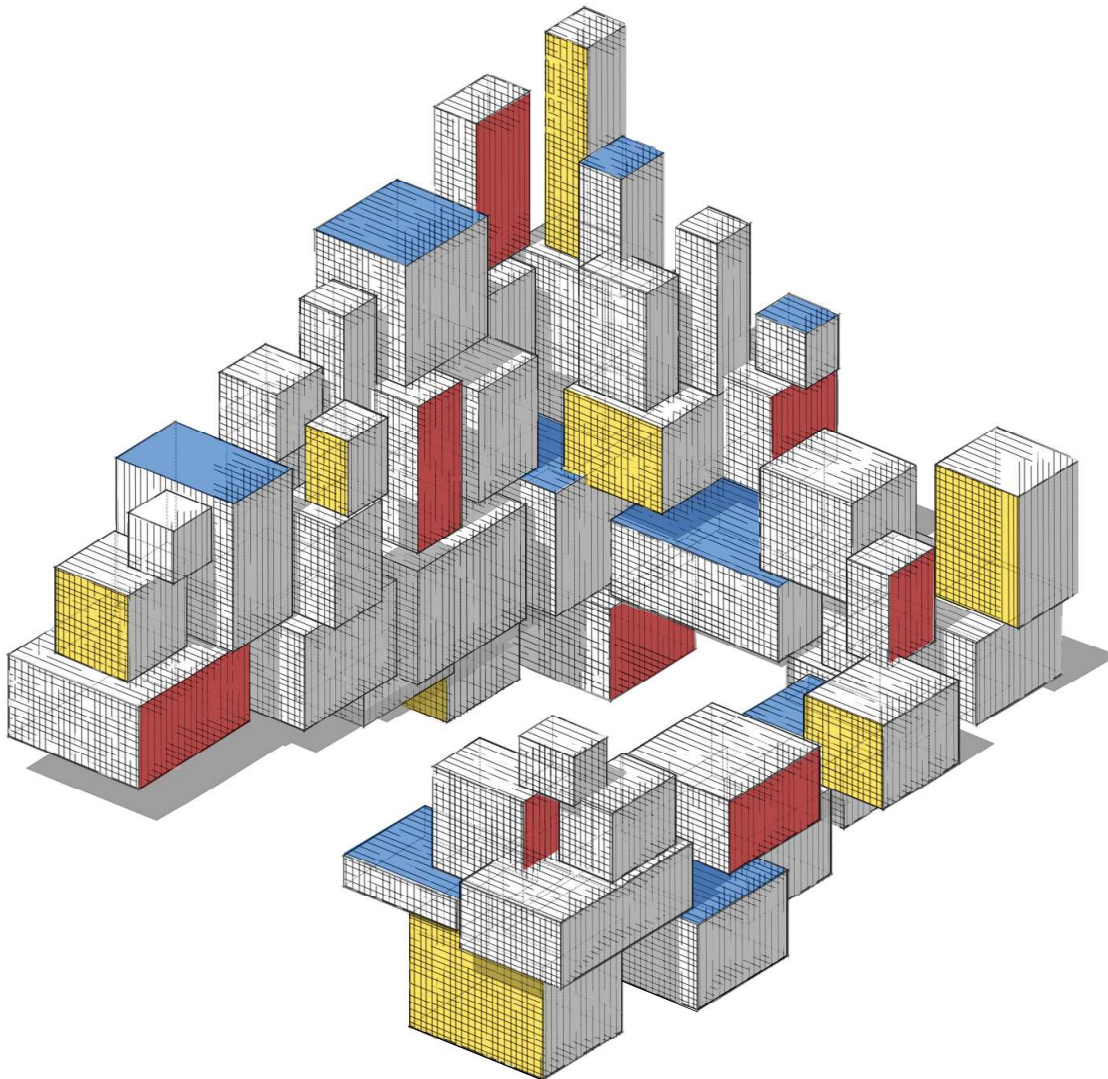


Data Science and Machine Learning

Mathematical and Statistical Methods



Dirk P. Kroese, Zdravko I. Botev, Thomas Taimre, Radislav Vaisman

8th May 2022

To my wife and daughters: Lesley, Elise, and Jessica

— DPK

To Sarah, Sofia, and my parents

— ZIB

To my grandparents: Arno, Harry, Jutta, and Maila

— TT

To Valerie

— RV

CONTENTS

Preface	xiii
Notation	xvii
1 Importing, Summarizing, and Visualizing Data	1
1.1 Introduction	1
1.2 Structuring Features According to Type	3
1.3 Summary Tables	6
1.4 Summary Statistics	7
1.5 Visualizing Data	8
1.5.1 Plotting Qualitative Variables	9
1.5.2 Plotting Quantitative Variables	9
1.5.3 Data Visualization in a Bivariate Setting	12
Exercises	15
2 Statistical Learning	19
2.1 Introduction	19
2.2 Supervised and Unsupervised Learning	20
2.3 Training and Test Loss	23
2.4 Tradeoffs in Statistical Learning	31
2.5 Estimating Risk	35
2.5.1 In-Sample Risk	35
2.5.2 Cross-Validation	37
2.6 Modeling Data	40
2.7 Multivariate Normal Models	44
2.8 Normal Linear Models	46
2.9 Bayesian Learning	47
Exercises	58
3 Monte Carlo Methods	67
3.1 Introduction	67
3.2 Monte Carlo Sampling	68
3.2.1 Generating Random Numbers	68
3.2.2 Simulating Random Variables	69
3.2.3 Simulating Random Vectors and Processes	74
3.2.4 Resampling	76
3.2.5 Markov Chain Monte Carlo	78
3.3 Monte Carlo Estimation	85

3.3.1	Crude Monte Carlo	85
3.3.2	Bootstrap Method	88
3.3.3	Variance Reduction	92
3.4	Monte Carlo for Optimization	96
3.4.1	Simulated Annealing	96
3.4.2	Cross-Entropy Method	100
3.4.3	Splitting for Optimization	103
3.4.4	Noisy Optimization	105
	Exercises	113
4	Unsupervised Learning	121
4.1	Introduction	121
4.2	Risk and Loss in Unsupervised Learning	122
4.3	Expectation–Maximization (EM) Algorithm	128
4.4	Empirical Distribution and Density Estimation	131
4.5	Clustering via Mixture Models	135
4.5.1	Mixture Models	135
4.5.2	EM Algorithm for Mixture Models	137
4.6	Clustering via Vector Quantization	142
4.6.1	K-Means	144
4.6.2	Clustering via Continuous Multiextremal Optimization	146
4.7	Hierarchical Clustering	147
4.8	Principal Component Analysis (PCA)	153
4.8.1	Motivation: Principal Axes of an Ellipsoid	153
4.8.2	PCA and Singular Value Decomposition (SVD)	155
	Exercises	160
5	Regression	167
5.1	Introduction	167
5.2	Linear Regression	169
5.3	Analysis via Linear Models	171
5.3.1	Parameter Estimation	171
5.3.2	Model Selection and Prediction	172
5.3.3	Cross-Validation and Predictive Residual Sum of Squares	173
5.3.4	In-Sample Risk and Akaike Information Criterion	175
5.3.5	Categorical Features	177
5.3.6	Nested Models	180
5.3.7	Coefficient of Determination	181
5.4	Inference for Normal Linear Models	182
5.4.1	Comparing Two Normal Linear Models	183
5.4.2	Confidence and Prediction Intervals	186
5.5	Nonlinear Regression Models	188
5.6	Linear Models in Python	191
5.6.1	Modeling	191
5.6.2	Analysis	193
5.6.3	Analysis of Variance (ANOVA)	195

5.6.4	Confidence and Prediction Intervals	198
5.6.5	Model Validation	198
5.6.6	Variable Selection	199
5.7	Generalized Linear Models	204
	Exercises	207
6	Regularization and Kernel Methods	215
6.1	Introduction	215
6.2	Regularization	216
6.3	Reproducing Kernel Hilbert Spaces	222
6.4	Construction of Reproducing Kernels	225
6.4.1	Reproducing Kernels via Feature Mapping	225
6.4.2	Kernels from Characteristic Functions	225
6.4.3	Reproducing Kernels Using Orthonormal Features	227
6.4.4	Kernels from Kernels	229
6.5	Representer Theorem	231
6.6	Smoothing Cubic Splines	235
6.7	Gaussian Process Regression	239
6.8	Kernel PCA	243
	Exercises	246
7	Classification	253
7.1	Introduction	253
7.2	Classification Metrics	255
7.3	Classification via Bayes' Rule	259
7.4	Linear and Quadratic Discriminant Analysis	261
7.5	Logistic Regression and Softmax Classification	268
7.6	K -Nearest Neighbors Classification	270
7.7	Support Vector Machine	271
7.8	Classification with Scikit-Learn	279
	Exercises	281
8	Decision Trees and Ensemble Methods	289
8.1	Introduction	289
8.2	Top-Down Construction of Decision Trees	291
8.2.1	Regional Prediction Functions	292
8.2.2	Splitting Rules	293
8.2.3	Termination Criterion	294
8.2.4	Basic Implementation	296
8.3	Additional Considerations	300
8.3.1	Binary Versus Non-Binary Trees	300
8.3.2	Data Preprocessing	300
8.3.3	Alternative Splitting Rules	300
8.3.4	Categorical Variables	301
8.3.5	Missing Values	301
8.4	Controlling the Tree Shape	302
8.4.1	Cost-Complexity Pruning	305

8.4.2	Advantages and Limitations of Decision Trees	306
8.5	Bootstrap Aggregation	307
8.6	Random Forests	311
8.7	Boosting	315
	Exercises	323
9	Deep Learning	325
9.1	Introduction	325
9.2	Feed-Forward Neural Networks	328
9.3	Back-Propagation	332
9.4	Methods for Training	336
9.4.1	Steepest Descent	336
9.4.2	Levenberg–Marquardt Method	337
9.4.3	Limited-Memory BFGS Method	338
9.4.4	Adaptive Gradient Methods	340
9.5	Examples in Python	342
9.5.1	Simple Polynomial Regression	342
9.5.2	Image Classification	346
	Exercises	350
A	Linear Algebra and Functional Analysis	357
A.1	Vector Spaces, Bases, and Matrices	357
A.2	Inner Product	362
A.3	Complex Vectors and Matrices	363
A.4	Orthogonal Projections	364
A.5	Eigenvalues and Eigenvectors	365
A.5.1	Left- and Right-Eigenvectors	366
A.6	Matrix Decompositions	370
A.6.1	(P)LU Decomposition	370
A.6.2	Woodbury Identity	372
A.6.3	Cholesky Decomposition	375
A.6.4	QR Decomposition and the Gram–Schmidt Procedure	377
A.6.5	Singular Value Decomposition	378
A.6.6	Solving Structured Matrix Equations	381
A.7	Functional Analysis	386
A.8	Fourier Transforms	392
A.8.1	Discrete Fourier Transform	394
A.8.2	Fast Fourier Transform	396
B	Multivariate Differentiation and Optimization	399
B.1	Multivariate Differentiation	399
B.1.1	Taylor Expansion	402
B.1.2	Chain Rule	402
B.2	Optimization Theory	404
B.2.1	Convexity and Optimization	405
B.2.2	Lagrangian Method	408
B.2.3	Duality	409

B.3	Numerical Root-Finding and Minimization	410
B.3.1	Newton-Like Methods	411
B.3.2	Quasi-Newton Methods	413
B.3.3	Normal Approximation Method	415
B.3.4	Nonlinear Least Squares	416
B.4	Constrained Minimization via Penalty Functions	417
C	Probability and Statistics	423
C.1	Random Experiments and Probability Spaces	423
C.2	Random Variables and Probability Distributions	424
C.3	Expectation	428
C.4	Joint Distributions	429
C.5	Conditioning and Independence	430
C.5.1	Conditional Probability	430
C.5.2	Independence	430
C.5.3	Expectation and Covariance	431
C.5.4	Conditional Density and Conditional Expectation	433
C.6	Functions of Random Variables	433
C.7	Multivariate Normal Distribution	436
C.8	Convergence of Random Variables	441
C.9	Law of Large Numbers and Central Limit Theorem	447
C.10	Markov Chains	453
C.11	Statistics	455
C.12	Estimation	456
C.12.1	Method of Moments	457
C.12.2	Maximum Likelihood Method	458
C.13	Confidence Intervals	459
C.14	Hypothesis Testing	460
D	Python Primer	465
D.1	Getting Started	465
D.2	Python Objects	467
D.3	Types and Operators	468
D.4	Functions and Methods	470
D.5	Modules	471
D.6	Flow Control	473
D.7	Iteration	474
D.8	Classes	475
D.9	Files	477
D.10	NumPy	480
D.10.1	Creating and Shaping Arrays	480
D.10.2	Slicing	482
D.10.3	Array Operations	482
D.10.4	Random Numbers	484
D.11	Matplotlib	485
D.11.1	Creating a Basic Plot	485

D.12 Pandas	487
D.12.1 Series and DataFrame	487
D.12.2 Manipulating Data Frames	489
D.12.3 Extracting Information	490
D.12.4 Plotting	492
D.13 Scikit-learn	492
D.13.1 Partitioning the Data	493
D.13.2 Standardization	493
D.13.3 Fitting and Prediction	494
D.13.4 Testing the Model	494
D.14 System Calls, URL Access, and Speed-Up	495
Bibliography	497
Index	505

PREFACE

In our present world of automation, cloud computing, algorithms, artificial intelligence, and big data, few topics are as relevant as *data science* and *machine learning*. Their recent popularity lies not only in their applicability to real-life questions, but also in their natural blending of many different disciplines, including mathematics, statistics, computer science, engineering, science, and finance.

To someone starting to learn these topics, the multitude of computational techniques and mathematical ideas may seem overwhelming. Some may be satisfied with only learning how to use off-the-shelf recipes to apply to practical situations. But what if the assumptions of the black-box recipe are violated? Can we still trust the results? How should the algorithm be adapted? To be able to truly understand data science and machine learning it is important to appreciate the underlying mathematics and statistics, as well as the resulting algorithms.

The purpose of this book is to provide an accessible, yet comprehensive, account of data science and machine learning. It is intended for anyone interested in gaining a better understanding of the mathematics and statistics that underpin the rich variety of ideas and machine learning algorithms in data science. Our viewpoint is that computer languages come and go, but the underlying key ideas and algorithms will remain forever and will form the basis for future developments.

Before we turn to a description of the topics in this book, we would like to say a few words about its philosophy. This book resulted from various courses in data science and machine learning at the Universities of Queensland and New South Wales, Australia. When we taught these courses, we noticed that students were eager to learn not only how to apply algorithms but also to understand how these algorithms actually work. However, many existing textbooks assumed either too much background knowledge (e.g., measure theory and functional analysis) or too little (everything is a black box), and the information overload from often disjointed and contradictory internet sources made it more difficult for students to gradually build up their knowledge and understanding. We therefore wanted to write a book about data science and machine learning that can be read as a linear story, with a substantial “backstory” in the appendices. The main narrative starts very simply and builds up gradually to quite an advanced level. The backstory contains all the necessary

background, as well as additional information, from linear algebra and functional analysis (Appendix A), multivariate differentiation and optimization (Appendix B), and probability and statistics (Appendix C). Moreover, to make the abstract ideas come alive, we believe it is important that the reader sees actual implementations of the algorithms, directly translated from the theory. After some deliberation we have chosen Python as our programming language. It is freely available and has been adopted as the programming language of choice for many practitioners in data science and machine learning. It has many useful packages for data manipulation (often ported from R) and has been designed to be easy to program. A gentle introduction to Python is given in Appendix D.

KEYWORDS

To keep the book manageable in size we had to be selective in our choice of topics. Important ideas and connections between various concepts are highlighted via *keywords* and page references (indicated by a 📖) in the margin. Key definitions and theorems are highlighted in boxes. Whenever feasible we provide proofs of theorems. Finally, we place great importance on *notation*. It is often the case that once a consistent and concise system of notation is in place, seemingly difficult ideas suddenly become obvious. We use different fonts to distinguish between different types of objects. Vectors are denoted by letters in boldface italics, \mathbf{x} , \mathbf{X} , and matrices by uppercase letters in boldface roman font, \mathbf{A} , \mathbf{K} . We also distinguish between random vectors and their values by using upper and lower case letters, e.g., \mathbf{X} (random vector) and \mathbf{x} (its value or outcome). Sets are usually denoted by calligraphic letters \mathcal{G} , \mathcal{H} . The symbols for probability and expectation are \mathbb{P} and \mathbb{E} , respectively. Distributions are indicated by sans serif font, as in Bin and Gamma; exceptions are the ubiquitous notations \mathcal{N} and \mathcal{U} for the normal and uniform distributions. A summary of the most important symbols and abbreviations is given on Pages xvii–xxi.

📖 xvii

Data science provides the language and techniques necessary for understanding and dealing with data. It involves the design, collection, analysis, and interpretation of numerical data, with the aim of extracting patterns and other useful information. Machine learning, which is closely related to data science, deals with the design of algorithms and computer resources to learn from data. The organization of the book follows roughly the typical steps in a data science project: Gathering data to gain information about a research question; cleaning, summarization, and visualization of the data; modeling and analysis of the data; translating decisions about the model into decisions and predictions about the research question. As this is a mathematics and statistics oriented book, most emphasis will be on modeling and analysis.

We start in Chapter 1 with the reading, structuring, summarization, and visualization of data using the data manipulation package **pandas** in Python. Although the material covered in this chapter requires no mathematical knowledge, it forms an obvious starting point for data science: to better understand the nature of the available data. In Chapter 2, we introduce the main ingredients of *statistical learning*. We distinguish between *supervised* and *unsupervised* learning techniques, and discuss how we can assess the predictive performance of (un)supervised learning methods. An important part of statistical learning is the *modeling* of data. We introduce various useful models in data science including linear, multivariate Gaussian, and Bayesian models. Many algorithms in machine learning and data science make use of Monte Carlo techniques, which is the topic of Chapter 3. Monte Carlo can be used for simulation, estimation, and optimization. Chapter 4 is concerned with unsupervised learning, where we discuss techniques such as density estimation, clustering, and principal component analysis. We then turn our attention to supervised learning

in Chapter 5, and explain the ideas behind a broad class of regression models. Therein, we also describe how Python's **statsmodels** package can be used to define and analyze linear models. Chapter 6 builds upon the previous regression chapter by developing the powerful concepts of kernel methods and regularization, which allow the fundamental ideas of Chapter 5 to be expanded in an elegant way, using the theory of reproducing kernel Hilbert spaces. In Chapter 7, we proceed with the classification task, which also belongs to the supervised learning framework, and consider various methods for classification, including Bayes classification, linear and quadratic discriminant analysis, K -nearest neighbors, and support vector machines. In Chapter 8 we consider versatile methods for regression and classification that make use of tree structures. Finally, in Chapter 9, we consider the workings of neural networks and deep learning, and show that these learning algorithms have a simple mathematical interpretation. An extensive range of exercises is provided at the end of each chapter.



Python code and data sets for each chapter can be downloaded from the GitHub site:
<https://github.com/DSML-book>

Acknowledgments

Some of the Python code for Chapters 1 and 5 was adapted from [73]. We thank Benoit Lique for making this available, and Lauren Jones for translating the R code into Python.

We thank all who through their comments, feedback, and suggestions have contributed to this book, including Qibin Duan, Luke Taylor, Rémi Mouzayek, Harry Goodman, Bryce Stansfield, Ryan Tongs, Dillon Steyl, Bill Rudd, Nan Ye, Christian Hirsch, Chris van der Heide, Sarat Moka, Aapeli Vuorinen, Joshua Ross, Giang Nguyen, and the anonymous referees. David Grubbs deserves a special accolade for his professionalism and attention to detail in his role as Editor for this book.

The book was test-run during the 2019 *Summer School of the Australian Mathematical Sciences Institute*. More than 80 bright upper-undergraduate (Honours) students used the book for the course *Mathematical Methods for Machine Learning*, taught by Zdravko Botev. We are grateful for the valuable feedback that they provided.

Our special thanks go out to Robert Salomone, Liam Berry, Robin Carrick, and Sam Daley, who commented in great detail on earlier versions of the entire book and wrote and improved our Python code. Their enthusiasm, perceptiveness, and kind assistance have been invaluable.

Of course, none of this work would have been possible without the loving support, patience, and encouragement from our families, and we thank them with all our hearts.

This book was financially supported by the Australian Research Council *Centre of Excellence for Mathematical & Statistical Frontiers*, under grant number CE140100049.

Dirk Kroese, Zdravko Botev,
Thomas Taimre, and Radislav Vaisman
Brisbane and Sydney

NOTATION

We could, of course, use any notation we want; do not laugh at notations; invent them, they are powerful. In fact, mathematics is, to a large extent, invention of better notations.

Richard P. Feynman

We have tried to use a notation system that is, in order of importance, simple, descriptive, consistent, and compatible with historical choices. Achieving all of these goals all of the time would be impossible, but we hope that our notation helps to quickly recognize the type or “flavor” of certain mathematical objects (vectors, matrices, random vectors, probability measures, etc.) and clarify intricate ideas.

We make use of various typographical aids, and it will be beneficial for the reader to be aware of some of these.

- Boldface font is used to indicate composite objects, such as column vectors $\mathbf{x} = [x_1, \dots, x_n]^\top$ and matrices $\mathbf{X} = [x_{ij}]$. Note also the difference between the upright bold font for matrices and the slanted bold font for vectors.
- Random variables are generally specified with upper case roman letters X, Y, Z and their outcomes with lower case letters x, y, z . Random vectors are thus denoted in upper case slanted bold font: $\mathbf{X} = [X_1, \dots, X_n]^\top$.
- Sets of vectors are generally written in calligraphic font, such as \mathcal{X} , but the set of real numbers uses the common blackboard bold font \mathbb{R} . Expectation and probability also use the latter font.
- Probability distributions use a sans serif font, such as Bin and Gamma . Exceptions to this rule are the “standard” notations \mathcal{N} and \mathcal{U} for the normal and uniform distributions.
- We often omit brackets when it is clear what the argument is of a function or operator. For example, we prefer $\mathbb{E}X^2$ to $\mathbb{E}[X^2]$.

- We employ color to emphasize that certain words refer to a **dataset**, **function**, or **package** in Python. All code is written in typewriter font. To be compatible with past notation choices, we introduced a special blue symbol **X** for the model (design) matrix of a linear model.
- Important notation such as \mathcal{T} , g , g^* is often defined in a mnemonic way, such as \mathcal{T} for “training”, g for “guess”, g^* for the “star” (that is, optimal) guess, and ℓ for “loss”.
- We will occasionally use a Bayesian notation convention in which the *same* symbol is used to denote different (conditional) probability densities. In particular, instead of writing $f_X(x)$ and $f_{X|Y}(x|y)$ for the probability density function (pdf) of X and the conditional pdf of X given Y , we simply write $f(x)$ and $f(x|y)$. This particular style of notation can be of great descriptive value, despite its apparent ambiguity.

General font/notation rules

x	scalar
\mathbf{x}	vector
\mathbf{X}	random vector
\mathbf{X}	matrix
\mathcal{X}	set
\widehat{x}	estimate or approximation
x^*	optimal
\bar{x}	average

Common mathematical symbols

\forall	for all
\exists	there exists
\propto	is proportional to
\perp	is perpendicular to
\sim	is distributed as
$\overset{\text{iid}}{\sim}, \sim_{\text{iid}}$	are independent and identically distributed as
$\overset{\text{approx.}}{\sim}$	is approximately distributed as
∇f	gradient of f
$\nabla^2 f$	Hessian of f
$f \in C^p$	f has continuous derivatives of order p
\approx	is approximately
\simeq	is asymptotically
\ll	is much smaller than
\oplus	direct sum

\odot	elementwise product
\cap	intersection
\cup	union
$:=, =:$	is defined as
$\xrightarrow{\text{a.s.}}$	converges almost surely to
\xrightarrow{d}	converges in distribution to
$\xrightarrow{\mathbb{P}}$	converges in probability to
$\xrightarrow{L_p}$	converges in L_p -norm to
$\ \cdot\ $	Euclidean norm
$\lceil x \rceil$	smallest integer larger than x
$\lfloor x \rfloor$	largest integer smaller than x
x_+	$\max\{x, 0\}$

Matrix/vector notation

$\mathbf{A}^\top, \mathbf{x}^\top$	transpose of matrix \mathbf{A} or vector \mathbf{x}
\mathbf{A}^{-1}	inverse of matrix \mathbf{A}
\mathbf{A}^+	pseudo-inverse of matrix \mathbf{A}
$\mathbf{A}^{-\top}$	inverse of matrix \mathbf{A}^\top or transpose of \mathbf{A}^{-1}
$\mathbf{A} > 0$	matrix \mathbf{A} is positive definite
$\mathbf{A} \geq 0$	matrix \mathbf{A} is positive semidefinite
$\dim(\mathbf{x})$	dimension of vector \mathbf{x}
$\det(\mathbf{A})$	determinant of matrix \mathbf{A}
$ \mathbf{A} $	absolute value of the determinant of matrix \mathbf{A}
$\text{tr}(\mathbf{A})$	trace of matrix \mathbf{A}

Reserved letters and words

\mathbb{C}	set of complex numbers
d	differential symbol
\mathbb{E}	expectation
e	the number 2.71828...
f	probability density (discrete or continuous)
g	prediction function
$\mathbb{1}\{A\}$ or $\mathbb{1}_A$	indicator function of set A
i	the square root of -1
ℓ	risk: expected loss

Loss	loss function
\ln	(natural) logarithm
\mathbb{N}	set of natural numbers $\{0, 1, \dots\}$
\mathcal{O}	big-O order symbol: $f(x) = \mathcal{O}(g(x))$ if $ f(x) \leq \alpha g(x)$ for some constant α as $x \rightarrow a$
o	little-o order symbol: $f(x) = o(g(x))$ if $f(x)/g(x) \rightarrow 0$ as $x \rightarrow a$
\mathbb{P}	probability measure
π	the number 3.14159...
\mathbb{R}	set of real numbers (one-dimensional Euclidean space)
\mathbb{R}^n	n -dimensional Euclidean space
\mathbb{R}_+	positive real line: $[0, \infty)$
τ	deterministic training set
\mathcal{T}	random training set
\mathbf{X}	model (design) matrix
\mathbb{Z}	set of integers $\{\dots, -1, 0, 1, \dots\}$

Probability distributions

Ber	Bernoulli
Beta	beta
Bin	binomial
Exp	exponential
Geom	geometric
Gamma	gamma
F	Fisher–Snedecor F
\mathcal{N}	normal or Gaussian
Pareto	Pareto
Poi	Poisson
t	Student's t
\mathcal{U}	uniform

Abbreviations and acronyms

cdf	cumulative distribution function
CMC	crude Monte Carlo
CE	cross-entropy
EM	expectation–maximization
GP	Gaussian process
KDE	Kernel density estimate/estimator

KL	Kullback–Leibler
KKT	Karush–Kuhn–Tucker
iid	independent and identically distributed
MAP	maximum <i>a posteriori</i>
MCMC	Markov chain Monte Carlo
MLE	maximum likelihood estimator/estimate
OOB	out-of-bag
PCA	principal component analysis
pdf	probability density function (discrete or continuous)
SVD	singular value decomposition

IMPORTING, SUMMARIZING, AND VISUALIZING DATA

This chapter describes where to find useful data sets, how to load them into Python, and how to (re)structure the data. We also discuss various ways in which the data can be summarized via tables and figures. Which type of plots and numerical summaries are appropriate depends on the type of the variable(s) in play. Readers unfamiliar with Python are advised to read Appendix D first.

1.1 Introduction

Data comes in many shapes and forms, but can generally be thought of as being the result of some random experiment — an experiment whose outcome cannot be determined in advance, but whose workings are still subject to analysis. Data from a random experiment are often stored in a table or spreadsheet. A statistical convention is to denote variables — often called *features* — as *columns* and the individual items (or units) as *rows*. It is useful to think of three types of columns in such a spreadsheet:

FEATURES

1. The first column is usually an identifier or index column, where each unit/row is given a unique name or ID.
2. Certain columns (features) can correspond to the design of the experiment, specifying, for example, to which experimental group the unit belongs. Often the entries in these columns are *deterministic*; that is, they stay the same if the experiment were to be repeated.
3. Other columns represent the observed measurements of the experiment. Usually, these measurements exhibit *variability*; that is, they would change if the experiment were to be repeated.

There are many data sets available from the Internet and in software packages. A well-known repository of data sets is the Machine Learning Repository maintained by the University of California at Irvine (UCI), found at <https://archive.ics.uci.edu/>.

These data sets are typically stored in a CSV (comma separated values) format, which can be easily read into Python. For example, to access the **abalone** data set from this website with Python, download the file to your working directory, import the **pandas** package via

```
import pandas as pd
```

and read in the data as follows:

```
abalone = pd.read_csv('abalone.data', header = None)
```

It is important to add `header = None`, as this lets Python know that the first line of the CSV does not contain the names of the features, as it assumes so by default. The data set was originally used to predict the age of abalone from physical measurements, such as shell weight and diameter.

Another useful repository of over 1000 data sets from various packages in the R programming language, collected by Vincent Arel-Bundock, can be found at:

<https://vincentarelbundock.github.io/Rdatasets/datasets.html>.

For example, to read Fisher's famous **iris** data set from R's **datasets** package into Python, type:

```
urlprefix = 'https://vincentarelbundock.github.io/Rdatasets/csv/'
dataname = 'datasets/iris.csv'
iris = pd.read_csv(urlprefix + dataname)
```

The **iris** data set contains four physical measurements (sepal/petal length/width) on 50 specimens (each) of 3 species of iris: setosa, versicolor, and virginica. Note that in this case the headers are included. The output of `read_csv` is a **DataFrame** object, which is **pandas**'s implementation of a spreadsheet; see Section D.12.1. The **DataFrame** method `head` gives the first few rows of the **DataFrame**, including the feature names. The number of rows can be passed as an argument and is 5 by default. For the **iris** **DataFrame**, we have:

```
iris.head()
```

	Unnamed: 0	Sepal.Length	...	Petal.Width	Species
0	1	5.1	...	0.2	setosa
1	2	4.9	...	0.2	setosa
2	3	4.7	...	0.2	setosa
3	4	4.6	...	0.2	setosa
4	5	5.0	...	0.2	setosa

[5 rows x 6 columns]

The names of the features can be obtained via the `columns` attribute of the **DataFrame** object, as in `iris.columns`. Note that the first column is a duplicate index column, whose name (assigned by **pandas**) is 'Unnamed: 0'. We can drop this column and reassign the **iris** object as follows:

```
iris = iris.drop('Unnamed: 0', 1)
```

The data for each feature (corresponding to its specific name) can be accessed by using Python’s *slicing* notation `[]`. For example, the object `iris['Sepal.Length']` contains the 150 sepal lengths.

The first three rows of the **abalone** data set from the UCI repository can be found as follows:

abalone.head(3)									
	0	1	2	3	4	5	6	7	8
0	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
1	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
2	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9

Here, the missing headers have been assigned according to the order of the natural numbers. The names should correspond to Sex, Length, Diameter, Height, Whole weight, Shucked weight, Viscera weight, Shell weight, and Rings, as described in the file with the name `abalone.names` on the UCI website. We can manually add the names of the features to the DataFrame by reassigning the columns attribute, as in:

```
abalone.columns = ['Sex', 'Length', 'Diameter', 'Height',
                   'Whole weight', 'Shucked weight', 'Viscera weight', 'Shell weight',
                   'Rings']
```

1.2 Structuring Features According to Type

We can generally classify features as either quantitative or qualitative. *Quantitative* features possess “numerical quantity”, such as height, age, number of births, etc., and can either be *continuous* or *discrete*. Continuous quantitative features take values in a continuous range of possible values, such as height, voltage, or crop yield; such features capture the idea that measurements can always be made more precisely. Discrete quantitative features have a countable number of possibilities, such as a count.

QUANTITATIVE

In contrast, *qualitative* features do not have a numerical meaning, but their possible values can be divided into a fixed number of categories, such as {M,F} for gender or {blue, black, brown, green} for eye color. For this reason such features are also called *categorical*. A simple rule of thumb is: if it does not make sense to average the data, it is categorical. For example, it does not make sense to average eye colors. Of course it is still possible to represent categorical data with numbers, such as 1 = blue, 2 = black, 3 = brown, but such numbers carry no quantitative meaning. Categorical features are often called *factors*.

QUALITATIVE

CATEGORICAL

FACTORS

When manipulating, summarizing, and displaying data, it is important to correctly specify the type of the variables (features). We illustrate this using the `nutrition_elderly` data set from [73], which contains the results of a study involving nutritional measurements of thirteen features (columns) for 226 elderly individuals (rows). The data set can be obtained from:

http://www.biostatisticien.eu/springerR/nutrition_elderly.xls.

Excel files can be read directly into **pandas** via the `read_excel` method:

```
xls = 'http://www.biostatisticien.eu/springerR/nutrition_elderly.xls'
nutri = pd.read_excel(xls)
```

This creates a DataFrame object **nutri**. The first three rows are as follows:

```
pd.set_option('display.max_columns', 8) # to fit display
nutri.head(3)
```

	gender	situation	tea	...	cooked_fruit_veg	chocol	fat
0	2	1	0	...	4	5	6
1	2	1	1	...	5	1	4
2	2	1	0	...	2	5	4

```
[3 rows x 13 columns]
```

You can check the type (or structure) of the variables via the **info** method of **nutri**.

```
nutri.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 226 entries, 0 to 225
Data columns (total 13 columns):
gender                226 non-null int64
situation             226 non-null int64
tea                   226 non-null int64
coffee               226 non-null int64
height                226 non-null int64
weight                226 non-null int64
age                   226 non-null int64
meat                  226 non-null int64
fish                  226 non-null int64
raw_fruit             226 non-null int64
cooked_fruit_veg      226 non-null int64
chocol                226 non-null int64
fat                   226 non-null int64
dtypes: int64(13)
memory usage: 23.0 KB
```

All 13 features in **nutri** are (at the moment) interpreted by Python as *quantitative* variables, indeed as integers, simply because they have been entered as whole numbers. The *meaning* of these numbers becomes clear when we consider the description of the features, given in Table 1.2. Table 1.1 shows how the variable types should be classified.

Table 1.1: The feature types for the data frame **nutri**.

Qualitative	gender, situation, fat meat, fish, raw_fruit, cooked_fruit_veg, chocol
Discrete quantitative	tea, coffee
Continuous quantitative	height, weight, age

Note that the categories of the qualitative features in the second row of Table 1.1, meat, ..., chocol have a natural order. Such qualitative features are sometimes called *ordinal*, in

Table 1.2: Description of the variables in the nutritional study [73].

Feature	Description	Unit or Coding
gender	Gender	1=Male; 2=Female
situation	Family status	1=Single 2=Living with spouse 3=Living with family 4=Living with someone else
tea	Daily consumption of tea	Number of cups
coffee	Daily consumption of coffee	Number of cups
height	Height	cm
weight	Weight (actually: mass)	kg
age	Age at date of interview	Years
meat	Consumption of meat	0=Never 1=Less than once a week 2=Once a week 3=2–3 times a week 4=4–6 times a week 5=Every day
fish	Consumption of fish	As in meat
raw_fruit	Consumption of raw fruits	As in meat
cooked_fruit_veg	Consumption of cooked fruits and vegetables	As in meat
chocol	Consumption of chocolate	As in meat
fat	Type of fat used for cooking	1=Butter 2=Margarine 3=Peanut oil 4=Sunflower oil 5=Olive oil 6=Mix of vegetable oils (e.g., Isio4) 7=Colza oil 8=Duck or goose fat

contrast to qualitative features without order, which are called *nominal*. We will not make such a distinction in this book.

We can modify the Python value and type for each categorical feature, using the `replace` and `astype` methods. For categorical features, such as `gender`, we can replace the value 1 with 'Male' and 2 with 'Female', and change the type to 'category' as follows.

```
DICT = {1:'Male', 2:'Female'} # dictionary specifies replacement
nutri['gender'] = nutri['gender'].replace(DICT).astype('category')
```

The structure of the other categorical-type features can be changed in a similar way. Continuous features such as `height` should have type `float`:

```
nutri['height'] = nutri['height'].astype(float)
```

We can repeat this for the other variables (see Exercise 2) and save this modified data frame as a CSV file, by using the **pandas** method `to_csv`.

```
nutri.to_csv('nutri.csv', index=False)
```

1.3 Summary Tables

It is often useful to summarize a large spreadsheet of data in a more condensed form. A table of counts or a table of frequencies makes it easier to gain insight into the underlying distribution of a variable, especially if the data are qualitative. Such tables can be obtained with the methods `describe` and `value_counts`.

As a first example, we load the **nutri** DataFrame, which we restructured and saved (see previous section) as 'nutri.csv', and then construct a summary for the feature (column) 'fat'.

```
nutri = pd.read_csv('nutri.csv')
nutri['fat'].describe()
```

```
count      226
unique       8
top    sunflower
freq        68
Name: fat, dtype: object
```

We see that there are 8 different types of fat used and that sunflower has the highest count, with 68 out of 226 individuals using this type of cooking fat. The method `value_counts` gives the counts for the different fat types.

```
nutri['fat'].value_counts()
```

```
sunflower    68
peanut       48
olive        40
margarine    27
Isio4        23
butter       15
duck         4
colza        1
Name: fat, dtype: int64
```



Column labels are also attributes of a DataFrame, and `nutri.fat`, for example, is exactly the same object as `nutri['fat']`.

It is also possible to use **crosstab** to *cross tabulate* between two or more variables, giving a *contingency table*:

CROSS TABULATE

pd.crosstab(nutri.gender, nutri.situation)			
situation	Couple	Family	Single
gender			
Female	56	7	78
Male	63	2	20

We see, for example, that the proportion of single men is substantially smaller than the proportion of single women in the data set of elderly people. To add row and column totals to a table, use `margins=True`.

pd.crosstab(nutri.gender, nutri.situation, margins=True)				
situation	Couple	Family	Single	All
gender				
Female	56	7	78	141
Male	63	2	20	85
All	119	9	98	226

1.4 Summary Statistics

In the following, $\mathbf{x} = [x_1, \dots, x_n]^T$ is a column vector of n numbers. For our **nutri** data, the vector \mathbf{x} could, for example, correspond to the heights of the $n = 226$ individuals.

The *sample mean* of \mathbf{x} , denoted by \bar{x} , is simply the average of the data values:

SAMPLE MEAN

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i.$$

Using the **mean** method in Python for the **nutri** data, we have, for instance:

nutri['height'].mean()	
163.96017699115043	

The *p-sample quantile* ($0 < p < 1$) of \mathbf{x} is a value x such that at least a fraction p of the data is less than or equal to x and at least a fraction $1 - p$ of the data is greater than or equal to x . The *sample median* is the sample 0.5-quantile. The *p-sample quantile* is also called the $100 \times p$ *percentile*. The 25, 50, and 75 sample percentiles are called the first, second, and third *quartiles* of the data. For the **nutri** data they are obtained as follows.

SAMPLE QUANTILE

SAMPLE MEDIAN

QUARTILES

nutri['height'].quantile(q=[0.25, 0.5, 0.75])	
0.25	157.0
0.50	163.0
0.75	170.0

SAMPLE RANGE
SAMPLE VARIANCE

The sample mean and median give information about the *location* of the data, while the distance between sample quantiles (say the 0.1 and 0.9 quantiles) gives some indication of the *dispersion* (spread) of the data. Other measures for dispersion are the *sample range*, $\max_i x_i - \min_i x_i$, the *sample variance*

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2, \quad (1.1)$$

SAMPLE
STANDARD
DEVIATION
457

and the *sample standard deviation* $s = \sqrt{s^2}$. For the **nutri** data, the range (in cm) is:

```
nutri['height'].max() - nutri['height'].min()
48.0
```

The variance (in cm²) is:

```
round(nutri['height'].var(), 2) # round to two decimal places
81.06
```

And the standard deviation can be found via:

```
round(nutri['height'].std(), 2)
9.0
```

We already encountered the **describe** method in the previous section for summarizing qualitative features, via the most frequent count and the number of unique elements. When applied to a *quantitative* feature, it returns instead the minimum, maximum, mean, and the three quartiles. For example, the 'height' feature in the **nutri** data has the following summary statistics.

```
nutri['height'].describe()
count    226.000000
mean     163.960177
std       9.003368
min      140.000000
25%      157.000000
50%      163.000000
75%      170.000000
max      188.000000
Name: height, dtype: float64
```

1.5 Visualizing Data

In this section we describe various methods for visualizing data. The main point we would like to make is that the way in which variables are visualized should always be adapted to the variable types; for example, qualitative data should be plotted differently from quantitative data.



For the rest of this section, it is assumed that `matplotlib.pyplot`, `pandas`, and `numpy`, have been imported in the Python code as follows.

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

1.5.1 Plotting Qualitative Variables

Suppose we wish to display graphically how many elderly people are living by themselves, as a couple, with family, or other. Recall that the data are given in the `situation` column of our `nutri` data. Assuming that we already *restructured the data*, as in Section 1.2, we can make a *barplot* of the number of people in each category via the `plt.bar` function of the standard `matplotlib` plotting library. The inputs are the *x*-axis positions, heights, and widths of each bar respectively.

3
BARPLOT

```
width = 0.35 # the width of the bars
x = [0, 0.8, 1.6] # the bar positions on x-axis
situation_counts=nutri['situation'].value_counts()
plt.bar(x, situation_counts, width, edgecolor = 'black')
plt.xticks(x, situation_counts.index)
plt.show()
```

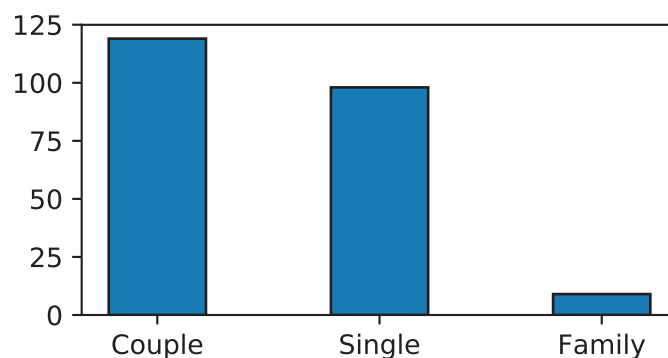


Figure 1.1: Barplot for the qualitative variable 'situation'.

1.5.2 Plotting Quantitative Variables

We now present a few useful methods for visualizing quantitative data, again using the `nutri` data set. We will first focus on continuous features (e.g., 'age') and then add some specific graphs related to discrete features (e.g., 'tea'). The aim is to describe the variability present in a single feature. This typically involves a central tendency, where observations tend to gather around, with fewer observations further away. The main aspects of the distribution are the *location* (or center) of the variability, the *spread* of the variability (how far the values extend from the center), and the *shape* of the variability; e.g., whether or not values are spread symmetrically on either side of the center.

1.5.2.1 Boxplot

BOXPLOT

A *boxplot* can be viewed as a graphical representation of the five-number summary of the data consisting of the minimum, maximum, and the first, second, and third quartiles. Figure 1.2 gives a boxplot for the 'age' feature of the **nutri** data.

```
plt.boxplot(nutri['age'], widths=width, vert=False)
plt.xlabel('age')
plt.show()
```

The `widths` parameter determines the width of the boxplot, which is by default plotted vertically. Setting `vert=False` plots the boxplot horizontally, as in Figure 1.2.

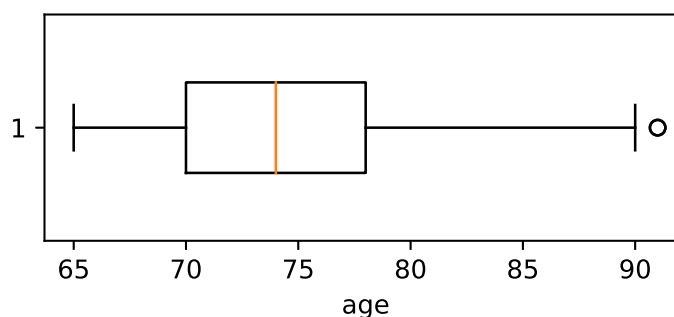


Figure 1.2: Boxplot for 'age'.

The box is drawn from the first quartile (Q_1) to the third quartile (Q_3). The vertical line inside the box signifies the location of the median. So-called “whiskers” extend to either side of the box. The size of the box is called the *interquartile range*: $IQR = Q_3 - Q_1$. The left whisker extends to the largest of (a) the minimum of the data and (b) $Q_1 - 1.5 IQR$. Similarly, the right whisker extends to the smallest of (a) the maximum of the data and (b) $Q_3 + 1.5 IQR$. Any data point outside the whiskers is indicated by a small hollow dot, indicating a suspicious or deviant point (outlier). Note that a boxplot may also be used for discrete quantitative features.

1.5.2.2 Histogram

HISTOGRAM

A *histogram* is a common graphical representation of the distribution of a quantitative feature. We start by breaking the range of the values into a number of *bins* or *classes*. We tally the counts of the values falling in each bin and then make the plot by drawing rectangles whose bases are the bin intervals and whose heights are the counts. In Python we can use the function `plt.hist`. For example, Figure 1.3 shows a histogram of the 226 ages in **nutri**, constructed via the following Python code.

```
weights = np.ones_like(nutri.age)/nutri.age.count()
plt.hist(nutri.age, bins=9, weights=weights, facecolor='cyan',
        edgecolor='black', linewidth=1)
plt.xlabel('age')
plt.ylabel('Proportion of Total')
plt.show()
```

Here 9 bins were used. Rather than using raw counts (the default), the vertical axis here gives the percentage in each class, defined by $\frac{\text{count}}{\text{total}}$. This is achieved by choosing the “weights” parameter to be equal to the vector with entries $1/266$, with length 226. Various plotting parameters have also been changed.

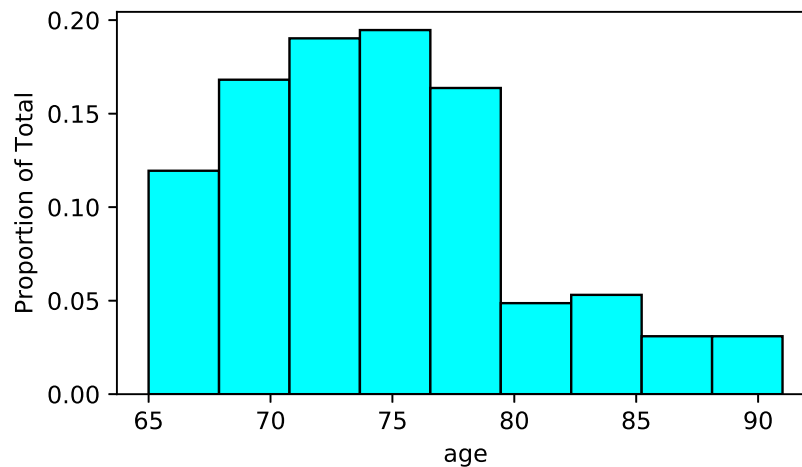


Figure 1.3: Histogram of 'age'.

Histograms can also be used for discrete features, although it may be necessary to explicitly specify the bins and placement of the ticks on the axes.

1.5.2.3 Empirical Cumulative Distribution Function

The *empirical cumulative distribution function*, denoted by F_n , is a step function which jumps an amount k/n at observation values, where k is the number of tied observations at that value. For observations x_1, \dots, x_n , $F_n(x)$ is the fraction of observations less than or equal to x , i.e.,

$$F_n(x) = \frac{\text{number of } x_i \leq x}{n} = \frac{1}{n} \sum_{i=1}^n \mathbb{1}\{x_i \leq x\}, \quad (1.2)$$

where $\mathbb{1}$ denotes the *indicator* function; that is, $\mathbb{1}\{x_i \leq x\}$ is equal to 1 when $x_i \leq x$ and 0 otherwise. To produce a plot of the empirical cumulative distribution function we can use the `plt.step` function. The result for the age data is shown in Figure 1.4. The empirical cumulative distribution function for a discrete quantitative variable is obtained in the same way.

EMPIRICAL
CUMULATIVE
DISTRIBUTION
FUNCTION

INDICATOR

```
x = np.sort(nutri.age)
y = np.linspace(0,1,len(nutri.age))
plt.xlabel('age')
plt.ylabel('Fn(x)')
plt.step(x,y)
plt.xlim(x.min(),x.max())
plt.show()
```

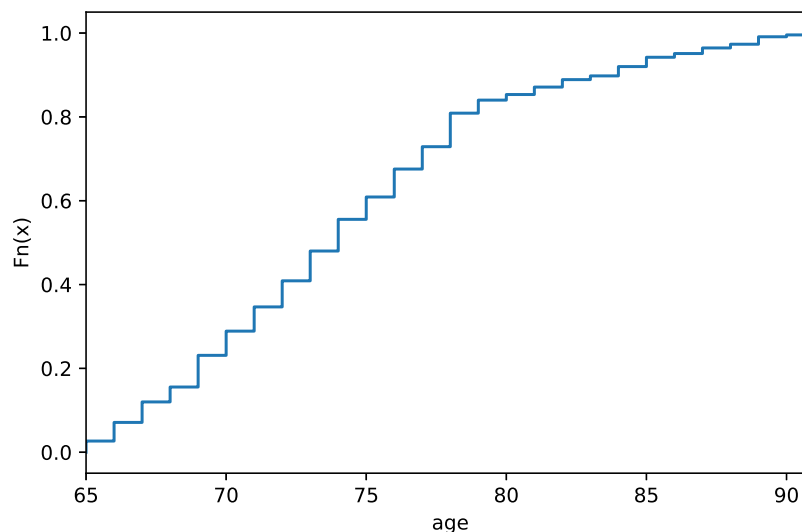


Figure 1.4: Plot of the empirical distribution function for the continuous quantitative feature 'age'.

1.5.3 Data Visualization in a Bivariate Setting

In this section, we present a few useful visual aids to explore relationships between two features. The graphical representation will depend on the type of the two features.

1.5.3.1 Two-way Plots for Two Categorical Variables

Comparing barplots for two categorical variables involves introducing subplots to the figure. Figure 1.5 visualizes the contingency table of Section 1.3, which cross-tabulates the family status (situation) with the gender of the elderly people. It simply shows two barplots next to each other in the same figure.

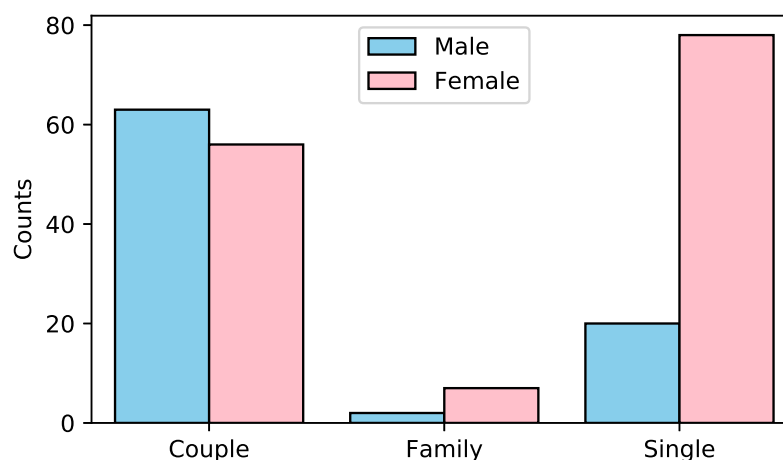


Figure 1.5: Barplot for two categorical variables.

The figure was made using the **seaborn** package, which was specifically designed to simplify statistical visualization tasks.

```
import seaborn as sns
sns.countplot(x='situation', hue = 'gender', data=nutri,
             hue_order = ['Male', 'Female'], palette = ['SkyBlue', 'Pink'],
             saturation = 1, edgecolor='black')
plt.legend(loc='upper center')
plt.xlabel('')
plt.ylabel('Counts')
plt.show()
```

1.5.3.2 Plots for Two Quantitative Variables

We can visualize patterns between two quantitative features using a *scatterplot*. This can be done with `plt.scatter`. The following code produces a scatterplot of 'weight' against 'height' for the **nutri** data.

SCATTERPLOT

```
plt.scatter(nutri.height, nutri.weight, s=12, marker='o')
plt.xlabel('height')
plt.ylabel('weight')
plt.show()
```

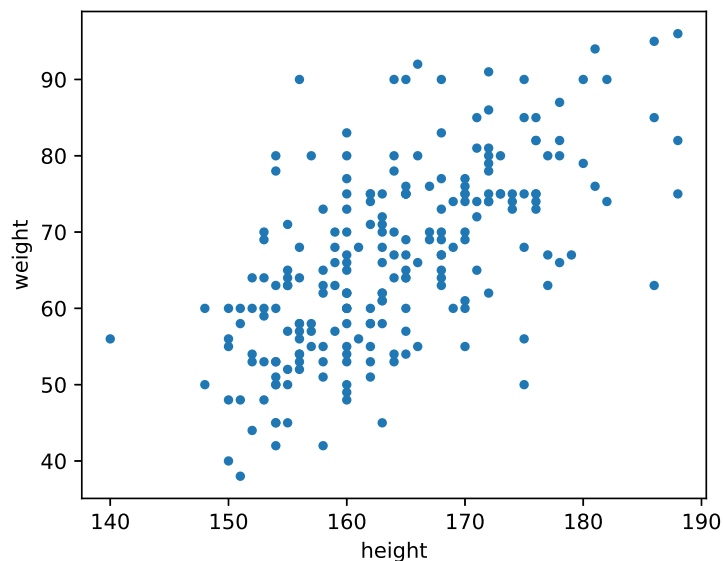


Figure 1.6: Scatterplot of 'weight' against 'height'.

The next Python code illustrates that it is possible to produce highly sophisticated scatter plots, such as in Figure 1.7. The figure shows the birth weights (mass) of babies whose mothers smoked (blue triangles) or not (red circles). In addition, straight lines were fitted to the two groups, suggesting that birth weight decreases with age when the mother smokes, but increases when the mother does not smoke! The question is whether these trends are statistically significant or due to chance. We will revisit this data set later on in the book.

```

urlprefix = 'https://vincentarelbundock.github.io/Rdatasets/csv/'
dataname = 'MASS/birthwt.csv'
bwt = pd.read_csv(urlprefix + dataname)
bwt = bwt.drop('Unnamed: 0',1) #drop unnamed column
styles = {0: ['o', 'red'], 1: ['^', 'blue']}
for k in styles:
    grp = bwt[bwt.smoke==k]
    m,b = np.polyfit(grp.age, grp.bwt, 1) # fit a straight line
    plt.scatter(grp.age, grp.bwt, c=styles[k][1], s=15, linewidth=0,
                marker = styles[k][0])
    plt.plot(grp.age, m*grp.age + b, '-', color=styles[k][1])

plt.xlabel('age')
plt.ylabel('birth weight (g)')
plt.legend(['non-smokers', 'smokers'],prop={'size':8},
           loc=(0.5,0.8))
plt.show()

```

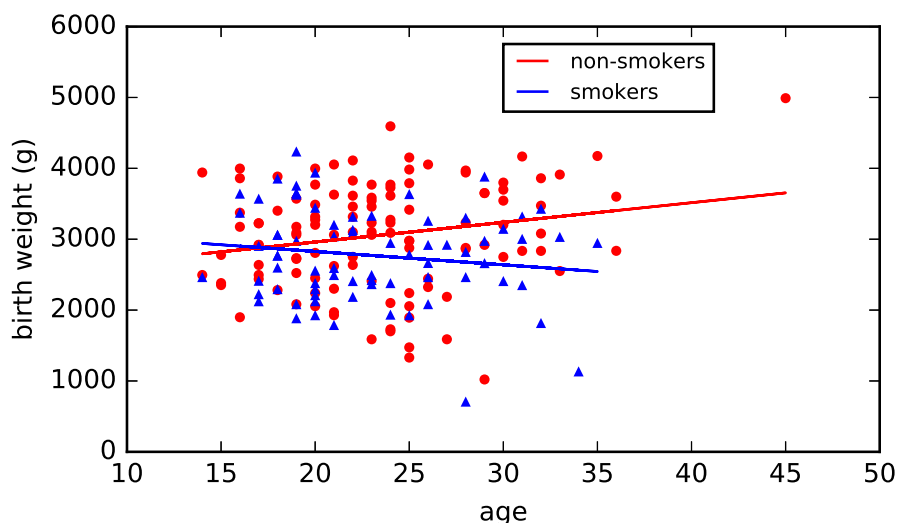


Figure 1.7: Birth weight against age for smoking and non-smoking mothers.

1.5.3.3 Plots for One Qualitative and One Quantitative Variable

In this setting, it is interesting to draw boxplots of the quantitative feature for each level of the categorical feature. Assuming the variables are structured correctly, the function `plt.boxplot` can be used to produce Figure 1.8, using the following code:

```

males = nutri[nutri.gender == 'Male']
females = nutri[nutri.gender == 'Female']
plt.boxplot([males.coffee, females.coffee], notch=True, widths
            =(0.5,0.5))
plt.xlabel('gender')
plt.ylabel('coffee')
plt.xticks([1,2], ['Male', 'Female'])
plt.show()

```

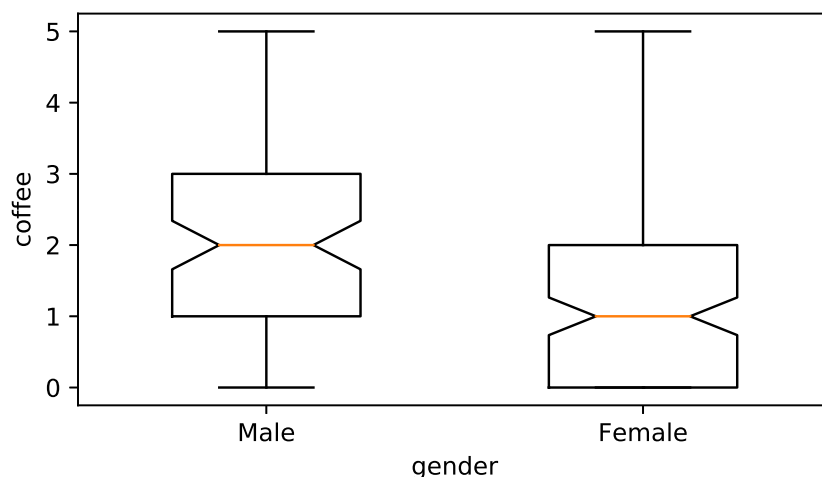



Figure 1.8: Boxplots of a quantitative feature 'coffee' as a function of the levels of a categorical feature 'gender'. Note that we used a different, “notched”, style boxplot this time.

Further Reading

The focus in this book is on the mathematical and statistical analysis of data, and for the rest of the book we assume that the data is available in a suitable form for analysis. However, a large part of practical data science involves the *cleaning* of data; that is, putting it into a form that is amenable to analysis with standard software packages. Standard Python modules such as **numpy** and **pandas** can be used to reformat rows, rename columns, remove faulty outliers, merge rows, and so on. McKinney, the creator of **pandas**, gives many practical case studies in [84]. Effective data visualization techniques are beautifully illustrated in [65].

Exercises

Before you attempt these exercises, make sure you have up-to-date versions of the relevant Python packages, specifically **matplotlib**, **pandas**, and **seaborn**. An easy way to ensure this is to update packages via the Anaconda Navigator, as explained in Appendix D.

1. Visit the UCI Repository <https://archive.ics.uci.edu/>. Read the description of the data and download the Mushroom data set `agaricus-lepiota.data`. Using **pandas**, read the data into a DataFrame called `mushroom`, via `read_csv`.

- (a) How many features are in this data set?
- (b) What are the initial names and types of the features?
- (c) Rename the first feature (index 0) to 'edibility' and the sixth feature (index 5) to 'odor' [Hint: the column names in **pandas** are immutable; so individual columns cannot be modified directly. However it is possible to assign the entire column names list via `mushroom.columns = newcols`.]

- (d) The 6th column lists the various odors of the mushrooms: encoded as 'a', 'c', Replace these with the names 'almond', 'creosote', etc. (categories corresponding to each letter can be found on the website). Also replace the 'edibility' categories 'e' and 'p' with 'edible' and 'poisonous'.
 - (e) Make a contingency table cross-tabulating 'edibility' and 'odor'.
 - (f) Which mushroom odors should be avoided, when gathering mushrooms for consumption?
 - (g) What proportion of odorless mushroom samples were safe to eat?
2. Change the type and value of variables in the **nutri** data set according to Table 1.2 and save the data as a CSV file. The modified data should have eight categorical features, three floats, and two integer features.
 3. It frequently happens that a table with data needs to be restructured before the data can be analyzed using standard statistical software. As an example, consider the test scores in Table 1.3 of 5 students before and after specialized tuition.

Table 1.3: Student scores.

Student	Before	After
1	75	85
2	30	50
3	100	100
4	50	52
5	60	65

This is not in the standard format described in Section 1.1. In particular, the student scores are divided over two columns, whereas the standard format requires that they are collected in one column, e.g., labelled 'Score'. Reformat (by hand) the table in standard format, using three features:

- 'Score', taking continuous values,
- 'Time', taking values 'Before' and 'After',
- 'Student', taking values from 1 to 5.

Useful methods for reshaping tables in **pandas** are **melt**, **stack**, and **unstack**.

4. Create a similar barplot as in Figure 1.5, but now plot the corresponding *proportions* of males and females in each of the three situation categories. That is, the heights of the bars should sum up to 1 for both barplots with the same 'gender' value. [Hint: **seaborn** does not have this functionality built in, instead you need to first create a contingency table and use **matplotlib.pyplot** to produce the figure.]

☞ 2

5. The **iris** data set, mentioned in Section 1.1, contains various features, including 'Petal.Length' and 'Sepal.Length', of three species of iris: setosa, versicolor, and virginica.

- (a) Load the data set into a **pandas** DataFrame object.
- (b) Using **matplotlib.pyplot**, produce boxplots of 'Petal.Length' for each the three species, in one figure.
- (c) Make a histogram with 20 bins for 'Petal.Length'.
- (d) Produce a similar scatterplot for 'Sepal.Length' against 'Petal.Length' to that of the left plot in Figure 1.9. Note that the points should be colored according to the 'Species' feature as per the legend in the right plot of the figure.
- (e) Using the **kdeplot** method of the **seaborn** package, reproduce the right plot of Figure 1.9, where kernel density plots for 'Petal.Length' are given.


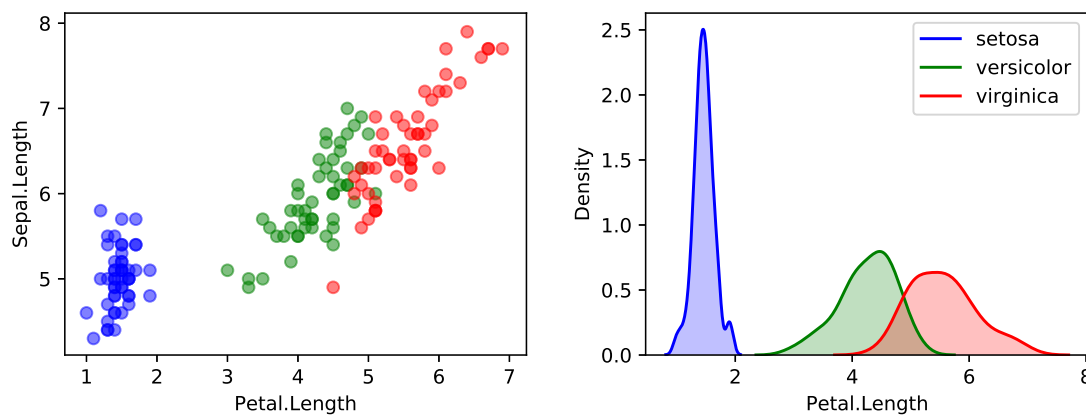
 131


Figure 1.9: Left: scatterplot of 'Sepal.Length' against 'Petal.Length'. Right: kernel density estimates of 'Petal.Length' for the three species of iris.

6. Import the data set **EuStockMarkets** from the same website as the **iris** data set above. The data set contains the daily closing prices of four European stock indices during the 1990s, for 260 working days per year.

- (a) Create a vector of times (working days) for the stock prices, between 1991.496 and 1998.646 with increments of 1/260.
- (b) Reproduce Figure 1.10. [Hint: Use a dictionary to map column names (stock indices) to colors.]

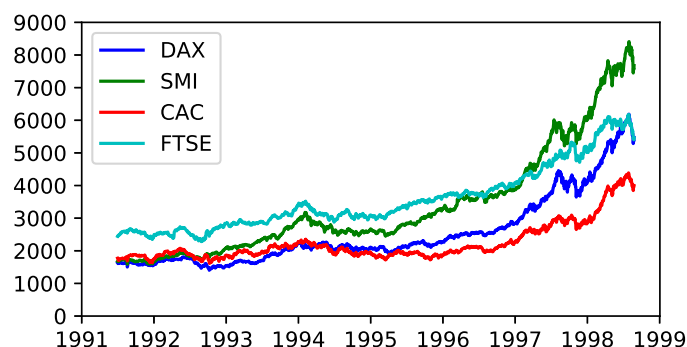


Figure 1.10: Closing stock indices for various European stock markets.

7. Consider the KASANDR data set from the UCI Machine Learning Repository, which can be downloaded from

<https://archive.ics.uci.edu/ml/machine-learning-databases/00385/de.tar.bz2>.

This archive file has a size of 900Mb, so it may take a while to download. Uncompressing the file (e.g., via 7-Zip) yields a directory `de` containing two large CSV files: `test_de.csv` and `train_de.csv`, with sizes 372Mb and 3Gb, respectively. Such large data files can still be processed efficiently in **pandas**, provided there is enough memory. The files contain records of user information from Kelkoo web logs in Germany as well as meta-data on users, offers, and merchants. The data sets have 7 attributes and 1919561 and 15844717 rows, respectively. The data sets are anonymized via hex strings.

(a) Load `train_de.csv` into a **pandas** DataFrame object `de`, using

```
read_csv('train_de.csv', delimiter = '\t').
```

If not enough memory is available, load `test_de.csv` instead. Note that entries are separated here by tabs, not commas. Time how long it takes for the file to load, using the **time** package. (It took 38 seconds for `train_de.csv` to load on one of our computers.)

(b) How many unique users and merchants are in this data set?

8. Visualizing data involving more than two features requires careful design, which is often more of an art than a science.

- Go to Vincent Arel-Bundocks's website (URL given in Section 1.1) and read the Orange data set into a **pandas** DataFrame object called `orange`. Remove its first (unnamed) column.
- The data set contains the circumferences of 5 orange trees at various stages in their development. Find the names of the features.
- In Python, import **seaborn** and visualize the growth curves (circumference against age) of the trees, using the **regplot** and **FacetGrid** methods.

STATISTICAL LEARNING

The purpose of this chapter is to introduce the reader to some common concepts and themes in statistical learning. We discuss the difference between supervised and unsupervised learning, and how we can assess the predictive performance of supervised learning. We also examine the central role that the linear and Gaussian properties play in the modeling of data. We conclude with a section on Bayesian learning. The required probability and statistics background is given in Appendix C.

2.1 Introduction

Although structuring and visualizing data are important aspects of data science, the main challenge lies in the mathematical analysis of the data. When the goal is to interpret the model and quantify the uncertainty in the data, this analysis is usually referred to as *statistical learning*. In contrast, when the emphasis is on making predictions using large-scale data, then it is common to speak about *machine learning* or *data mining*.

There are two major goals for modeling data: 1) to accurately predict some future quantity of interest, given some observed data, and 2) to discover unusual or interesting patterns in the data. To achieve these goals, one must rely on knowledge from three important pillars of the mathematical sciences.

Function approximation. Building a mathematical model for data usually means understanding how one data variable depends on another data variable. The most natural way to represent the relationship between variables is via a mathematical function or map. We usually assume that this mathematical function is not completely known, but can be approximated well given enough computing power and data. Thus, data scientists have to understand how best to approximate and represent functions using the least amount of computer processing and memory.

Optimization. Given a class of mathematical models, we wish to find the best possible model in that class. This requires some kind of efficient search or optimization procedure. The optimization step can be viewed as a process of fitting or calibrating a function to observed data. This step usually requires knowledge of optimization algorithms and efficient computer coding or programming.

STATISTICAL
LEARNING
MACHINE
LEARNING
DATA MINING

Probability and Statistics. In general, the data used to fit the model is viewed as a realization of a random process or numerical vector, whose probability law determines the accuracy with which we can predict future observations. Thus, in order to quantify the uncertainty inherent in making predictions about the future, and the sources of error in the model, data scientists need a firm grasp of probability theory and statistical inference.

2.2 Supervised and Unsupervised Learning

FEATURE
RESPONSE

Given an input or *feature* vector \mathbf{x} , one of the main goals of machine learning is to predict an output or *response* variable y . For example, \mathbf{x} could be a digitized signature and y a binary variable that indicates whether the signature is genuine or false. Another example is where \mathbf{x} represents the weight and smoking habits of an expecting mother and y the birth weight of the baby. The data science attempt at this prediction is encoded in a mathematical function g , called the *prediction function*, which takes as an input \mathbf{x} and outputs a guess $g(\mathbf{x})$ for y (denoted by \hat{y} , for example). In a sense, g encompasses all the information about the relationship between the variables \mathbf{x} and y , excluding the effects of chance and randomness in nature.

PREDICTION
FUNCTION

REGRESSION

In *regression* problems, the response variable y can take any real value. In contrast, when y can only lie in a finite set, say $y \in \{0, \dots, c-1\}$, then predicting y is conceptually the same as classifying the input \mathbf{x} into one of c categories, and so prediction becomes a *classification* problem.

CLASSIFICATION

LOSS FUNCTION

We can measure the accuracy of a prediction \hat{y} with respect to a given response y by using some *loss function* $\text{Loss}(y, \hat{y})$. In a regression setting the usual choice is the squared-error loss $(y - \hat{y})^2$. In the case of classification, the zero-one (also written 0–1) loss function $\text{Loss}(y, \hat{y}) = \mathbb{1}\{y \neq \hat{y}\}$ is often used, which incurs a loss of 1 whenever the predicted class \hat{y} is not equal to the class y . Later on in this book, we will encounter various other useful loss functions, such as the cross-entropy and hinge loss functions (see, e.g., Chapter 7).



The word *error* is often used as a measure of distance between a “true” object y and some approximation \hat{y} thereof. If y is real-valued, the absolute error $|y - \hat{y}|$ and the squared error $(y - \hat{y})^2$ are both well-established error concepts, as are the norm $\|y - \hat{y}\|$ and squared norm $\|y - \hat{y}\|^2$ for vectors. The squared error $(y - \hat{y})^2$ is just one example of a loss function.

RISK

It is unlikely that any mathematical function g will be able to make accurate predictions for all possible pairs (\mathbf{x}, y) one may encounter in Nature. One reason for this is that, even with the same input \mathbf{x} , the output y may be different, depending on chance circumstances or randomness. For this reason, we adopt a probabilistic approach and assume that each pair (\mathbf{x}, y) is the outcome of a random pair (\mathbf{X}, Y) that has some joint probability density $f(\mathbf{x}, y)$. We then assess the predictive performance via the expected loss, usually called the *risk*, for g :

$$\ell(g) = \mathbb{E} \text{Loss}(Y, g(\mathbf{X})). \quad (2.1)$$

For example, in the classification case with zero-one loss function the risk is equal to the probability of incorrect classification: $\ell(g) = \mathbb{P}[Y \neq g(\mathbf{X})]$. In this context, the prediction

function g is called a *classifier*. Given the distribution of (X, Y) and any loss function, we can in principle find the best possible $g^* := \operatorname{argmin}_g \mathbb{E} \operatorname{Loss}(Y, g(X))$ that yields the smallest risk $\ell^* := \ell(g^*)$. We will see in Chapter 7 that in the classification case with $y \in \{0, \dots, c-1\}$ and $\ell(g) = \mathbb{P}[Y \neq g(X)]$, we have

$$g^*(\mathbf{x}) = \operatorname{argmax}_{y \in \{0, \dots, c-1\}} f(y | \mathbf{x}),$$

where $f(y | \mathbf{x}) = \mathbb{P}[Y = y | X = \mathbf{x}]$ is the conditional probability of $Y = y$ given $X = \mathbf{x}$. As already mentioned, for regression the most widely-used loss function is the squared-error loss. In this setting, the optimal prediction function g^* is often called the *regression function*. The following theorem specifies its exact form.

Theorem 2.1: Optimal Prediction Function for Squared-Error Loss

For the squared-error loss $\operatorname{Loss}(y, \hat{y}) = (y - \hat{y})^2$, the optimal prediction function g^* is equal to the conditional expectation of Y given $X = \mathbf{x}$:

$$g^*(\mathbf{x}) = \mathbb{E}[Y | X = \mathbf{x}].$$

Proof: Let $g^*(\mathbf{x}) = \mathbb{E}[Y | X = \mathbf{x}]$. For any function g , the squared-error risk satisfies

$$\begin{aligned} \mathbb{E}(Y - g(X))^2 &= \mathbb{E}[(Y - g^*(X) + g^*(X) - g(X))^2] \\ &= \mathbb{E}(Y - g^*(X))^2 + 2\mathbb{E}[(Y - g^*(X))(g^*(X) - g(X))] + \mathbb{E}(g^*(X) - g(X))^2 \\ &\geq \mathbb{E}(Y - g^*(X))^2 + 2\mathbb{E}[(Y - g^*(X))(g^*(X) - g(X))] \\ &= \mathbb{E}(Y - g^*(X))^2 + 2\mathbb{E}\{(g^*(X) - g(X))\mathbb{E}[Y - g^*(X) | X]\}. \end{aligned}$$

In the last equation we used the tower property. By the definition of the conditional expectation, we have $\mathbb{E}[Y - g^*(X) | X] = 0$. It follows that $\mathbb{E}(Y - g(X))^2 \geq \mathbb{E}(Y - g^*(X))^2$, showing that g^* yields the smallest squared-error risk. \square

One consequence of Theorem 2.1 is that, conditional on $X = \mathbf{x}$, the (random) response Y can be written as

$$Y = g^*(\mathbf{x}) + \varepsilon(\mathbf{x}), \quad (2.2)$$

where $\varepsilon(\mathbf{x})$ can be viewed as the random deviation of the response from its conditional mean at \mathbf{x} . This random deviation satisfies $\mathbb{E} \varepsilon(\mathbf{x}) = 0$. Further, the conditional variance of the response Y at \mathbf{x} can be written as $\mathbb{V}ar \varepsilon(\mathbf{x}) = v^2(\mathbf{x})$ for some unknown positive function v . Note that, in general, the probability distribution of $\varepsilon(\mathbf{x})$ is unspecified.

Since, the optimal prediction function g^* depends on the typically unknown joint distribution of (X, Y) , it is not available in practice. Instead, all that we have available is a finite number of (usually) independent realizations from the joint density $f(\mathbf{x}, y)$. We denote this sample by $\mathcal{T} = \{(X_1, Y_1), \dots, (X_n, Y_n)\}$ and call it the *training set* (\mathcal{T} is a mnemonic for training) with n examples. It will be important to distinguish between a random training set \mathcal{T} and its (deterministic) outcome $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$. We will use the notation τ for the latter. We will also add the subscript n in τ_n when we wish to emphasize the size of the training set.

Our goal is thus to “learn” the unknown g^* using the n examples in the training set \mathcal{T} . Let us denote by $g_{\mathcal{T}}$ the best (by some criterion) approximation for g^* that we can construct

CLASSIFIER

253

REGRESSION
FUNCTION

433

TRAINING SET

LEARNER

from \mathcal{T} . Note that $g_{\mathcal{T}}$ is a random function. A particular outcome is denoted by g_{τ} . It is often useful to think of a teacher–learner metaphor, whereby the function $g_{\mathcal{T}}$ is a *learner* who learns the unknown functional relationship $g^* : \mathbf{x} \mapsto y$ from the training data \mathcal{T} . We can imagine a “teacher” who provides n examples of the true relationship between the output Y_i and the input X_i for $i = 1, \dots, n$, and thus “trains” the learner $g_{\mathcal{T}}$ to predict the output of a new input X , for which the correct output Y is not provided by the teacher (is unknown).

SUPERVISED
LEARNING

The above setting is called *supervised learning*, because one tries to learn the functional relationship between the feature vector \mathbf{x} and response y in the presence of a teacher who provides n examples. It is common to speak of “explaining” or predicting y on the basis of \mathbf{x} , where \mathbf{x} is a vector of *explanatory variables*.

EXPLANATORY
VARIABLES

An example of supervised learning is email spam detection. The goal is to train the learner $g_{\mathcal{T}}$ to accurately predict whether any future email, as represented by the feature vector \mathbf{x} , is spam or not. The training data consists of the feature vectors of a number of different email examples as well as the corresponding labels (spam or not spam). For instance, a feature vector could consist of the number of times sales-pitch words like “free”, “sale”, or “miss out” occur within a given email.

As seen from the above discussion, most questions of interest in supervised learning can be answered if we know the conditional pdf $f(y|\mathbf{x})$, because we can then in principle work out the function value $g^*(\mathbf{x})$.

UNSUPERVISED
LEARNING

In contrast, *unsupervised learning* makes no distinction between response and explanatory variables, and the objective is simply to learn the structure of the unknown distribution of the data. In other words, we need to learn $f(\mathbf{x})$. In this case the guess $g(\mathbf{x})$ is an approximation of $f(\mathbf{x})$ and the risk is of the form

$$\ell(g) = \mathbb{E} \text{Loss}(f(\mathbf{X}), g(\mathbf{X})).$$

An example of unsupervised learning is when we wish to analyze the purchasing behaviors of the customers of a grocery shop that has a total of, say, a hundred items on sale. A feature vector here could be a binary vector $\mathbf{x} \in \{0, 1\}^{100}$ representing the items bought by a customer on a visit to the shop (a 1 in the k -th position if a customer bought item $k \in \{1, \dots, 100\}$ and a 0 otherwise). Based on a training set $\tau = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$, we wish to find any interesting or unusual purchasing patterns. In general, it is difficult to know if an unsupervised learner is doing a good job, because there is no teacher to provide examples of accurate predictions.

☞ 121

The main methodologies for unsupervised learning include *clustering*, *principal component analysis*, and *kernel density estimation*, which will be discussed in Chapter 4.

☞ 167

☞ 253

In the next three sections we will focus on supervised learning. The main supervised learning methodologies are *regression* and *classification*, to be discussed in detail in Chapters 5 and 7. More advanced supervised learning techniques, including *reproducing kernel Hilbert spaces*, *tree methods*, and *deep learning*, will be discussed in Chapters 6, 8, and 9.

2.3 Training and Test Loss

Given an arbitrary prediction function g , it is typically not possible to compute its risk $\ell(g)$ in (2.1). However, using the training sample \mathcal{T} , we can approximate $\ell(g)$ via the empirical (sample average) risk

$$\ell_{\mathcal{T}}(g) = \frac{1}{n} \sum_{i=1}^n \text{Loss}(Y_i, g(X_i)), \quad (2.3)$$

which we call the *training loss*. The training loss is thus an unbiased estimator of the risk (the expected loss) for a prediction function g , based on the training data.

TRAINING LOSS

To approximate the optimal prediction function g^* (the minimizer of the risk $\ell(g)$) we first select a suitable collection of approximating functions \mathcal{G} and then take our *learner* to be the function in \mathcal{G} that minimizes the training loss; that is,

$$g_{\mathcal{T}}^{\mathcal{G}} = \operatorname{argmin}_{g \in \mathcal{G}} \ell_{\mathcal{T}}(g). \quad (2.4)$$

For example, the simplest and most useful \mathcal{G} is the set of *linear* functions of \mathbf{x} ; that is, the set of all functions $g : \mathbf{x} \mapsto \boldsymbol{\beta}^{\top} \mathbf{x}$ for some real-valued vector $\boldsymbol{\beta}$.

We suppress the superscript \mathcal{G} when it is clear which function class is used. Note that minimizing the training loss over all possible functions g (rather than over all $g \in \mathcal{G}$) does not lead to a meaningful optimization problem, as any function g for which $g(X_i) = Y_i$ for all i gives minimal training loss. In particular, for a squared-error loss, the training loss will be 0. Unfortunately, such functions have a poor ability to predict new (that is, independent from \mathcal{T}) pairs of data. This poor generalization performance is called *overfitting*.

OVERFITTING



By choosing g a function that predicts the training data exactly (and is, for example, 0 otherwise), the squared-error training loss is zero. Minimizing the training loss is not the ultimate goal!

The prediction accuracy of new pairs of data is measured by the *generalization risk* of the learner. For a *fixed* training set τ it is defined as

GENERALIZATION
RISK

$$\ell(g_{\tau}^{\mathcal{G}}) = \mathbb{E} \text{Loss}(Y, g_{\tau}^{\mathcal{G}}(X)), \quad (2.5)$$

where (X, Y) is distributed according to $f(\mathbf{x}, y)$. In the discrete case the generalization risk is therefore: $\ell(g_{\tau}^{\mathcal{G}}) = \sum_{\mathbf{x}, y} \text{Loss}(y, g_{\tau}^{\mathcal{G}}(\mathbf{x})) f(\mathbf{x}, y)$ (replace the sum with an integral for the continuous case). The situation is illustrated in Figure 2.1, where the distribution of (X, Y) is indicated by the red dots. The training set (points in the shaded regions) determines a fixed prediction function shown as a straight line. Three possible outcomes of (X, Y) are shown (black dots). The amount of loss for each point is shown as the length of the dashed lines. The generalization risk is the average loss over all possible pairs (\mathbf{x}, y) , weighted by the corresponding $f(\mathbf{x}, y)$.

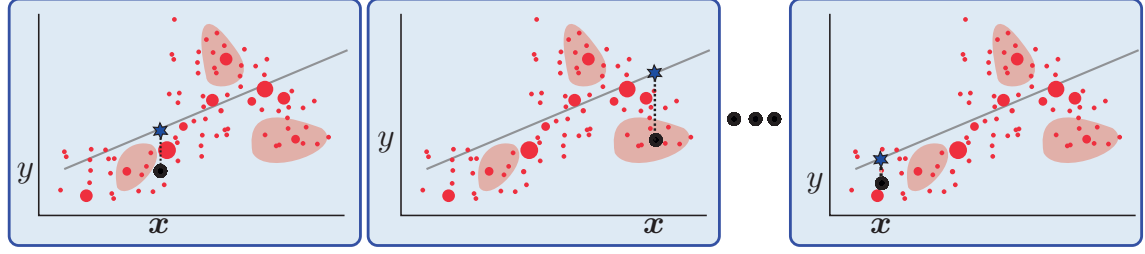


Figure 2.1: The generalization risk for a fixed training set is the weighted-average loss over all possible pairs (\mathbf{x}, y) .

For a *random* training set \mathcal{T} , the generalization risk is thus a random variable that depends on \mathcal{T} (and \mathcal{G}). If we average the generalization risk over all possible instances of \mathcal{T} , we obtain the *expected generalization risk*:

EXPECTED
GENERALIZATION
RISK

$$\mathbb{E} \ell(g_{\mathcal{T}}^{\mathcal{G}}) = \mathbb{E} \text{Loss}(Y, g_{\mathcal{T}}^{\mathcal{G}}(X)), \quad (2.6)$$

where (X, Y) in the expectation above is independent of \mathcal{T} . In the discrete case, we have $\mathbb{E} \ell(g_{\mathcal{T}}^{\mathcal{G}}) = \sum_{\mathbf{x}, y, \mathbf{x}_1, y_1, \dots, \mathbf{x}_n, y_n} \text{Loss}(y, g_{\mathcal{T}}^{\mathcal{G}}(\mathbf{x})) f(\mathbf{x}, y) f(\mathbf{x}_1, y_1) \cdots f(\mathbf{x}_n, y_n)$. Figure 2.2 gives an illustration.

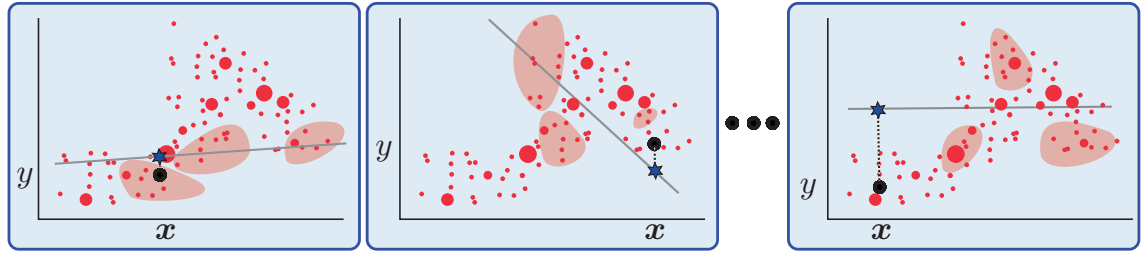


Figure 2.2: The expected generalization risk is the weighted-average loss over all possible pairs (\mathbf{x}, y) and over all training sets.

For any outcome τ of the training data, we can estimate the generalization risk without bias by taking the sample average

$$\ell_{\mathcal{T}'}(g_{\tau}^{\mathcal{G}}) := \frac{1}{n'} \sum_{i=1}^{n'} \text{Loss}(Y'_i, g_{\tau}^{\mathcal{G}}(X'_i)), \quad (2.7)$$

TEST SAMPLE

TEST LOSS

where $\{(X'_1, Y'_1), \dots, (X'_{n'}, Y'_{n'})\} =: \mathcal{T}'$ is a so-called *test sample*. The test sample is completely separate from \mathcal{T} , but is drawn in the same way as \mathcal{T} ; that is, via independent draws from $f(\mathbf{x}, y)$, for some sample size n' . We call the estimator (2.7) the *test loss*. For a random training set \mathcal{T} we can define $\ell_{\mathcal{T}'}(g_{\mathcal{T}}^{\mathcal{G}})$ similarly. It is then crucial to assume that \mathcal{T} is independent of \mathcal{T}' . Table 2.1 summarizes the main definitions and notation for supervised learning.

Table 2.1: Summary of definitions for supervised learning.

\mathbf{x}	Fixed explanatory (feature) vector.
\mathbf{X}	Random explanatory (feature) vector.
y	Fixed (real-valued) response.
Y	Random response.
$f(\mathbf{x}, y)$	Joint pdf of \mathbf{X} and Y , evaluated at (\mathbf{x}, y) .
$f(y \mathbf{x})$	Conditional pdf of Y given $\mathbf{X} = \mathbf{x}$, evaluated at y .
τ or τ_n	Fixed training data $\{(\mathbf{x}_i, y_i), i = 1, \dots, n\}$.
\mathcal{T} or \mathcal{T}_n	Random training data $\{(\mathbf{X}_i, Y_i), i = 1, \dots, n\}$.
\mathbf{X}	Matrix of explanatory variables, with n rows $\mathbf{x}_i^\top, i = 1, \dots, n$ and $\dim(\mathbf{x})$ feature columns; one of the features may be the constant 1.
\mathbf{y}	Vector of response variables $(y_1, \dots, y_n)^\top$.
g	Prediction (guess) function.
$\text{Loss}(y, \hat{y})$	Loss incurred when predicting response y with \hat{y} .
$\ell(g)$	Risk for prediction function g ; that is, $\mathbb{E} \text{Loss}(Y, g(\mathbf{X}))$.
g^*	Optimal prediction function; that is, $\text{argmin}_g \ell(g)$.
$g^{\mathcal{G}}$	Optimal prediction function in function class \mathcal{G} ; that is, $\text{argmin}_{g \in \mathcal{G}} \ell(g)$.
$\ell_\tau(g)$	Training loss for prediction function g ; that is, the sample average estimate of $\ell(g)$ based on a fixed training sample τ .
$\ell_{\mathcal{T}}(g)$	The same as $\ell_\tau(g)$, but now for a random training sample \mathcal{T} .
$g_\tau^{\mathcal{G}}$ or g_τ	The <i>learner</i> : $\text{argmin}_{g \in \mathcal{G}} \ell_\tau(g)$. That is, the optimal prediction function based on a fixed training set τ and function class \mathcal{G} .
	We suppress the superscript \mathcal{G} if the function class is implicit.
$g_{\mathcal{T}}^{\mathcal{G}}$ or $g_{\mathcal{T}}$	The learner, where we have replaced τ with a random training set \mathcal{T} .

To compare the predictive performance of various learners in the function class \mathcal{G} , as measured by the test loss, we can use the *same* fixed training set τ and test set τ' for all learners. When there is an abundance of data, the “overall” data set is usually (randomly) divided into a training and test set, as depicted in Figure 2.3. We then use the training data to construct various learners $g_\tau^{\mathcal{G}_1}, g_\tau^{\mathcal{G}_2}, \dots$, and use the test data to select the best (with the smallest test loss) among these learners. In this context the test set is called the *validation set*. Once the best learner has been chosen, a third “test” set can be used to assess the predictive performance of the best learner. The training, validation, and test sets can again be obtained from the overall data set via a random allocation. When the overall data set is of modest size, it is customary to perform the validation phase (model selection) on the training set only, using cross-validation. This is the topic of Section 2.5.2.

VALIDATION SET

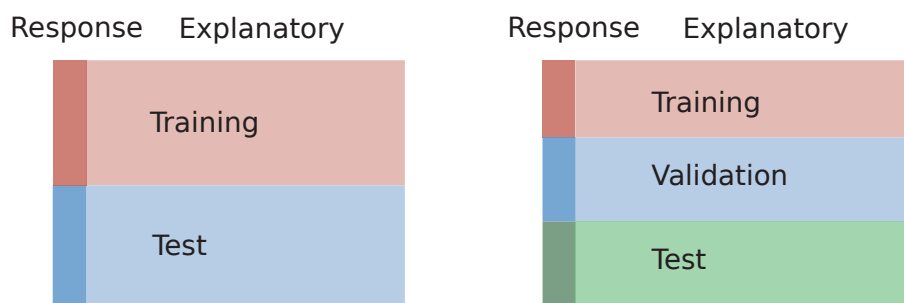


Figure 2.3: Statistical learning algorithms often require the data to be divided into training and test data. If the latter is used for model selection, a third set is needed for testing the performance of the selected model.

We next consider a concrete example that illustrates the concepts introduced so far.

■ **Example 2.1 (Polynomial Regression)** In what follows, it will appear that we have arbitrarily replaced the symbols $\mathbf{x}, g, \mathcal{G}$ with u, h, \mathcal{H} , respectively. The reason for this switch of notation will become clear at the end of the example.

The data (depicted as dots) in Figure 2.4 are $n = 100$ points $(u_i, y_i), i = 1, \dots, n$ drawn from iid random points $(U_i, Y_i), i = 1, \dots, n$, where the $\{U_i\}$ are uniformly distributed on the interval $(0, 1)$ and, given $U_i = u_i$, the random variable Y_i has a normal distribution with expectation $10 - 140u_i + 400u_i^2 - 250u_i^3$ and variance $\ell^* = 25$. This is an example of a *polynomial regression model*. Using a squared-error loss, the optimal prediction function $h^*(u) = \mathbb{E}[Y | U = u]$ is thus

$$h^*(u) = 10 - 140u + 400u^2 - 250u^3,$$

which is depicted by the dashed curve in Figure 2.4.

POLYNOMIAL
REGRESSION
MODEL

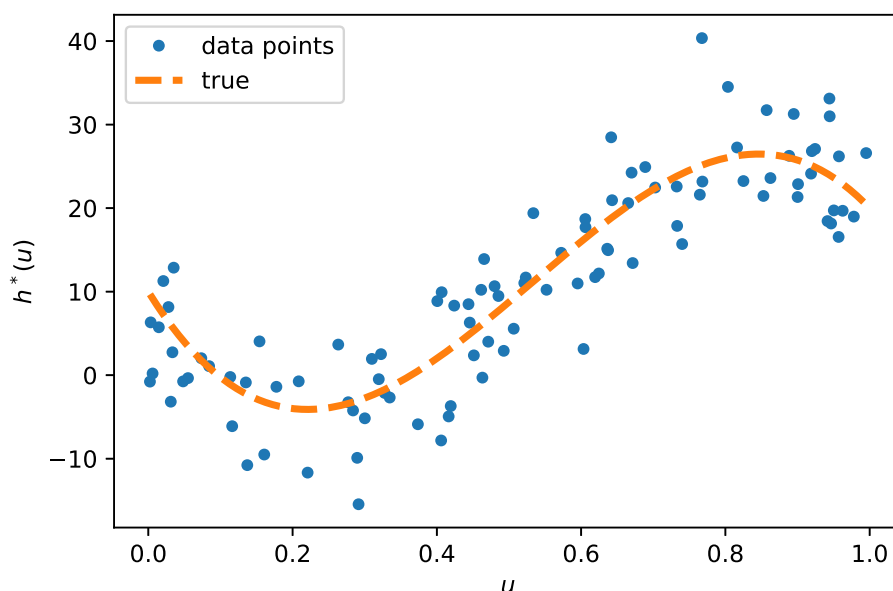


Figure 2.4: Training data and the optimal polynomial prediction function h^* .

To obtain a good estimate of $h^*(u)$ based on the training set $\tau = \{(u_i, y_i), i = 1, \dots, n\}$, we minimize the outcome of the training loss (2.3):

$$\ell_\tau(h) = \frac{1}{n} \sum_{i=1}^n (y_i - h(u_i))^2, \quad (2.8)$$

over a suitable set \mathcal{H} of candidate functions. Let us take the set \mathcal{H}_p of polynomial functions in u of order $p - 1$:

$$h(u) := \beta_1 + \beta_2 u + \beta_3 u^2 + \dots + \beta_p u^{p-1} \quad (2.9)$$

for $p = 1, 2, \dots$ and parameter vector $\beta = [\beta_1, \beta_2, \dots, \beta_p]^\top$. This function class contains the best possible $h^*(u) = \mathbb{E}[Y | U = u]$ for $p \geq 4$. Note that optimization over \mathcal{H}_p is a parametric optimization problem, in that we need to find the best β . Optimization of (2.8) over \mathcal{H}_p is not straightforward, unless we notice that (2.9) is a *linear* function in β . In particular, if we map each feature u to a feature vector $\mathbf{x} = [1, u, u^2, \dots, u^{p-1}]^\top$, then the right-hand side of (2.9) can be written as the function

$$g(\mathbf{x}) = \mathbf{x}^\top \beta,$$

which is linear in \mathbf{x} (as well as β). The optimal $h^*(u)$ in \mathcal{H}_p for $p \geq 4$ then corresponds to the function $g^*(\mathbf{x}) = \mathbf{x}^\top \beta^*$ in the set \mathcal{G}_p of linear functions from \mathbb{R}^p to \mathbb{R} , where $\beta^* = [10, -140, 400, -250, 0, \dots, 0]^\top$. Thus, instead of working with the set \mathcal{H}_p of polynomial functions we may prefer to work with the set \mathcal{G}_p of linear functions. This brings us to a very important idea in statistical learning:



Expand the feature space to obtain a *linear* prediction function.

Let us now reformulate the learning problem in terms of the new explanatory (feature) variables $\mathbf{x}_i = [1, u_i, u_i^2, \dots, u_i^{p-1}]^\top$, $i = 1, \dots, n$. It will be convenient to arrange these feature vectors into a matrix \mathbf{X} with rows $\mathbf{x}_1^\top, \dots, \mathbf{x}_n^\top$:

$$\mathbf{X} = \begin{bmatrix} 1 & u_1 & u_1^2 & \dots & u_1^{p-1} \\ 1 & u_2 & u_2^2 & \dots & u_2^{p-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & u_n & u_n^2 & \dots & u_n^{p-1} \end{bmatrix}. \quad (2.10)$$

Collecting the responses $\{y_i\}$ into a column vector \mathbf{y} , the training loss (2.3) can now be written compactly as

$$\frac{1}{n} \|\mathbf{y} - \mathbf{X}\beta\|^2. \quad (2.11)$$

To find the optimal learner (2.4) in the class \mathcal{G}_p we need to find the minimizer of (2.11):

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \|\mathbf{y} - \mathbf{X}\beta\|^2, \quad (2.12)$$

which is called the *ordinary least-squares* solution. As is illustrated in Figure 2.5, to find $\hat{\beta}$, we choose $\mathbf{X}\hat{\beta}$ to be equal to the orthogonal projection of \mathbf{y} onto the linear space spanned by the columns of the matrix \mathbf{X} ; that is, $\mathbf{X}\hat{\beta} = \mathbf{P}\mathbf{y}$, where \mathbf{P} is the *projection matrix*.

ORDINARY
LEAST-SQUARES

PROJECTION
MATRIX

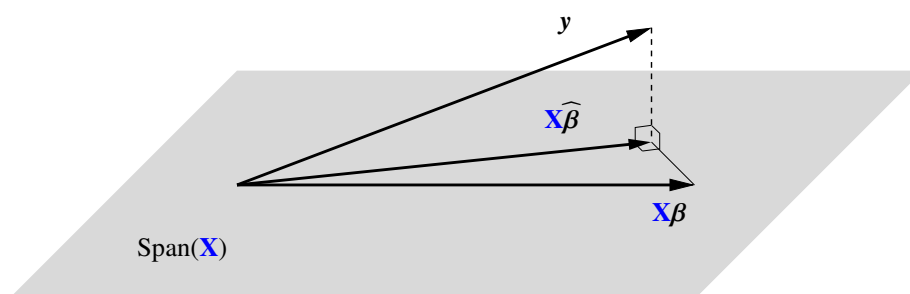


Figure 2.5: $\mathbf{X}\hat{\boldsymbol{\beta}}$ is the orthogonal projection of \mathbf{y} onto the linear space spanned by the columns of the matrix \mathbf{X} .

364

According to Theorem A.4, the projection matrix is given by

$$\mathbf{P} = \mathbf{X}\mathbf{X}^+, \quad (2.13)$$

362

PSEUDO-INVERSE

358

where the $p \times n$ matrix \mathbf{X}^+ in (2.13) is the *pseudo-inverse* of \mathbf{X} . If \mathbf{X} happens to be of *full column rank* (so that none of the columns can be expressed as a linear combination of the other columns), then $\mathbf{X}^+ = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$.

In any case, from $\mathbf{X}\hat{\boldsymbol{\beta}} = \mathbf{P}\mathbf{y}$ and $\mathbf{P}\mathbf{X} = \mathbf{X}$, we can see that $\hat{\boldsymbol{\beta}}$ satisfies the *normal equations*:

NORMAL
EQUATIONS

$$\mathbf{X}^\top \mathbf{X} \hat{\boldsymbol{\beta}} = \mathbf{X}^\top \mathbf{P} \mathbf{y} = (\mathbf{P} \mathbf{X})^\top \mathbf{y} = \mathbf{X}^\top \mathbf{y}. \quad (2.14)$$

This is a set of linear equations, which can be solved very fast and whose solution can be written explicitly as:

$$\hat{\boldsymbol{\beta}} = \mathbf{X}^+ \mathbf{y}. \quad (2.15)$$

Figure 2.6 shows the trained learners for various values of p :

$$h_{\tau}^{\mathcal{H}_p}(u) = g_{\tau}^{\mathcal{G}_p}(x) = \mathbf{x}^\top \hat{\boldsymbol{\beta}}$$

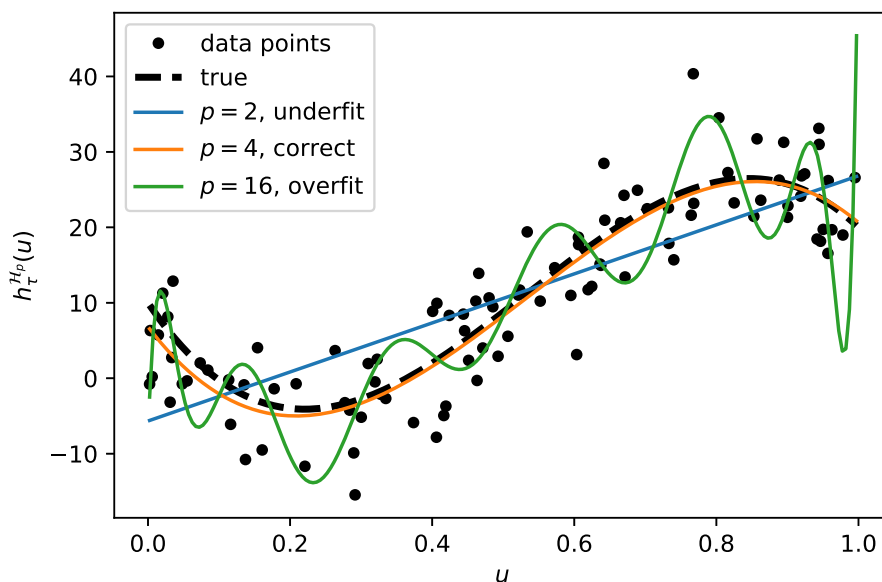


Figure 2.6: Training data with fitted curves for $p = 2, 4$, and 16 . The true cubic polynomial curve for $p = 4$ is also plotted (dashed line).

We see that for $p = 16$ the fitted curve lies closer to the data points, but is further away from the dashed true polynomial curve, indicating that we overfit. The choice $p = 4$ (the true cubic polynomial) is much better than $p = 16$, or indeed $p = 2$ (straight line).

Each function class \mathcal{G}_p gives a different learner $g_{\tau}^{\mathcal{G}_p}$, $p = 1, 2, \dots$. To assess which is better, we should not simply take the one that gives the smallest training loss. We can always get a *zero* training loss by taking $p = n$, because for any set of n points there exists a polynomial of degree $n - 1$ that interpolates all points!

Instead, we assess the predictive performance of the learners using the test loss (2.7), computed from a test data set. If we collect all n' test feature vectors in a matrix \mathbf{X}' and the corresponding test responses in a vector \mathbf{y}' , then, similar to (2.11), the test loss can be written compactly as

$$\ell_{\tau'}(g_{\tau}^{\mathcal{G}_p}) = \frac{1}{n'} \|\mathbf{y}' - \mathbf{X}'\widehat{\boldsymbol{\beta}}\|^2,$$

where $\widehat{\boldsymbol{\beta}}$ is given by (2.15), using the training data.

Figure 2.7 shows a plot of the test loss against the number of parameters in the vector $\boldsymbol{\beta}$; that is, p . The graph has a characteristic “bath-tub” shape and is at its lowest for $p = 4$, correctly identifying the polynomial order 3 for the true model. Note that the test loss, as an estimate for the generalization risk (2.7), becomes numerically unreliable after $p = 16$ (the graph goes down, where it should go up). The reader may check that the graph for the training loss exhibits a similar numerical instability for large p , and in fact fails to numerically decrease to 0 for large p , contrary to what it should do in theory. The numerical problems arise from the fact that for large p the columns of the (Vandermonde) matrix \mathbf{X} are of vastly different magnitudes and so floating point errors quickly become very large.

Finally, observe that the lower bound for the test loss is here around 21, which corresponds to an estimate of the minimal (squared-error) risk $\ell^* = 25$.

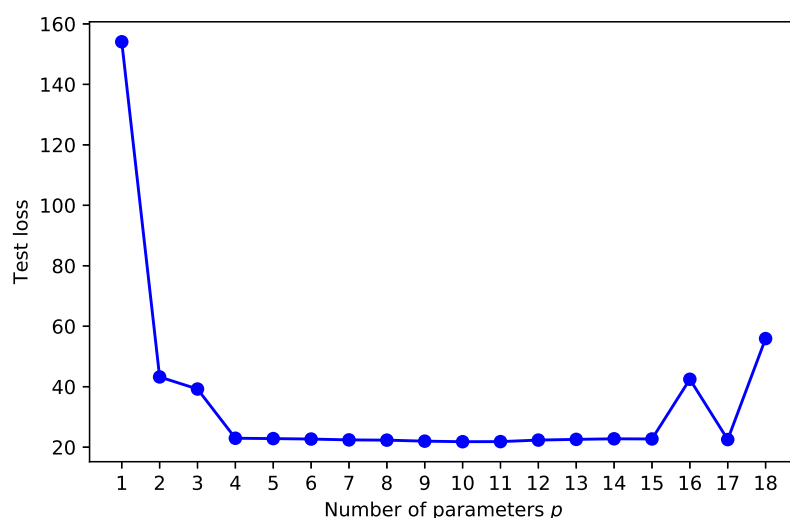


Figure 2.7: Test loss as function of the number of parameters p of the model.

This script shows how the training data were generated and plotted in Python:

polyreg1.py

```

import numpy as np
from numpy.random import rand , randn
from numpy.linalg import norm , solve
import matplotlib.pyplot as plt
def generate_data(beta , sig, n):
    u = np.random.rand(n, 1)
    y = (u ** np.arange(0, 4)) @ beta + sig * np.random.randn(n, 1)
    return u, y

np.random.seed(12)
beta = np.array([[10, -140, 400, -250]]).T
n = 100
sig = 5
u, y = generate_data(beta , sig, n)
xx = np.arange(np.min(u), np.max(u)+5e-3, 5e-3)
yy = np.polyval(np.flip(beta), xx)
plt.plot(u, y, '.', markersize=8)
plt.plot(xx, yy, '--', linewidth=3)
plt.xlabel(r'$u$')
plt.ylabel(r'$h^{(u)}$')
plt.legend(['data points', 'true'])
plt.show()

```

The following code, which imports the code above, fits polynomial models with $p = 1, \dots, K = 18$ parameters to the training data and plots a selection of fitted curves, as shown in Figure 2.6.

polyreg2.py

```

from polyreg1 import *

max_p = 18
p_range = np.arange(1, max_p + 1, 1)
X = np.ones((n, 1))
betahat, trainloss = {}, {}

for p in p_range: # p is the number of parameters
    if p > 1:
        X = np.hstack((X, u**(p-1))) # add column to matrix

        betahat[p] = solve(X.T @ X, X.T @ y)
        trainloss[p] = (norm(y - X @ betahat[p])**2/n)

p = [2, 4, 16] # select three curves

#replot the points and true line and store in the list "plots"
plots = [plt.plot(u, y, 'k.', markersize=8)[0],
         plt.plot(xx, yy, 'k--', linewidth=3)[0]]
# add the three curves
for i in p:
    yy = np.polyval(np.flip(betahat[i]), xx)
    plots.append(plt.plot(xx, yy)[0])

```



```
plt.xlabel(r'$u$')
plt.ylabel(r'$h^{\mathcal{H}_p}_{\tau}(u)$')
plt.legend(plots, ('data points', 'true', '$p=2$, underfit',
                  '$p=4$, correct', '$p=16$, overfit'))
plt.savefig('polyfitpy.pdf', format='pdf')
plt.show()
```

The last code snippet which imports the previous code, generates the test data and plots the graph of the test loss, as shown in Figure 2.7.

polyreg3.py

```
from polyreg2 import *

# generate test data
u_test, y_test = generate_data(beta, sig, n)

MSE = []
X_test = np.ones((n, 1))

for p in p_range:
    if p > 1:
        X_test = np.hstack((X_test, u_test**(p-1)))

    y_hat = X_test @ betahat[p] # predictions
    MSE.append(np.sum((y_test - y_hat)**2/n))

plt.plot(p_range, MSE, 'b', p_range, MSE, 'bo')
plt.xticks(ticks=p_range)
plt.xlabel('Number of parameters $p$')
plt.ylabel('Test loss')
```

2.4 Tradeoffs in Statistical Learning

The art of machine learning in the supervised case is to make the generalization risk (2.5) or expected generalization risk (2.6) as small as possible, while using as few computational resources as possible. In pursuing this goal, a suitable class \mathcal{G} of prediction functions has to be chosen. This choice is driven by various factors, such as

- the complexity of the class (e.g., is it rich enough to adequately approximate, or even contain, the optimal prediction function g^*),
- the ease of training the learner via the optimization program (2.4),
- how accurately the training loss (2.3) estimates the risk (2.1) within class \mathcal{G} ,
- the feature types (categorical, continuous, etc.).

As a result, the choice of a suitable function class \mathcal{G} usually involves a tradeoff between conflicting factors. For example, a learner from a simple class \mathcal{G} can be trained very

quickly, but may not approximate g^* very well, whereas a learner from a rich class \mathcal{G} that contains g^* may require a lot of computing resources to train.

To better understand the relation between model complexity, computational simplicity, and estimation accuracy, it is useful to decompose the generalization risk into several parts, so that the tradeoffs between these parts can be studied. We will consider two such decompositions: the approximation–estimation tradeoff and the bias–variance tradeoff.

We can decompose the generalization risk (2.5) into the following three components:

$$\ell(g_\tau^\mathcal{G}) = \underbrace{\ell^*}_{\text{irreducible risk}} + \underbrace{\ell(g^\mathcal{G}) - \ell^*}_{\text{approximation error}} + \underbrace{\ell(g_\tau^\mathcal{G}) - \ell(g^\mathcal{G})}_{\text{statistical error}}, \quad (2.16)$$

IRREDUCIBLE RISK

where $\ell^* := \ell(g^*)$ is the *irreducible risk* and $g^\mathcal{G} := \operatorname{argmin}_{g \in \mathcal{G}} \ell(g)$ is the best learner within class \mathcal{G} . No learner can predict a new response with a smaller risk than ℓ^* .

APPROXIMATION ERROR

The second component is the *approximation error*; it measures the difference between the irreducible risk and the best possible risk that can be obtained by selecting the best prediction function in the selected class of functions \mathcal{G} . Determining a suitable class \mathcal{G} and minimizing $\ell(g)$ over this class is purely a problem of numerical and functional analysis, as the training data τ are not present. For a fixed \mathcal{G} that does not contain the optimal g^* , the approximation error cannot be made arbitrarily small and may be the dominant component in the generalization risk. The only way to reduce the approximation error is by expanding the class \mathcal{G} to include a larger set of possible functions.

STATISTICAL (ESTIMATION) ERROR

The third component is the *statistical (estimation) error*. It depends on the training set τ and, in particular, on how well the learner $g_\tau^\mathcal{G}$ estimates the best possible prediction function, $g^\mathcal{G}$, within class \mathcal{G} . For any sensible estimator this error should decay to zero (in probability or expectation) as the training size tends to infinity.

☞ 441

APPROXIMATION– ESTIMATION TRADEOFF

The *approximation–estimation tradeoff* pits two competing demands against each other. The first is that the class \mathcal{G} has to be simple enough so that the statistical error is not too large. The second is that the class \mathcal{G} has to be rich enough to ensure a small approximation error. Thus, there is a tradeoff between the approximation and estimation errors.

For the special case of the squared-error loss, the generalization risk is equal to $\ell(g_\tau^\mathcal{G}) = \mathbb{E}(Y - g_\tau^\mathcal{G}(X))^2$; that is, the expected squared error¹ between the predicted value $g_\tau^\mathcal{G}(X)$ and the response Y . Recall that in this case the optimal prediction function is given by $g^*(\mathbf{x}) = \mathbb{E}[Y | X = \mathbf{x}]$. The decomposition (2.16) can now be interpreted as follows.

1. The first component, $\ell^* = \mathbb{E}(Y - g^*(X))^2$, is the *irreducible error*, as no prediction function will yield a smaller expected squared error.
2. The second component, the approximation error $\ell(g^\mathcal{G}) - \ell(g^*)$, is equal to $\mathbb{E}(g^\mathcal{G}(X) - g^*(X))^2$. We leave the proof (which is similar to that of Theorem 2.1) as an exercise; see Exercise 2. Thus, the approximation error (defined as a risk difference) can here be interpreted as the expected squared error between the optimal predicted value and the optimal predicted value within the class \mathcal{G} .
3. For the third component, the statistical error, $\ell(g_\tau^\mathcal{G}) - \ell(g^\mathcal{G})$ there is no direct interpretation as an expected squared error *unless* \mathcal{G} is the class of *linear* functions; that is, $g(\mathbf{x}) = \mathbf{x}^\top \boldsymbol{\beta}$ for some vector $\boldsymbol{\beta}$. In this case we can write (see Exercise 3) the statistical error as $\ell(g_\tau^\mathcal{G}) - \ell(g^\mathcal{G}) = \mathbb{E}(g_\tau^\mathcal{G}(X) - g^\mathcal{G}(X))^2$.

¹Colloquially called *mean squared error*.

Thus, when using a squared-error loss, the generalization risk for a linear class \mathcal{G} can be decomposed as:

$$\ell(g_{\tau}^{\mathcal{G}}) = \mathbb{E}(g_{\tau}^{\mathcal{G}}(X) - Y)^2 = \ell^* + \underbrace{\mathbb{E}(g^{\mathcal{G}}(X) - g^*(X))^2}_{\text{approximation error}} + \underbrace{\mathbb{E}(g_{\tau}^{\mathcal{G}}(X) - g^{\mathcal{G}}(X))^2}_{\text{statistical error}}. \quad (2.17)$$

Note that in this decomposition the statistical error is the only term that depends on the training set.

■ **Example 2.2 (Polynomial Regression (cont.))** We continue Example 2.1. Here $\mathcal{G} = \mathcal{G}_p$ is the class of linear functions of $\mathbf{x} = [1, u, u^2, \dots, u^{p-1}]^{\top}$, and $g^*(\mathbf{x}) = \mathbf{x}^{\top} \boldsymbol{\beta}^*$. Conditional on $X = \mathbf{x}$ we have that $Y = g^*(\mathbf{x}) + \varepsilon(\mathbf{x})$, with $\varepsilon(\mathbf{x}) \sim \mathcal{N}(0, \ell^*)$, where $\ell^* = \mathbb{E}(Y - g^*(X))^2 = 25$ is the irreducible error. We wish to understand how the approximation and statistical errors behave as we change the complexity parameter p .

First, we consider the approximation error. Any function $g \in \mathcal{G}_p$ can be written as

$$g(\mathbf{x}) = h(u) = \beta_1 + \beta_2 u + \dots + \beta_p u^{p-1} = [1, u, \dots, u^{p-1}] \boldsymbol{\beta},$$

and so $g(X)$ is distributed as $[1, U, \dots, U^{p-1}] \boldsymbol{\beta}$, where $U \sim \mathcal{U}(0, 1)$. Similarly, $g^*(X)$ is distributed as $[1, U, U^2, U^3] \boldsymbol{\beta}^*$. It follows that an expression for the approximation error is: $\int_0^1 ([1, u, \dots, u^{p-1}] \boldsymbol{\beta} - [1, u, u^2, u^3] \boldsymbol{\beta}^*)^2 du$. To minimize this error, we set the gradient with respect to $\boldsymbol{\beta}$ to zero and obtain the p linear equations

$$\begin{aligned} \int_0^1 ([1, u, \dots, u^{p-1}] \boldsymbol{\beta} - [1, u, u^2, u^3] \boldsymbol{\beta}^*) du &= 0, \\ \int_0^1 ([1, u, \dots, u^{p-1}] \boldsymbol{\beta} - [1, u, u^2, u^3] \boldsymbol{\beta}^*) u du &= 0, \\ &\vdots \\ \int_0^1 ([1, u, \dots, u^{p-1}] \boldsymbol{\beta} - [1, u, u^2, u^3] \boldsymbol{\beta}^*) u^{p-1} du &= 0. \end{aligned}$$

Let

$$\mathbf{H}_p = \int_0^1 [1, u, \dots, u^{p-1}]^{\top} [1, u, \dots, u^{p-1}] du$$

be the $p \times p$ *Hilbert matrix*, which has (i, j) -th entry given by $\int_0^1 u^{i+j-2} du = 1/(i+j-1)$.

HILBERT MATRIX

Then, the above system of linear equations can be written as $\mathbf{H}_p \boldsymbol{\beta} = \tilde{\mathbf{H}} \boldsymbol{\beta}^*$, where $\tilde{\mathbf{H}}$ is the $p \times 4$ upper left sub-block of $\mathbf{H}_{\tilde{p}}$ and $\tilde{p} = \max\{p, 4\}$. The solution, which we denote by $\boldsymbol{\beta}_p$, is:

$$\boldsymbol{\beta}_p = \begin{cases} \begin{bmatrix} \frac{65}{6}, \\ \end{bmatrix} & p = 1, \\ \begin{bmatrix} -\frac{20}{3}, 35 \end{bmatrix}^{\top}, & p = 2, \\ \begin{bmatrix} -\frac{5}{2}, 10, 25 \end{bmatrix}^{\top}, & p = 3, \\ \begin{bmatrix} 10, -140, 400, -250, 0, \dots, 0 \end{bmatrix}^{\top}, & p \geq 4. \end{cases} \quad (2.18)$$

Hence, the approximation error $\mathbb{E}(g^{\mathcal{G}_p}(X) - g^*(X))^2$ is given by

$$\int_0^1 ([1, u, \dots, u^{p-1}] \boldsymbol{\beta}_p - [1, u, u^2, u^3] \boldsymbol{\beta}^*)^2 du = \begin{cases} \frac{32225}{252} \approx 127.9, & p = 1, \\ \frac{1625}{63} \approx 25.8, & p = 2, \\ \frac{625}{28} \approx 22.3, & p = 3, \\ 0, & p \geq 4. \end{cases} \quad (2.19)$$

Notice how the approximation error becomes smaller as p increases. In this particular example the approximation error is in fact zero for $p \geq 4$. In general, as the class of approximating functions \mathcal{G} becomes more complex, the approximation error goes down.

Next, we illustrate the typical behavior of the statistical error. Since $g_\tau(\mathbf{x}) = \mathbf{x}^\top \widehat{\boldsymbol{\beta}}$, the statistical error can be written as

$$\int_0^1 ([1, \dots, u^{p-1}] (\widehat{\boldsymbol{\beta}} - \boldsymbol{\beta}_p))^2 du = (\widehat{\boldsymbol{\beta}} - \boldsymbol{\beta}_p)^\top \mathbf{H}_p (\widehat{\boldsymbol{\beta}} - \boldsymbol{\beta}_p). \quad (2.20)$$

Figure 2.8 illustrates the decomposition (2.17) of the generalization risk for the *same* training set that was used to compute the test loss in Figure 2.7. Recall that test loss gives an estimate of the generalization risk, using independent test data. Comparing the two figures, we see that in this case the two match closely. The global minimum of the statistical error is approximately 0.28, with minimizer $p = 4$. Since the approximation error is monotonically decreasing to zero, $p = 4$ is also the global minimizer of the generalization risk.

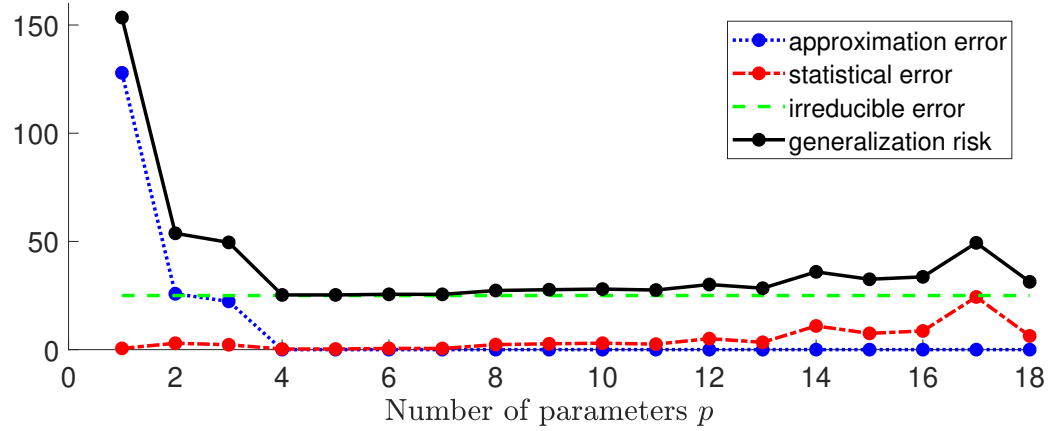


Figure 2.8: The generalization risk for a particular training set is the sum of the irreducible error, the approximation error, and the statistical error. The approximation error decreases to zero as p increases, whereas the statistical error has a tendency to increase after $p = 4$.

Note that the statistical error depends on the estimate $\widehat{\boldsymbol{\beta}}$, which in its turn depends on the training set τ . We can obtain a better understanding of the statistical error by considering its *expected* behavior; that is, averaged over many training sets. This is explored in Exercise 11. ■

Using again a squared-error loss, a second decomposition (for general \mathcal{G}) starts from

$$\ell(g_\tau^\mathcal{G}) = \ell^* + \ell(g_\tau^\mathcal{G}) - \ell(g^*),$$

where the statistical error and approximation error are combined. Using similar reasoning as in the proof of Theorem 2.1, we have

$$\ell(g_\tau^\mathcal{G}) = \mathbb{E}(g_\tau^\mathcal{G}(\mathbf{X}) - Y)^2 = \ell^* + \mathbb{E}(g_\tau^\mathcal{G}(\mathbf{X}) - g^*(\mathbf{X}))^2 = \ell^* + \mathbb{E}D^2(\mathbf{X}, \tau),$$

where $D(\mathbf{x}, \tau) := g_{\tau}^{\mathcal{G}}(\mathbf{x}) - g^*(\mathbf{x})$. Now consider the random variable $D(\mathbf{x}, \mathcal{T})$ for a random training set \mathcal{T} . The expectation of its square is:

$$\begin{aligned} \mathbb{E} \left(g_{\mathcal{T}}^{\mathcal{G}}(\mathbf{x}) - g^*(\mathbf{x}) \right)^2 &= \mathbb{E} D^2(\mathbf{x}, \mathcal{T}) = (\mathbb{E} D(\mathbf{x}, \mathcal{T}))^2 + \text{Var } D(\mathbf{x}, \mathcal{T}) \\ &= \underbrace{(\mathbb{E} g_{\mathcal{T}}^{\mathcal{G}}(\mathbf{x}) - g^*(\mathbf{x}))^2}_{\text{pointwise squared bias}} + \underbrace{\text{Var } g_{\mathcal{T}}^{\mathcal{G}}(\mathbf{x})}_{\text{pointwise variance}}. \end{aligned} \quad (2.21)$$

If we view the learner $g_{\mathcal{T}}^{\mathcal{G}}(\mathbf{x})$ as a function of a random training set, then the *pointwise squared bias* term is a measure for how close $g_{\mathcal{T}}^{\mathcal{G}}(\mathbf{x})$ is on average to the true $g^*(\mathbf{x})$, whereas the *pointwise variance* term measures the deviation of $g_{\mathcal{T}}^{\mathcal{G}}(\mathbf{x})$ from its expected value $\mathbb{E} g_{\mathcal{T}}^{\mathcal{G}}(\mathbf{x})$. The squared bias can be reduced by making the class of functions \mathcal{G} more complex. However, decreasing the bias by increasing the complexity often leads to an increase in the variance term. We are thus seeking learners that provide an optimal balance between the bias and variance, as expressed via a minimal generalization risk. This is called the *bias–variance tradeoff*.

POINTWISE
SQUARED BIAS
POINTWISE
VARIANCE

Note that the *expected* generalization risk (2.6) can be written as $\ell^* + \mathbb{E} D^2(\mathbf{X}, \mathcal{T})$, where \mathbf{X} and \mathcal{T} are independent. It therefore decomposes as

$$\mathbb{E} \ell(g_{\mathcal{T}}^{\mathcal{G}}) = \ell^* + \underbrace{\mathbb{E} (\mathbb{E}[g_{\mathcal{T}}^{\mathcal{G}}(\mathbf{X}) | \mathbf{X}] - g^*(\mathbf{X}))^2}_{\text{expected squared bias}} + \underbrace{\mathbb{E} [\text{Var}[g_{\mathcal{T}}^{\mathcal{G}}(\mathbf{X}) | \mathbf{X}]]}_{\text{expected variance}}. \quad (2.22)$$

BIAS–VARIANCE
TRADEOFF

2.5 Estimating Risk

The most straightforward way to quantify the generalization risk (2.5) is to estimate it via the test loss (2.7). However, the generalization risk depends inherently on the training set, and so different training sets may yield significantly different estimates. Moreover, when there is a limited amount of data available, reserving a substantial proportion of the data for testing rather than training may be uneconomical. In this section we consider different methods for estimating risk measures which aim to circumvent these difficulties.

2.5.1 In-Sample Risk

We mentioned that, due to the phenomenon of overfitting, the training loss of the learner, $\ell_{\tau}(g_{\tau})$ (for simplicity, here we omit \mathcal{G} from $g_{\tau}^{\mathcal{G}}$), is not a good estimate of the generalization risk $\ell(g_{\tau})$ of the learner. One reason for this is that we use the same data for both training the model and assessing its risk. How should we then estimate the generalization risk or expected generalization risk?

To simplify the analysis, suppose that we wish to estimate the average accuracy of the predictions of the learner g_{τ} at the n feature vectors $\mathbf{x}_1, \dots, \mathbf{x}_n$ (these are part of the training set τ). In other words, we wish to estimate the *in-sample risk* of the learner g_{τ} :

IN-SAMPLE RISK

$$\ell_{\text{in}}(g_{\tau}) = \frac{1}{n} \sum_{i=1}^n \mathbb{E} \text{Loss}(Y'_i, g_{\tau}(\mathbf{x}_i)), \quad (2.23)$$

where each response Y'_i is drawn from $f(y | \mathbf{x}_i)$, independently. Even in this simplified setting, the training loss of the learner will be a poor estimate of the in-sample risk. Instead, the

proper way to assess the prediction accuracy of the learner at the feature vectors $\mathbf{x}_1, \dots, \mathbf{x}_n$, is to draw new response values $Y'_i \sim f(y|\mathbf{x}_i)$, $i = 1, \dots, n$, that are independent from the responses y_1, \dots, y_n in the training data, and then estimate the in-sample risk of g_τ via

$$\frac{1}{n} \sum_{i=1}^n \text{Loss}(Y'_i, g_\tau(\mathbf{x}_i)).$$

For a fixed training set τ , we can compare the training loss of the learner with the in-sample risk. Their difference,

$$\text{op}_\tau = \ell_{\text{in}}(g_\tau) - \ell_\tau(g_\tau),$$

is called the *optimism* (of the training loss), because it measures how much the training loss underestimates (is optimistic about) the unknown in-sample risk. Mathematically, it is simpler to work with the *expected optimism*:

EXPECTED
OPTIMISM

$$\mathbb{E}[\text{op}_\tau | \mathbf{X}_1 = \mathbf{x}_1, \dots, \mathbf{X}_n = \mathbf{x}_n] =: \mathbb{E}_{\mathbf{X}} \text{op}_\tau,$$

where the expectation is taken over a random training set \mathcal{T} , conditional on $\mathbf{X}_i = \mathbf{x}_i$, $i = 1, \dots, n$. For ease of notation, we have abbreviated the expected optimism to $\mathbb{E}_{\mathbf{X}} \text{op}_\tau$, where $\mathbb{E}_{\mathbf{X}}$ denotes the expectation operator conditional on $\mathbf{X}_i = \mathbf{x}_i$, $i = 1, \dots, n$. As in Example 2.1, the feature vectors are stored as the rows of an $n \times p$ matrix \mathbf{X} . It turns out that the expected optimism for various loss functions can be expressed in terms of the (conditional) covariance between the observed and predicted response.

Theorem 2.2: Expected Optimism

For the squared-error loss and 0–1 loss with 0–1 response, the expected optimism is

$$\mathbb{E}_{\mathbf{X}} \text{op}_\tau = \frac{2}{n} \sum_{i=1}^n \text{Cov}_{\mathbf{X}}(g_\tau(\mathbf{x}_i), Y_i). \quad (2.24)$$

Proof: In what follows, all expectations are taken conditional on $\mathbf{X}_1 = \mathbf{x}_1, \dots, \mathbf{X}_n = \mathbf{x}_n$. Let Y_i be the response for \mathbf{x}_i and let $\widehat{Y}_i = g_\tau(\mathbf{x}_i)$ be the predicted value. Note that the latter depends on Y_1, \dots, Y_n . Also, let Y'_i be an independent copy of Y_i for the *same* \mathbf{x}_i , as in (2.23). In particular, Y'_i has the same distribution as Y_i and is statistically independent of all $\{Y_j\}$, including Y_i , and therefore is also independent of \widehat{Y}_i . We have

$$\begin{aligned} \mathbb{E}_{\mathbf{X}} \text{op}_\tau &= \frac{1}{n} \sum_{i=1}^n \mathbb{E}_{\mathbf{X}} [(Y'_i - \widehat{Y}_i)^2 - (Y_i - \widehat{Y}_i)^2] = \frac{2}{n} \sum_{i=1}^n \mathbb{E}_{\mathbf{X}} [(Y_i - Y'_i) \widehat{Y}_i] \\ &= \frac{2}{n} \sum_{i=1}^n (\mathbb{E}_{\mathbf{X}} [Y_i \widehat{Y}_i] - \mathbb{E}_{\mathbf{X}} Y_i \mathbb{E}_{\mathbf{X}} \widehat{Y}_i) = \frac{2}{n} \sum_{i=1}^n \text{Cov}_{\mathbf{X}}(\widehat{Y}_i, Y_i). \end{aligned}$$

The proof for the 0–1 loss with 0–1 response is left as Exercise 4. \square

In summary, the expected optimism indicates how much, on average, the training loss deviates from the expected in-sample risk. Since the covariance of independent random variables is zero, the expected optimism is zero if the learner g_τ is statistically independent from the responses Y_1, \dots, Y_n .

■ **Example 2.3 (Polynomial Regression (cont.))** We continue Example 2.2, where the components of the response vector $\mathbf{Y} = [Y_1, \dots, Y_n]^\top$ are independent and normally distributed with variance $\ell^* = 25$ (the irreducible error) and expectations $\mathbb{E}_{\mathbf{X}} Y_i = g^*(\mathbf{x}_i) = \mathbf{x}_i^\top \boldsymbol{\beta}^*$, $i = 1, \dots, n$. Using the formula (2.15) for the least-squares estimator $\widehat{\boldsymbol{\beta}}$, the expected optimism (2.24) is

$$\begin{aligned} \frac{2}{n} \sum_{i=1}^n \text{Cov}_{\mathbf{X}}(\mathbf{x}_i^\top \widehat{\boldsymbol{\beta}}, Y_i) &= \frac{2}{n} \text{tr}(\text{Cov}_{\mathbf{X}}(\mathbf{X} \widehat{\boldsymbol{\beta}}, \mathbf{Y})) = \frac{2}{n} \text{tr}(\text{Cov}_{\mathbf{X}}(\mathbf{X} \mathbf{X}^+ \mathbf{Y}, \mathbf{Y})) \\ &= \frac{2 \text{tr}(\mathbf{X} \mathbf{X}^+ \text{Cov}_{\mathbf{X}}(\mathbf{Y}, \mathbf{Y}))}{n} = \frac{2 \ell^* \text{tr}(\mathbf{X} \mathbf{X}^+)}{n} = \frac{2 \ell^* p}{n}. \end{aligned}$$

In the last equation we used the cyclic property of the trace (Theorem A.1): $\text{tr}(\mathbf{X} \mathbf{X}^+) = \text{tr}(\mathbf{X}^+ \mathbf{X}) = \text{tr}(\mathbf{I}_p)$, assuming that $\text{rank}(\mathbf{X}) = p$. Therefore, an estimate for the in-sample risk (2.23) is:

$$\widehat{\ell}_{\text{in}}(g_\tau) = \ell_\tau(g_\tau) + 2 \ell^* p/n, \quad (2.25)$$

where we have assumed that the irreducible risk ℓ^* is known. Figure 2.9 shows that this estimate is very close to the test loss from Figure 2.7. Hence, instead of computing the test loss to assess the best model complexity p , we could simply have minimized the training loss plus the correction term $2 \ell^* p/n$. In practice, ℓ^* also has to be estimated somehow.

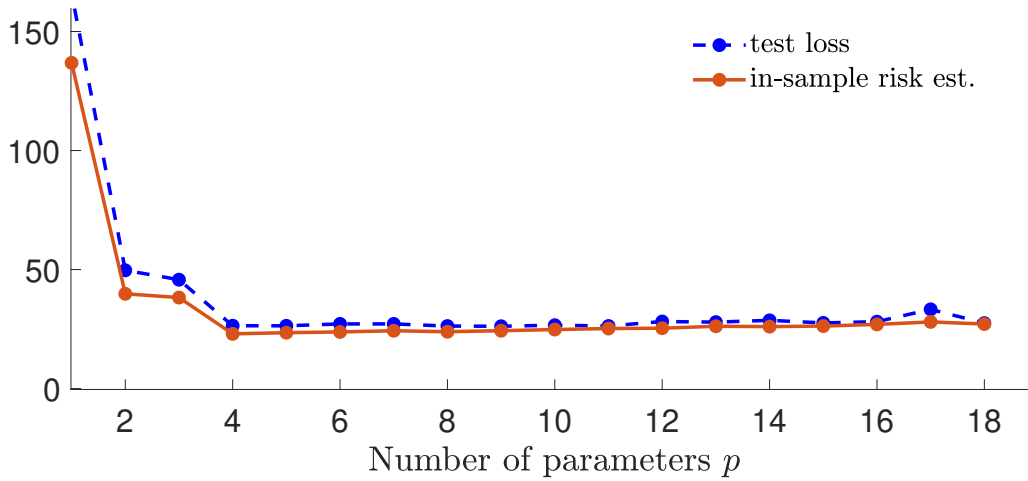


Figure 2.9: In-sample risk estimate $\widehat{\ell}_{\text{in}}(g_\tau)$ as a function of the number of parameters p of the model. The test loss is superimposed as a blue dashed curve.

2.5.2 Cross-Validation

In general, for complex function classes \mathcal{G} , it is very difficult to derive simple formulas of the approximation and statistical errors, let alone for the generalization risk or expected generalization risk. As we saw, when there is an abundance of data, the easiest way to assess the generalization risk for a given training set τ is to obtain a test set τ' and evaluate the test loss (2.7). When a sufficiently large test set is not available but computational resources are cheap, one can instead gain direct knowledge of the expected generalization risk via a computationally intensive method called *cross-validation*.

The idea is to make multiple identical copies of the data set, and to partition each copy into different training and test sets, as illustrated in Figure 2.10. Here, there are four copies of the data set (consisting of response and explanatory variables). Each copy is divided into a test set (colored blue) and training set (colored pink). For each of these sets, we estimate the model parameters using only training data and then predict the responses for the test set. The average loss between the predicted and observed responses is then a measure for the predictive power of the model.

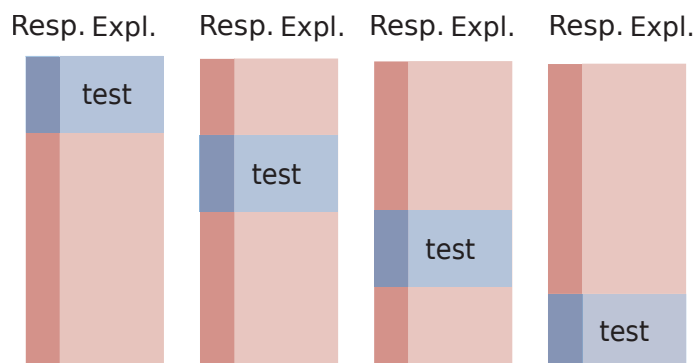


Figure 2.10: An illustration of four-fold cross-validation, representing four copies of the same data set. The data in each copy is partitioned into a training set (pink) and a test set (blue). The darker columns represent the response variable and the lighter ones the explanatory variables.

FOLDS

In particular, suppose we partition a data set \mathcal{T} of size n into K folds C_1, \dots, C_K of sizes n_1, \dots, n_K (hence, $n_1 + \dots + n_K = n$). Typically $n_k \approx n/K$, $k = 1, \dots, K$.

Let ℓ_{C_k} be the test loss when using C_k as test data and all remaining data, denoted \mathcal{T}_{-k} , as training data. Each ℓ_{C_k} is an unbiased estimator of the generalization risk for training set \mathcal{T}_{-k} ; that is, for $\ell(g_{\mathcal{T}_{-k}})$.

K-FOLD CROSS-VALIDATION

The K -fold cross-validation loss is the weighted average of these risk estimators:

$$\begin{aligned} \text{CV}_K &= \sum_{k=1}^K \frac{n_k}{n} \ell_{C_k}(g_{\mathcal{T}_{-k}}) \\ &= \frac{1}{n} \sum_{k=1}^K \sum_{i \in C_k} \text{Loss}(g_{\mathcal{T}_{-k}}(\mathbf{x}_i), y_i) \\ &= \frac{1}{n} \sum_{i=1}^n \text{Loss}(g_{\mathcal{T}_{-\kappa(i)}}(\mathbf{x}_i), y_i), \end{aligned}$$

where the function $\kappa : \{1, \dots, n\} \mapsto \{1, \dots, K\}$ indicates to which of the K folds each of the n observations belongs. As the average is taken over varying training sets $\{\mathcal{T}_{-k}\}$, it estimates the expected generalization risk $\mathbb{E} \ell(g_{\mathcal{T}})$, rather than the generalization risk $\ell(g_{\mathcal{T}})$ for the particular training set \mathcal{T} .

■ **Example 2.4 (Polynomial Regression (cont.))** For the polynomial regression example, we can calculate a K -fold cross-validation loss with a nonrandom partitioning of the training set using the following code, which imports the previous code for the polynomial regression example. We omit the full plotting code.

polyregCV.py

```

from polyreg3 import *

K_vals = [5, 10, 100] # number of folds
cv = np.zeros((len(K_vals), max_p)) # cv loss
X = np.ones((n, 1))

for p in p_range:
    if p > 1:
        X = np.hstack((X, u**(p-1)))
    j = 0
    for K in K_vals:
        loss = []
        for k in range(1, K+1):
            # integer indices of test samples
            test_ind = ((n/K)*(k-1) + np.arange(1,n/K+1)-1).astype('int')
            train_ind = np.setdiff1d(np.arange(n), test_ind)

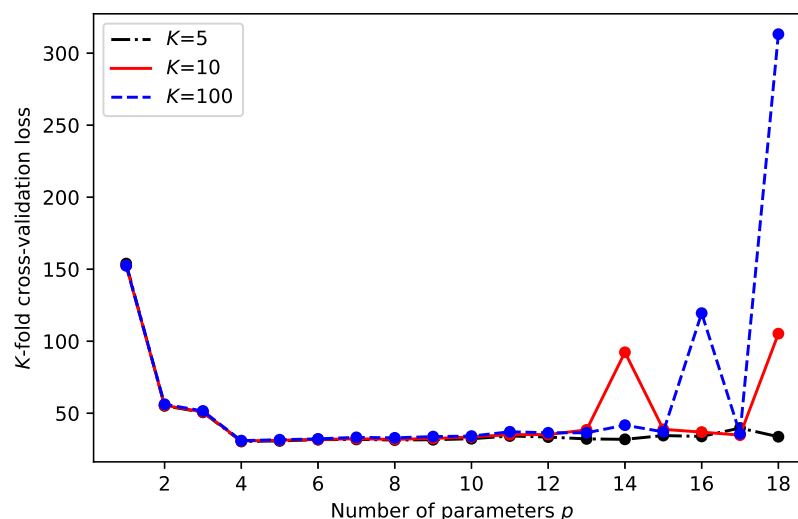
            X_train, y_train = X[train_ind, :], y[train_ind, :]
            X_test, y_test = X[test_ind, :], y[test_ind]

            # fit model and evaluate test loss
            betahat = solve(X_train.T @ X_train, X_train.T @ y_train)
            loss.append(norm(y_test - X_test @ betahat) ** 2)

        cv[j, p-1] = sum(loss)/n
        j += 1

# basic plotting
plt.plot(p_range, cv[0, :], 'k-.')
plt.plot(p_range, cv[1, :], 'r')
plt.plot(p_range, cv[2, :], 'b--')
plt.show()

```

Figure 2.11: K -fold cross-validation for the polynomial regression example.

LEAVE-ONE-OUT
CROSS-VALIDATION

174

Figure 2.11 shows the cross-validation loss for $K \in \{5, 10, 100\}$. The case $K = 100$ corresponds to the *leave-one-out cross-validation*, which can be computed more efficiently using the formula in Theorem 5.1. ■

2.6 Modeling Data

MODEL

The first step in any data analysis is to *model* the data in one form or another. For example, in an *unsupervised* learning setting with data represented by a vector $\mathbf{x} = [x_1, \dots, x_p]^\top$, a very general model is to assume that \mathbf{x} is the outcome of a random vector $\mathbf{X} = [X_1, \dots, X_p]^\top$ with some unknown pdf f . The model can then be refined by assuming a specific form of f .

431

When given a sequence of such data vectors $\mathbf{x}_1, \dots, \mathbf{x}_n$, one of the simplest models is to assume that the corresponding random vectors $\mathbf{X}_1, \dots, \mathbf{X}_n$ are *independent and identically distributed* (iid). We write

$$\mathbf{X}_1, \dots, \mathbf{X}_n \stackrel{\text{iid}}{\sim} f \quad \text{or} \quad \mathbf{X}_1, \dots, \mathbf{X}_n \stackrel{\text{iid}}{\sim} \text{Dist},$$

431

to indicate that the random vectors form an iid sample from a sampling pdf f or sampling distribution Dist. This model formalizes the notion that the knowledge about one variable does not provide extra information about another variable. The main theoretical use of independent data models is that the joint density of the random vectors $\mathbf{X}_1, \dots, \mathbf{X}_n$ is simply the *product* of the marginal ones; see Theorem C.1. Specifically,

$$f_{\mathbf{X}_1, \dots, \mathbf{X}_n}(\mathbf{x}_1, \dots, \mathbf{x}_n) = f(\mathbf{x}_1) \cdots f(\mathbf{x}_n).$$

427

In most models of this kind, our approximation or model for the sampling distribution is specified up to a small number of parameters. That is, $g(\mathbf{x})$ is of the form $g(\mathbf{x}|\boldsymbol{\beta})$ which is known up to some parameter vector $\boldsymbol{\beta}$. Examples for the one-dimensional case ($p = 1$) include the $\mathcal{N}(\mu, \sigma^2)$, $\text{Bin}(n, p)$, and $\text{Exp}(\lambda)$ distributions. See Tables C.1 and C.2 for other common sampling distributions.

11

Typically, the parameters are unknown and must be estimated from the data. In a non-parametric setting the whole sampling distribution would be unknown. To visualize the underlying sampling distribution from outcomes $\mathbf{x}_1, \dots, \mathbf{x}_n$ one can use graphical representations such as histograms, density plots, and empirical cumulative distribution functions, as discussed in Chapter 1.

If the order in which the data were collected (or their labeling) is not informative or relevant, then the joint pdf of $\mathbf{X}_1, \dots, \mathbf{X}_n$ satisfies the symmetry:

$$f_{\mathbf{X}_1, \dots, \mathbf{X}_n}(\mathbf{x}_1, \dots, \mathbf{x}_n) = f_{\mathbf{X}_{\pi_1}, \dots, \mathbf{X}_{\pi_n}}(\mathbf{x}_{\pi_1}, \dots, \mathbf{x}_{\pi_n}) \quad (2.26)$$

EXCHANGEABLE

for any permutation π_1, \dots, π_n of the integers $1, \dots, n$. We say that the infinite sequence $\mathbf{X}_1, \mathbf{X}_2, \dots$ is *exchangeable* if this permutational invariance (2.26) holds for any finite subset of the sequence. As we shall see in Section 2.9 on Bayesian learning, it is common to assume that the random vectors $\mathbf{X}_1, \dots, \mathbf{X}_n$ are a subset of an exchangeable sequence and thus satisfy (2.26). Note that while iid random variables are exchangeable, the converse is not necessarily true. Thus, the assumption of an exchangeable sequence of random vectors is weaker than the assumption of iid random vectors.

Figure 2.12 illustrates the modeling tradeoffs. The keywords within the triangle represent various modeling paradigms. A few keywords have been highlighted, symbolizing their importance in modeling. The specific meaning of the keywords does not concern us here, but the point is there are many models to choose from, depending on what assumptions are made about the data.

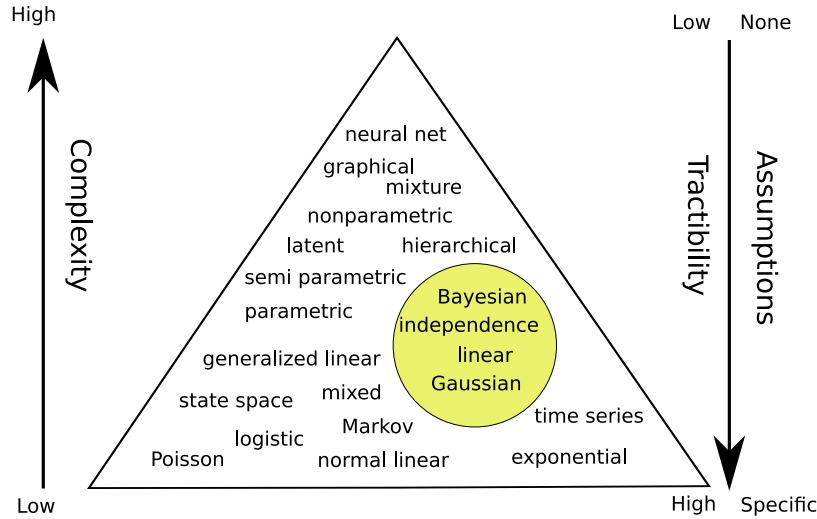


Figure 2.12: Illustration of the modeling dilemma. Complex models are more generally applicable, but may be difficult to analyze. Simple models may be highly tractable, but may not describe the data accurately. The triangular shape signifies that there are a great many specific models but not so many generic ones.

On the one hand, models that make few assumptions are more widely applicable, but at the same time may not be very mathematically tractable or provide insight into the nature of the data. On the other hand, very specific models may be easy to handle and interpret, but may not match the data very well. This tradeoff between the tractability and applicability of the model is very similar to the approximation–estimation tradeoff described in Section 2.4.

In the typical *unsupervised* setting we have a training set $\tau = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ that is viewed as the outcome of n iid random variables $\mathbf{X}_1, \dots, \mathbf{X}_n$ from some unknown pdf f . The objective is then to learn or estimate f from the finite training data. To put the learning in a similar framework as for supervised learning discussed in the preceding Sections 2.3–2.5, we begin by specifying a class of probability density functions $\mathcal{G}_p := \{g(\cdot | \boldsymbol{\theta}), \boldsymbol{\theta} \in \Theta\}$, where $\boldsymbol{\theta}$ is a parameter in some subset Θ of \mathbb{R}^p . We now seek the best g in \mathcal{G}_p to minimize some risk. Note that \mathcal{G}_p may not necessarily contain the true f even for very large p .



We stress that our notation $g(\mathbf{x})$ has a different meaning in the supervised and unsupervised case. In the supervised case, g is interpreted as a prediction function for a response y ; in the unsupervised setting, g is an approximation of a density f .

For each \mathbf{x} we measure the discrepancy between the true model $f(\mathbf{x})$ and the hypothesized model $g(\mathbf{x} | \boldsymbol{\theta})$ using the loss function

$$\text{Loss}(f(\mathbf{x}), g(\mathbf{x} | \boldsymbol{\theta})) = \ln \frac{f(\mathbf{x})}{g(\mathbf{x} | \boldsymbol{\theta})} = \ln f(\mathbf{x}) - \ln g(\mathbf{x} | \boldsymbol{\theta}).$$

The expected value of this loss (that is, the risk) is thus

$$\ell(g) = \mathbb{E} \ln \frac{f(X)}{g(X|\theta)} = \int f(x) \ln \frac{f(x)}{g(x|\theta)} dx. \quad (2.27)$$

KULLBACK–
LEIBLER
DIVERGENCE

The integral in (2.27) provides a fundamental way to measure the distance between two densities and is called the *Kullback–Leibler (KL) divergence*² between f and $g(\cdot|\theta)$. Note that the KL divergence is not symmetric in f and $g(\cdot|\theta)$. Moreover, it is always greater than or equal to 0 (see Exercise 15) and equal to 0 when $f = g(\cdot|\theta)$.

Using similar notation as for the supervised learning setting in Table 2.1, define $g^{\mathcal{G}_p}$ as the global minimizer of the risk in the class \mathcal{G}_p ; that is, $g^{\mathcal{G}_p} = \operatorname{argmin}_{g \in \mathcal{G}_p} \ell(g)$. If we define

$$\begin{aligned} \theta^* &= \operatorname{argmin}_{\theta} \mathbb{E} \operatorname{Loss}(f(X), g(X|\theta)) = \operatorname{argmin}_{\theta} \int (\ln f(x) - \ln g(x|\theta)) f(x) dx \\ &= \operatorname{argmax}_{\theta} \int f(x) \ln g(x|\theta) dx = \operatorname{argmax}_{\theta} \mathbb{E} \ln g(X|\theta), \end{aligned}$$

then $g^{\mathcal{G}_p} = g(\cdot|\theta^*)$ and learning $g^{\mathcal{G}_p}$ is equivalent to learning (or estimating) θ^* . To learn θ^* from a training set $\tau = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ we then minimize the training loss,

$$\frac{1}{n} \sum_{i=1}^n \operatorname{Loss}(f(\mathbf{x}_i), g(\mathbf{x}_i|\theta)) = -\frac{1}{n} \sum_{i=1}^n \ln g(\mathbf{x}_i|\theta) + \frac{1}{n} \sum_{i=1}^n \ln f(\mathbf{x}_i),$$

giving:

$$\widehat{\theta}_n := \operatorname{argmax}_{\theta} \frac{1}{n} \sum_{i=1}^n \ln g(\mathbf{x}_i|\theta). \quad (2.28)$$

As the logarithm is an increasing function, this is equivalent to

$$\widehat{\theta}_n := \operatorname{argmax}_{\theta} \prod_{i=1}^n g(\mathbf{x}_i|\theta),$$

where $\prod_{i=1}^n g(\mathbf{x}_i|\theta)$ is the *likelihood* of the data; that is, the joint density of the $\{\mathbf{X}_i\}$ evaluated at the points $\{\mathbf{x}_i\}$. We therefore have recovered the classical *maximum likelihood estimate* of θ^* .

MAXIMUM
LIKELIHOOD
ESTIMATE

When the risk $\ell(g(\cdot|\theta))$ is convex in θ over a convex set Θ , we can find the maximum likelihood estimator by setting the gradient of the training loss to zero; that is, we solve

$$-\frac{1}{n} \sum_{i=1}^n \mathbf{S}(\mathbf{x}_i|\theta) = \mathbf{0},$$

where $\mathbf{S}(\mathbf{x}|\theta) := \frac{\partial \ln g(\mathbf{x}|\theta)}{\partial \theta}$ is the gradient of $\ln g(\mathbf{x}|\theta)$ with respect to θ and is often called the *score*.

SCORE

■ **Example 2.5 (Exponential Model)** Suppose we have the training data $\tau_n = \{x_1, \dots, x_n\}$, which is modeled as a realization of n positive iid random variables: $X_1, \dots, X_n \sim_{\text{iid}} f(x)$. We select the class of approximating functions \mathcal{G} to be the parametric class $\{g : g(x|\theta) =$

²Sometimes called cross-entropy distance.

$\theta \exp(-x\theta), x > 0, \theta > 0\}$. In other words, we look for the best $g^{\mathcal{G}}$ within the family of exponential distributions with unknown parameter $\theta > 0$. The likelihood of the data is

$$\prod_{i=1}^n g(x_i | \theta) = \prod_{i=1}^n \theta \exp(-\theta x_i) = \exp(-\theta n \bar{x}_n + n \ln \theta)$$

and the score is $S(x | \theta) = -x + \theta^{-1}$. Thus, maximizing the likelihood with respect to θ is the same as maximizing $-\theta n \bar{x}_n + n \ln \theta$ or solving $-\sum_{i=1}^n S(x_i | \theta)/n = \bar{x}_n - \theta^{-1} = 0$. In other words, the solution to (2.28) is the maximum likelihood estimate $\hat{\theta}_n = 1/\bar{x}_n$. ■

In a *supervised* setting, where the data is represented by a vector \mathbf{x} of explanatory variables and a response y , the general model is that (\mathbf{x}, y) is an outcome of $(\mathbf{X}, Y) \sim f$ for some unknown f . And for a training sequence $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$ the default model assumption is that $(\mathbf{X}_1, Y_1), \dots, (\mathbf{X}_n, Y_n) \sim_{\text{iid}} f$. As explained in Section 2.2, the analysis primarily involves the conditional pdf $f(y | \mathbf{x})$ and in particular (when using the squared-error loss) the conditional expectation $g^*(\mathbf{x}) = \mathbb{E}[Y | \mathbf{X} = \mathbf{x}]$. The resulting representation (2.2) allows us to then write the response at $\mathbf{X} = \mathbf{x}$ as a function of the feature \mathbf{x} plus an error term: $Y = g^*(\mathbf{x}) + \varepsilon(\mathbf{x})$.

This leads to the simplest and most important model for supervised learning, where we choose a *linear* class \mathcal{G} of prediction or guess functions and assume that it is rich enough to contain the true g^* . If we further assume that, conditional on $\mathbf{X} = \mathbf{x}$, the error term ε does not depend on \mathbf{x} , that is, $\mathbb{E} \varepsilon = 0$ and $\text{Var} \varepsilon = \sigma^2$, then we obtain the following model.

Definition 2.1: Linear Model

In a *linear model* the response Y depends on a p -dimensional explanatory variable $\mathbf{x} = [x_1, \dots, x_p]^\top$ via the linear relationship

$$Y = \mathbf{x}^\top \boldsymbol{\beta} + \varepsilon, \quad (2.29)$$

where $\mathbb{E} \varepsilon = 0$ and $\text{Var} \varepsilon = \sigma^2$.

LINEAR MODEL

Note that (2.29) is a model for a single pair (\mathbf{x}, Y) . The model for the training set $\{(\mathbf{x}_i, Y_i)\}$ is simply that each Y_i satisfies (2.29) (with $\mathbf{x} = \mathbf{x}_i$) and that the $\{Y_i\}$ are independent. Gathering all responses in the vector $\mathbf{Y} = [Y_1, \dots, Y_n]^\top$, we can write

$$\mathbf{Y} = \mathbf{X} \boldsymbol{\beta} + \boldsymbol{\varepsilon}, \quad (2.30)$$

where $\boldsymbol{\varepsilon} = [\varepsilon_1, \dots, \varepsilon_n]^\top$ is a vector of iid copies of ε and \mathbf{X} is the so-called *model matrix*, with rows $\mathbf{x}_1^\top, \dots, \mathbf{x}_n^\top$. Linear models are fundamental building blocks of statistical learning algorithms. For this reason, a large part of Chapter 5 is devoted to linear regression models.

MODEL MATRIX

👉 167

■ **Example 2.6 (Polynomial Regression (cont.))** For our running Example 2.1, we see that the data is described by a linear model of the form (2.30), with model matrix \mathbf{X} given in (2.10). ■

👉 26

Before we discuss a few other models in the following sections, we would like to emphasize a number of points about modeling.

- Any model for data is likely to be *wrong*. For example, real data (as opposed to computer-generated data) are often assumed to come from a normal distribution, which is never exactly true. However, an important advantage of using a normal distribution is that it has many nice mathematical properties, as we will see in Section 2.7.
- Most data models depend on a number of unknown parameters, which need to be estimated from the observed data.
- Any model for real-life data needs to be *checked* for suitability. An important criterion is that data simulated from the model should resemble the observed data, at least for a certain choice of model parameters.

Here are some guidelines for choosing a model. Think of the data as a spreadsheet or data frame, as in Chapter 1, where rows represent the data units and the columns the data features (variables, groups).

- First establish the *type* of the features (quantitative, qualitative, discrete, continuous, etc.).
- Assess whether the data can be assumed to be independent across rows or columns.
- Decide on the level of generality of the model. For example, should we use a simple model with a few unknown parameters or a more generic model that has a large number of parameters? Simple specific models are easier to fit to the data (low estimation error) than more general models, but the fit itself may not be accurate (high approximation error). The tradeoffs discussed in Section 2.4 play an important role here.
- Decide on using a classical (frequentist) or Bayesian model. Section 2.9 gives a short introduction to Bayesian learning.

47

2.7 Multivariate Normal Models

A standard model for numerical observations x_1, \dots, x_n (forming, e.g., a column in a spreadsheet or data frame) is that they are the outcomes of iid normal random variables

$$X_1, \dots, X_n \stackrel{\text{iid}}{\sim} \mathcal{N}(\mu, \sigma^2).$$

It is helpful to view a normally distributed random variable as a simple transformation of a standard normal random variable. To wit, if Z has a standard normal distribution, then $X = \mu + \sigma Z$ has a $\mathcal{N}(\mu, \sigma^2)$ distribution. The generalization to n dimensions is discussed in Appendix C.7. We summarize the main points: Let $Z_1, \dots, Z_n \stackrel{\text{iid}}{\sim} \mathcal{N}(0, 1)$. The pdf of $\mathbf{Z} = [Z_1, \dots, Z_n]^\top$ (that is, the joint pdf of Z_1, \dots, Z_n) is given by

436

$$f_{\mathbf{Z}}(\mathbf{z}) = \prod_{i=1}^n \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}z_i^2} = (2\pi)^{-\frac{n}{2}} e^{-\frac{1}{2}\mathbf{z}^\top \mathbf{z}}, \quad \mathbf{z} \in \mathbb{R}^n. \quad (2.31)$$