

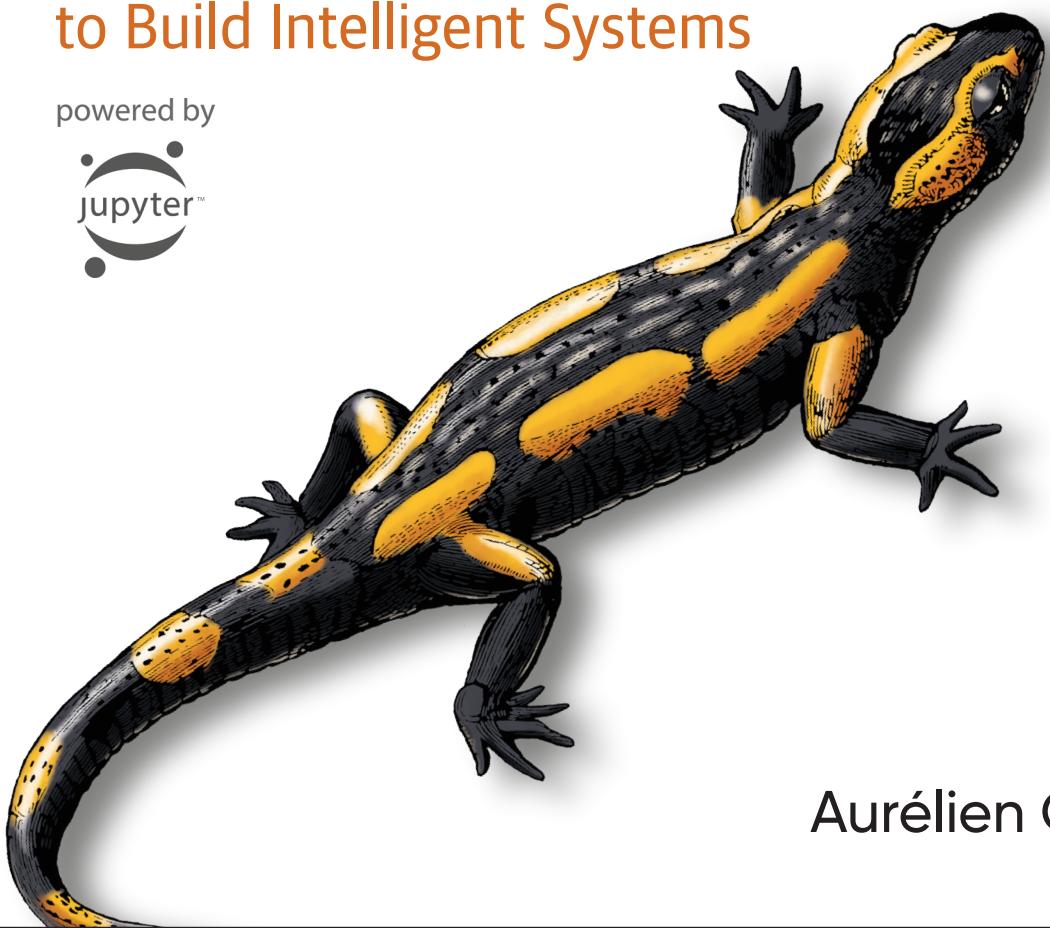
O'REILLY®

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



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



5 7 4 9 9  
9 781492 032649



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](mailto: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

|  |           |
|--|-----------|
| <b>1. The Machine Learning Landscape.....</b>      | <b>1</b>  |
| 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        |
| <b>2. End-to-End Machine Learning Project.....</b> | <b>35</b> |
| 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        |
| <b>3. Classification.....</b>                    | <b>85</b> |
| 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        |
| <b>4. Training Models.....</b>         | <b>111</b> |
| 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        |
| <b>5. Support Vector Machines.....</b> | <b>153</b> |
| 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        |
| <b>6. Decision Trees.....</b>                       | <b>175</b> |
| 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        |
| <b>7. Ensemble Learning and Random Forests.....</b> | <b>189</b> |
| 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        |
| <b>8. Dimensionality Reduction.....</b>             | <b>213</b> |
| 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        |
| <b>9. Unsupervised Learning Techniques.....</b>    | <b>235</b> |
| 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

|   |            |
|---|------------|
| <b>10. Introduction to Artificial Neural Networks with Keras.....</b> | <b>279</b> |
| 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        |
| <b>11. Training Deep Neural Networks.....</b>              | <b>331</b> |
| 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        |
| $\ell_1$ and $\ell_2$ Regularization                       | 364        |
| Dropout  | 365        |
| Monte Carlo (MC) Dropout                                   | 368        |
| Max-Norm Regularization                                    | 370        |
| Summary and Practical Guidelines                           | 371        |
| Exercises  | 373        |
| <b>12. Custom Models and Training with TensorFlow.....</b> | <b>375</b> |
| 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        |
| <b>13. Loading and Preprocessing Data with TensorFlow.....</b>           | <b>413</b> |
| 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        |
| <b>14. Deep Computer Vision Using Convolutional Neural Networks.....</b> | <b>445</b> |
| 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        |
| <b>15. Processing Sequences Using RNNs and CNNs.....</b>            | <b>497</b> |
| 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        |
| <b>16. Natural Language Processing with RNNs and Attention.....</b> | <b>525</b> |
| 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        |
| <b>17. Representation Learning and Generative Learning Using Autoencoders and GANs.</b> | <b>567</b> |
| 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        |
| <b>18. Reinforcement Learning.</b>  | <b>609</b> |
| 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        |
| <b>19. Training and Deploying TensorFlow Models at Scale.....</b> | <b>667</b> |
| 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        |
| <br>  |            |
| <b>A. Exercise Solutions.</b>                           | <b>719</b> |
| <br>  |            |
| <b>B. Machine Learning Project Checklist.</b>           | <b>755</b> |
| <br>  |            |
| <b>C. SVM Dual Problem.</b>                             | <b>761</b> |
| <br>  |            |
| <b>D. Autodiff.</b>                                     | <b>765</b> |
| <br>  |            |
| <b>E. Other Popular ANN Architectures.</b>              | <b>773</b> |
| <br>  |            |
| <b>F. Special Data Structures.</b>                      | <b>783</b> |
| <br>  |            |
| <b>G. TensorFlow Graphs.</b>                            | <b>791</b> |
| <br>  |            |
| <b>Index.</b>   | <b>801</b> |



---

# Preface

## The Machine Learning Tsunami

In 2006, Geoffrey Hinton et al. published [a paper<sup>1</sup>](#) 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,<sup>2</sup> 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!

---

<sup>1</sup> Geoffrey E. Hinton et al., “A Fast Learning Algorithm for Deep Belief Nets,” *Neural Computation* 18 (2006): 1527–1554.

<sup>2</sup> 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](https://www.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* (MLBook) 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](mailto: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](mailto: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 Mongaghan 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



# 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.

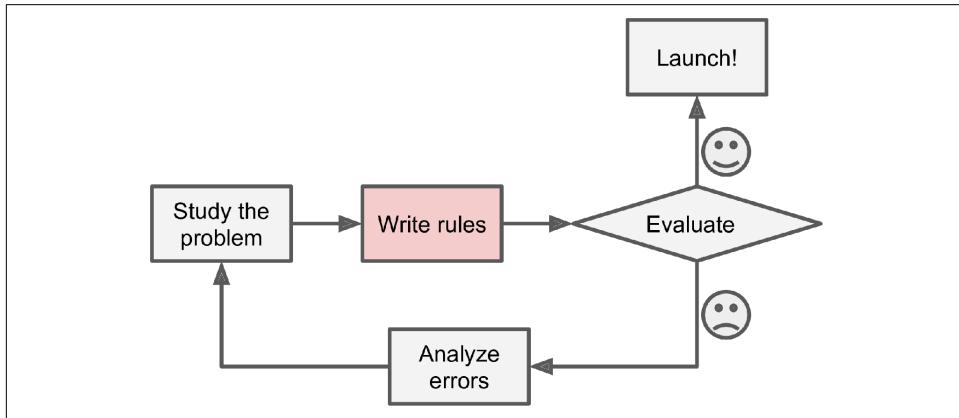
If you just download a copy of Wikipedia, your computer has a lot more data, but it is not suddenly better at any task. Thus, downloading a copy of Wikipedia is not Machine Learning.

## 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*.

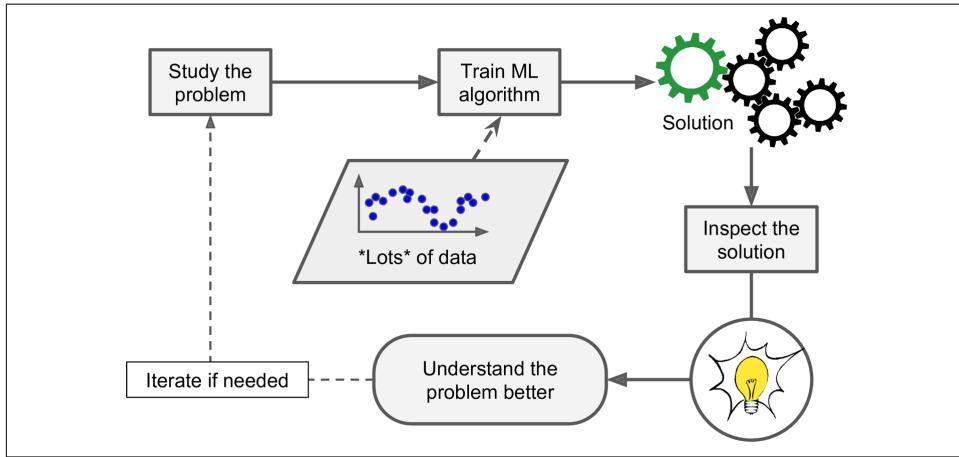


Figure 1-4. Machine Learning can help humans learn

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).

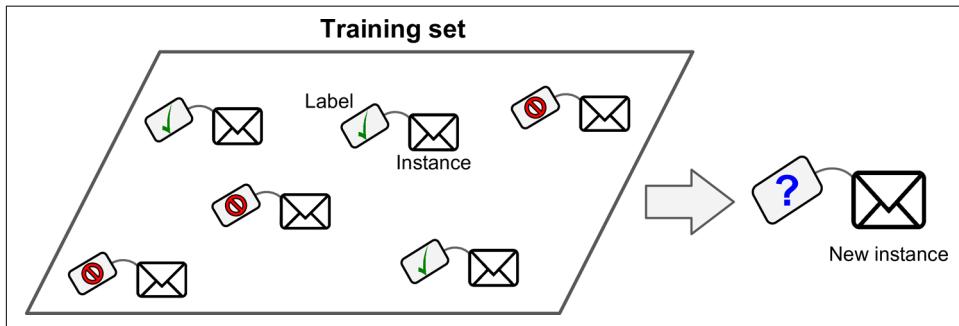


Figure 1-5. A labeled training set for spam classification (an example of supervised learning)

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).<sup>1</sup> 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).

---

<sup>1</sup> 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<sup>2</sup>

## 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.

---

<sup>2</sup> 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.

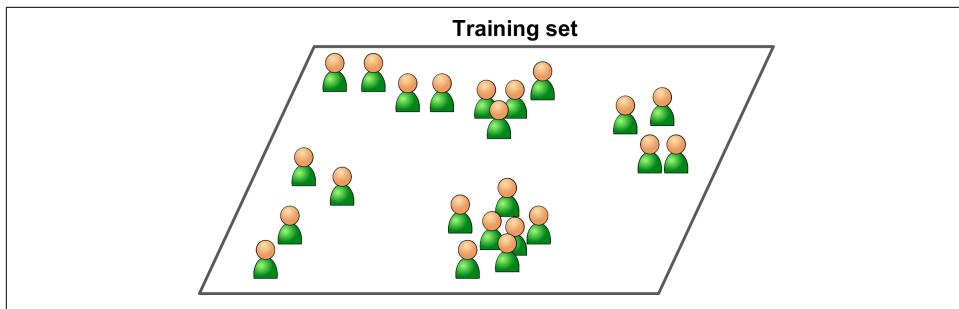
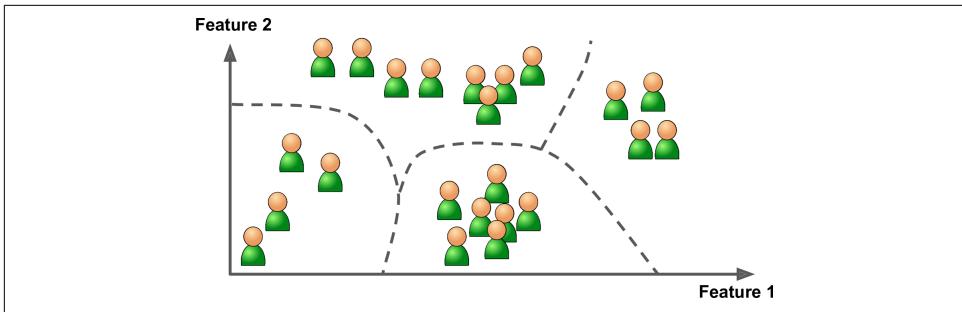


Figure 1-7. An unlabeled training set for unsupervised learning

Here are some of the most important unsupervised learning algorithms (most of these are covered in Chapters 8 and 9):

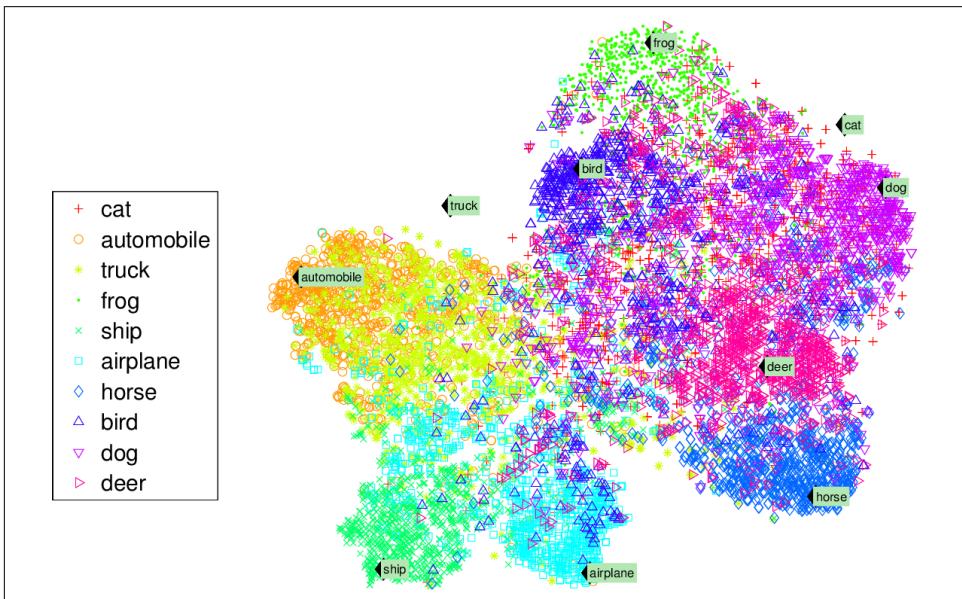
- Clustering
  - K-Means
  - DBSCAN
  - Hierarchical Cluster Analysis (HCA)
- Anomaly detection and novelty detection
  - One-class SVM
  - Isolation Forest
- Visualization and dimensionality reduction
  - Principal Component Analysis (PCA)
  - Kernel PCA
  - Locally Linear Embedding (LLE)
  - t-Distributed 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<sup>3</sup>*

<sup>3</sup> 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).

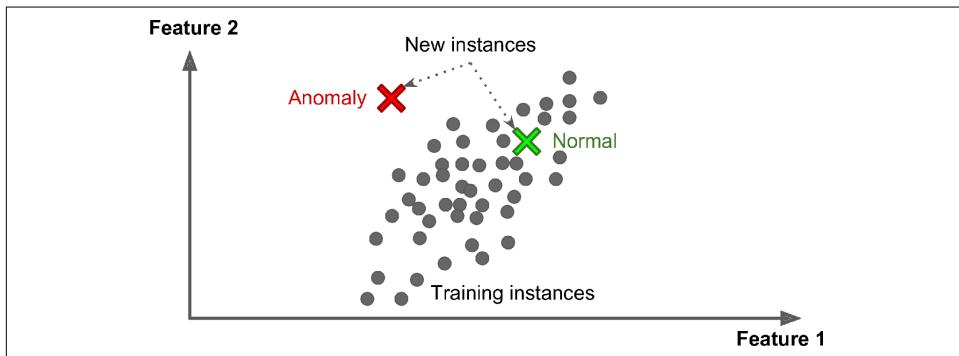


Figure 1-10. Anomaly detection

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).

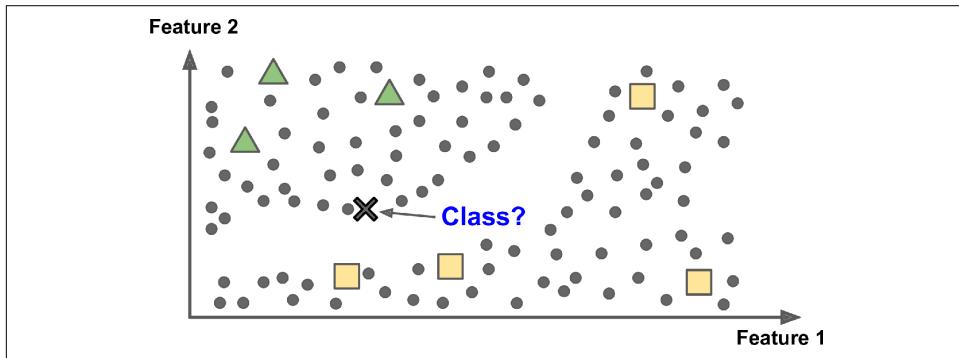


Figure 1-11. Semisupervised learning with two classes (triangles and squares): the unlabeled examples (circles) help classify a new instance (the cross) into the triangle class rather than the square class, even though it is closer to the labeled squares

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<sup>4</sup> 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.

<sup>4</sup> 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.

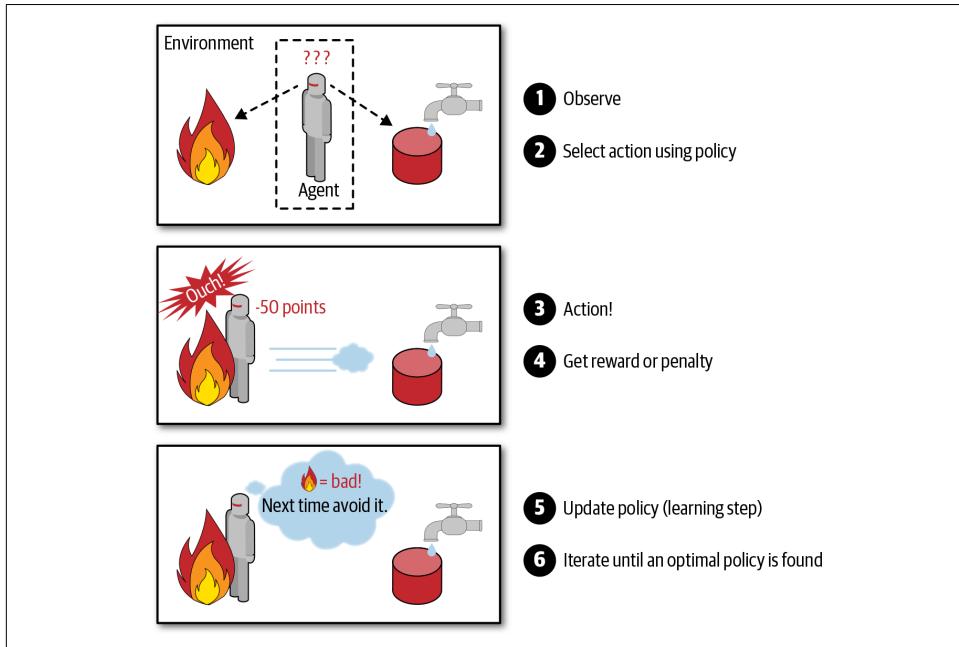


Figure 1-12. Reinforcement Learning

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).

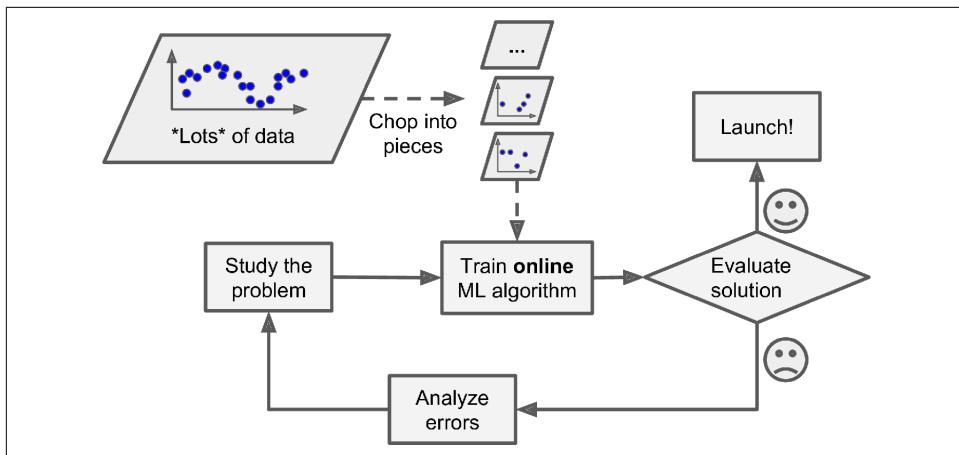


Figure 1-14. Using online learning to handle huge datasets

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.

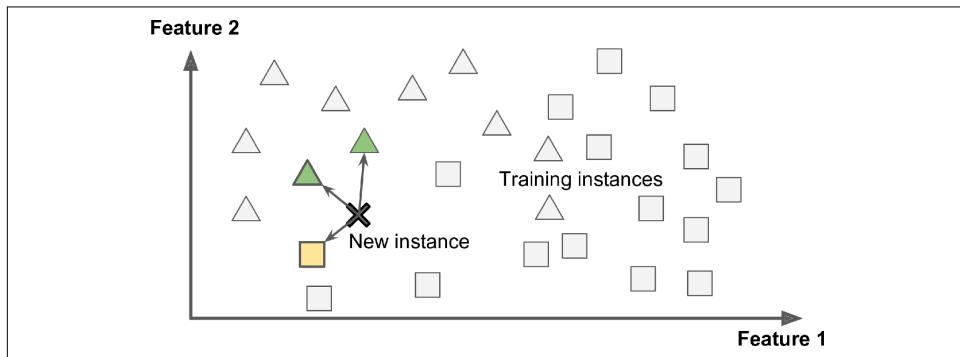
There are two main approaches to generalization: instance-based learning and model-based learning.

### 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.

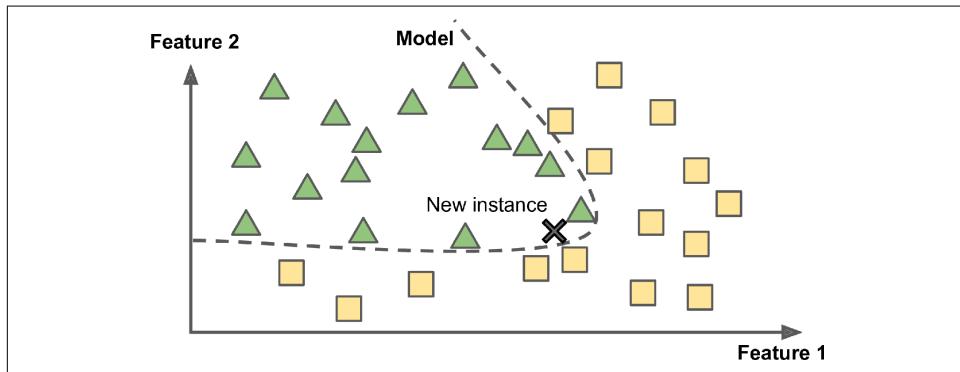
This is called *instance-based learning*: the system learns the examples by heart, then generalizes to new cases by using a similarity measure to compare them to the learned examples (or a subset of them). For example, in [Figure 1-15](#) the new instance would be classified as a triangle because the 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](#)).



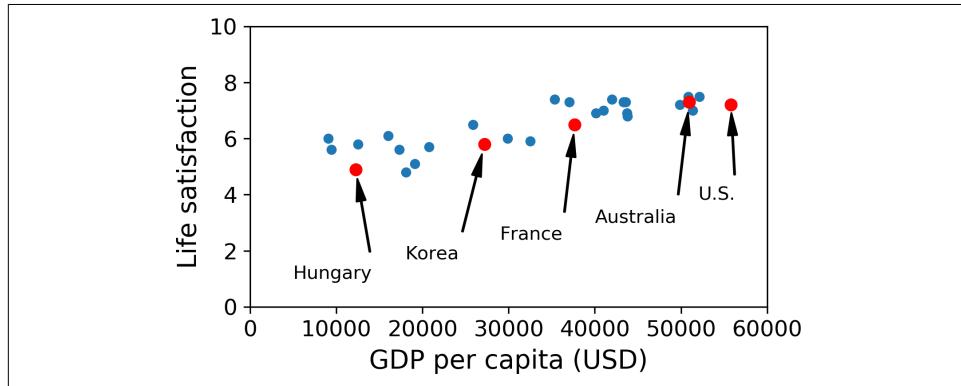
*Figure 1-16. Model-based learning*

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*,  $\theta_0$  and  $\theta_1$ .<sup>5</sup> By tweaking these parameters, you can make your model represent any linear function, as shown in Figure 1-18.

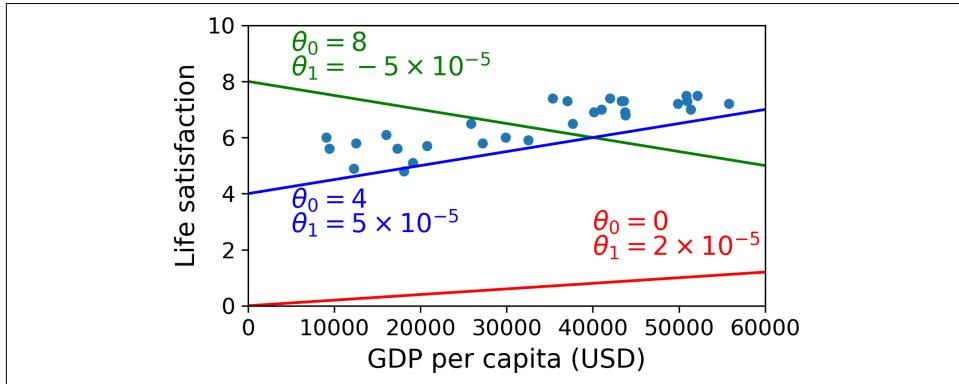


Figure 1-18. A few possible linear models

Before you can use your model, you need to define the parameter values  $\theta_0$  and  $\theta_1$ . 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.

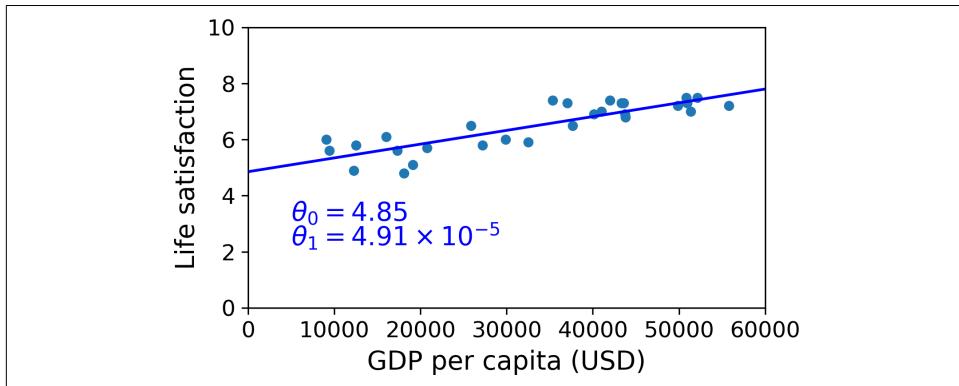
This is where the Linear Regression algorithm comes in: you feed it your training examples, and it finds the parameters that make the linear model fit best to your data. This is called *training* the model. In our case, the algorithm finds that the optimal parameter values are  $\theta_0 = 4.85$  and  $\theta_1 = 4.91 \times 10^{-5}$ .



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).

<sup>5</sup> By convention, the Greek letter  $\theta$  (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,<sup>6</sup> creates a scatterplot for visualization, and then trains a linear model and makes a prediction.<sup>7</sup>

#### *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')
```

<sup>6</sup> 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.

<sup>7</sup> 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.

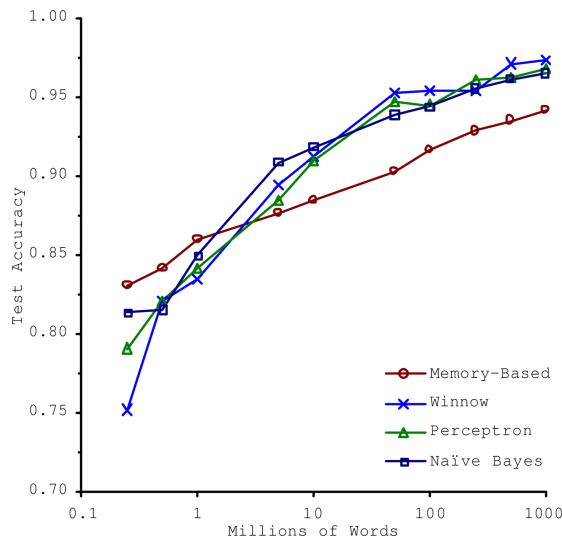
### Insufficient Quantity of Training Data

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<sup>8</sup> once they were given enough data (as you can see in [Figure 1-20](#)).



*Figure 1-20. The importance of data versus algorithms<sup>9</sup>*

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.<sup>10</sup> 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.

<sup>8</sup> For example, knowing whether to write “to,” “two,” or “too,” depending on the context.

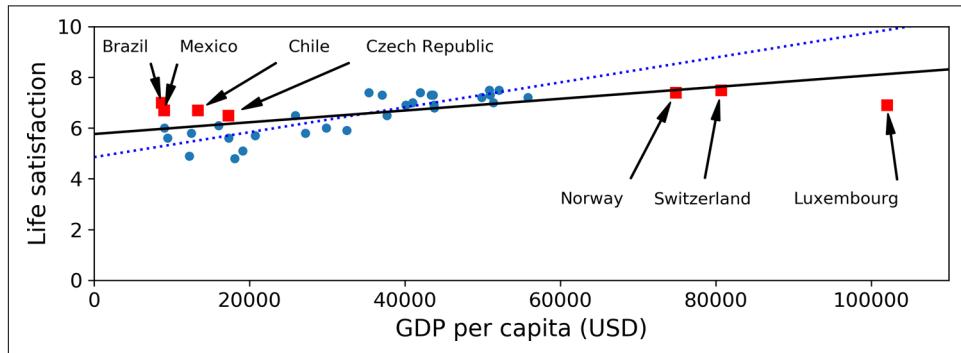
<sup>9</sup> Figure reproduced with permission from Michele Banko and Eric Brill, “Scaling to Very Very Large Corpora for Natural Language Disambiguation,” *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics* (2001): 26–33.

<sup>10</sup> 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?

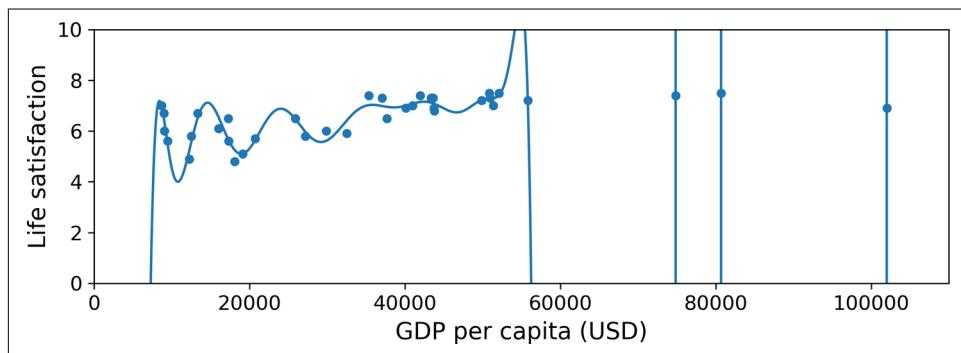


Figure 1-22. Overfitting the training data

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,  $\theta_0$  and  $\theta_1$ . This gives the learning algorithm two *degrees of freedom* to adapt the model to the training data: it can tweak both the height ( $\theta_0$ ) and the slope ( $\theta_1$ ) 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  $\theta_1$  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).

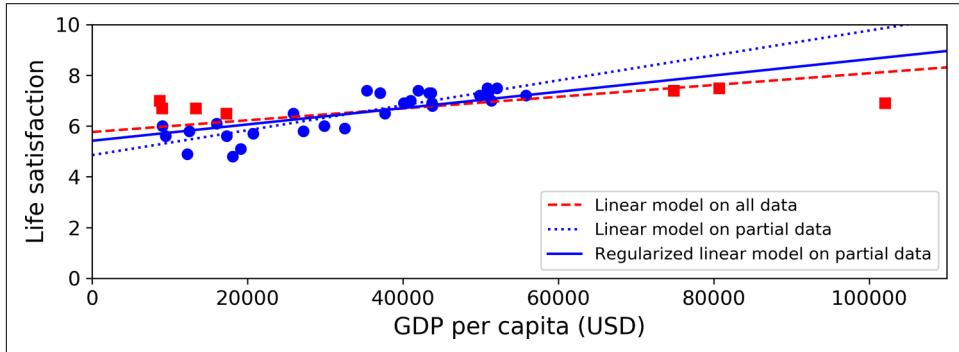


Figure 1-23. Regularization reduces the risk of overfitting

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](#)<sup>11</sup> 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?

---

<sup>11</sup> 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](#).

---

# 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.<sup>1</sup> 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:

---

<sup>1</sup> 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<sup>2</sup> (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.

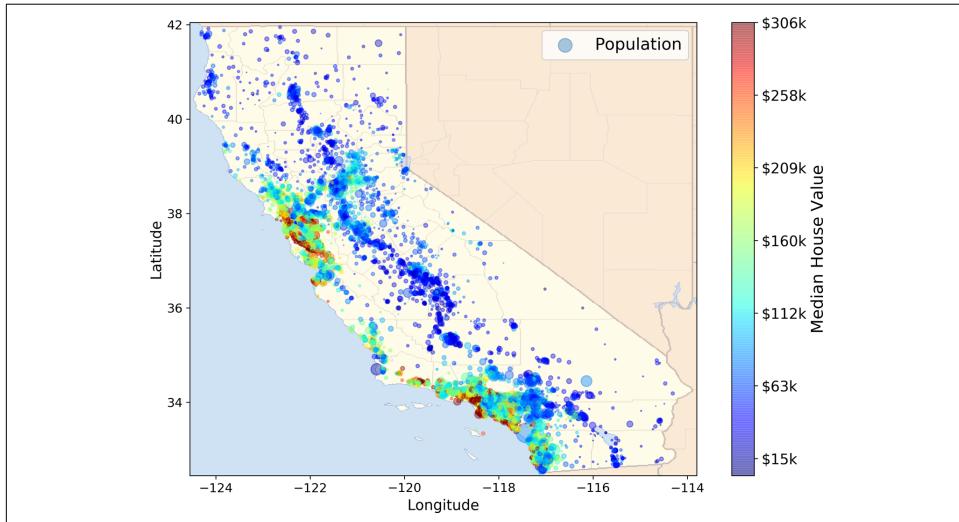


Figure 2-1. California housing prices

---

<sup>2</sup> 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.<sup>3</sup> 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.

---

<sup>3</sup> 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.

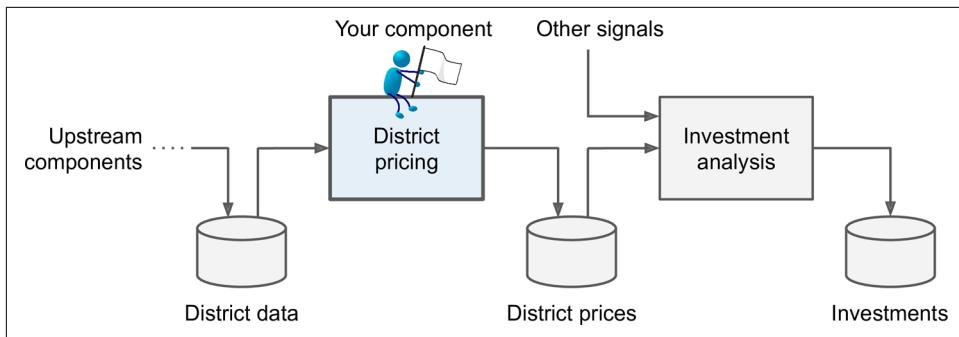


Figure 2-2. A Machine Learning pipeline for real estate investments

## 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)*

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(\mathbf{x}^{(i)}) - y^{(i)})^2}$$

## 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  $i^{\text{th}}$  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  $-118.29^{\circ}$ , latitude  $33.91^{\circ}$ , 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  $i^{\text{th}}$  row is equal to the transpose of  $\mathbf{x}^{(i)}$ , noted  $(\mathbf{x}^{(i)})^T$ .<sup>4</sup>
  - For example, if the first district is as just described, then the matrix  $\mathbf{X}$  looks like this:

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

---

<sup>4</sup> 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  $\hat{y}^{(1)} - y^{(1)} = 2,000$ .
- $\text{RMSE}(\mathbf{X}, h)$  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)*

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m} \sum_{i=1}^m |h(\mathbf{x}^{(i)}) - y^{(i)}|$$

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  $\ell_2$  norm, noted  $\|\cdot\|_2$  (or just  $\|\cdot\|$ ).
- Computing the sum of absolutes (MAE) corresponds to the  $\ell_1$  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  $\ell_k$  norm of a vector  $\mathbf{v}$  containing  $n$  elements is defined as  $\|\mathbf{v}\|_k = (\|v_0\|^k + \|v_1\|^k + \dots + \|v_n\|^k)^{1/k}$ .  $\ell_0$  gives the number of nonzero elements in the vector, and  $\ell_\infty$  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/>.<sup>5</sup>

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

---

<sup>5</sup> 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).<sup>6</sup> 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):<sup>7</sup>

```
$ 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<sup>8</sup> 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:

---

<sup>6</sup> 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.

<sup>7</sup> 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).

<sup>8</sup> Alternative tools include venv (very similar to virtualenv and included in the standard library), virtualenv-wrapper (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<sup>9</sup> (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.

---

<sup>9</sup> Note that Jupyter can handle multiple versions of Python, and even many other languages such as R or Octave.

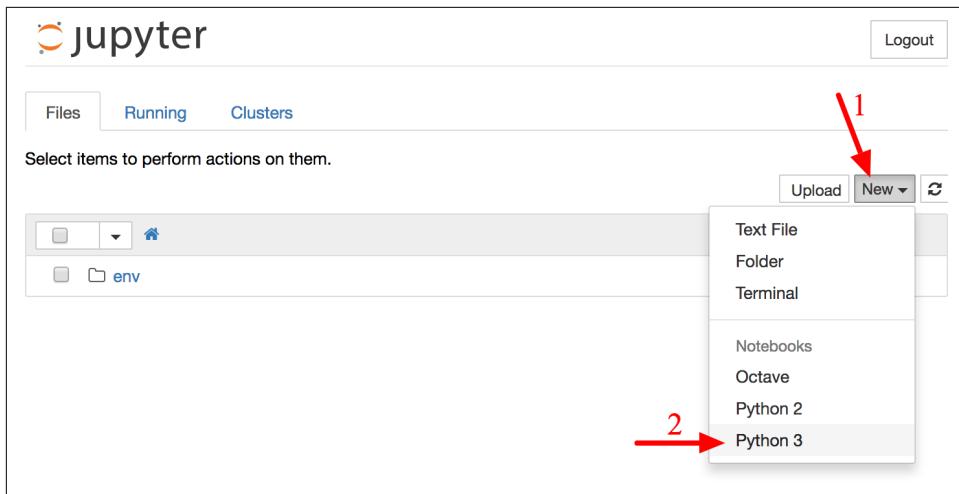


Figure 2-3. Your workspace in Jupyter

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.

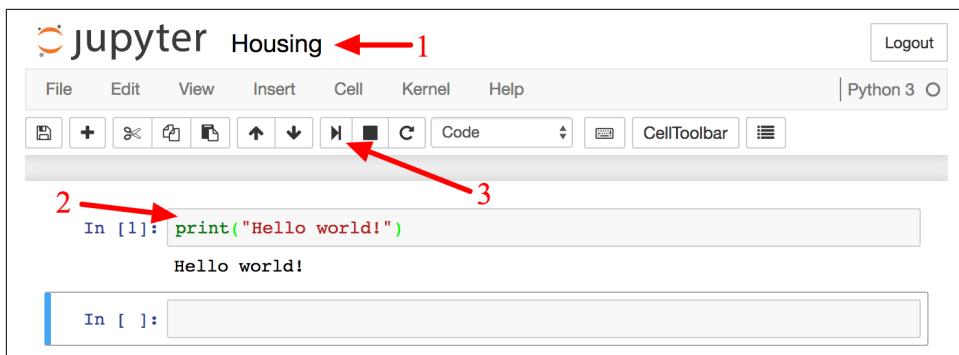


Figure 2-4. Hello world Python notebook

## 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<sup>10</sup> 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:<sup>11</sup>

```
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.

---

<sup>10</sup> You might also need to check legal constraints, such as private fields that should never be copied to unsafe data stores.

<sup>11</sup> 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).

The screenshot shows a Jupyter Notebook interface. In the 'In [5]' cell, the code `housing = load_housing_data()  
housing.head()` is written. In the 'Out[5]' cell, the resulting DataFrame is displayed, showing the first five rows of data. The columns are labeled: longitude, latitude, housing\_median\_age, total\_rooms, total\_bedrooms, and population. The data rows are indexed from 0 to 4.

|   | longitude | latitude | housing_median_age | total_rooms | total_bedrooms | population |
|---|-----------|----------|--------------------|-------------|----------------|------------|
| 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_bedrooms |
| <b>count</b> | 20640.000000       | 20640.000000 | 20640.000000 |                    | 20640.000000 | 20433.000000   |
| <b>mean</b>  | -119.569704        | 35.631861    | 28.639486    |                    | 2635.763081  | 537.870553     |
| <b>std</b>   | 2.003532           | 2.135952     | 12.585558    |                    | 2181.615252  | 421.385070     |
| <b>min</b>   | -124.350000        | 32.540000    | 1.000000     |                    | 2.000000     | 1.000000       |
| <b>25%</b>   | -121.800000        | 33.930000    | 18.000000    |                    | 1447.750000  | 296.000000     |
| <b>50%</b>   | -118.490000        | 34.260000    | 29.000000    |                    | 2127.000000  | 435.000000     |
| <b>75%</b>   | -118.010000        | 37.710000    | 37.000000    |                    | 3148.000000  | 647.000000     |
| <b>max</b>   | -114.310000        | 41.950000    | 52.000000    |                    | 39320.000000 | 6445.000000    |

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.<sup>12</sup> 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()
```

---

<sup>12</sup> The standard deviation is generally denoted  $\sigma$  (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  $1\sigma$  of the mean, 95% within  $2\sigma$ , and 99.7% within  $3\sigma$ .



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.

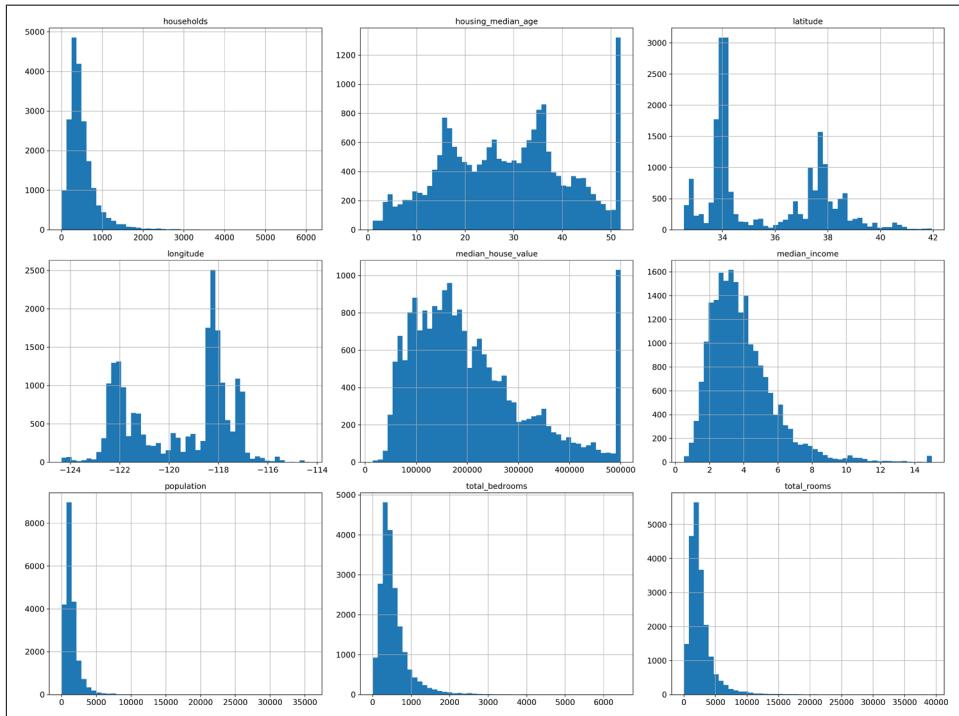


Figure 2-8. A histogram for each numerical attribute

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:<sup>13</sup>

```

>>> 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)`)<sup>14</sup> 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

```

---

<sup>13</sup> 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.

<sup>14</sup> 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:<sup>15</sup>

```
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

---

<sup>15</sup> 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.

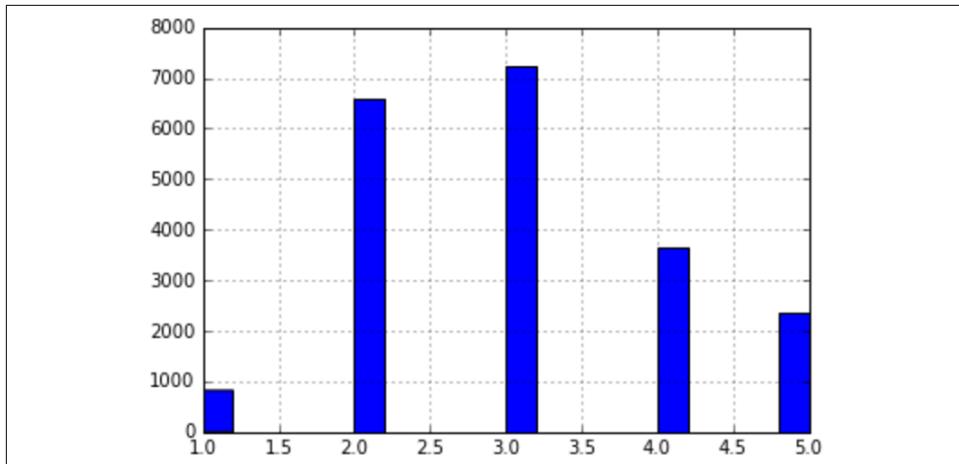
that was either less than 49% female or more than 54% female. Either way, the survey results would be significantly biased.

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.

|          | <b>Overall</b> | <b>Stratified</b> | <b>Random</b> | <b>Rand. %error</b> | <b>Strat. %error</b> |
|----------|----------------|-------------------|---------------|---------------------|----------------------|
| <b>1</b> | 0.039826       | 0.039729          | 0.040213      | 0.973236            | -0.243309            |
| <b>2</b> | 0.318847       | 0.318798          | 0.324370      | 1.732260            | -0.015195            |
| <b>3</b> | 0.350581       | 0.350533          | 0.358527      | 2.266446            | -0.013820            |
| <b>4</b> | 0.176308       | 0.176357          | 0.167393      | -5.056334           | 0.027480             |
| <b>5</b> | 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")
```

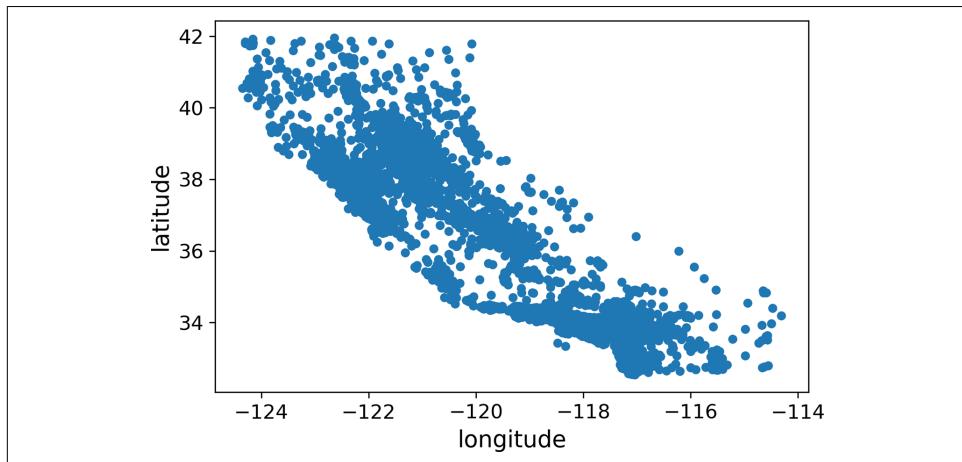
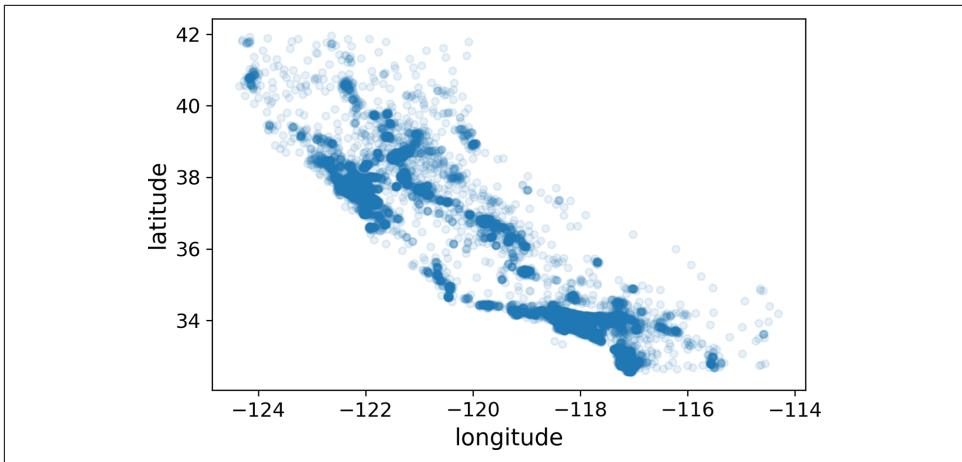


Figure 2-11. A geographical scatterplot of the data

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):<sup>16</sup>

```
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()
```

---

<sup>16</sup> 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.

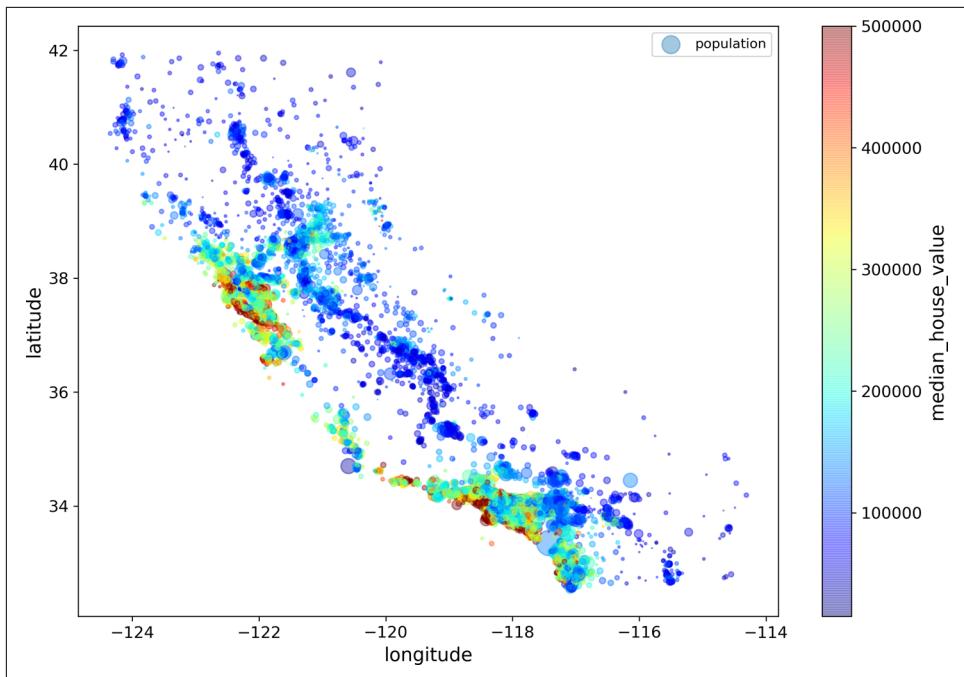


Figure 2-13. California housing prices: red is expensive, blue is cheap, larger circles indicate areas with a larger population

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:

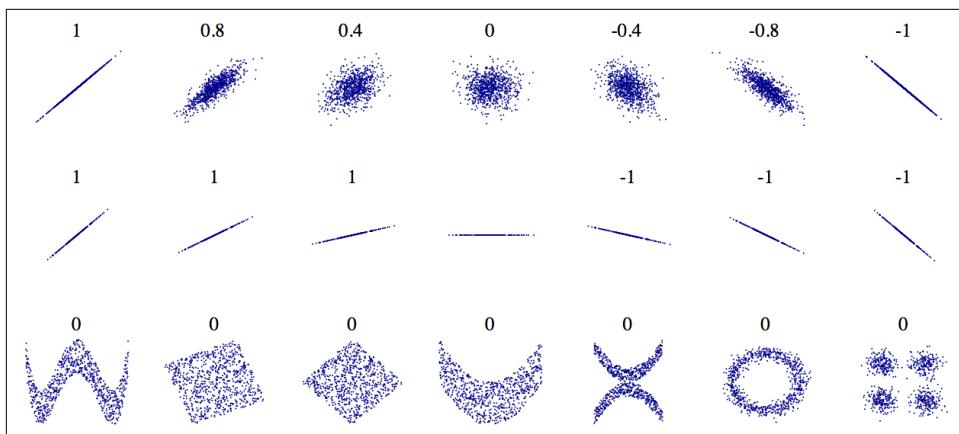
```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income        0.687170
total_rooms          0.135231
housing_median_age   0.114220
```

```

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))
```

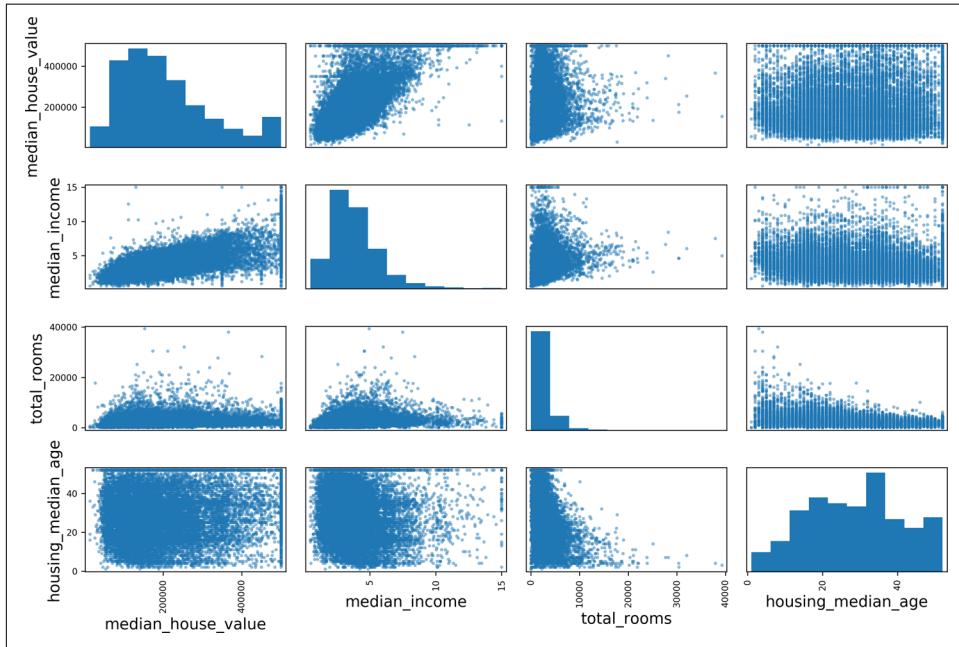


Figure 2-15. This scatter matrix plots every numerical attribute against every other numerical attribute, plus a histogram of each numerical attribute

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)
```

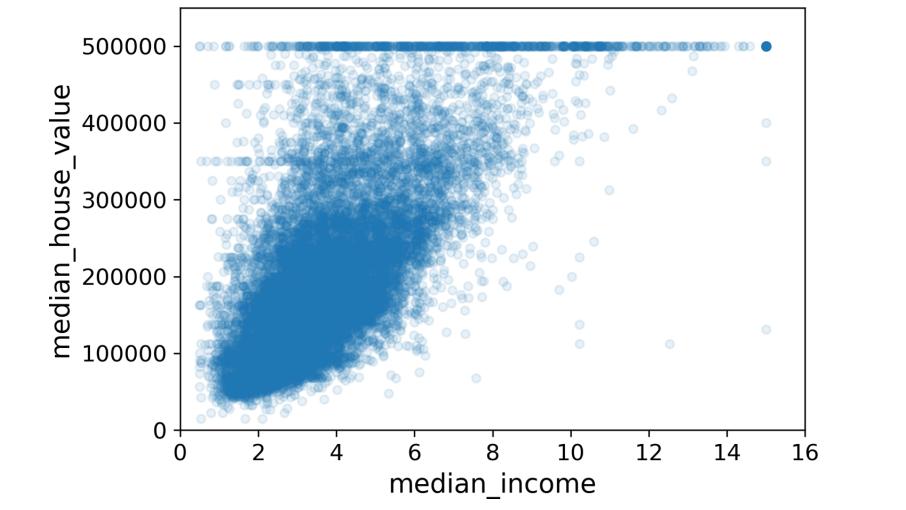


Figure 2-16. Median income versus median house value

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](#):<sup>17</sup>

### 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

<sup>17</sup> 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).<sup>18</sup>

#### *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:

---

<sup>18</sup> 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:<sup>19</sup>

```
>>> 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

---

<sup>19</sup> 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:<sup>20</sup>

```
>>> 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,<sup>21</sup> 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)]
```

---

<sup>20</sup> 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.

<sup>21</sup> 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 `**kwargs` 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 **kwargs
        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")),
    ('attrbs_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,<sup>22</sup> 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 "pass through" 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`

---

<sup>22</sup> 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  $\pm 2,439$ . 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  $3 \times 4 = 12$  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  $2 \times 3 = 6$  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  $18 \times 5 = 90$  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])
```

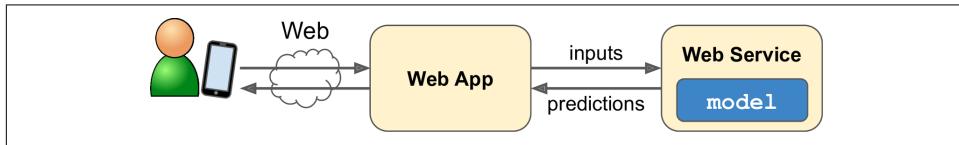
If you did a lot of hyperparameter tuning, the performance will usually be slightly worse than what you measured using cross-validation (because your system ends up fine-tuned to perform well on the validation data and will likely not perform as well on unknown datasets). It is not the case in this example, 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<sup>23</sup> (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 takes 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.

---

<sup>23</sup> 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.<sup>24</sup>

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.

---

<sup>24</sup> 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:<sup>1</sup>

```
>>> 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'])
```

---

<sup>1</sup> 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  $28 \times 28$  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  $28 \times 28$  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.<sup>2</sup>

---

<sup>2</sup> 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 ( $\text{precision} = 1/1 = 100\%$ ). 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.

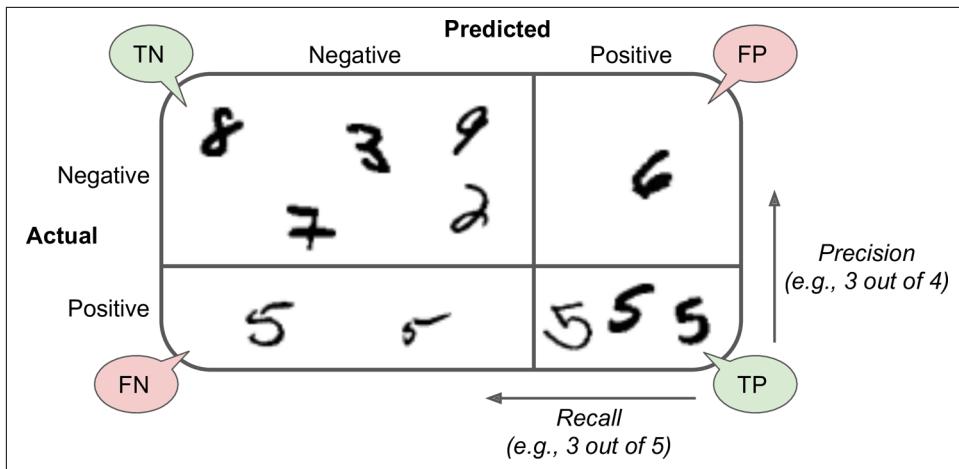


Figure 3-2. An illustrated confusion matrix shows examples of true negatives (top left), false positives (top right), false negatives (lower left), and true positives (lower right)

## 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<sub>1</sub> score*, in particular if you need a simple way to compare two classifiers. The F<sub>1</sub> 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<sub>1</sub> 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.

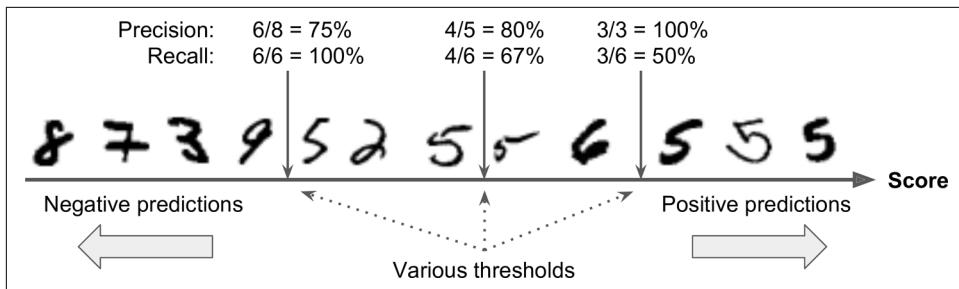


Figure 3-3. In this precision/recall trade-off, images are ranked by their classifier score, and those above the chosen decision threshold are considered positive; the higher the threshold, the lower the recall, but (in general) the higher the precision

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()

```

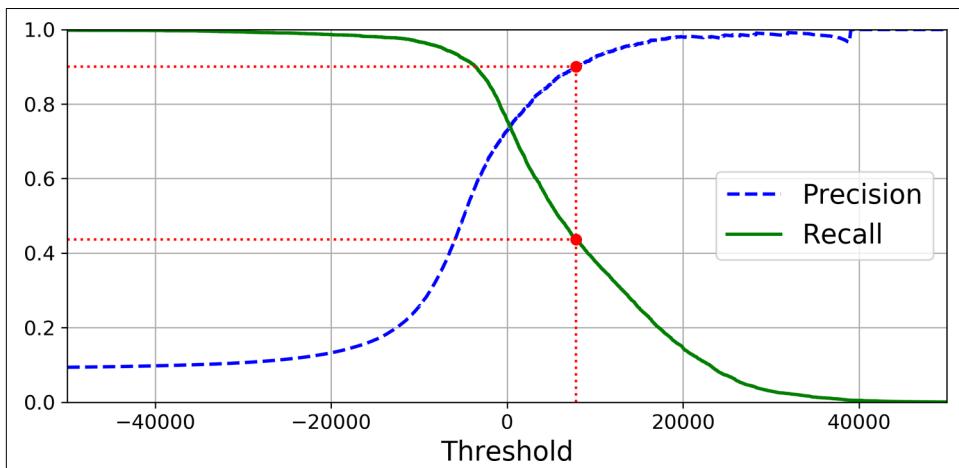


Figure 3-4. Precision and recall versus the decision threshold



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).

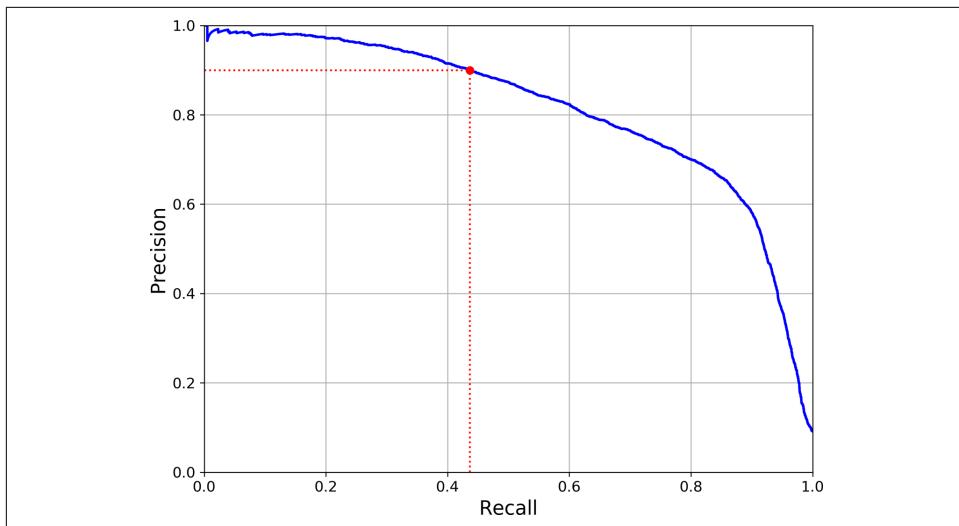


Figure 3-5. Precision versus recall

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  $1 - \text{true negative rate}$  (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  $1 - \text{specificity}$ .

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).

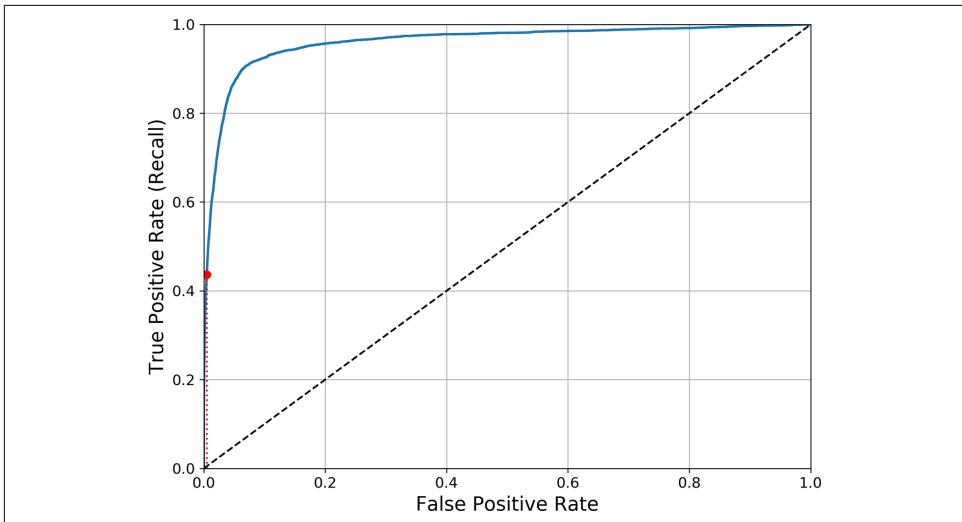


Figure 3-6. This ROC curve plots the false positive rate against the true positive rate for all possible thresholds; the red circle highlights the chosen ratio (at 43.68% recall)

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_5, 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()
```

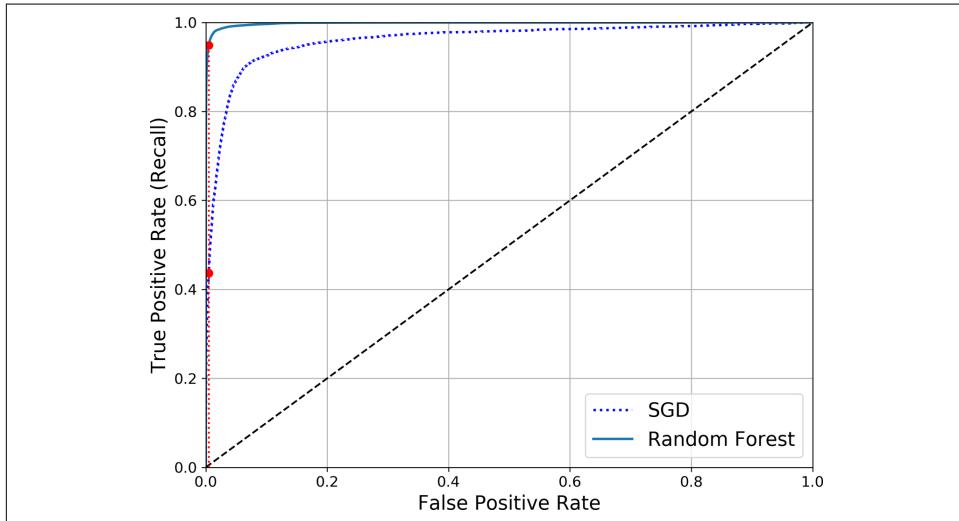


Figure 3-7. Comparing ROC curves: the Random Forest classifier is superior to the SGD classifier because its ROC curve is much closer to the top-left corner, and it has a greater AUC

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 multi-class 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

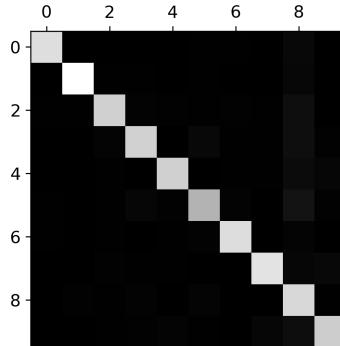
If this were a real project, you would now follow the steps in your Machine Learning project checklist (see [Appendix B](#)). You'd explore data preparation options, try out multiple models (shortlisting the best ones and fine-tuning their hyperparameters using `GridSearchCV`), and automate as much as 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()
```



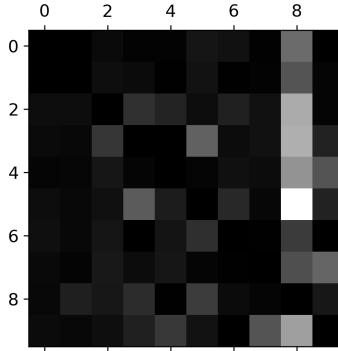
This confusion matrix looks pretty good, since most images are on the main diagonal, which means that they were classified correctly. The 5s look slightly darker than the other digits, which could mean that there are fewer images of 5s in the dataset or that the classifier does not perform as well on 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.

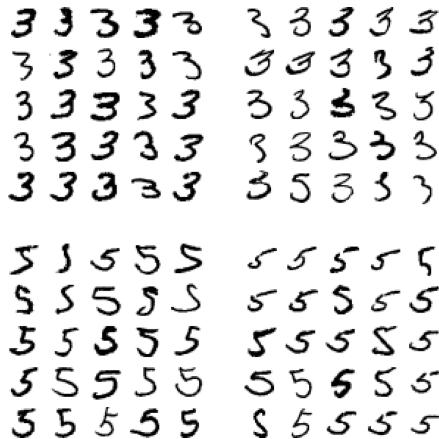
Analyzing individual errors can also be a good way to gain insights on what your classifier is doing and why it is failing, but it is more difficult and time-consuming. For example, let's plot examples of 3s and 5s (the `plot_digits()` function just uses Matplotlib's `imshow()` function; see this chapter's 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  $5 \times 5$  blocks on the left show digits classified as 3s, and the two  $5 \times 5$  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.<sup>3</sup> 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.

---

<sup>3</sup> 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.<sup>4</sup>

## 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):

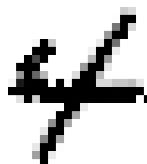
---

<sup>4</sup> 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.<sup>5</sup> 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*.

---

<sup>5</sup> 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 non-linear 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:  $\text{life\_satisfaction} = \theta_0 + \theta_1 \times \text{GDP\_per\_capita}$ .

This model is just a linear function of the input feature `GDP_per_capita`.  $\theta_0$  and  $\theta_1$  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  $i^{\text{th}}$  feature value.
- $\theta_j$  is the  $j^{\text{th}}$  model parameter (including the bias term  $\theta_0$  and the feature weights  $\theta_1, \theta_2, \dots, \theta_n$ ).

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)*

$$\hat{y} = h_{\theta}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

In this equation:

- $\boldsymbol{\theta}$  is the model's *parameter vector*, containing the bias term  $\theta_0$  and the feature weights  $\theta_1$  to  $\theta_n$ .
- $\mathbf{x}$  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  $\theta_0x_0 + \theta_1x_1 + \theta_2x_2 + \dots + \theta_nx_n$ .
- $h_{\theta}$  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  $\hat{y} = \boldsymbol{\theta}^T \mathbf{x}$ , where  $\boldsymbol{\theta}^T$  is the *transpose* of  $\boldsymbol{\theta}$  (a row vector instead of a column vector) and  $\boldsymbol{\theta}^T \mathbf{x}$  is the matrix multiplication of  $\boldsymbol{\theta}^T$  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).<sup>1</sup>

---

<sup>1</sup> 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  $h_{\theta}$  on a training set  $\mathbf{X}$  is calculated using [Equation 4-3](#).

*Equation 4-3. MSE cost function for a Linear Regression model*

$$\text{MSE}(\mathbf{X}, h_{\theta}) = \frac{1}{m} \sum_{i=1}^m (\theta^\top \mathbf{x}^{(i)} - y^{(i)})^2$$

Most of these notations were presented in [Chapter 2](#) (see “[Notations](#)” on page 40). The only difference is that we write  $h_{\theta}$  instead of just  $h$  to make it clear that the model is parametrized by the vector  $\theta$ . To simplify notations, we will just write  $\text{MSE}(\theta)$  instead of  $\text{MSE}(\mathbf{X}, h_{\theta})$ .

## The Normal Equation

To find the value of  $\theta$  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*

$$\hat{\theta} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

In this equation:

- $\hat{\theta}$  is the value of  $\theta$  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)
```

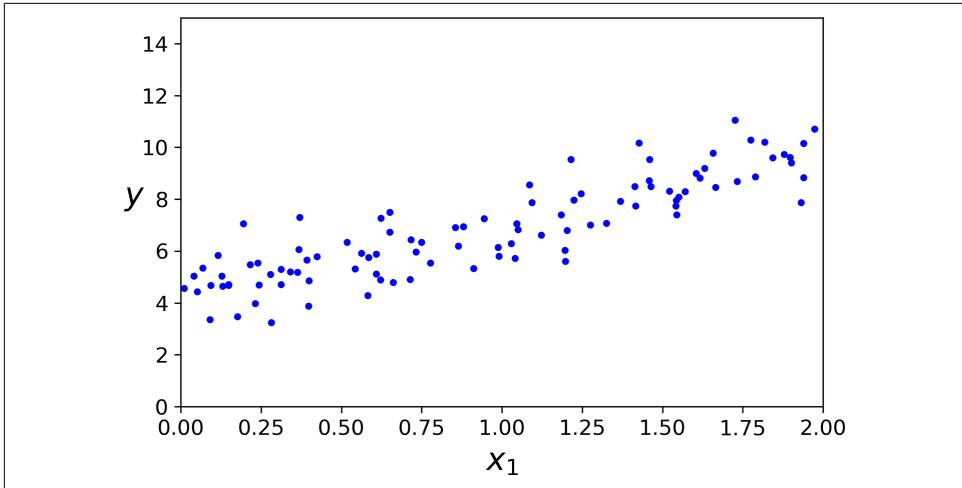


Figure 4-1. Randomly generated linear dataset

Now let's compute  $\hat{\theta}$  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  $y = 4 + 3x_1 + \text{Gaussian noise}$ . 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  $\hat{\theta}$ :

```
>>> 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()
```

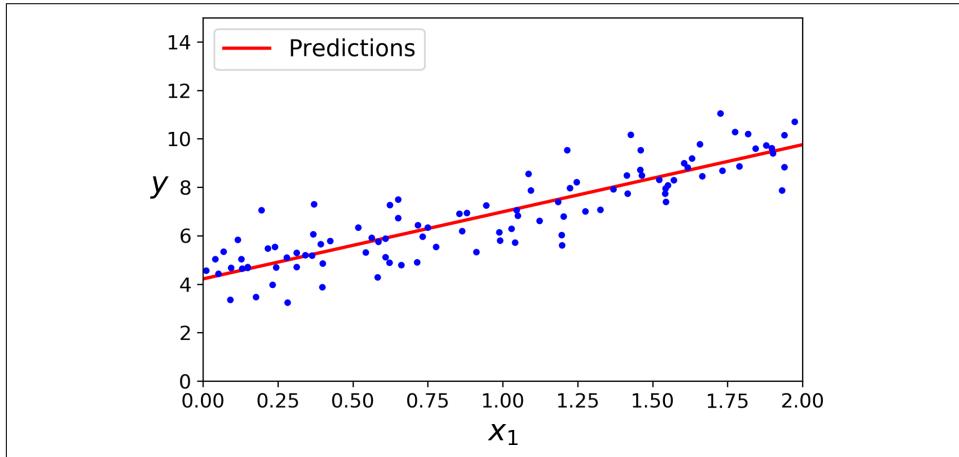


Figure 4-2. Linear Regression model predictions

Performing Linear Regression using Scikit-Learn is simple:<sup>2</sup>

```
>>> 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  $\hat{\theta} = \mathbf{X}^+ \mathbf{y}$ , 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:

---

<sup>2</sup> 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  $\mathbf{U} \Sigma \mathbf{V}^T$  (see `numpy.linalg.svd()`). The pseudoinverse is computed as  $\mathbf{X}^+ = \mathbf{V}\Sigma^+\mathbf{U}^T$ . To compute the matrix  $\Sigma^+$ , the algorithm takes  $\Sigma$  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  $\mathbf{X}^T\mathbf{X}$  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  $\mathbf{X}^T \mathbf{X}$ , 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  $\theta$ , and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum!

Concretely, you start by filling  $\theta$  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).

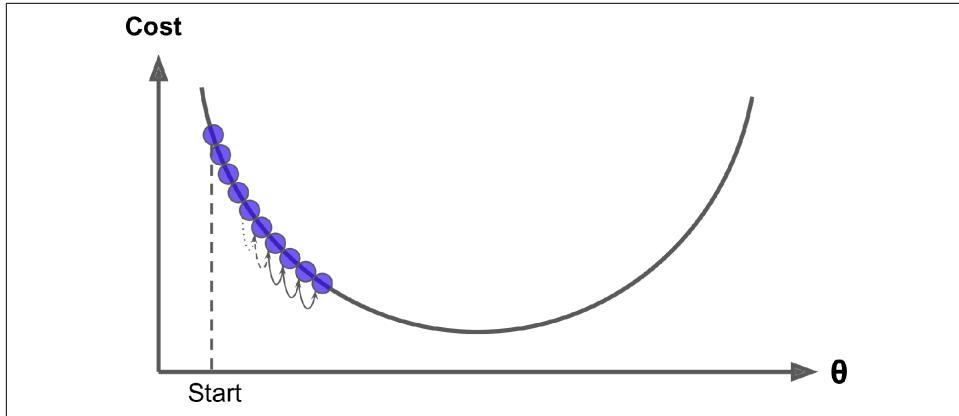


Figure 4-4. The learning rate is too small

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).

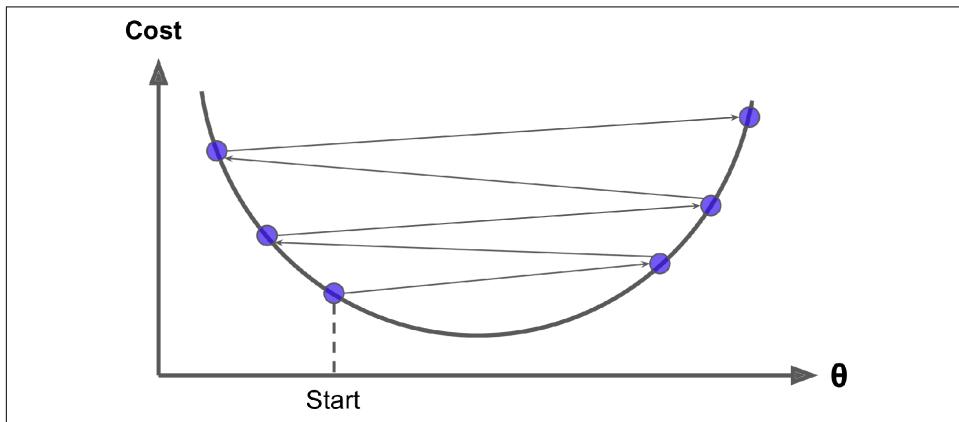


Figure 4-5. The learning rate is too large

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.<sup>3</sup> 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).<sup>4</sup>

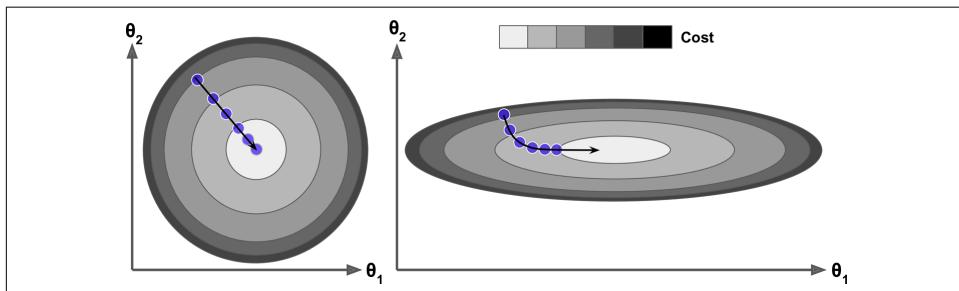


Figure 4-7. Gradient Descent with (left) and without (right) feature scaling

<sup>3</sup> Technically speaking, its derivative is *Lipschitz continuous*.

<sup>4</sup> Since feature 1 is smaller, it takes a larger change in  $\theta_1$  to affect the cost function, which is why the bowl is elongated along the  $\theta_1$  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  $\theta_j$ . In other words, you need to calculate how much the cost function will change if you change  $\theta_j$  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  $\theta_j$ , noted  $\frac{\partial \text{MSE}(\boldsymbol{\theta})}{\partial \theta_j}$ .

*Equation 4-5. Partial derivatives of the cost function*

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \sum_{i=1}^m (\boldsymbol{\theta}^\top \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

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*

$$\nabla_{\theta} \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T (\mathbf{X}\theta - \mathbf{y})$$



Notice that this formula involves calculations over the full training set  $\mathbf{X}$ , 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  $\nabla_{\theta} \text{MSE}(\theta)$  from  $\theta$ . This is where the learning rate  $\eta$  comes into play:<sup>5</sup> multiply the gradient vector by  $\eta$  to determine the size of the downhill step ([Equation 4-7](#)).

*Equation 4-7. Gradient Descent step*

$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

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
```

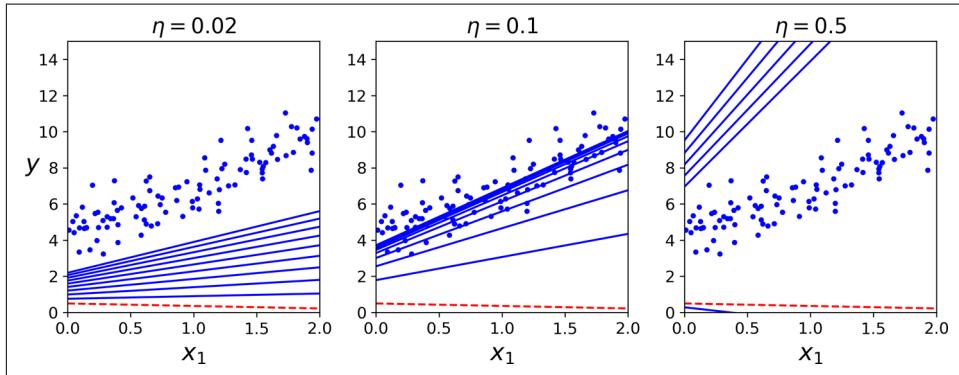
---

<sup>5</sup> Eta ( $\eta$ ) 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  $\epsilon$  (called the *tolerance*)—because this happens when Gradient Descent has (almost) reached the minimum.

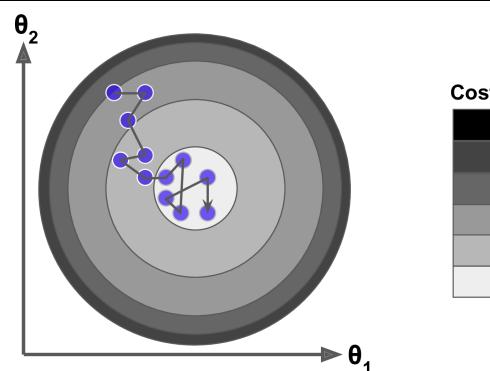
## Convergence Rate

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  $\epsilon$ , 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)

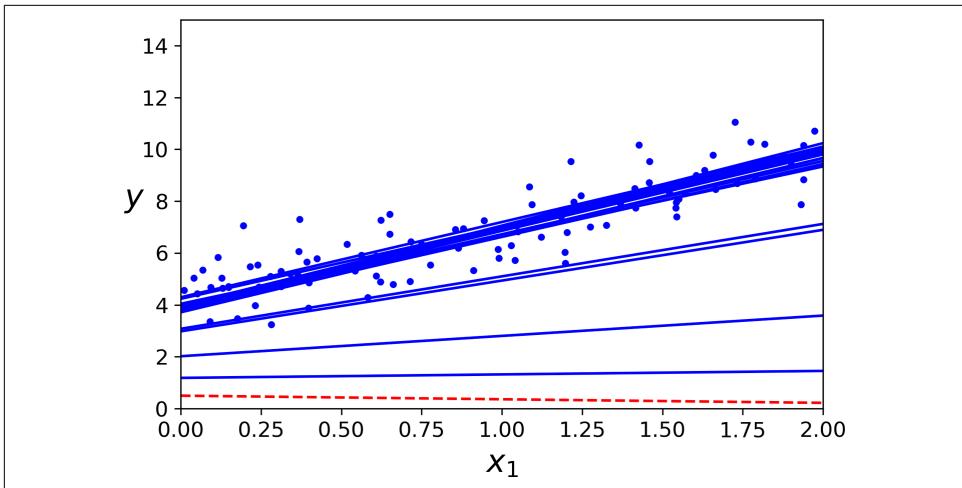
theta = np.random.randn(2,1) # random initialization

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:

```

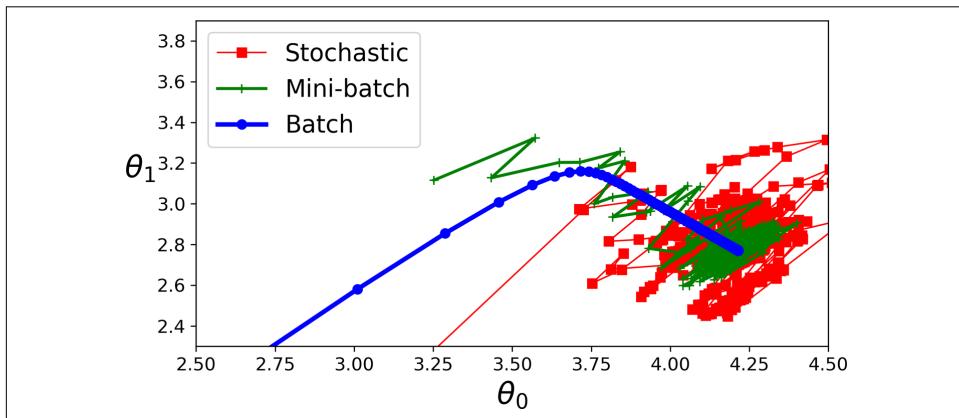
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))

```

## 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<sup>6</sup> (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               | <code>LinearRegression</code> |
| Batch GD        | Slow      | No                  | Fast      | 2           | Yes              | <code>SGDRegressor</code>     |
| Stochastic GD   | Fast      | Yes                 | Fast      | $\geq 2$    | Yes              | <code>SGDRegressor</code>     |
| Mini-batch GD   | Fast      | Yes                 | Fast      | $\geq 2$    | Yes              | <code>SGDRegressor</code>     |



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*<sup>7</sup> (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)
```

<sup>6</sup> 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.

<sup>7</sup> A quadratic equation is of the form  $y = ax^2 + bx + c$ .

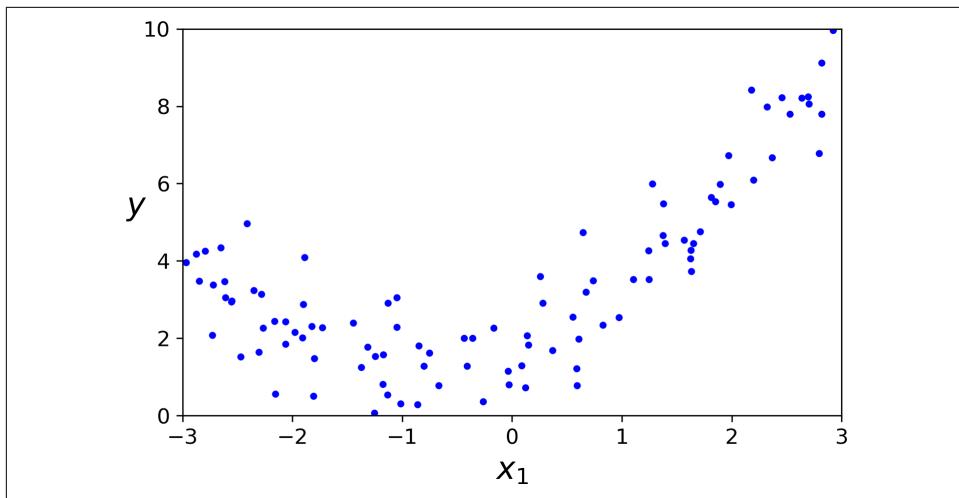


Figure 4-12. Generated nonlinear and noisy dataset

Clearly, a straight line will never fit this data properly. So let's use Scikit-Learn's `Poly`  
`nomicFeatures` 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]]))
```

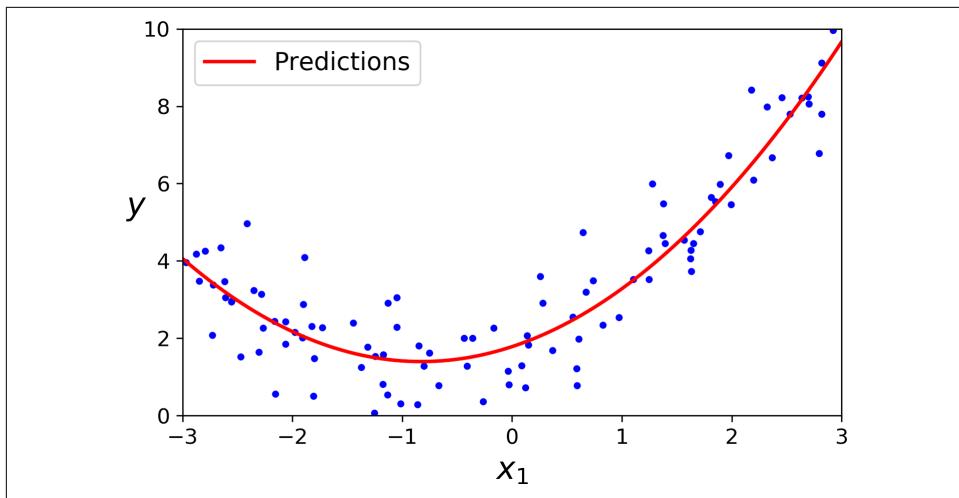


Figure 4-13. Polynomial Regression model predictions

Not bad: the model estimates  $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$  when in fact the original function was  $y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise}$ .

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  $1 \times 2 \times 3 \times \dots \times n$ . 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.

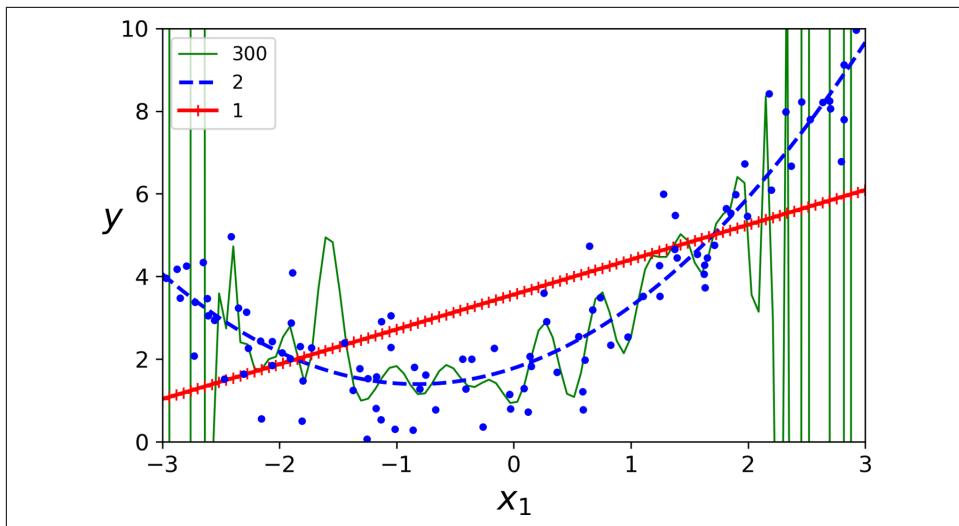


Figure 4-14. High-degree Polynomial Regression

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)

```

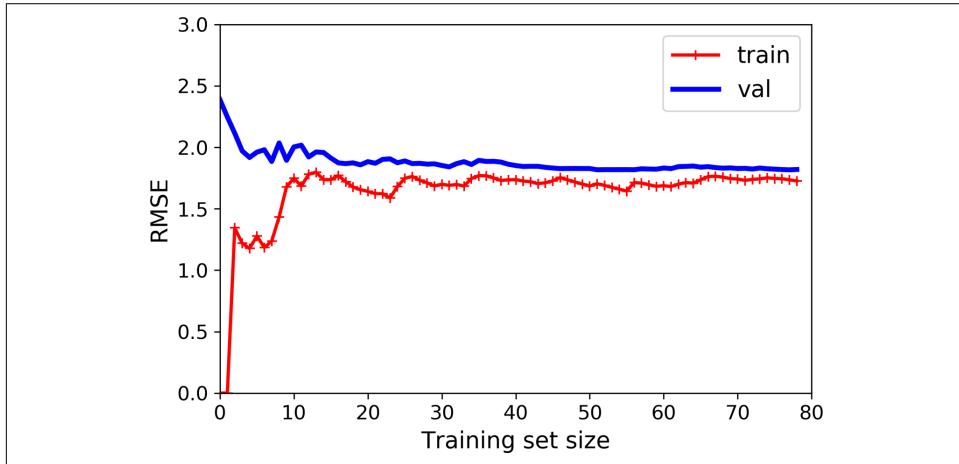


Figure 4-15. Learning curves

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.

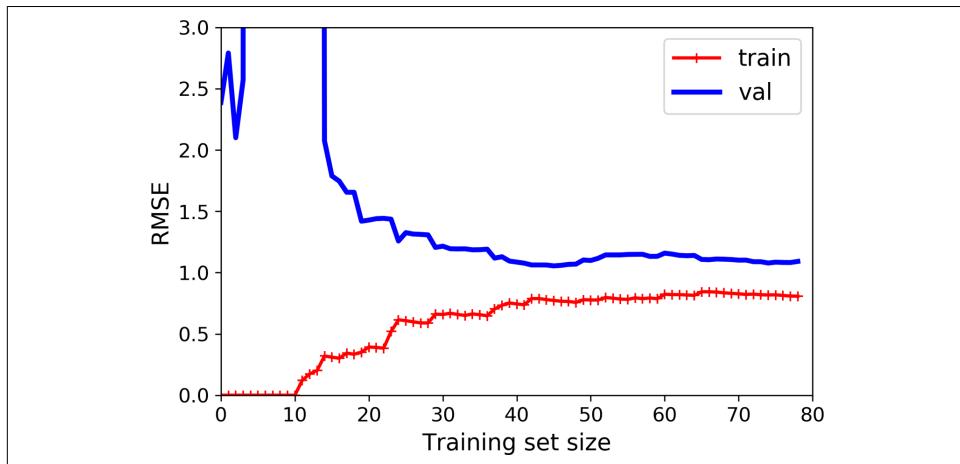


If your model is underfitting the training data, adding more training examples will not help. You need to use a more complex model or come up with better features.

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.<sup>8</sup>

### *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

---

<sup>8</sup> 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  $\alpha$  controls how much you want to regularize the model. If  $\alpha = 0$ , then Ridge Regression is just Linear Regression. If  $\alpha$  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.<sup>9</sup>

*Equation 4-8. Ridge Regression cost function*

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$$

Note that the bias term  $\theta_0$  is not regularized (the sum starts at  $i = 1$ , not 0). If we define  $\mathbf{w}$  as the vector of feature weights ( $\theta_1$  to  $\theta_n$ ), then the regularization term is

---

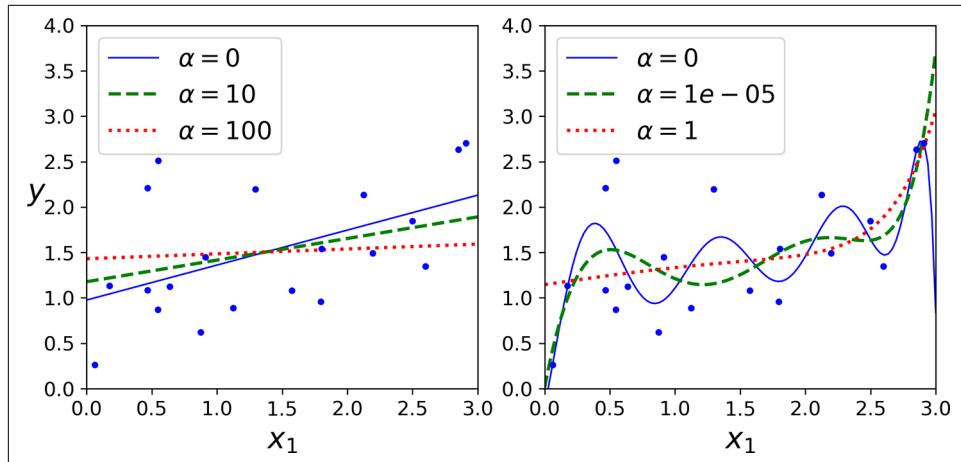
<sup>9</sup> It is common to use the notation  $J(\boldsymbol{\theta})$  for cost functions that don't have a short name; we will often use this notation throughout the rest of this book. The context will make it clear which cost function is being discussed.

equal to  $\frac{1}{2}(\|\mathbf{w}\|_2)^2$ , where  $\|\mathbf{w}\|_2$  represents the  $\ell_2$  norm of the weight vector.<sup>10</sup> 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  $\alpha$  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  $\alpha$  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

---

<sup>10</sup> 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*,<sup>11</sup> except with a 0 in the top-left cell, corresponding to the bias term.

*Equation 4-9. Ridge Regression closed-form solution*

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T \mathbf{X} + \alpha \mathbf{A})^{-1} \mathbf{X}^T \mathbf{y}$$

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:<sup>12</sup>

```
>>> 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  $\ell_2$  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  $\ell_1$  norm of the weight vector instead of half the square of the  $\ell_2$  norm (see [Equation 4-10](#)).

*Equation 4-10. Lasso Regression cost function*

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^n |\theta_i|$$

---

<sup>11</sup> A square matrix full of 0s except for 1s on the main diagonal (top left to bottom right).

<sup>12</sup> 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  $\alpha$  values.

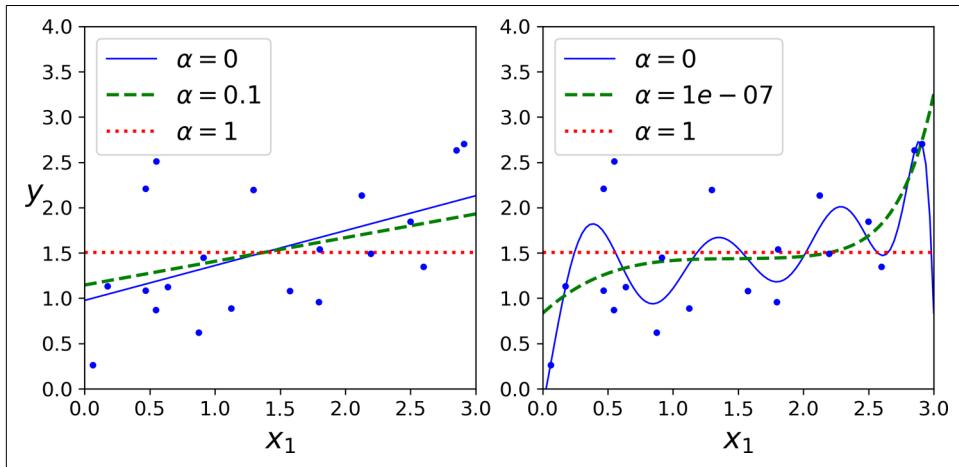


Figure 4-18. A linear model (left) and a polynomial model (right), both using various levels of Lasso regularization

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  $\ell_1$  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  $\theta_2$  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  $\ell_1$  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  $\ell_1$  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  $\alpha$ , the global optimum would move left along the dashed yellow line, while

if we decreased  $\alpha$ , the global optimum would move right (in this example, the optimal parameters for the unregularized MSE are  $\theta_1 = 2$  and  $\theta_2 = 0.5$ ).

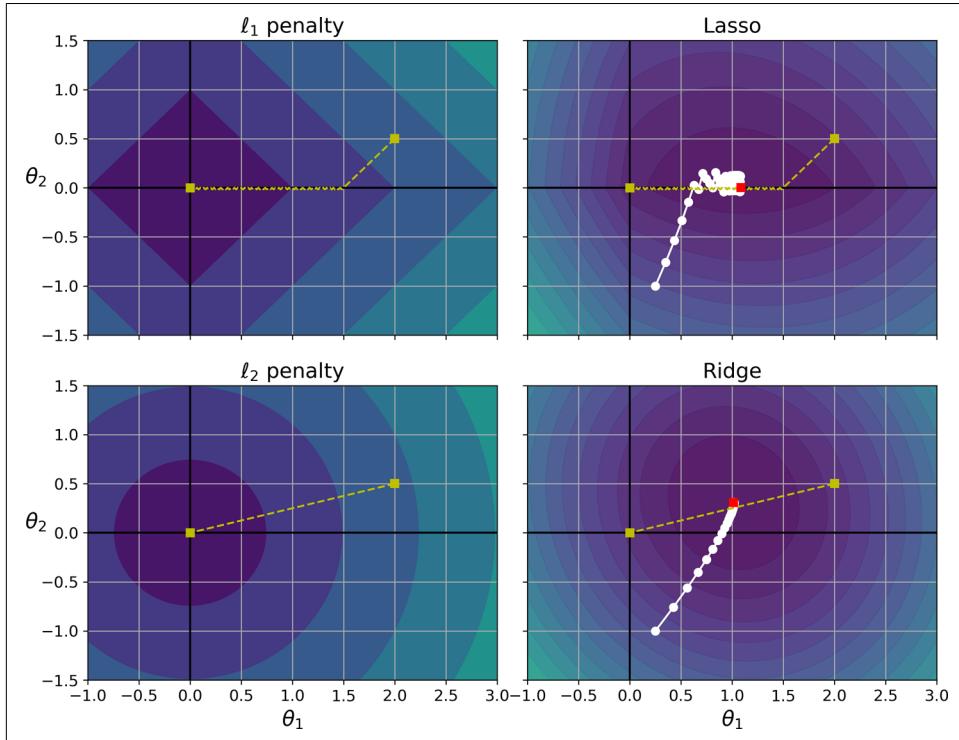


Figure 4-19. Lasso versus Ridge regularization

The two bottom plots show the same thing but with an  $\ell_2$  penalty instead. In the bottom-left plot, you can see that the  $\ell_2$  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  $\ell_2$  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  $\alpha$ , 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  $i = 1, 2, \dots, n$ ), but Gradient Descent still works fine if you use a *subgradient vector*  $\mathbf{g}$ <sup>13</sup> 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*

$$g(\boldsymbol{\theta}, J) = \nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) + \alpha \begin{pmatrix} \text{sign}(\theta_1) \\ \text{sign}(\theta_2) \\ \vdots \\ \text{sign}(\theta_n) \end{pmatrix} \quad \text{where } \text{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

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*

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + r\alpha \sum_{i=1}^n |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^n \theta_i^2$$

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

---

<sup>13</sup> 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.”

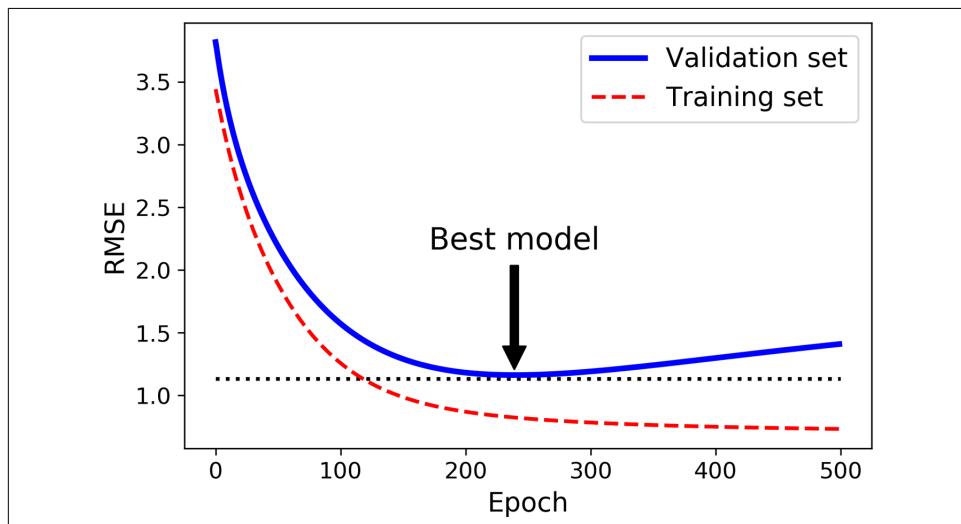


Figure 4-20. Early stopping regularization



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](#)).

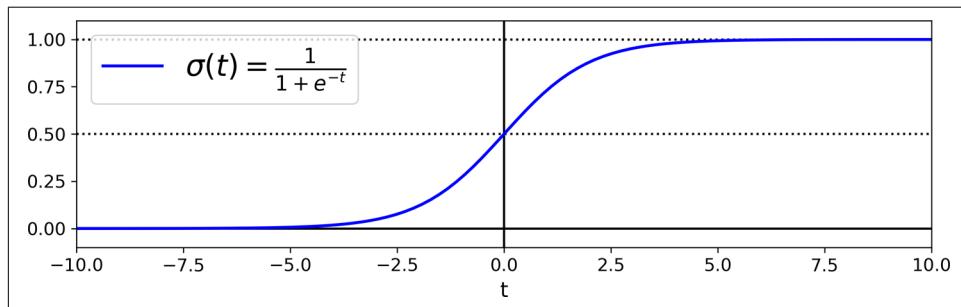
*Equation 4-13. Logistic Regression model estimated probability (vectorized form)*

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\mathbf{x}^T \boldsymbol{\theta})$$

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  $\hat{p} = h_{\theta}(\mathbf{x})$  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  $\mathbf{x}^T \boldsymbol{\theta}$  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  $\Theta$  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*

$$c(\Theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$

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)*

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)})]$$

The bad news is that there is no known closed-form equation to compute the value of  $\Theta$  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  $j^{\text{th}}$  model parameter  $\theta_j$  are given by [Equation 4-18](#).

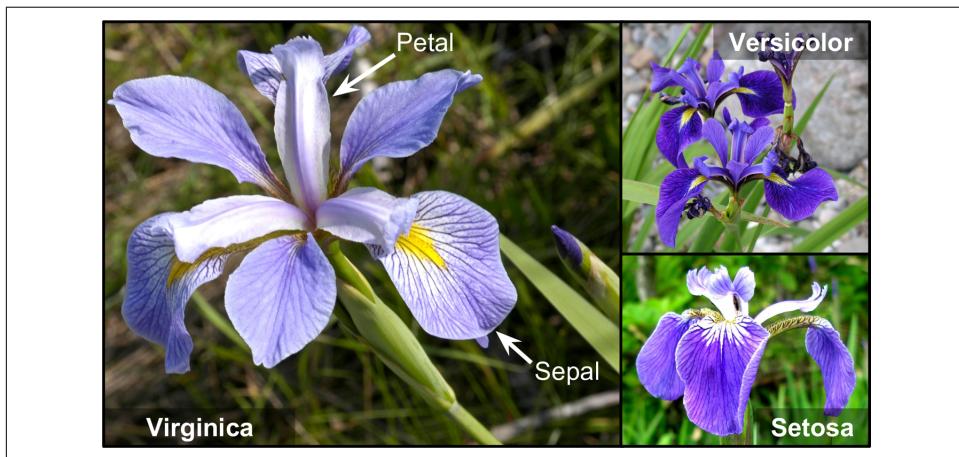
*Equation 4-18. Logistic cost function partial derivatives*

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m (\sigma(\boldsymbol{\theta}^\top \mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

This equation looks very much like [Equation 4-5](#): for each instance it computes the prediction error and multiplies it by the  $j^{\text{th}}$  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<sup>14</sup>*

<sup>14</sup> 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):<sup>15</sup>

```
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
```

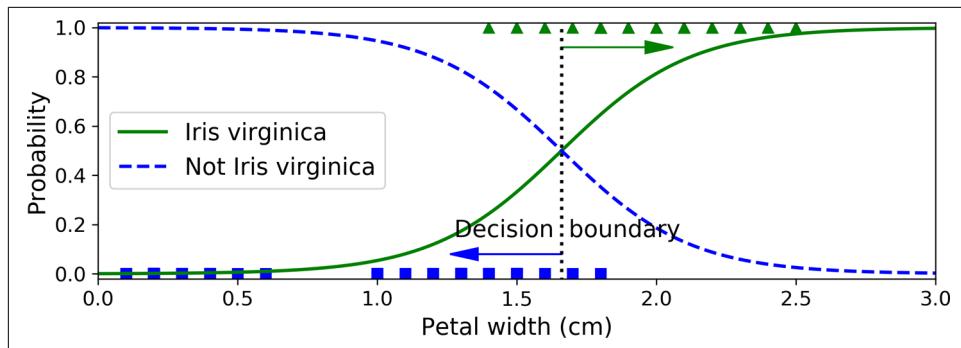


Figure 4-23. Estimated probabilities and decision boundary

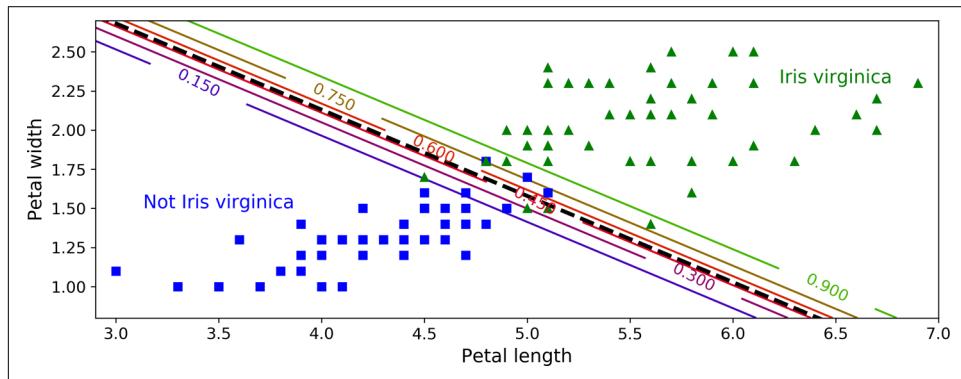
The petal width of *Iris virginica* flowers (represented by triangles) ranges from 1.4 cm to 2.5 cm, while the other iris flowers (represented by squares) generally have a smaller petal width, ranging from 0.1 cm to 1.8 cm. Notice that there is a bit of overlap. Above about 2 cm the classifier is highly 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”

<sup>15</sup> 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.<sup>16</sup> 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  $\ell_1$  or  $\ell_2$  penalties. Scikit-Learn actually adds an  $\ell_2$  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.

<sup>16</sup> It is the set of points  $\mathbf{x}$  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*

$$s_k(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\theta}^{(k)}$$

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*

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))}$$

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*

$$\hat{y} = \operatorname{argmax}_k \sigma(\mathbf{s}(\mathbf{x}))_k = \operatorname{argmax}_k s_k(\mathbf{x}) = \operatorname{argmax}_k \left( (\boldsymbol{\theta}^{(k)})^\top \mathbf{x} \right)$$

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*

$$J(\boldsymbol{\Theta}) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(\hat{p}_k^{(i)})$$

In this equation:

- $y_k^{(i)}$  is the target probability that the  $i^{\text{th}}$  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  $H(p, q) = -\sum_x p(x) \log q(x)$  (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  $\Theta^{(k)}$  is given by [Equation 4-23](#).

*Equation 4-23. Cross entropy gradient vector for class k*

$$\nabla_{\Theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^m (\hat{p}_k^{(i)} - y_k^{(i)}) \mathbf{x}^{(i)}$$

Now you can compute the gradient vector for every class, then use Gradient Descent (or any other optimization algorithm) to find the parameter matrix  $\Theta$  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  $\ell_2$  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%.

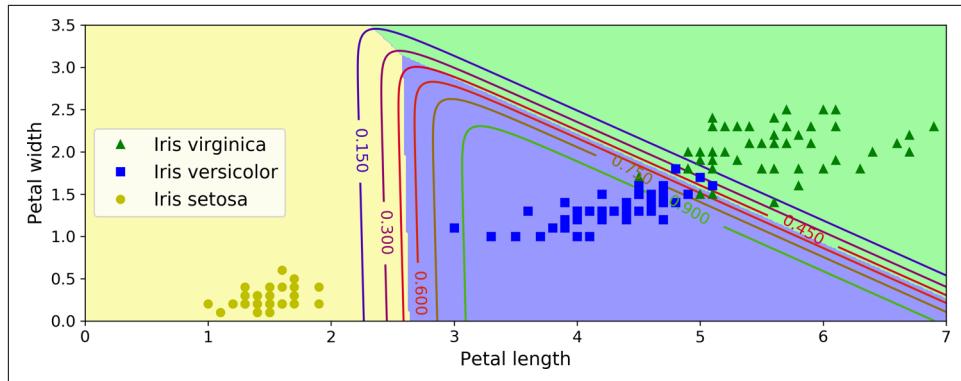


Figure 4-25. Softmax Regression decision boundaries

## 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  $\alpha$  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](#).

# 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*.

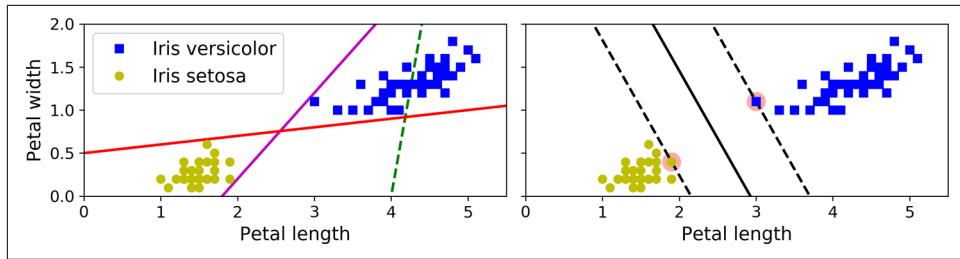


Figure 5-1. Large margin classification

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).

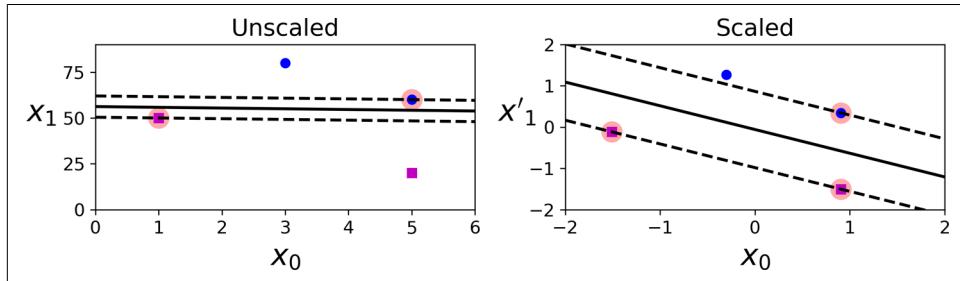


Figure 5-2. Sensitivity to feature scales



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.

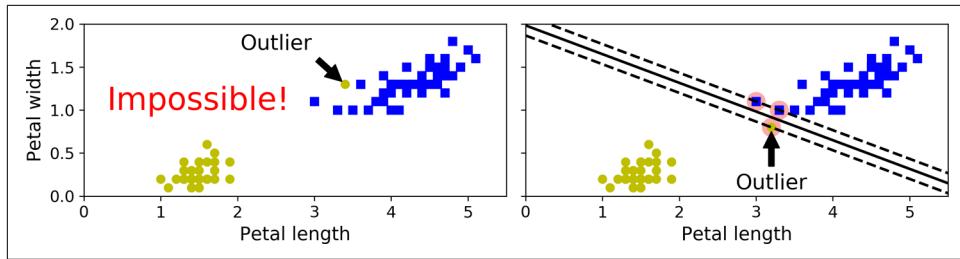


Figure 5-3. Hard margin sensitivity to outliers

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.

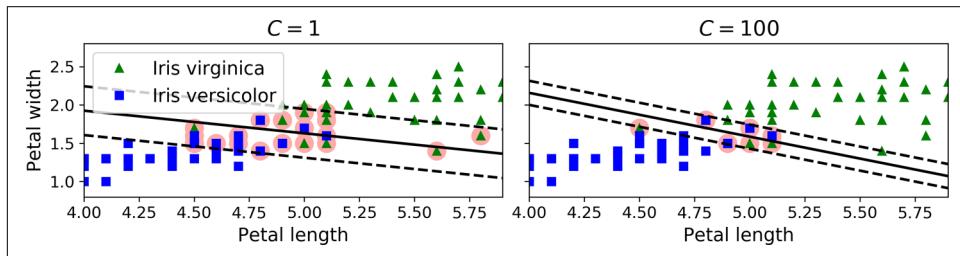


Figure 5-4. Large margin (left) versus fewer margin violations (right)



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.

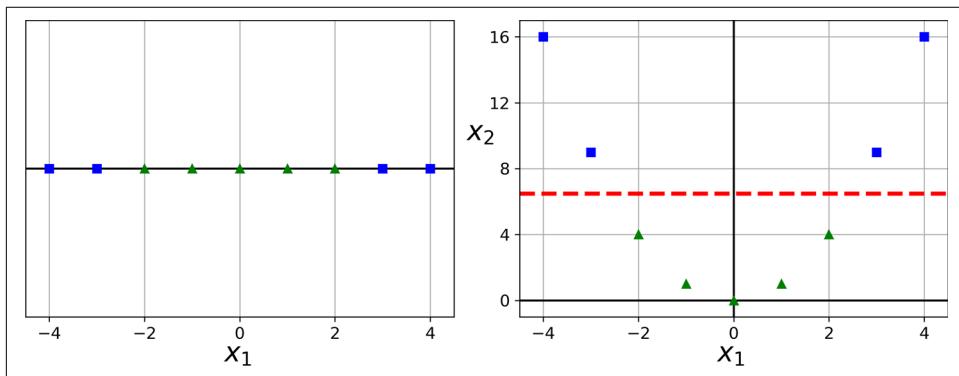
Instead of using the `LinearSVC` class, we could use the `SVC` class with a linear kernel. When creating the `SVC` model, we would write `SVC(kernel="linear", C=1)`. Or we could use the `SGDClassifier` class, with `SGDClassifier(loss="hinge", alpha=1/(m*C))`. This applies regular Stochastic Gradient 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)
```

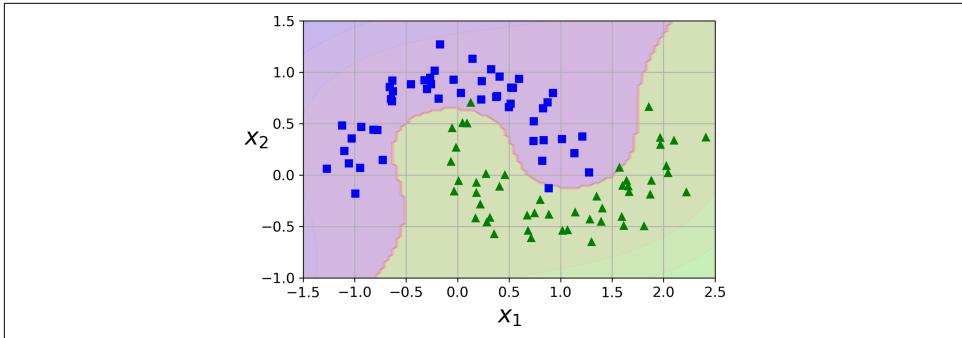


Figure 5-6. Linear SVM classifier using polynomial features

## 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.

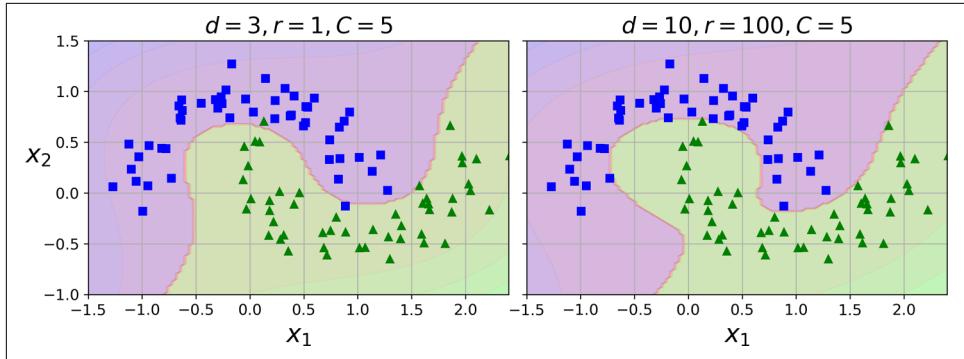


Figure 5-7. SVM classifiers with a polynomial kernel



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*

$$\phi_\gamma(\mathbf{x}, \ell) = \exp(-\gamma \|\mathbf{x} - \ell\|^2)$$

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.

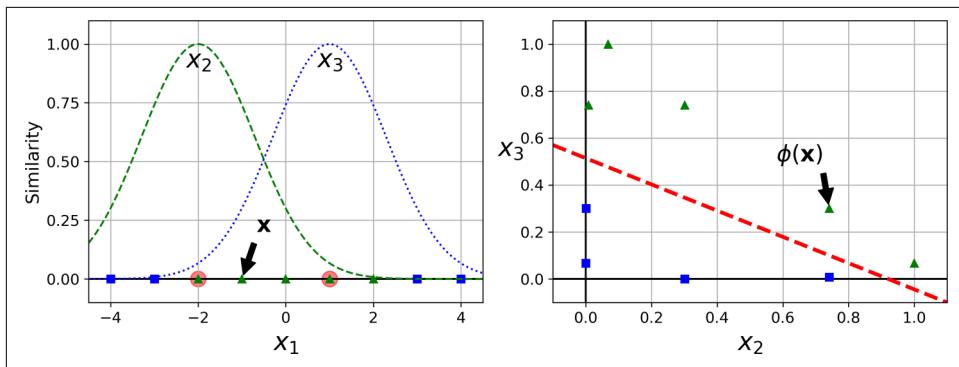


Figure 5-8. Similarity features using the Gaussian RBF

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` ( $\gamma$ ) 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  $\gamma$  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).

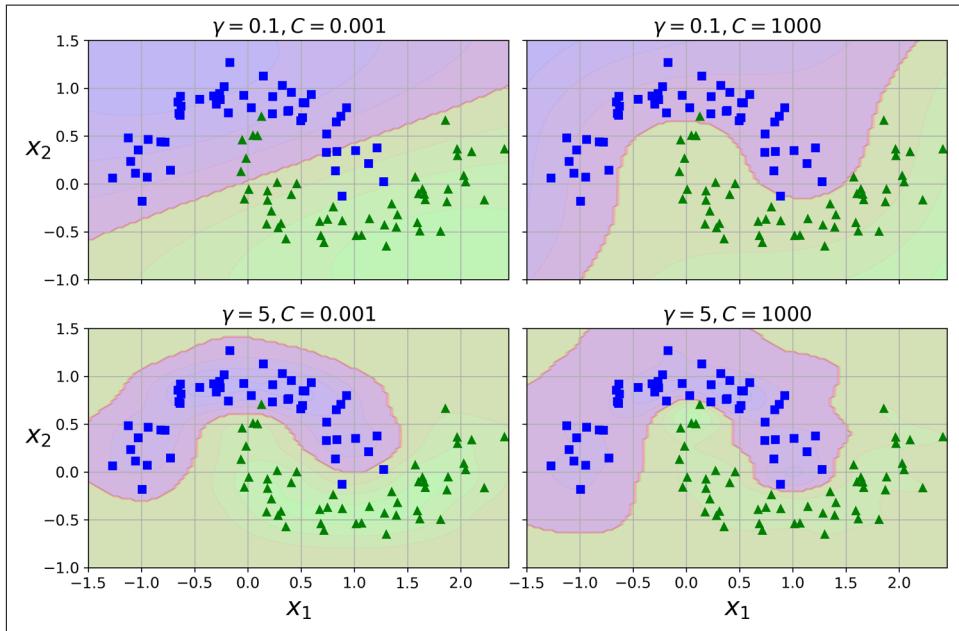


Figure 5-9. SVM classifiers using an RBF kernel

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.<sup>1</sup> 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  $\epsilon$  (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.<sup>2</sup> 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 |
|----------------------------|--|---------------------|------------------|--------------|
| <code>LinearSVC</code>     | $O(m \times n)$                        | No                  | Yes              | No           |
| <code>SGDClassifier</code> | $O(m \times n)$                        | Yes                 | Yes              | No           |
| <code>SVC</code>           | $O(m^2 \times n)$ 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,  $\epsilon$ . [Figure 5-10](#) shows two linear SVM

<sup>1</sup> 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.

<sup>2</sup> 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 ( $\epsilon=0.5$ ).

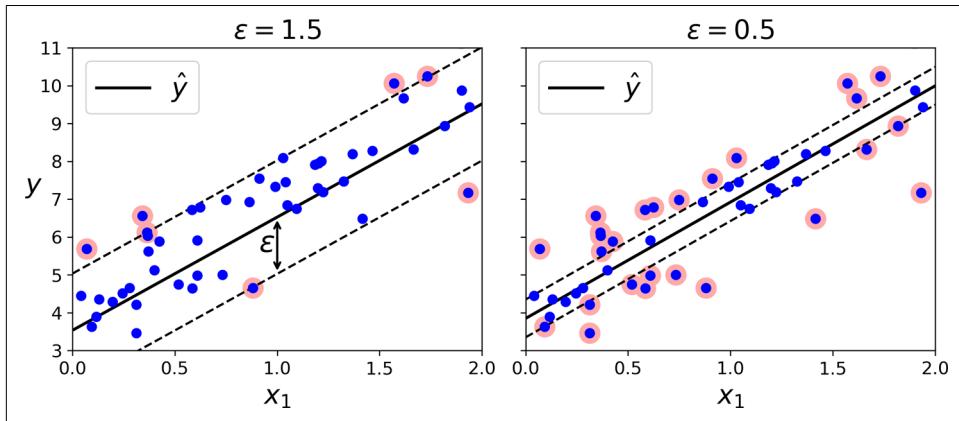


Figure 5-10. SVM Regression

Adding more training instances within the margin does not affect the model's predictions; thus, the model is said to be  *$\epsilon$ -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).

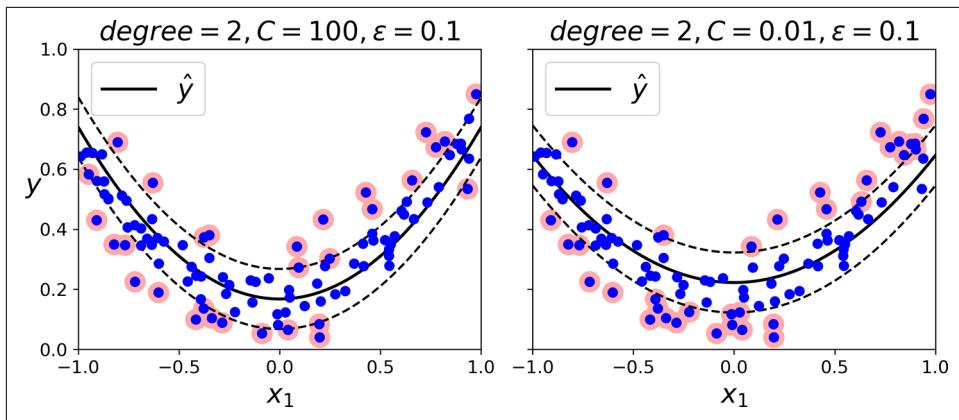


Figure 5-11. SVM Regression using a second-degree polynomial kernel

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  $\theta$ , including the bias term  $\theta_0$  and the input feature weights  $\theta_1$  to  $\theta_n$ , 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.

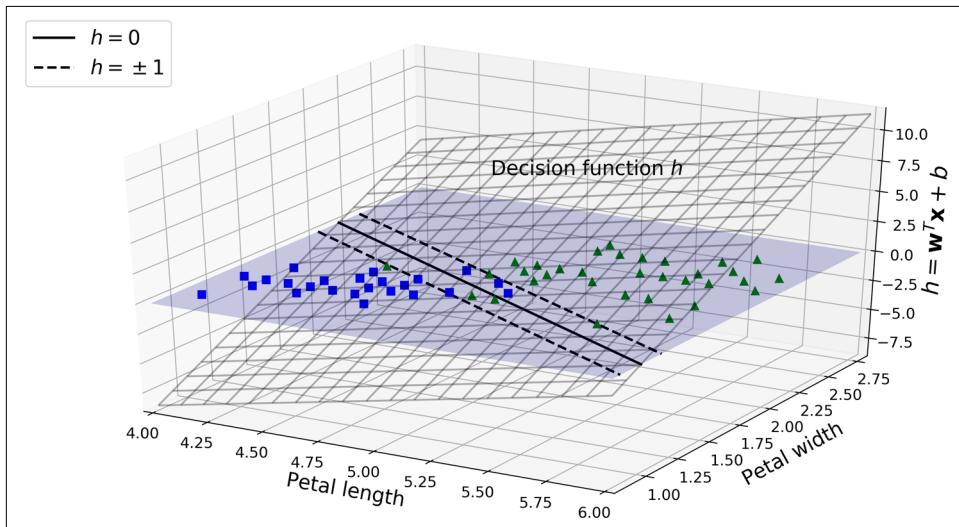
## Decision Function and Predictions

The linear SVM classifier model predicts the class of a new instance  $\mathbf{x}$  by simply computing the decision function  $\mathbf{w}^\top \mathbf{x} + b = w_1 x_1 + \dots + w_n x_n + b$ . 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).<sup>3</sup>



*Figure 5-12. Decision function for the iris dataset*

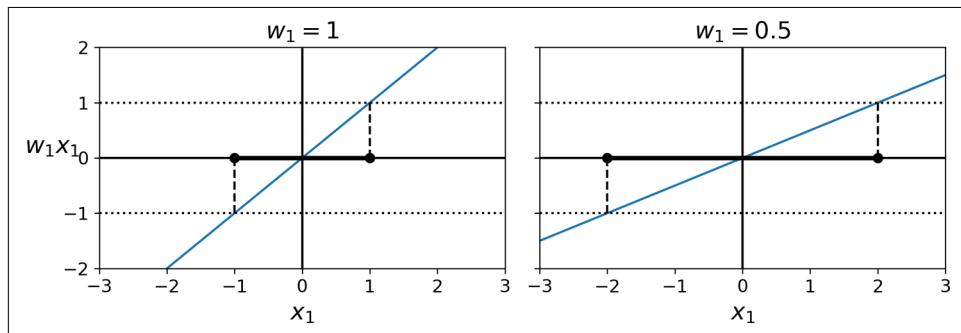
---

<sup>3</sup> 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,  $\|w\|$ . If we divide this slope by 2, the points where the decision function is equal to  $\pm 1$  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  $\|w\|$  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  $t^{(i)}(w^\top x^{(i)} + b) \geq 1$  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*

$$\begin{aligned} & \underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2} \mathbf{w}^\top \mathbf{w} \\ & \text{subject to} \quad t^{(i)} (\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$



We are minimizing  $\frac{1}{2} \mathbf{w}^\top \mathbf{w}$ , which is equal to  $\frac{1}{2} \|\mathbf{w}\|^2$ , rather than minimizing  $\|\mathbf{w}\|$ . Indeed,  $\frac{1}{2} \|\mathbf{w}\|^2$  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:<sup>4</sup>  $\zeta^{(i)}$  measures how much the  $i^{\text{th}}$  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  $\frac{1}{2} \mathbf{w}^\top \mathbf{w}$  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*

$$\begin{aligned} & \underset{\mathbf{w}, b, \zeta}{\text{minimize}} \quad \frac{1}{2} \mathbf{w}^\top \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)} \\ & \text{subject to} \quad t^{(i)} (\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

## 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.<sup>5</sup>

---

<sup>4</sup> Zeta ( $\zeta$ ) is the sixth letter of the Greek alphabet.

<sup>5</sup> 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*

$$\underset{\mathbf{p}}{\text{Minimize}} \quad \frac{1}{2} \mathbf{p}^T \mathbf{H} \mathbf{p} + \mathbf{f}^T \mathbf{p}$$

subject to  $\mathbf{A} \mathbf{p} \leq \mathbf{b}$

where  $\begin{cases} \mathbf{p} & \text{is an } n_p\text{-dimensional vector } (n_p = \text{number of parameters}), \\ \mathbf{H} & \text{is an } n_p \times n_p \text{ matrix,} \\ \mathbf{f} & \text{is an } n_p\text{-dimensional vector,} \\ \mathbf{A} & \text{is an } n_c \times n_p \text{ matrix } (n_c = \text{number of constraints}), \\ \mathbf{b} & \text{is an } n_c\text{-dimensional vector.} \end{cases}$

Note that the expression  $\mathbf{A} \mathbf{p} \leq \mathbf{b}$  defines  $n_c$  constraints:  $\mathbf{p}^T \mathbf{a}^{(i)} \leq b^{(i)}$  for  $i = 1, 2, \dots, n_c$ , where  $\mathbf{a}^{(i)}$  is the vector containing the elements of the  $i^{\text{th}}$  row of  $\mathbf{A}$  and  $b^{(i)}$  is the  $i^{\text{th}}$  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  $-1$ s.
- $\mathbf{a}^{(i)} = -t^{(i)} \dot{\mathbf{x}}^{(i)}$ , 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  $i = 1, 2, \dots, n$ . 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,<sup>6</sup> 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*

$$\underset{\alpha}{\text{minimize}} \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)\top} \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)}$$

subject to  $\alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m$

Once you find the vector  $\hat{\alpha}$  that minimizes this equation (using a QP solver), use [Equation 5-7](#) to compute  $\hat{\mathbf{w}}$  and  $\hat{b}$  that minimize the primal problem.

*Equation 5-7. From the dual solution to the primal solution*

$$\begin{aligned}\hat{\mathbf{w}} &= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)} \\ \hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left( t^{(i)} - \hat{\mathbf{w}}^\top \mathbf{x}^{(i)} \right)\end{aligned}$$

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  $\phi$  that you want to apply.

---

<sup>6</sup> The objective function is convex, and the inequality constraints are continuously differentiable and convex functions.

*Equation 5-8. Second-degree polynomial mapping*

$$\phi(\mathbf{x}) = \phi\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

Notice that the transformed vector is 3D instead of 2D. Now let's look at what happens to a couple of 2D vectors,  $\mathbf{a}$  and  $\mathbf{b}$ , if we apply this second-degree polynomial mapping and then compute the dot product<sup>7</sup> of the transformed vectors (See [Equation 5-9](#)).

*Equation 5-9. Kernel trick for a second-degree polynomial mapping*

$$\begin{aligned} \phi(\mathbf{a})^\top \phi(\mathbf{b}) &= \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^\top \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2 \\ &= (a_1b_1 + a_2b_2)^2 = \left( \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^\top \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^\top \mathbf{b})^2 \end{aligned}$$

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  $\phi$  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  $\phi$  is the second-degree polynomial transformation defined in [Equation 5-8](#), then you can replace this dot product of transformed vectors simply by  $(\mathbf{x}^{(i)\top} \mathbf{x}^{(j)})^2$ . 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  $\phi(\mathbf{a})^\top \phi(\mathbf{b})$ ,

---

<sup>7</sup> As explained in [Chapter 4](#), the dot product of two vectors  $\mathbf{a}$  and  $\mathbf{b}$  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.