ❷ and push the updated project to Heroku ❸.

Now when an error page appears, it should have the same styling as the rest of the site, making for a smoother user experience when errors arise.

Using the get_object_or_404() Method

At this point, if a user manually requests a topic or entry that doesn't exist, they'll get a 500 server error. Django tries to render the nonexistent page, but it doesn't have enough information to do so, and the result is a 500 error. This situation is more accurately handled as a 404 error, and we can implement this behavior using the Django shortcut function `get_object_or_404()`. This function tries to get the requested object from the database, but if that object doesn't exist, it raises a 404 exception. We'll import this function into *views.py* and use it in place of `get()`:

views.py

```
from django.shortcuts import render, redirect, get_object_or_404
from django.contrib.auth.decorators import login_required
--snip--
@login_required
def topic(request, topic_id):
    """Show a single topic and all its entries."""
    topic = get_object_or_404(Topic, id=topic_id)
    # Make sure the topic belongs to the current user.
    --snip--
```

Now when you request a topic that doesn't exist (for example, *http://localhost:8000/topics/999/*), you'll see a 404 error page. To deploy this change, make a new commit and then push the project to Heroku.

Ongoing Development

You might want to further develop Learning Log after your initial push to a live server or develop your own projects to deploy. There's a fairly consistent process for updating projects.

First, you'll make any changes needed to your local project. If your changes result in any new files, add those files to the Git repository using the command `git add .` (be sure to include the dot at the end of the command). Any change that requires a database migration will need this command, because each migration generates a new migration file.

Second, commit the changes to your repository using `git commit -am "commit message"`. Then push your changes to Heroku using the command `git push heroku master`. If you migrated your database locally, you'll need to migrate the live database as well. You can either use the one-off command `heroku run python manage.py migrate`, or open a remote terminal session with `heroku run bash` and run the command `python manage.py migrate`. Then visit your live project, and make sure the changes you expect to see have taken effect.

It's easy to make mistakes during this process, so don't be surprised when something goes wrong. If the code doesn't work, review what you've done and try to spot the mistake. If you can't find the mistake or you can't figure out how to undo the mistake, refer to the suggestions for getting help in Appendix C. Don't be shy about asking for help: everyone else learned to build projects by asking the same questions you're likely to ask, so someone will be happy to help you. Solving each problem that arises helps you steadily develop your skills until you're building meaningful, reliable projects and you're answering other people's questions as well.

The SECRET_KEY Setting

Django uses the value of the `SECRET_KEY` setting in *settings.py* to implement a number of security protocols. In this project, we've committed our settings file to the repository with the `SECRET_KEY` setting included. This is fine for a practice project, but the `SECRET_KEY` setting should be handled more carefully for a production site. If you build a project that's getting meaningful use, make sure you research how to handle your `SECRET_KEY` setting more securely.

Deleting a Project on Heroku

It's great practice to run through the deployment process a number of times with the same project or with a series of small projects to get the hang of deployment. But you'll need to know how to delete a project that's been deployed. Heroku also limits the number of projects you can host for free, and you don't want to clutter your account with practice projects.

Log in to the Heroku website (*https://heroku.com/*); you'll be redirected to a page showing a list of your projects. Click the project you want to delete. You'll see a new page with information about the project. Click the **Settings**

link, and scroll down until you see a link to delete the project. This action can't be reversed, so Heroku will ask you to confirm the request for deletion by manually entering the project's name.

If you prefer working from a terminal, you can also delete a project by issuing the `destroy` command:

```
(ll_env)learning_log$ heroku apps:destroy --app appname
```

Here, *appname* is the name of your project, which is either something like `secret-lowlands-82594` or `learning-log` if you've renamed the project. You'll be prompted to reenter the project name to confirm the deletion.

NOTE *Deleting a project on Heroku does nothing to your local version of the project. If no one has used your deployed project and you're just practicing the deployment process, it's perfectly reasonable to delete your project on Heroku and redeploy it.*

TRY IT YOURSELF

20-3. Live Blog: Deploy the Blog project you've been working on to Heroku. Make sure you set `DEBUG` to `False`, so users don't see the full Django error pages when something goes wrong.

20-4. More 404s: The `get_object_or_404()` function should also be used in the `new_entry()` and `edit_entry()` views. Make this change, test it by entering a URL like *http://localhost:8000/new_entry/999/*, and check that you see a 404 error.

20-5. Extended Learning Log: Add one feature to Learning Log, and push the change to your live deployment. Try a simple change, such as writing more about the project on the home page. Then try adding a more advanced feature, such as giving users the option of making a topic public. This would require an attribute called `public` as part of the `Topic` model (this should be set to `False` by default) and a form element on the new_topic page that allows the user to change a topic from private to public. You'd then need to migrate the project and revise *views.py* so any topic that's public is visible to unauthenticated users as well. Remember to migrate the live database after you've pushed your changes to Heroku.

Summary

In this chapter, you learned to give your projects a simple but professional appearance using the Bootstrap library and the django-bootstrap4 app. Using Bootstrap, the styles you choose will work consistently on almost any device people use to access your project.

You learned about Bootstrap's templates and used the *Navbar static* template to create a simple look and feel for Learning Log. You used a jumbotron to make a home page's message stand out and learned to style all the pages in a site consistently.

In the final part of the project, you learned how to deploy a project to Heroku's servers so anyone can access it. You made a Heroku account and installed some tools that help manage the deployment process. You used Git to commit the working project to a repository and then pushed the repository to Heroku's servers. Finally, you learned to begin securing your app by setting `DEBUG=False` on the live server.

Now that you've finished Learning Log, you can start building your own projects. Start simple, and make sure the project works before adding complexity. Enjoy your continued learning, and good luck with your projects!

AFTERWORD

Congratulations! You've learned the basics of Python and applied your knowledge to meaningful projects. You've made a game, visualized some data, and made a web application. From here, you can go in a number of different directions to continue developing your programming skills.

First, you should continue to work on meaningful projects that interest you. Programming is more appealing when you're solving relevant and significant problems, and you now have the skills to engage in a variety of projects. You could invent your own game or write your own version of a classic arcade game. You might want to explore some data that's important to you and make visualizations that show interesting patterns and connections. You could create your own web application or try to emulate one of your favorite apps.

Whenever possible, invite other people to try using your programs. If you write a game, let other people play it. If you make a visualization, show it to others and see if it makes sense to them. If you make a web app, deploy it online and invite others to try it out. Listen to your users and try to incorporate their feedback into your projects; you'll become a better programmer if you do.

When you work on your own projects, you'll run into problems that are challenging, or even impossible, to solve on your own. Keep finding ways to ask for help, and find your own place in the Python community. Join a local Python User Group or explore some online Python communities. Consider attending a PyCon near you as well.

You should strive to maintain a balance between working on projects that interest you and developing your Python skills in general. Many Python learning sources are available online, and a large number of Python books target intermediate programmers. Many of these resources will be accessible to you now that you know the basics and how to apply your skills. Working through Python tutorials and books will build directly on what you learned here and deepen your understanding of programming in general and Python in particular. Then when you go back to working on projects after focusing on learning about Python, you'll be capable of solving a wider variety of problems more efficiently.

Congratulations on how far you've come, and good luck with your continued learning!

A

INSTALLATION AND TROUBLESHOOTING

Python has several versions, and there are a number of ways to set it up on each operating system. Use this appendix to install Python if the approach in Chapter 1 didn't work or if you want to install a different version of Python than the one that came with your system.

Python on Windows

The instructions in Chapter 1 show you how to install Python using the official installer at *https://python.org/*. If you couldn't get Python to run after using the installer, the troubleshooting instructions in this section should help you get Python up and running.

Finding the Python Interpreter

If you've entered the simple command `python` and get an error, such as *python is not recognized as an internal or external command*, you most likely

forgot to select the *Add Python to PATH* option when you ran the installer. In this case, you'll need to tell Windows where to find the Python interpreter. To find it, open your C drive and find the folder that starts with the name *Python* (you might need to enter the word `python` in the Windows Explorer search bar to find the right folder, because it might be nested further down). Open the folder, and look for a file named *python* in lowercase. Right-click this file and choose **Properties**; the path to this file will be listed under the heading *Location*.

To tell Windows where to find the interpreter, open a terminal window and enter the path followed by the `--version` command, like so:

```
$ C:\Python37\python --version
Python 3.7.2
```

Your path might look something more like `C:\Users\username\Programs\Python37\python` on your system. Using this path, Windows should then run the Python interpreter.

Adding Python to Your Path Variable

It's annoying to type the full path each time you want to start a Python terminal session, so let's add the path to the system so you can just use the `python` command. Open your system's **Control Panel**, click **System and Security**, and then click **System**. Click **Advanced System Settings**. In the window that appears, click **Environment Variables**.

In the box labeled *System variables*, look for a variable called `Path`. Click the word **`Path`**, and then click **Edit**. You should see a list of locations that your system searches through when it looks for programs. Click **New**, and paste the path to your *python.exe* file in the text box that appears. If your system is set up like mine, that would be:

```
C:\Python37
```

Notice that we're not including the name of the *python.exe* file; we're just telling the system where to look for it.

Close your terminal window and open a new one. Doing so will load the new `Path` variable into your terminal session. Now when you enter `python --version`, you should see the version of Python you just added to your `Path` variable. You can now start a Python terminal session by just entering `python` at a command prompt.

NOTE *If you're using an earlier version of Windows, you might see a box labeled* Variable value *when you click* ***Edit****. If you see this box, use the right arrow key to scroll all the way to the right. Be careful not to overwrite the existing variable; if you do, click* ***Cancel*** *and try again. Add a semicolon and the path to your* python.exe *file to the existing variable:*

```
%SystemRoot%\system32\...\System32\WindowsPowerShell\v1.0\;C:\Python37
```

Reinstalling Python

If you're still unable to run Python, oftentimes uninstalling Python and running the installer again will address any problems that occurred in your first attempt.

To do this, open your system's **Control Panel** and click **Programs and Features**. Scroll down until you see the version of Python you just installed, and select it. Click **Uninstall/Change**, and then click **Uninstall** in the dialog that appears. Then run the installer again using the instructions in Chapter 1, but this time make sure you select the **Add Python to PATH** option and any other settings that are relevant to your system. If you're still running into trouble and aren't sure where to get help, see the suggestions in Appendix C.

Python on macOS

The installation instructions in Chapter 1 use the official Python installer at *https://python.org/*, which I recommend you use unless you have a specific reason not to. Another approach uses Homebrew, a tool you can use to install a variety of software on macOS. If you're already using Homebrew and want to use it to install Python, or if the people you're working with use Homebrew and you want a similar setup to what they're using, you can use the following instructions.

Installing Homebrew

Homebrew depends on some of the command line tools from Apple's Xcode package, so you'll first need to install the Xcode command line tools. Open a terminal and run this command:

```
$ xcode-select --install
```

Click through the confirmation dialogs that appear (this might take a while, depending on your connection speed). Next, install Homebrew by entering the following command:

```
$ /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

You can find this command at *https://brew.sh/*. Make sure you include a space between `curl -fsSL` and the URL.

NOTE *The `-e` in this command tells Ruby (the programming language Homebrew is written in) to execute the code that's downloaded here. You should only run commands like this from sources you trust.*

To confirm that Homebrew installed correctly, run this command:

```
$ brew doctor
Your system is ready to brew.
```

This output means you're ready to use Homebrew to install packages to your system.

Installing Python

To install the latest version of Python, enter the following command:

```
$ brew install python
```

Check which version was installed by using this command:

```
$ python3 --version
Python 3.7.2
$
```

Now you can start a Python terminal session using the command `python3`. You can also use the `python3` command in your text editor so it runs programs with the version of Python you just installed instead of the system's earlier version. If you need help configuring Sublime Text to use the version you just installed, see the instructions in Chapter 1.

Python on Linux

Python is included by default on almost every Linux system. But if the default version is earlier than Python 3.6, you should install the latest version. The following instructions should work for most apt-based systems.

We'll use a package called `deadsnakes`, which makes it easy to install multiple versions of Python. Enter the following commands:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt-get update
$ sudo apt install python3.7
```

These commands should install Python 3.7 onto your system.

Enter the following command to start a terminal session that runs Python 3.7:

```
$ python3.7
>>>
```

You'll also want to use this command when you configure your text editor and when you run programs from the terminal.

Python Keywords and Built-in Functions

Python comes with its own set of keywords and built-in functions. It's important to be aware of these when you're naming variables: your variable names cannot be the same as these keywords and shouldn't be the same as the function names, or you'll overwrite the functions.

In this section, we'll list Python's keywords and built-in function names, so you'll know which names to avoid.

Python Keywords

Each of the following keywords has a specific meaning, and you'll see an error if you try to use any of them as a variable name.

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Python Built-in Functions

You won't get an error if you use one of the following readily available built-in functions as a variable name, but you'll override the behavior of that function:

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

B

TEXT EDITORS AND IDEs

Programmers spend a lot of time writing, reading, and editing code, and using a text editor or an *integrated development environment (IDE)* to make this work as efficient as possible is essential. A good editor will do simple tasks, like highlight your code's structure so you can catch common bugs as you're working. But it won't do so much that it distracts you from your thinking. Editors also have useful features like automatic indenting, markers to show appropriate line length, and keyboard shortcuts for common operations.

An IDE is a text editor with a number of other tools included, like interactive debuggers and code introspection. An IDE examines your code as you enter it and tries to learn about the project you're building. For example, when you start typing the name of a function, an IDE might show you all the arguments that function accepts. This behavior can be very helpful when everything works and you understand what you're seeing. But it can also be overwhelming as a beginner and difficult to troubleshoot when you aren't sure why your code isn't working in the IDE.

I encourage you to use a simple text editor while you're learning to code. Text editors also put a much lighter load on your system; so if you're working on an older machine or one with fewer resources, a text editor will work better than an IDE. If you're already familiar with IDEs or if people around you use an IDE and you want to use a similar environment, by all means try them out.

Don't worry too much about choosing your tools at this point; your time will be better spent digging into the language and working on the projects you're interested in. Once you've mastered the basics, you'll have a better idea what tools work for you.

In this appendix, we'll set up the Sublime Text text editor to help you work more efficiently. We'll also take a brief look at a number of other editors you might consider using or see other Python programmers using.

Customizing Sublime Text Settings

In Chapter 1, you configured Sublime Text to use the Python version you want it to when running your programs. Now we'll configure it to do some of the tasks mentioned at the beginning of this appendix.

Converting Tabs to Spaces

If you use a mix of tabs and spaces in your code, it can cause problems in your programs that are difficult to diagnose. To avoid this, you can configure Sublime Text to always use spaces for indentation, even when you press the TAB key. Go to **View ▸ Indentation** and make sure the **Indent Using Spaces** option is selected. If it's not, select it. Also, make sure the **Tab Width** is set to **4** spaces.

If you already have a mix of tabs and spaces in one of your programs, you can convert all tabs to spaces by clicking **View ▸ Indentation ▸ Convert Tabs to Spaces**. You can also access these settings by clicking **Spaces** at the bottom right of the Sublime Text window.

Now you can use the TAB key to indent lines of code, and Sublime Text will insert spaces automatically to indent those lines.

Setting the Line Length Indicator

Most editors allow you to set up a visual cue, usually a vertical line, to show where your lines should end. In the Python community, the convention is to limit lines to 79 characters or less. To set this feature, select **View ▸ Ruler**, and then click **80**. Sublime Text will place a vertical line at the 80-character mark to help restrict your code lines to an appropriate length.

Indenting and Unindenting Code Blocks

To indent an entire block of code, highlight it and select **Edit ▸ Line ▸ Indent** or press CTRL-], or ⌘-] on macOS. To unindent a block of code, click **Edit ▸ Line ▸ Unindent** or press CTRL-[, or ⌘-[.

Commenting Out Blocks of Code

To temporarily disable a block of code, you can highlight the block and comment it so Python will ignore it. Select **Edit ▸ Comment ▸ Toggle Comment** (CTRL-/ or ⌘-/). The selected lines will be commented out with a hash mark (#) indented at the same level as the line of code to indicate these are not regular comments. When you want to uncomment the block of code, highlight the block and reissue the same command.

Saving Your Configuration

Some of the settings mentioned only affect the current file you're working in. To make your settings affect all files you open in Sublime Text, you'll need to define your user settings. Select **Sublime Text ▸ Preferences ▸ Settings**, and look for the file *Preferences.sublime-settings – User*. Enter the following in this file:

```
{
    "rulers": [80],
    "translate_tabs_to_spaces": true
}
```

Save this file, and your ruler and tab settings will apply to all files you work with in Sublime Text. If you add more settings to this file, make sure each line ends with a comma except the last line. You can look at other users' settings files online and customize your editor to the settings that help you work most efficiently.

Further Customizations

You can customize Sublime Text in many ways to help you work even more efficiently. As you're exploring the menus, keep an eye out for keyboard shortcuts for the menu items you use most often. Every time you use a keyboard shortcut instead of reaching for the mouse or trackpad, you become a bit more efficient. Don't try to learn everything at once; just try to become efficient with the actions you use most, and be on the lookout for other features that might help you develop your own workflow.

Other Text Editors and IDEs

You'll hear about and see people using a number of other text editors. Most of them can be configured to help you in the same way you customized Sublime Text. Here's a small selection of text editors you might hear about.

IDLE

IDLE is a text editor that's included with Python. It's a little less intuitive to work with than Sublime Text, but you'll see references to it in other tutorials aimed at beginners, so you might want to give it a try.

Geany

Geany is a simple text editor that lets you run all of your programs directly from the editor. It displays all of your output in a terminal window, which helps you become comfortable using terminals. Geany has a very simple interface, but it's powerful enough that a significant number of experienced programmers still use it.

Emacs and Vim

Emacs and Vim are two popular editors favored by many experienced programmers because they're designed to be used so your hands never have to leave the keyboard. This makes writing, reading, and modifying code very efficient once you learn how the editor works. It also means both editors have a fairly steep learning curve. Vim is included on most Linux and macOS machines, and both Emacs and Vim can be run entirely inside a terminal. For this reason, they're often used to write code on servers through a remote terminal session.

Programmers will often recommend that you give them a try. But many proficient programmers forget how much new programmers are already trying to learn. It's beneficial to be aware of these editors, but hold off on using them until you're comfortable working with code in a simpler editor that lets you focus on learning to program rather than learning to use an editor.

Atom

Atom is a text editor with some features that you'd normally find in an IDE. You can open an individual file you're working on, or you can open a project folder and Atom will instantly make all the files in that project easily accessible. Atom is integrated with Git and GitHub, so as you start to use version control you'll be able to work with local and remote repositories from within your editor instead of having to do so in a separate terminal.

Atom allows you to install packages as well, so you can extend its behavior in many ways. A number of these packages incorporate behavior that makes Atom behave more like an IDE.

Visual Studio Code

Visual Studio Code, also called VS Code, is another editor that acts more like an IDE. VS Code supports efficient use of a debugger, has integrated version control support, and also offers code completion tools.

PyCharm

PyCharm is a popular IDE among Python programmers because it was built to work specifically with Python. The full version requires a paid subscription, but a free version called the PyCharm Community Edition is also available that many developers find useful.

PyCharm features a *linter*, which checks that your coding style matches common Python conventions, and offers suggestions when you deviate from normal Python formatting. It also has an integrated debugger to help you resolve errors proficiently and modes that help you work efficiently with a number of popular Python libraries.

Jupyter Notebooks

Jupyter Notebook is a different kind of tool than traditional text editors or IDEs in that it's a web app primarily built of blocks; each block is either a code block or a text block. The text blocks are rendered in Markdown, so you can include simple formatting in your text blocks.

Jupyter Notebooks were developed to support the use of Python in scientific applications, but they have since expanded to become useful in a wide variety of situations. Rather than just writing comments inside a *.py* file, you can write clear text with simple formatting, such as headers, bulleted lists, and hyperlinks in between sections of code. Every code block can be run independently, allowing you to test small pieces of your program, or you can run all the code blocks at once. Each code block has its own output area, and you can toggle the output areas on or off as needed.

Jupyter Notebooks can be confusing at times because of the interactions between different cells. If you define a function in one cell, that function is available to other cells as well. This is beneficial most of the time, but it can be confusing in longer notebooks and if you don't fully understand how the Notebook environment works.

If you're doing any scientific or data-focused work in Python, you'll almost certainly see Jupyter Notebooks at some point.

C

GETTING HELP

Everyone gets stuck at some point when they're learning to program. So one of the most important skills to learn as a programmer is how to get unstuck efficiently. This appendix outlines several ways to help you get going again when programming gets confusing.

First Steps

When you're stuck, your first step should be to assess your situation. Before you ask for help from anyone else, answer the following three questions clearly:

- What are you trying to do?
- What have you tried so far?
- What results have you been getting?

Make your answers as specific as possible. For the first question, explicit statements like "I'm trying to install the latest version of Python on my Windows 10 laptop" are detailed enough for others in the Python community to help you. Statements like "I'm trying to install Python" don't provide enough information for others to offer much help.

Your answer to the second question should provide enough detail so you won't be advised to repeat what you've already tried: "I went to *https://python.org/downloads/* and clicked the Download button for my system. Then I ran the installer," is more helpful than "I went to the Python website and downloaded something."

For the third question, it's helpful to know the exact error messages you received so you can search online for a solution or provide them when asking for help.

Sometimes just answering these three questions before you ask for help from others allows you to see something you're missing and get you unstuck without having to go any further. Programmers even have a name for this: it's called *rubber duck debugging*. The idea is that if you clearly explain your situation to a rubber duck (or any inanimate object), and ask it a specific question, you'll often be able to answer your own question. Some programming shops even keep a real rubber duck around to encourage people to "talk to the duck."

Try It Again

Just going back to the start and trying again can be enough to solve many problems. Say you're trying to write a `for` loop based on an example in this book. You might have only missed something simple, like a colon at the end of the `for` line. Going through the steps again might help you avoid repeating the same mistake.

Take a Break

If you've been working on the same problem for a while, taking a break is one of the best tactics you can try. When we work on the same task for long periods of time, our brains start to zero in on only one solution. We lose sight of the assumptions we've made, and taking a break helps us get a fresh perspective on the problem. It doesn't need to be a long break, just something that gets you out of your current mind-set. If you've been sitting for a long time, do something physical: take a short walk or go outside for a bit; perhaps drink a glass of water or eat a light and healthy snack.

If you're getting frustrated, it might be worth putting your work away for the day. A good night's sleep almost always makes a problem more approachable.

Refer to This Book's Resources

The online resources for this book, available at *https://nostarch.com/pythoncrashcourse2e/*, include a number of helpful sections about setting

up your system and working through each chapter. If you haven't done so already, take a look at these resources and see if there's anything that helps your situation.

Searching Online

Chances are that someone else has had the same problem you're having and has written about it online. Good searching skills and specific inquiries will help you find existing resources to solve the issue you're facing. For example, if you're struggling to install the latest version of Python on Windows 10, searching for *install python windows 10* and limiting the results to resources from the last year might direct you to a clear answer.

Searching the exact error message can be extremely helpful too. For example, say you get the following error when you try to start a Python terminal session:

```
> python
'python' is not recognized as an internal or external command,
operable program or batch file
>
```

Searching for the full phrase "python is not recognized as an internal or external command" will probably yield some good advice.

When you start searching for programming-related topics, a few sites will appear repeatedly. I'll describe some of these sites briefly, so you'll know how helpful they're likely to be.

Stack Overflow

Stack Overflow (*https://stackoverflow.com/*) is one of the most popular question-and-answer sites for programmers, and will often appear in the first page of results on Python-related searches. Members post questions when they're stuck, and other members try to give helpful responses. Users can vote for the responses they find most helpful, so the best answers are usually the first ones you'll find.

Many basic Python questions have very clear answers on Stack Overflow, because the community has refined them over time. Users are encouraged to post updates too, so responses tend to stay relatively current. At the time of this writing, over one million Python-related questions have been answered on Stack Overflow.

The Official Python Documentation

The official Python documentation (*https://docs.python.org/*) is a bit more hit or miss for beginners, because its purpose is more to document the language than to provide explanations. The examples in the official documentation should work, but you might not understand everything shown. Still, it's a good resource to check when it comes up in your searches and will become more useful to you as you continue building your understanding of Python.

Official Library Documentation

If you're using a specific library, such as Pygame, Matplotlib, Django, and so on, links to the official documentation for that project will often appear in searches—for example, *https://docs.djangoproject.com/* is very helpful. If you're planning to work with any of these libraries, it's a good idea to become familiar with their official documentation.

r/learnpython

Reddit is made up of a number of subforums called *subreddits*. The *r/learnpython* subreddit (*https://reddit.com/r/learnpython/*) is fairly active and supportive. Here you can read others' questions and post your own.

Blog Posts

Many programmers maintain blogs and share posts about the parts of the language they're working with. You should skim the first few comments on a blog post to see what reactions other people have had before taking any advice. If no comments appear, take the post with a grain of salt. It's possible no one else has verified the advice.

Internet Relay Chat

Many programmers interact in real time through Internet Relay Chat (IRC). If you're stuck on a problem and searching online isn't providing answers, asking in an IRC channel might be a good option. Most people who hang out in these channels are polite and helpful, especially if you can be specific about what you're trying to do, what you've already tried, and what results you're getting.

Making an IRC Account

To create an account on IRC, go to *https://webchat.freenode.net/*. Choose a nickname, fill out the CAPTCHA box, and click **Connect**. You'll see a message welcoming you to the freenode IRC server. In the box at the bottom of the window, enter the following command:

```
/msg nickserv register password email
```

Enter your own password and email address in place of `password` and `email`. Choose a password that you don't use for any other account. You'll receive an email with instructions to verify your account. The email will provide you with a command like this:

```
/msg nickserv verify register nickname verification_code
```

Paste this line into the IRC site with `nickname` as the name you chose earlier and a value for `verification_code`. Now you're ready to join a channel.

If you have trouble logging into your account at some point, you can issue the following command:

```
/msg nickserv identify nickname password
```

Replace *nickname* and *password* with your own nickname and password. This will authenticate you on the network, and you'll be able to access channels that require an authenticated nickname.

Channels to Join

To join the main Python channel, enter **/join #python** in the input box. You'll see a confirmation that you joined the channel and some general information about the channel.

The channel ##*learnpython* (with two hashtags) is usually quite active as well. This channel is associated with *https://reddit.com/r/learnpython/*, so you'll see messages about posts on *r/learnpython* too. You might want to join the #*django* channel if you're working on web applications.

After you've joined a channel, you can read the conversations other people are having and ask your own questions as well.

IRC Culture

To get effective help, you should know a few details about IRC culture. Focusing on the three questions at the beginning of this appendix will definitely help guide you to a successful solution. People will be happy to help you if you can explain precisely what you're trying to do, what you've already tried, and the exact results you're getting. If you need to share code or output, IRC members use external sites made for this purpose, such as *https://bpaste.net/+python*. (This is where `#python` sends you to share code and output.) This keeps the channels from being flooded with code and also makes it much easier to read the code that people share.

Being patient will always make people more likely to help you. Ask your question concisely, and then wait for someone to respond. Often, people are in the middle of many conversations, but usually someone will address you in a reasonable amount of time. If few people are in the channel, it might take a while to get a response.

Slack

Slack is like a modern-day reinvention of IRC. It's often used for internal company communications, but there are also many public groups you can join. If you want to check out Python Slack groups, start with *https://pyslackers.com/*. Click the **Slack** link at the top of the page, and enter your email address to get an invitation.

Once you're in the Python Developers workspace, you'll see a list of channels. Click **Channels**, and then choose the topics that interest you. You might want to start with the #*learning_python* and #*django* channels.

Discord

Discord is another online chat environment with a Python community where you can ask for help and follow Python-related discussions.

To check it out, head to *https://pythondiscord.com/* and click the **Chat Now** link. You should see a screen with an automatically generated invitation; click **Accept Invite**. If you already have a Discord account, you can log in with your existing account. If you don't have an account, enter a username and follow the prompts to complete your Discord registration.

If this is your first time visiting the Python Discord, you'll need to accept the rules for the community before participating fully. Once you've done that, you can join any of the channels that interest you. If you're looking for help, be sure to post in one of the Python Help channels.

D

USING GIT FOR VERSION CONTROL

Version control software allows you to take snapshots of a project whenever it's in a working state. When you make changes to a project—for example, when you implement a new feature—you can revert back to a previous working state if the project's current state isn't functioning well.

Using version control software gives you the freedom to work on improvements and make mistakes without worrying about ruining your project. This is especially critical in large projects, but can also be helpful in smaller projects, even when you're working on programs contained in a single file.

In this appendix, you'll learn to install Git and use it for version control in the programs you're working on now. Git is the most popular version control software in use today. Many of its advanced tools help teams collaborate on large projects, but its most basic features also work well for solo developers. Git implements version control by tracking the changes made to every file in a project; if you make a mistake, you can just return to a previously saved state.

Installing Git

Git runs on all operating systems, but there are different approaches to installing it on each system. The following sections provide specific instructions for each operating system.

Installing Git on Windows

You can download an installer for Git at *https://git-scm.com/*. You should see a download link for an installer that's appropriate for your system.

Installing Git on macOS

Git might already be installed on your system, so try issuing the command **`git --version`**. If you see output listing a specific version number, Git is installed on your system. If you see a message prompting you to install or update Git, simply follow the onscreen directions.

You can also visit *https://git-scm.com/*, where you should see a download link for an appropriate installer for your system.

Installing Git on Linux

To install Git on Linux, enter the following command:

```
$ sudo apt install git-all
```

That's it. You can now use Git in your projects.

Configuring Git

Git keeps track of who makes changes to a project, even when only one person is working on the project. To do this, Git needs to know your username and email. You must provide a username, but you can make up a fake email address:

```
$ git config --global user.name "username"
$ git config --global user.email "username@example.com"
```

If you forget this step, Git will prompt you for this information when you make your first commit.

Making a Project

Let's make a project to work with. Create a folder somewhere on your system called *git_practice*. Inside the folder, make a simple Python program:

hello_git.py

```
print("Hello Git world!")
```

We'll use this program to explore Git's basic functionality.

Ignoring Files

Files with the extension *.pyc* are automatically generated from *.py* files, so we don't need Git to keep track of them. These files are stored in a directory called *__pycache__*. To tell Git to ignore this directory, make a special file called *.gitignore*—with a dot at the beginning of the filename and no file extension—and add the following line to it:

.gitignore

```
__pycache__/
```

This file tells Git to ignore any file in the *__pycache__* directory. Using a *.gitignore* file will keep your project clutter free and easier to work with.

You might need to modify your text editor's settings so it will show hidden files in order to open *.gitignore*. Some editors are set to ignore filenames that begin with a dot.

Initializing a Repository

Now that you have a directory containing a Python file and a *.gitignore* file, you can initialize a Git repository. Open a terminal, navigate to the *git_practice* folder, and run the following command:

```
git_practice$ git init
Initialized empty Git repository in git_practice/.git/
git_practice$
```

The output shows that Git has initialized an empty repository in *git_practice*. A *repository* is the set of files in a program that Git is actively tracking. All the files Git uses to manage the repository are located in the hidden directory *.git*, which you won't need to work with at all. Just don't delete that directory, or you'll lose your project's history.

Checking the Status

Before doing anything else, let's look at the project's status:

```
  git_practice$ git status
❶ On branch master

  No commits yet

❷ Untracked files:
    (use "git add <file>..." to include in what will be committed)

    .gitignore
    hello_git.py

❸ nothing added to commit but untracked files present (use "git add" to track)
  git_practice$
```

In Git, a *branch* is a version of the project you're working on; here you can see that we're on a branch named `master` ❶. Each time you check your project's status, it should show that you're on the branch `master`. You then see that we're about to make the initial commit. A *commit* is a snapshot of the project at a particular point in time.

Git informs us that untracked files are in the project ❷, because we haven't told it which files to track yet. Then we're told that there's nothing added to the current commit, but untracked files are present that we might want to add to the repository ❸.

Adding Files to the Repository

Let's add the two files to the repository, and check the status again:

```
❶ git_practice$ git add .
❷ git_practice$ git status
  On branch master

  No commits yet

  Changes to be committed:
    (use "git rm --cached <file>..." to unstage)

❸   new file:   .gitignore
    new file:   hello_git.py

  git_practice$
```

The command `git add .` adds all files within a project that aren't already being tracked to the repository ❶. It doesn't commit the files; it just tells Git to start paying attention to them. When we check the status of the project now, we can see that Git recognizes some changes that need to be committed ❷. The label *new file* means these files were newly added to the repository ❸.

Making a Commit

Let's make the first commit:

```
❶ git_practice$ git commit -m "Started project."
❷ [master (root-commit) ee76419] Started project.
❸  2 files changed, 4 insertions(+)
   create mode 100644 .gitignore
   create mode 100644 hello_git.py
❹ git_practice$ git status
  On branch master
  nothing to commit, working tree clean
  git_practice$
```

We issue the command `git commit -m "message"` ❶ to take a snapshot of the project. The `-m` flag tells Git to record the message that follows (`"Started project."`) in the project's log. The output shows that we're on the `master` branch ❷ and that two files have changed ❸.

When we check the status now, we can see that we're on the `master` branch, and we have a clean working tree ❹. This is the message you want to see each time you commit a working state of your project. If you get a different message, read it carefully; it's likely you forgot to add a file before making a commit.

Checking the Log

Git keeps a log of all commits made to the project. Let's check the log:

```
git_practice$ git log
commit a9d74d87f1aa3b8f5b2688cb586eac1a908cfc7f (HEAD -> master)
Author: Eric Matthes <eric@example.com>
Date:   Mon Jan 21 21:24:28 2019 -0900

    Started project.
git_practice$
```

Each time you make a commit, Git generates a unique, 40-character reference ID. It records who made the commit, when it was made, and the message recorded. You won't always need all of this information, so Git provides an option to print a simpler version of the log entries:

```
git_practice$ git log --pretty=oneline
ee76419954379819f3f2cacafd15103ea900ecb2 (HEAD -> master) Started project.
git_practice$
```

The `--pretty=oneline` flag provides the two most important pieces of information: the reference ID of the commit and the message recorded for the commit.

The Second Commit

To see the real power of version control, we need to make a change to the project and commit that change. Here we'll just add another line to *hello_git.py*:

hello_git.py

```
print("Hello Git world!")
print("Hello everyone.")
```

When we check the status of the project, we'll see that Git has noticed the file that changed:

```
  git_practice$ git status
❶ On branch master
```

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

❷  modified:   hello_git.py

❸ no changes added to commit (use "git add" and/or "git commit -a")
git_practice$
```

We see the branch we're working on ❶, the name of the file that was modified ❷, and that no changes have been committed ❸. Let's commit the change and check the status again:

```
❶ git_practice$ git commit -am "Extended greeting."
[master 51f0fe5] Extended greeting.
 1 file changed, 1 insertion(+), 1 deletion(-)
❷ git_practice$ git status
On branch master
nothing to commit, working tree clean
❸ git_practice$ git log --pretty=oneline
51f0fe5884e045b91c12c5449fabf4ad0eef8e5d (HEAD -> master) Extended greeting.
ee76419954379819f3f2cacafd15103ea900ecb2 Started project.
git_practice$
```

We make a new commit, passing the `-am` flags when we use the command `git commit` ❶. The `-a` flag tells Git to add all modified files in the repository to the current commit. (If you create any new files between commits, simply reissue the `git add .` command to include the new files in the repository.) The `-m` flag tells Git to record a message in the log for this commit.

When we check the project's status, we see that we once again have a clean working directory ❷. Finally, we see the two commits in the log ❸.

Reverting a Change

Now let's look at how to abandon a change and revert back to the previous working state. First, add a new line to *hello_git.py*:

hello_git.py

```
print("Hello Git world!")
print("Hello everyone.")

print("Oh no, I broke the project!")
```

Save and run this file.

We check the status and see that Git notices this change:

```
git_practice$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
```

```
  (use "git checkout -- <file>..." to discard changes in working directory)

❶     modified:   hello_git.py

no changes added to commit (use "git add" and/or "git commit -a")
git_practice$
```

Git sees that we modified *hello_git.py* ❶, and we can commit the change if we want to. But this time, instead of committing the change, we'll revert back to the last commit when we knew our project was working. We won't do anything to *hello_git.py*: we won't delete the line or use the Undo feature in the text editor. Instead, enter the following commands in your terminal session:

```
git_practice$ git checkout .
git_practice$ git status
On branch master
nothing to commit, working tree clean
git_practice$
```

The command `git checkout` allows you to work with any previous commit. The command `git checkout .` abandons any changes made since the last commit and restores the project to the last committed state.

When you return to your text editor, you'll see that *hello_git.py* has changed back to this:

```
print("Hello Git world!")
print("Hello everyone.")
```

Although going back to a previous state might seem trivial in this simple project, if we were working on a large project with dozens of modified files, all the files that had changed since the last commit would be reverted. This feature is incredibly useful: you can make as many changes as you want when implementing a new feature, and if they don't work, you can discard them without affecting the project. You don't have to remember those changes and manually undo them. Git does all of that for you.

NOTE *You might have to refresh the file in your editor to see the previous version.*

Checking Out Previous Commits

You can check out any commit in your log, not just the most recent, by including the first six characters of the reference ID instead of a dot. By checking out and reviewing an earlier commit, you can then return to the latest commit or abandon your recent work and pick up development from the earlier commit:

```
git_practice$ git log --pretty=oneline
51f0fe5884e045b91c12c5449fabf4ad0eef8e5d (HEAD -> master) Extended greeting.
```

```
ee76419954379819f3f2cacafd15103ea900ecb2 Started project.
git_practice$ git checkout ee7641
Note: checking out 'ee7641'.

❶ You are in 'detached HEAD' state. You can look around, make experimental
  changes and commit them, and you can discard any commits you make in this
  state without impacting any branches by performing another checkout.

  If you want to create a new branch to retain commits you create, you may
  do so (now or later) by using -b with the checkout command again. Example:

    git checkout -b <new-branch-name>

  HEAD is now at ee7641... Started project.
  git_practice$
```

When you check out a previous commit, you leave the master branch and enter what Git refers to as a *detached HEAD* state ❶. *HEAD* is the current committed state of the project; you're *detached* because you've left a named branch (`master`, in this case).

To get back to the `master` branch, you check it out:

```
git_practice$ git checkout master
Previous HEAD position was ee76419 Started project.
Switched to branch 'master'
git_practice$
```

This command brings you back to the `master` branch. Unless you want to work with some more advanced features of Git, it's best not to make any changes to your project when you've checked out an old commit. However, if you're the only one working on a project and you want to discard all of the more recent commits and go back to a previous state, you can reset the project to a previous commit. Working from the `master` branch, enter the following:

```
❶ git_practice$ git status
  On branch master
  nothing to commit, working directory clean
❷ git_practice$ git log --pretty=oneline
  51f0fe5884e045b91c12c5449fabf4ad0eef8e5d (HEAD -> master) Extended greeting.
  ee76419954379819f3f2cacafd15103ea900ecb2 Started project.
❸ git_practice$ git reset --hard ee76419
  HEAD is now at ee76419 Started project.
❹ git_practice$ git status
  On branch master
  nothing to commit, working directory clean
❺ git_practice$ git log --pretty=oneline
  ee76419954379819f3f2cacafd15103ea900ecb2 (HEAD -> master) Started project.
  git_practice$
```

We first check the status to make sure we're on the `master` branch ❶. When we look at the log, we see both commits ❷. We then issue the `git`

`reset --hard` command with the first six characters of the reference ID of the commit we want to revert to permanently ❸. We check the status again and see we're on the `master` branch with nothing to commit ❹. When we look at the log again, we see that we're at the commit we wanted to start over from ❺.

Deleting the Repository

Sometimes you'll mess up your repository's history and won't know how to recover it. If this happens, first consider asking for help using the methods discussed in Appendix C. If you can't fix it and you're working on a solo project, you can continue working with the files but get rid of the project's history by deleting the *.git* directory. This won't affect the current state of any of the files, but it will delete all commits, so you won't be able to check out any other states of the project.

To do this, either open a file browser and delete the *.git* repository or delete it from the command line. Afterwards, you'll need to start over with a fresh repository to start tracking your changes again. Here's what this entire process looks like in a terminal session:

```
❶ git_practice$ git status
  On branch master
  nothing to commit, working directory clean
❷ git_practice$ rm -rf .git
❸ git_practice$ git status
  fatal: Not a git repository (or any of the parent directories): .git
❹ git_practice$ git init
  Initialized empty Git repository in git_practice/.git/
❺ git_practice$ git status
  On branch master

  No commits yet

  Untracked files:
    (use "git add <file>..." to include in what will be committed)

    .gitignore
    hello_git.py

  nothing added to commit but untracked files present (use "git add" to track)
❻ git_practice$ git add .
  git_practice$ git commit -m "Starting over."
  [master (root-commit) 6baf231] Starting over.
   2 files changed, 4 insertions(+)
   create mode 100644 .gitignore
   create mode 100644 hello_git.py
❼ git_practice$ git status
  On branch master
  nothing to commit, working tree clean
  git_practice$
```

We first check the status and see that we have a clean working tree ❶. Then we use the command `rm -rf .git` to delete the *.git* directory (`rmdir /s .git` on Windows) ❷. When we check the status after deleting the *.git* folder, we're told that this is not a Git repository ❸. All the information Git uses to track a repository is stored in the *.git* folder, so removing it deletes the entire repository.

We're then free to use `git init` to start a fresh repository ❹. Checking the status shows that we're back at the initial stage, awaiting the first commit ❺. We add the files and make the first commit ❻. Checking the status now shows us that we're on the new `master` branch with nothing to commit ❼.

Using version control takes a bit of practice, but once you start using it you'll never want to work without it again.

INDEX

Symbols

+ (addition), 26
* (asterisk) operator, 147
{} (braces), 92, 105
/ (division), 26
** (double asterisk) operator, 149
== (equality operator), 72–73
** (exponent), 26
// (floor division), 260
> (greater than), 75
>= (greater than or equal to), 75
(hash mark), for comments, 29
!= (inequality operator), 74
< (less than), 75
<= (less than or equal to), 75
% (modulo operator), 116–117, 122
* (multiplication), 26
\n (newline), 22
! (not), 74
+= operator, 115
[] (square brackets), 34
- (subtraction), 26
\t (tab), 22

A

addition (+), 26
aliases, 152
alice.py, 197–199
Alien Invasion project. *See also* Pygame
  aliens
    checking edges, 266
    collisions, with bullets, 268–269, 291–292
    collisions, with ship, 272–275
    controlling fleet direction, 266
    creating an alien, 256
    creating the fleet, 258–264
    dropping the fleet, 267
    reaching bottom of screen, 276
    rebuilding the fleet, 270
  bullets, 246–252
    collisions, with aliens, 268–269, 291–292
    deleting old, 250
    firing, 249
    limiting number of, 251
    making larger, 270
    settings, 247
    speeding up, 271
  classes
    Alien, 256–258
    Bullet, 247–248
    Button, 280–281
    GameStats, 273
    Scoreboard, 288–289
    Settings, 231
    Ship, 233–235
  ending the game, 276
  files
    alien_invasion.py, 229
    bullet.py, 247
    button.py, 280
    game_stats.py, 273
    scoreboard.py, 288
    settings.py, 231
    ship.bmp, 233
  initializing dynamic settings, 286
  levels
    adding, 285–287
    modifying speed settings, 285
    resetting the speed, 287
  planning, 228

Alien Invasion project, *continued*
Play button
adding, 280–285
deactivating, 284
drawing, 281
hiding the mouse cursor, 284
resetting the game, 283
starting the game, 283
scoring, 288–300
all hits, 292
high score, 294
increasing point values, 292
level, 296–298
number of ships, 298–300
resetting, 291
rounding and formatting, 293–294
`score` attribute, 288
updating, 291
settings, storing, 231
ship
adjusting speed, 241–243
continuous movement, 239–241
finding an image, 232
limiting range, 243
amusement_park.py, 80–83
`and` keyword, 75
API (application programming interface), 359
calls, 359
for GitHub, 371
for Hacker News, 372–375
processing responses, 361–365
rate limits, 365
requesting data, 360
visualizing results, 366–371
apostrophe.py, 24
`append()` method, 37
application programming interface. *See* API (application programming interface)
arguments, 131–137. *See also* functions: arguments
arithmetic, 26
`as` keyword, 152
assert methods, 212, 216
asterisk (*) operator, 147
attributes, 159. *See also* classes: attributes

B

banned_users.py, 77
bicycles.py, 34–36
.bmp (bitmap) image files, 232
body
of a function, 130
of an HTML file, 440
Boolean values, 77, 456
Bootstrap, 438–447
braces ({}), 92, 105
built-in functions, 471

C

CamelCase, 181
car.py, 162–179
cars.py, 43–45, 72
cities.py, 121
classes
attributes, 159
accessing, 160
default values, 163
modifying, 164–166
creating, 158–162
importing, 174–179
all classes from a module, 177
multiple classes, 175–177
single class, 174–175
inheritance, 167–173
attributes and methods, 169
child classes, 167
`__init__()` method, 167–169
instances as attributes, 170–172
overriding methods, 170
parent classes, 167
subclasses, 168
`super()` function, 168
superclasses, 168
instances, 157
methods, 159
calling, 160
`__init__()` method, 159

modeling real-world objects, 173
multiple instances, 161
naming conventions, 159
objects, 157
styling guidelines, 181
comma-separated value files. *See* CSV (comma-separated value) files
comment.py, 29
comments, 29–30
conditional tests, 72–77. *See also* `if` statements
confirmed_users.py, 124
constants, 28
counting.py, 118, 122
CSV (comma-separated value) files, 334–346
error-checking, 343–345
parsing headers, 334–335
reading data, 336

D

data analysis, 305
databases. *See* Django: databases
data visualization, 305. *See also* Matplotlib; Plotly
`datetime` module, 337–339
death_valley_highs_lows.py, 343–345
decorators, 429
default values, 134
class attributes, 163
function parameters, 134
`def` keyword, 130
`del` statement, 39
dice_visual.py, 328–330
dictionaries
defining, 92
empty, 94
formatting larger, 97
`get()` method, 98
`KeyError`, 98
key-value pairs, 92–99
adding, 93
removing, 96
looping through, 99–105
keys, 101
keys in order, 103
key-value pairs, 99
values, 104
ordering in, 94
sorting a list of, 374
values
accessing, 93
modifying, 95
die.py, 324
die_visual.py, 325–327
dimensions.py, 66–67
Discord, 48
div (HTML), 441
division (`/`), 26
division_calculator.py, 194–197
Django, 379. *See also* Heroku; Learning Log project
admin site, 387–392
registering models, 388, 391
associating data with users, 435
Bootstrap, 438–447
card, 446
collapsible navigation, 440
container element, 443
django-bootstrap4 app, 438
HTML headers, 439–440
jumbotron, 443
navigation bar, 440–442
styling forms, 444–445
commands
`createsuperuser`, 388
`flush`, 433
`makemigrations`, 387, 391, 432
`migrate`, 383
`shell`, 392
`startapp`, 385, 421
`startproject`, 382
creating a project, 381
databases
cascading delete, 390
creating, 382
foreign keys, 390
many-to-one relationships, 390
migrating, 383, 391
queries, 404, 433
querysets, 392–393, 401
required (non-nullable) fields, 432

Django, *continued*
databases, *continued*
resetting, 433
SQLite, 383
decorators, 429
deployment. *See* Heroku
development server, 383, 389
documentation
Django, 379
models, 386
queries, 394
templates, 406
forms, 410–420
`action` argument, 413
cross-site request forgery, 413
displaying, 413
GET and POST requests, 412
`ModelForm`, 410, 414
pre-filling with data, 419
processing, 412, 416
validation, 410
widgets, 414
`get_object_or_404()` method, 460
hashes (for passwords), 388
HTML
div element, 441
main element, 442
margin, 443
padding, 443
span element, 442
HTTP 404 error, 434
`INSTALLED_APPS`, 386, 421, 438
installing, 381
localhost, 383
logging out, 424
login page, 422
`@login_required`, 429
login template, 422
mapping URLs, 395–396
migrating the database, 383, 391
models, 385–393, 431
privileges, 387
projects (vs. apps), 384
`redirect()` function, 411
registration page, 426–428
release cycle, 381
restricting access to data, 433–435
restricting access to pages, 428–435
settings.py
`INSTALLED APPS`, 386, 421, 438
`LOGIN_URL`, 429
`SECRET_KEY`, 461
shell, 392, 431
starting an app, 385
starting a new project, 382
static files, 448
styling. *See* Django: Bootstrap
superuser, 387
templates, 397
anchor tags, 399
`block` tags, 399
context dictionary, 401
filters, 405
`for` loop, 402
indentation in, 399
inheritance, 398
`linebreaks` filter, 405
template tags, 399
third-party apps, 438
URLs
capturing values, 404
namespaces, 399
URL patterns, 395–396
`url` template tag, 399
user ID values, 431
users
default login view, 422
displaying message to logged-in user, 424
logging in a user, 427
`UserCreationForm`, 427
versions, 381
views, 396
retrieving objects, 401, 404
docstrings, 130
dog.py, 158–162
dot notation, 151, 160
double asterisk (**) operator, 149

E

earthquakes. *See* mapping earthquakes
electric_car.py, 168–173
electric_car.py module, 178
`enumerate()` function, 335
environment variables, 456
epoch time, 366
eq_explore_data.py, 348–351
equality operator (`==`), 72–73
eq_world_map.py, 351–357
even_numbers.py, 58
even_or_odd.py, 117
exceptions, 183, 194–202
  deciding which errors to report, 201
  `else` blocks, 196
  failing silently, 200
  `FileNotFoundError`, 197
  handling, 194
  `try-except` blocks, 194
  using to prevent crashes, 195
  `ZeroDivisionError`, 194
exponents (**), 26

F

favorite_languages.py, 97–98
`FileNotFoundError`, 197
file_reader.py, 184–188
files
  closing, 185
  file paths, 185
    absolute, 186
    relative, 186
  opening, 184
    append mode, 192
    read mode, 192
    write mode, 192
  reading from, 184–190
    entire files, 184–185
    line by line, 187
    making a list of lines, 188
    working with contents, 188
    working with large files, 189
  writing to
    by appending, 193
    empty files, 191
    multiple lines, 192
first_numbers.py, 57
flags, 120
floats, 26
foods.py, 63–65
`for` loops, 49. *See also* dictionaries; lists
formatted_name.py, 138–140
full_name.py, 21–22
functions, 129
  alias (`as`), 152
  arguments, 131–137
    arbitrary keyword, 148
    arbitrary number of, 147
    avoiding errors, 136
    keyword, 133
    lists as, 143–146
    mixing positional and arbitrary, 148
    optional, 138
    order of, 133
    positional, 132–133
  built-in, 471
  calling, 130–137
    equivalent calls, 135–136
    multiple times, 132
  defining, 130
  dictionaries, returning, 140
  lists in
    modifying, 143–145
    protecting, 145
  modules, 150–155
    alias (`as`), 153
    importing all functions (*), 153
    importing entire modules, 150
    importing specific functions, 152
  parameters, default values for, 134
  passing information to. *See* functions: arguments
  return values, 137–142
  styling, 154

G

games. *See* Alien Invasion project; Pygame
Geany, 476
GET requests, 412
getting help
  Discord, 484
  IRC (Internet Relay Chat), 482–483
  official Python documentation, 481
  online resources, 480
  r/learnpython, 482
  rubber duck debugging, 480
  Slack, 483
  Stack Overflow, 481
  three main questions, 479
Git, 360, 450
  branches, 452, 488
  commits, 360, 450
    checking out, 491–493
    making, 452, 457, 460, 488, 490, 493
  configuring, 451, 486
  detached HEAD, 492
  files
    adding, 452, 460, 488, 493
    ignoring, 451, 487
  HEAD, 492
  installing, 450, 486
  log, checking, 489
  repositories, 360
    deleting, 493
    initializing, 452, 487, 493
  reverting changes, 490
  status, checking, 452, 457, 487–493
GitHub, 360
greater than (>), 75
greater than or equal to (>=), 75
greeter.py, 114, 130–131
greet_users.py, 143
`gunicorn` package, 448

H

Hacker News, 372
hash mark (#), for comments, 29
head, of an HTML file, 440
HEAD (Git), 492
Hello World, 9
hello_git.py, 486–491
hello_world.py, 10, 15–19
Heroku, 437. *See also* Django; Git; Learning Log project
  Bash shell, 454
  CLI, installing, 448
  commands
    `config`, 458
    `destroy`, 462
    `login`, 453
    `open`, 453
    `ps`, 453
    `rename`, 455
    `run`, 454
    `set`, 458
  databases, setting up, 454
  `django-heroku` package, 448
  documentation, 453
  environment variables, setting, 456–458
  error pages, custom, 458–460
  free plan, limitations of, 448, 456
  making an account, 448
  Procfile, 450
  projects
    deleting, 461
    pushing to Heroku, 452–453, 457
    viewing live, 453
  Python runtime, specifying, 449
  requirements.txt, 448–449
  securing, 456
  settings.py, modifying for, 450, 456, 459
  superuser, creating, 454–455
  URLs, user-friendly, 455
hidden files, 451

hn_article.py, 372
hn_submissions.py, 373
Homebrew, 469

I

IDE (integrated development environment), 473
IDLE, 475
`if` statements
  `and` keyword, 75
  Boolean expressions, 77
  checking for
    equality (`==`), 72
    equality, ignoring case, 73
    empty lists, 87
    inequality (`!=`), 74
    items in a list, 76
    items not in a list, 77
  conditional tests, 72–77
  `elif` statement, 80–84
  `else` statement, 79
  lists and, 85–88
  numerical comparisons, 74–76
  `or` keyword, 76
  simple, 78
  styling guidelines, 90
  testing multiple conditions, 83–84
immutable, 65
`import *`, 153
`import this`, 30
indentation errors, 53–56
index errors, 47
inequality operator (`!=`), 74
infinite loops, 122
inheritance, 167. *See also* classes: inheritance
`input()` function, 114–117
  numerical input, 115–116
  prompts, 114
`insert()` method, 38
IRC (Internet Relay Chat), 482–483
`itemgetter()` function, 374
`items()` method, 100

J

JSON files
  examining data, 347
  geoJSON file format, 349
  `json.dump()` function, 348
  `json.load()` function, 204
jumbotron, 443

K

`keys()` method, 101
key-value pairs, 92–99. *See also* dictionaries
keyword arguments, 132, 133. *See also* functions
keywords, 471

L

language_survey.py, 217
Learning Log project, 379. *See also* Django; Heroku deployment. *See* Git, Heroku
  files
    404.html, 458
    500.html, 459
    admin.py, 388
    base.html, 399, 403, 423, 425, 428, 439–443
    edit_entry.html, 419
    forms.py, 410, 414
    .gitignore, 451
    index.html, 397, 443
    learning_logs/urls.py, 395, 401, 404, 411, 415, 418
    learning_log/urls.py, 395, 422
    logged_out.html, 425
    login.html, 422–423, 444
    models.py, 385, 390, 431
    new_entry.html, 416
    new_topic.html, 412
    Procfile, 450
    register.html, 427
    requirements.txt, 448
    runtime.txt, 449
    settings.py, 386, 421, 429, 450, 456, 459

Learning Log project, *continued*
files, *continued*
topic.html, 404–405, 417, 420
topics.html, 402, 405, 413, 445
users/urls.py, 422, 426
users/views.py, 426
views.py, 396, 401, 404, 411, 415, 418, 429, 430, 433, 460
pages
edit entry, 418–420
home, 394–398
login, 422–424
logout, 424
new entry, 414–417
new topic, 410–413
register, 426
topic, 403
topics, 400–403
`users` app, 421–428
virtual environment, 380–381
writing a specification (spec), 380
`len()` function, 45
less than (`<`), 75
less than or equal to (`<=`), 75
Linux
Hello World, running, 10
Python
checking installed version, 8
installing, 470
troubleshooting installation, 11
Sublime Text, installing, 9
terminal
running programs from, 12
starting Python session, 9
lists, 33
as arguments, 143–146
copying, 63–65
elements
accessing, 34
accessing last, 35
adding with `append()`, 37
adding with `insert()`, 38
identifying unique, 104
modifying, 36
removing with `del`, 39
removing with `pop()`, 39–41
removing with `remove()`, 41
empty, 38
`enumerate()` function, 335
`for` loops, 49–56
nested, 109, 263
`if` statements and, 85–88
indentation errors, 53–56
indexes, 35
errors, 46–47
negative, 35
`len()` function, 45
list comprehensions, 59
naming, 34
numerical lists, 57–60
`range()` function, 58–59
removing all occurrences of a value, 125
slices, 61–63
sorting, 43–46
`reverse()` method, 45
`sorted()` function, 44
`sort()` method, 43
logical errors, 54

M

macOS
Hello World, running, 10
Python
checking installed version, 7
installing, with Homebrew, 469–470
installing, official version, 7–8
troubleshooting installation, 11
Sublime Text, installing, 8
terminal
running programs from, 12
starting a Python session, 8
magicians.py, 50–53
magic_number.py, 75
making_pizzas.py, 151–153
mapping earthquakes, 347–357
building a world map, 351
colorscales, 354

data
downloading, 347, 358
examining JSON data, 347
extracting locations, 351
extracting magnitudes, 350
geoJSON file format, 349
hover text, 356
latitude-longitude ordering, 349
magnitudes, representing, 353
Scattergeo chart type, 352
Matplotlib, 306–323, 336–347
axes
`axis()` method, 313
`ax` variable, 307
removing, 321
`fig` variable, 307
formatting plots
alpha argument, 342
built-in styles, 310
colormaps, 314
custom colors, 314
labels, 307–308
line thickness, 307
shading, 342
size, 322
gallery, 306
installing, 306
line graphs, 306
`plot()` method, 307, 309
`pyplot` module, 307
saving plots, 315
scatter plots, 310–313
`subplots()` function, 307
methods, 20, 236–237. *See also* classes
modules, 150. *See also* classes: modules; functions: modules
modulo operator (`%`), 116–117, 122
motorcycles.py, 37–42
mountain_poll.py, 126
mpl_squares.py, 306–310
multiplication (`*`), 26
my_car.py, 175
my_cars.py, 177–179
my_electric_car.py, 176

N

name errors, 17
name_function.py, 210–215
name.py, 20
names.py, 210
nesting, 106–112
depth, 110
dictionaries in dictionaries, 110–111
dictionaries in lists, 106–108
lists in dictionaries, 108–110
newline (`\n`), 22
`None`, 99
not (`!`), 74
number_reader.py, 203
numbers, 25–28
arithmetic, 26
comparisons, 74–76
exponents, 26
floats, 26
floor division (`//`), 260
integers, 26
mixing integers and floats, 27
order of operations, 26
`round()` function, 294
underscores in, 28
number_writer.py, 203

O

object-oriented programming, 157. *See also* classes
`open()` function, 184
`or` keyword, 76

P

parameters, 131
parent classes, 167. *See also* classes: inheritance
parrot.py, 114, 118–121
`pass` statement, 200
PEP 8, 68–70, 90, 154
person.py, 140–142
pets.py, 125, 132–136

pip, 228
installing Django, 381
installing Matplotlib, 306
installing Plotly, 324
installing Pygame, 228
installing Requests, 361
pi_string.py, 188–190
pizza.py, 147–148
players.py, 61–63
Plotly, 306
`Bar()` class, 327
`data`, long format, 353
formatting plots
bars, 368
colorscales, 354
hover text, 356, 369–370
marker color, 368
marker size, 353
`Layout()` class, 327
layout, long format, 367
x-axis, 368
y-axis, 368
gallery, 324
histogram, 326
installing, 324
`offline.plot()` function, 327
Python figure reference, 371
Scattergeo chart type, 352
user guide, 371
positional arguments. *See* functions: arguments
POST requests, 412
printing_models.py, 143–146
Project Gutenberg, 198
.py file extension, 16
Pygame. *See also* Alien Invasion project
background colors, 230–231
collisions, 268–269, 272–275, 291–292
colors, 231
creating empty windows, 229
cursor, hiding, 285
displaying text, 280
ending games, 276
event loops, 230
fullscreen mode, 244
groups, 248
drawing all elements in, 259
emptying, 271
looping through, 250
removing elements from, 251
storing elements in, 248
updating all elements in, 249
images, drawing to screen, 235
images, loading, 234
installing, 228
`print()` calls in, 251
quitting, 244
rect objects
creating from scratch, 247
positioning, 234, 247–248, 257–258, 260–261, 263–264
responding to input, 230
keypresses, 238
mouse clicks, 283
screen coordinates, 234
surfaces, 230
testing games, 270
`pyplot` module, 307
Python
`>>>` prompt, 4
built-in functions, 471
documentation, 481
installing
on Linux, 470
on macOS, official, 7–8
on macOS, using Homebrew, 469–470
on Windows, 5–6, 467–469
interpreter, 16
keywords, 471
PEP 8, 68–70, 90, 154
standard library, 180–181
terminal sessions, 4
versions, 4
Zen of, 30–31
Python Enhancement Proposal (PEP), 68
python_repos.py, 361–365
python_repos_visual.py, 366–371

Q

quit values, 118–120

R

random_walk.py, 316–317
random walks, 315–323
    `choice()` function, 317
    coloring points, 319
    `fill_walk()` method, 316
    multiple walks, generating, 318
    plotting, 317
    `RandomWalk` class, 316
    starting and ending points, 320
`range()` function, 57–59
`readlines()` method, 188
`read()` method, 185
refactoring, 206–208, 236
remember_me.py, 204–208
Requests package, 361
return values, 137
r/learnpython, 482
rollercoaster.py, 116
rolling dice, 323–330
    analyzing results, 325
    `Die` class, 324
    different sizes, 329
    `randint()` function, 324
    two dice, 328
rubber duck debugging, 480
rw_visual.py, 317–323

S

scatter_squares.py, 311–315
sets, 104
sitka_highs_lows.py, 340–342
sitka_highs.py, 334–340
Slack, 483
`sleep()` function, 274
slice, 61
`split()` method, 198
square brackets ([]), 34
squares.py, 58, 60
Stack Overflow, 481
storing data, 202–205. *See also* JSON
strings, 19–25
    changing case, 20
    `format()` method, 22
    f-strings, 21
    newlines in, 22
    single and double quotes, 19, 24
    tabs in, 22
    using variables in, 21
    whitespace in, 22–24
`strptime()` method, 338
style guidelines, 68–70
    blank lines, 69
    CamelCase, 181
    classes, 181
    functions, 154
    `if` statements, 90
    indentation, 69
    line length, 69
    PEP 8, 68
Sublime Text, 4–10, 474–475
    commenting out code, 475
    configuring, 9
    customizing, 474
    indenting and unindenting code blocks, 474
    installing, 7–9
    line length indicator, 474
    running Python programs, 9–10
    saving your configuration, 475
    tabs and spaces, 474
subtraction (-), 26
superclasses, 168. *See also* classes: inheritance
survey.py, 217
syntax errors, 24
syntax highlighting, 16

T

tab (\t), 22
testing code, 209–222
    adding tests, 214
    assert methods, 216
    coverage, 211
    failing tests, 212–214
    passing tests, 211–212
    `setUp()` method, 220
    test case, 211
    testing classes, 216–221
    testing functions, 210–215
    `unittest` module, 209
    unit tests, 211

test_name_function.py, 211–215
test_survey.py, 218–221
text editors and IDEs
    Atom, 476
    Emacs and Vim, 476
    Geany, 476
    IDLE, 475
    Jupyter Notebooks, 477
    PyCharm, 476
    Sublime Text, 4–10, 474–475
    Visual Studio Code, 476
toppings.py, 74, 83–88
traceback, 18
`try-except` blocks, 194–202. *See also* exceptions
tuples, 65–67
type errors, 66

U

`unittest` module, 209
unit tests, 211
Unix time, 366
user_profile.py, 148

V

`values()` method, 104
variables, 16–19, 28
    constants, 28
    as labels, 18
    multiple assignment, 28
    naming conventions, 17
    values, 16
version control system, 485. *See also* Git
virtual environment (`venv`), 380
voting.py, 79–80

W

weather data, 334–347
web framework, 379
`while` loops, 118–127
    `active` flag, 120–121
    `break` statement, 121
    infinite, 122
    moving items between lists, 124
    quit values, 118–120
whitespace, 22–24
Windows
    Hello World, running, 10
    Python
        checking installed version, 5
        installing, 5–6, 467–469
        troubleshooting installation, 11, 467–469
    Sublime Text, installing, 7
    terminal
        running programs from, 12
        starting a Python session, 6
`with` statement, 185
word_count.py, 199–201
write_message.py, 191–193
`write()` method, 192

Z

Zen of Python, 30–31
`ZeroDivisionError`, 194

Python Crash Course, 2nd Edition is set in New Baskerville, Futura, Dogma, and The Sans Mono Condensed.

RESOURCES

Visit *https://nostarch.com/pythoncrashcourse2e/* for resources, errata, and more information.

More no-nonsense books from **NO STARCH PRESS**

SERIOUS PYTHON
Black-Belt Advice on Deployment, Scalability, Testing, and More
by JULIEN DANJOU
DECEMBER 2018, 240 PP., $34.95
ISBN 978-1-59327-878-6

PYTHON FLASH CARDS
Syntax, Concepts, and Examples
by ERIC MATTHES
JANUARY 2019, 101 CARDS, $27.95
ISBN 978-1-59327-896-0
full color

MATH ADVENTURES WITH PYTHON
An Illustrated Guide to Exploring Math with Code
by PETER FARRELL
JANUARY 2019, 304 PP., $29.95
ISBN 978-1-59327-867-0
full color

IMPRACTICAL PYTHON PROJECTS
Playful Programming Activities to Make You Smarter
by LEE VAUGHAN
NOVEMBER 2018, 424 PP., $29.95
ISBN 978-1-59327-890-8

LEARN ROBOTICS WITH RASPBERRY PI
Build and Code Your Own Moving, Sensing, Thinking Robots
by MATT TIMMONS-BROWN
JANUARY 2019, 240 PP., $24.95
ISBN 978-1-59327-920-2
full color

MISSION PYTHON
Code a Space Adventure Game!
by SEAN MCMANUS
OCTOBER 2018, 280 PP., $29.95
ISBN 978-1-59327-857-1
full color

1.800.420.7240 or 1.415.863.9900 | sales@nostarch.com | www.nostarch.com

WORLDWIDE BEST SELLER

OVER 500,000 COPIES SOLD

LEARN PYTHON—FAST!

Python Crash Course is the world's best-selling guide to the Python programming language. This fast-paced, thorough introduction to programming with Python will have you writing programs, solving problems, and making things that work in no time.

In the first half of the book, you'll learn basic programming concepts, such as variables, lists, classes, and loops, and practice writing clean code with exercises for each topic. You'll also learn how to make your programs interactive and test your code safely before adding it to a project. In the second half, you'll put your new knowledge into practice with three substantial projects: a *Space Invaders*–inspired arcade game, a set of data visualizations with Python's handy libraries, and a simple web app you can deploy online.

As you work through the book, you'll learn how to:

- Use powerful Python libraries and tools, including Pygame, Matplotlib, Plotly, and Django
- Make 2D games that respond to keypresses and mouse clicks, and that increase in difficulty
- Use data to generate interactive visualizations
- Create and customize web apps and deploy them safely online
- Deal with mistakes and errors so you can solve your own programming problems

This **updated second edition** has been thoroughly revised to reflect the latest in Python code and practices. The first half of the book includes improved coverage of topics like f-strings, constants, and managing data. In the second half, the code for the projects has been updated with better structure, cleaner syntax, and more popular and up-to-date libraries and tools, like Plotly and the latest version of Django. (For a full list of updates, see the Preface.)

If you've been thinking about digging into programming, *Python Crash Course* will get you writing real programs fast. Why wait any longer? Start your engines and code!

ABOUT THE AUTHOR

Eric Matthes is a high school science, math, and programming teacher living in Alaska. He has been writing programs since he was five years old and is the author of the *Python Flash Cards*, also from No Starch Press.

COVERS PYTHON 3.*X*

no starch press

THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com

$39.95 ($53.95 CDN)

ISBN: 978-1-59327-928-8

9 781593 279288 53995

SHELVE IN: PROGRAMMING LANGUAGES/PYTHON