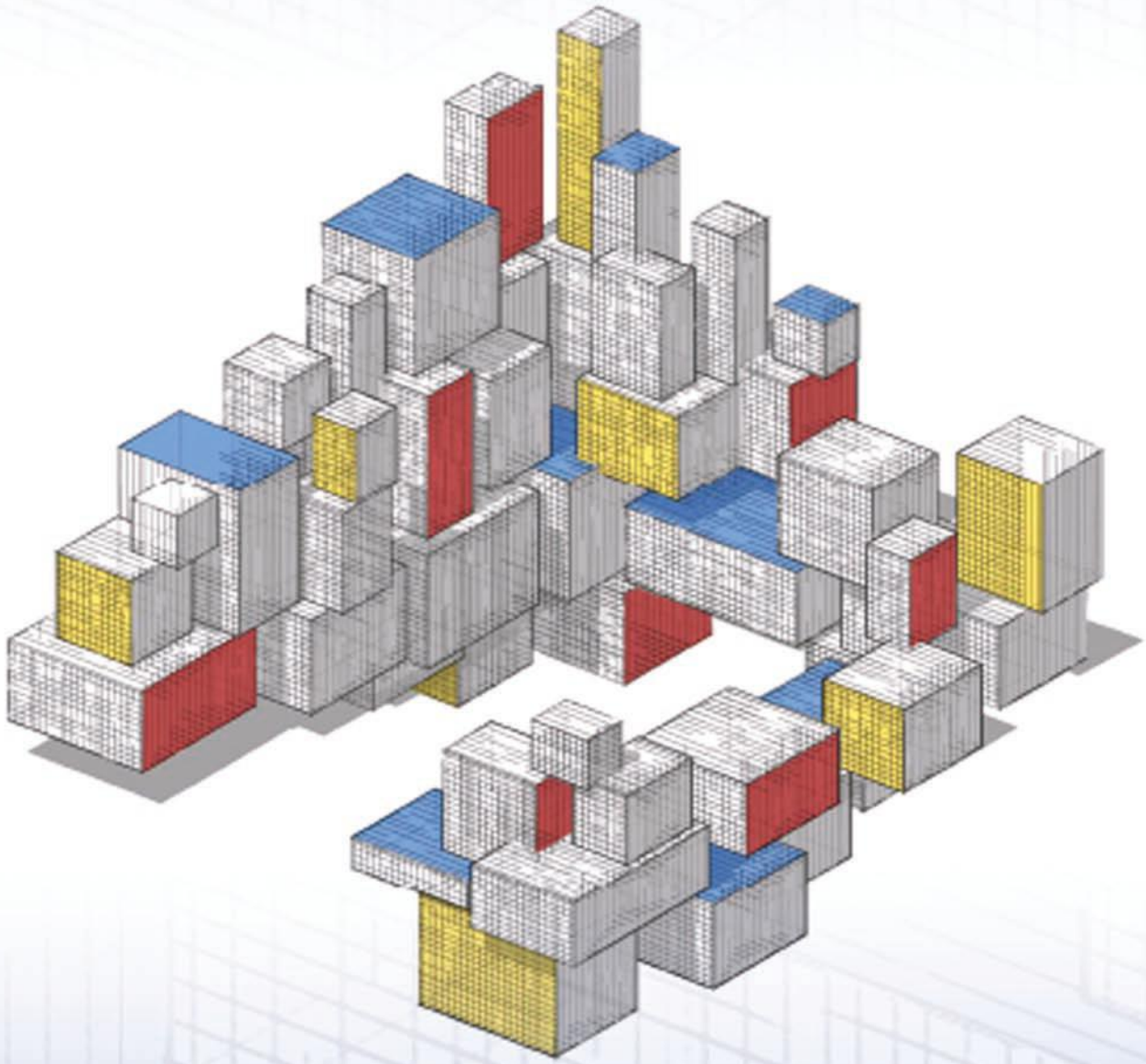
Chapman & Hall/CRC
Machine Learning & Pattern Recognition Series

DATA SCIENCE AND MACHINE LEARNING

MATHEMATICAL AND STATISTICAL METHODS



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

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Data Science and Machine Learning

Mathematical and Statistical Methods

Chapman & Hall/CRC Machine Learning & Pattern Recognition

Introduction to Machine Learning with Applications in Information Security
Mark Stamp

A First Course in Machine Learning
Simon Rogers, Mark Girolami

Statistical Reinforcement Learning: Modern Machine Learning Approaches
Masashi Sugiyama

Sparse Modeling: Theory, Algorithms, and Applications
Irina Rish, Genady Grabarnik

Computational Trust Models and Machine Learning
Xin Liu, Anwitaman Datta, Ee-Peng Lim

Regularization, Optimization, Kernels, and Support Vector Machines
Johan A.K. Suykens, Marco Signoretto, Andreas Argyriou

Machine Learning: An Algorithmic Perspective, Second Edition
Stephen Marsland

Bayesian Programming
Pierre Bessiere, Emmanuel Mazer, Juan Manuel Ahuactzin, Kamel Mekhnacha

Multilinear Subspace Learning: Dimensionality Reduction of Multidimensional Data
Haiping Lu, Konstantinos N. Plataniotis, Anastasios Venetsanopoulos

Data Science and Machine Learning: Mathematical and Statistical Methods
Dirk P. Kroese, Zdravko I. Botev, Thomas Taimre, Radislav Vaisman

For more information on this series please visit: https://www.crcpress.com/Chapman--HallCRC-Machine-Learning--Pattern-Recognition/book-series/erie

Data Science and Machine Learning

Mathematical and Statistical Methods

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


CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

Front cover image reproduced with permission from J. A. Kroese.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper

International Standard Book Number-13: 978-1-138-49253-0 (Hardback)

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

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

— *DPK*

To Sarah, Sofia, and my parents

— *ZIB*

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

— *TT*

To Valerie

— *RV*


Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CONTENTS

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.

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, $\boldsymbol{x}, \boldsymbol{X}$, 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., $\boldsymbol{X}$ (random vector) and $\boldsymbol{x}$ (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.

KEYWORDS

☞ 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 Liquet 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 accollade 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 I. 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 I. Botev
Thomas Taimre, and Radislav Vaisman
Brisbane and Sydney


Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

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 $\boldsymbol{x} = [x_1, \ldots, x_n]^\top$ 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: $\boldsymbol{X} = [X_1, \ldots, X_n]^\top$.

• 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 $\mathbf{X}$ 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
$\boldsymbol{x}$	vector
$\boldsymbol{X}$	random vector
$\mathbf{X}$	matrix
$\mathcal{X}$	set
$\widehat{x}$	estimate or approximation
x^*	optimal
$\overline{x}$	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
$\xrightarrow{\text{d}}$	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

$\mathbf{A}^\top, \boldsymbol{x}^\top$	transpose of matrix $\mathbf{A}$ or vector $\boldsymbol{x}$
$\mathbf{A}^{-1}$	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 $\mathbf{A}^{-1}$
$\mathbf{A} \succ 0$	matrix $\mathbf{A}$ is positive definite
$\mathbf{A} \succeq 0$	matrix $\mathbf{A}$ is positive semidefinite
$\dim(\boldsymbol{x})$	dimension of vector $\boldsymbol{x}$
$\det(\mathbf{A})$	determinant of matrix $\mathbf{A}$
$\lvert\mathbf{A}\rvert$	absolute value of the determinant of matrix $\mathbf{A}$
$\operatorname{tr}(\mathbf{A})$	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 $\{0, 1, \ldots\}$
O	big-O order symbol: $f(x) = O(g(x))$ if $\lvert f(x)\rvert \leqslant \alpha g(x)$ for some constant α as $x \to a$
o	little-o order symbol: $f(x) = o(g(x))$ if $f(x)/g(x) \to 0$ as $x \to a$
$\mathbb{P}$	probability measure
π	the number $3.14159\ldots$
$\mathbb{R}$	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 $\{\ldots, -1, 0, 1, \ldots\}$

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 *a posteriori*
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

CHAPTER 1

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
☞ 485
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/springeR/nutrition_elderly.xls`.

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

```
xls = 'http://www.biostatisticien.eu/springeR/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, $\boldsymbol{x} = [x_1, \ldots, x_n]^\mathsf{T}$ is a column vector of n numbers. For our **nutri** data, the vector $\boldsymbol{x}$ could, for example, correspond to the heights of the $n = 226$ individuals.

The *sample mean* of $\boldsymbol{x}$, denoted by $\overline{x}$, is simply the average of the data values:

SAMPLE MEAN

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

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 $\boldsymbol{x}$ 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*

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

SAMPLE STANDARD DEVIATION

☞ 455

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 cm^2) 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()
```

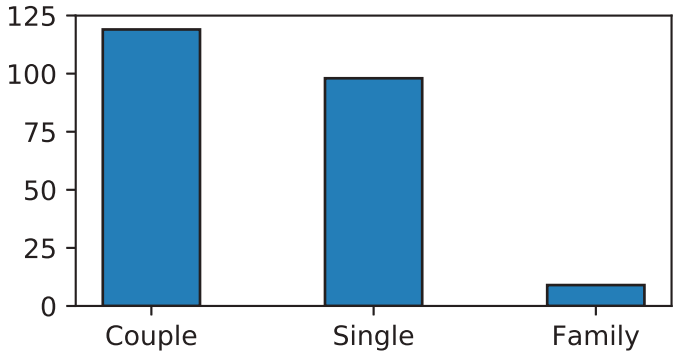


Figure 1.1: Barplot for the qualitative variable `'situation'`.

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.

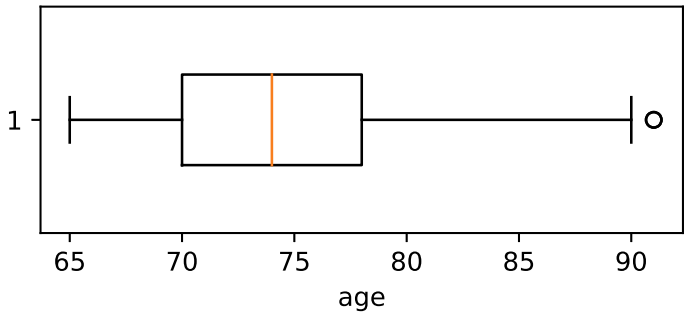


Figure 1.2: Boxplot for `'age'`.

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) $Q_1 - 1.5\,\text{IQR}$. Similarly, the right whisker extends to the smallest of (a) the maximum of the data and (b) $Q_3 + 1.5\,\text{IQR}$. 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.

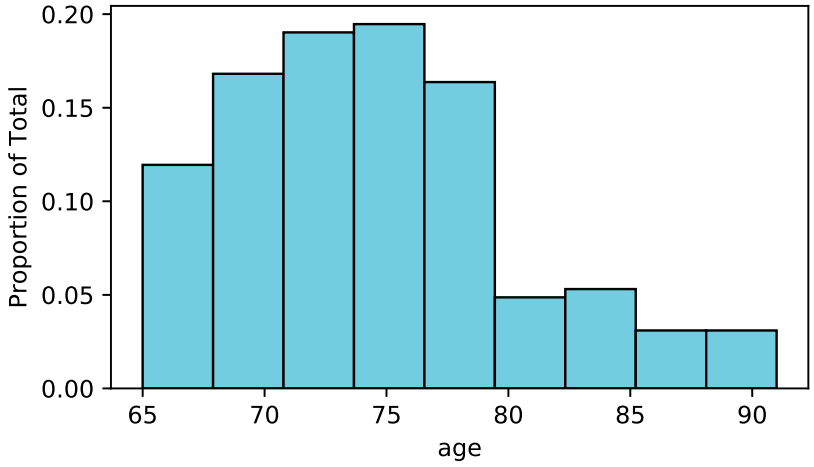


Figure 1.3: Histogram of `'age'`.

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

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 $x_1, \ldots, x_n$, $F_n(x)$ is the fraction of observations less than or equal to x, i.e.,

$$F_n(x) = \frac{\text{number of } x_i \leqslant x}{n} = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}\left\{x_i \leqslant x\right\}, \tag{1.2}$$

INDICATOR

where $\mathbb{1}$ denotes the *indicator* function; that is, $\mathbb{1}\left\{x_i \leqslant x\right\}$ is equal to 1 when $x_i \leqslant x$ 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.

```
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()
```

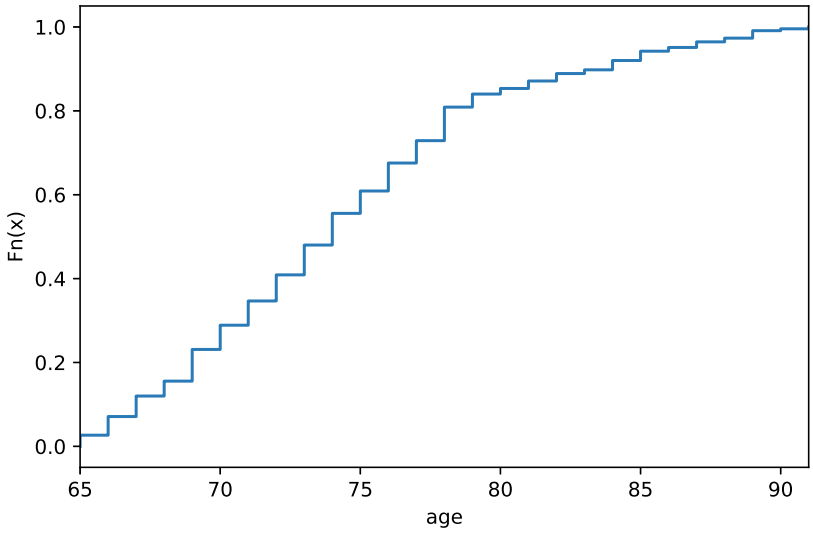


Figure 1.4: Plot of the empirical distribution function for the continuous quantitative feature `'age'`.

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.

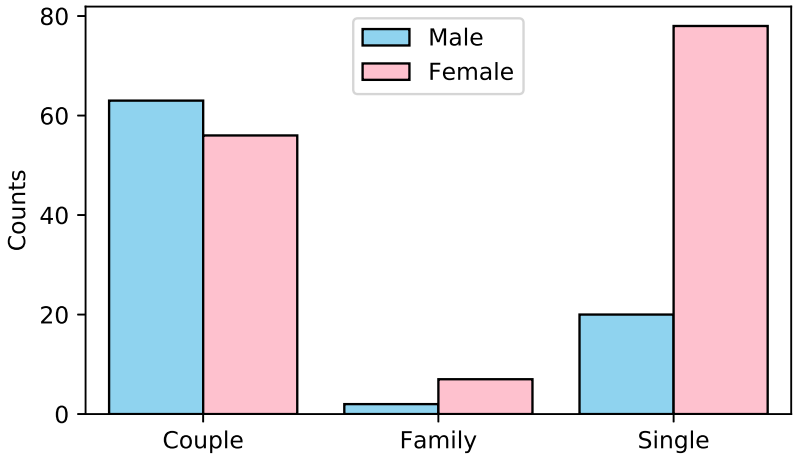


Figure 1.5: Barplot for two categorical variables.

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()
```

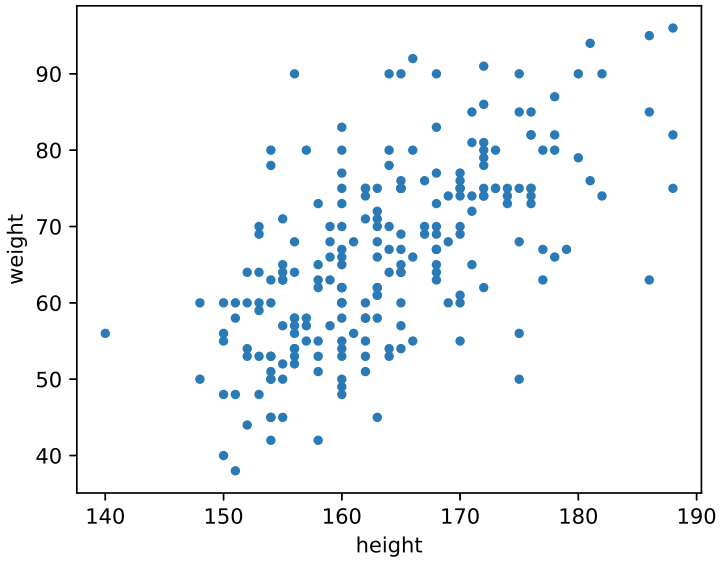


Figure 1.6: Scatterplot of `'weight'` against `'height'`.

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.

☞ 199

```
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()
```

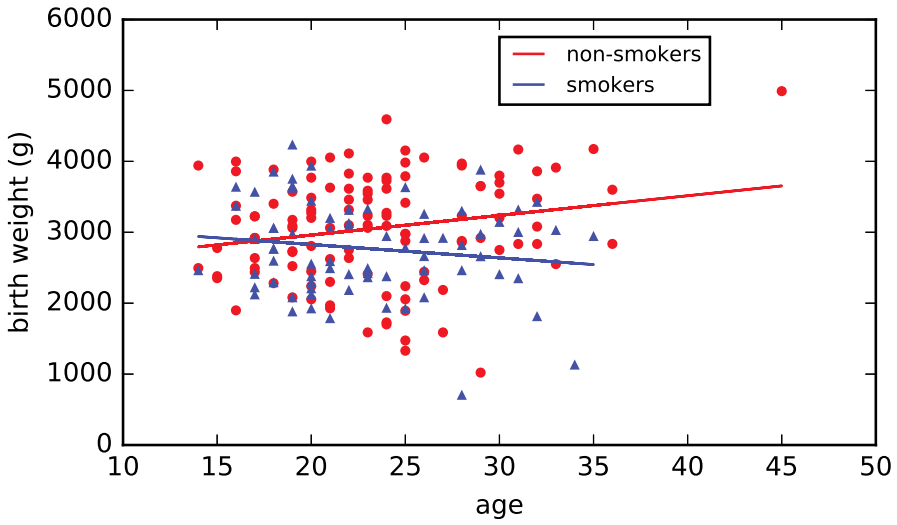


Figure 1.7: Birth weight against age for smoking and non-smoking mothers.

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()
```

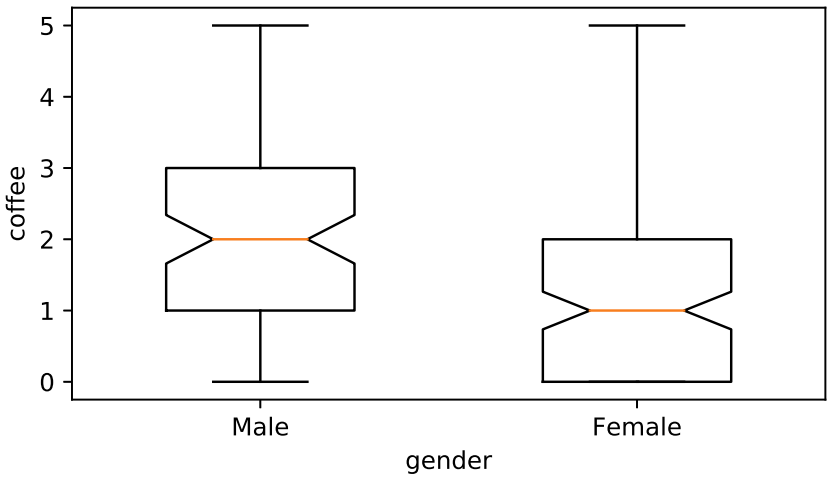


Figure 1.8: Boxplots of a quantitative feature `'coffee'` as a function of the levels of a categorical feature `'gender'`. Note that we used a different, "notched", style boxplot this time.

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

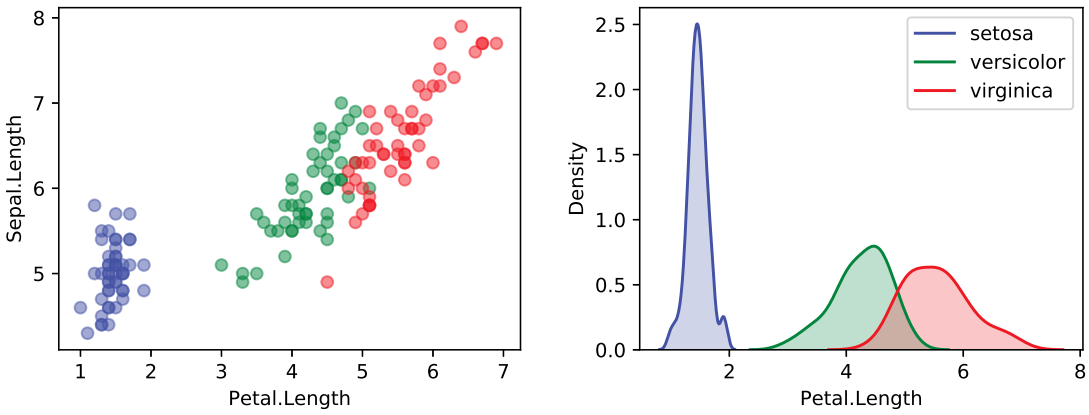


Figure 1.9: Left: scatterplot of `'Sepal.Length'` against `'Petal.Length'`. Right: kernel density estimates of `'Petal.Length'` for the three species of iris.

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.]

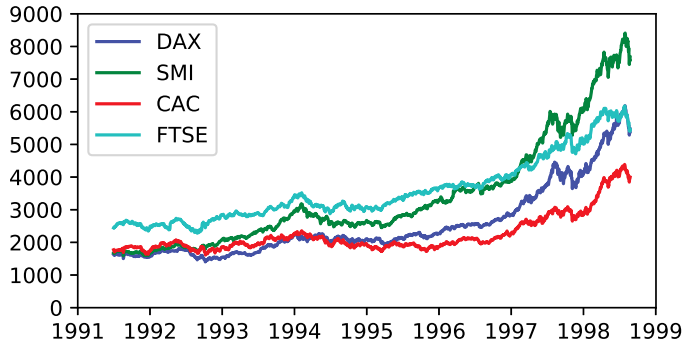


Figure 1.10: Closing stock indices for various European stock markets.

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.

(a) 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.

(b) The data set contains the circumferences of 5 orange trees at various stages in their development. Find the names of the features.

(c) In Python, import seaborn and visualize the growth curves (circumference against age) of the trees, using the regplot and FacetGrid methods.

CHAPTER 2

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*.

STATISTICAL LEARNING

MACHINE LEARNING

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.

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 $\boldsymbol{x}$, one of the main goals of machine learning is to predict an output or *response* variable y. For example, $\boldsymbol{x}$ could be a digitized signature and y a binary variable that indicates whether the signature is genuine or false. Another example is where $\boldsymbol{x}$ 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 $\boldsymbol{x}$ and outputs a guess $g(\boldsymbol{x})$ for y (denoted by $\widehat{y}$, for example). In a sense, g encompasses all the information about the relationship between the variables $\boldsymbol{x}$ 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 $y \in \{0, \ldots, c-1\}$, then predicting y is conceptually the same as classifying the input $\boldsymbol{x}$ into one of c categories, and so prediction becomes a *classification* problem.

CLASSIFICATION

LOSS FUNCTION

We can measure the accuracy of a prediction $\widehat{y}$ with respect to a given response y by using some *loss function* $\text{Loss}(y, \widehat{y})$. In a regression setting the usual choice is the squared-error loss $(y-\widehat{y})^2$. In the case of classification, the zero–one (also written 0–1) loss function $\text{Loss}(y, \widehat{y}) = \mathbb{1}\{y \neq \widehat{y}\}$ is often used, which incurs a loss of 1 whenever the predicted class $\widehat{y}$ 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 $\widehat{y}$ thereof. If y is real-valued, the absolute error $|y-\widehat{y}|$ and the squared error $(y-\widehat{y})^2$ are both well-established error concepts, as are the norm $\|\boldsymbol{y}-\widehat{\boldsymbol{y}}\|$ and squared norm $\|\boldsymbol{y}-\widehat{\boldsymbol{y}}\|^2$ for vectors. The squared error $(y-\widehat{y})^2$ is just one example of a loss function.

It is unlikely that any mathematical function g will be able to make accurate predictions for all possible pairs $(\boldsymbol{x}, y)$ one may encounter in Nature. One reason for this is that, even with the same input $\boldsymbol{x}$, 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 $(\boldsymbol{x}, y)$ is the outcome of a random pair $(\boldsymbol{X}, Y)$ that has some joint probability density $f(\boldsymbol{x}, y)$. We then assess the predictive performance via the expected loss, usually called the *risk*, for g:

RISK

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

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

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

CLASSIFIER

☞ 251

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

where $f(y \,|\, \boldsymbol{x}) = \mathbb{P}[Y = y \,|\, \boldsymbol{X} = \boldsymbol{x}]$ is the conditional probability of $Y = y$ given $\boldsymbol{X} = \boldsymbol{x}$. 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.

REGRESSION FUNCTION

Theorem 2.1: Optimal Prediction Function for Squared-Error Loss

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

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

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

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

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

☞ 431

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

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

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

Since, the optimal prediction function g^* depends on the typically unknown joint distribution of $(\boldsymbol{X}, Y)$, it is not available in practice. Instead, all that we have available is a finite number of (usually) independent realizations from the joint density $f(\boldsymbol{x}, y)$. We denote this sample by $\mathcal{T} = \{(\boldsymbol{X}_1, Y_1), \ldots, (\boldsymbol{X}_n, Y_n)\}$ 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 $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$. 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.

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

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 $g^* : \boldsymbol{x} \mapsto y$ 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 $\boldsymbol{X}_i$ for $i = 1, \ldots, n$, and thus "trains" the learner $g_{\mathcal{T}}$ to predict the output of a new input $\boldsymbol{X}$, for which the correct output Y is not provided by the teacher (is unknown).

LEARNER

SUPERVISED LEARNING

The above setting is called *supervised learning*, because one tries to learn the functional relationship between the feature vector $\boldsymbol{x}$ 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 $\boldsymbol{x}$, where $\boldsymbol{x}$ 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 $\boldsymbol{x}$, 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 $f(y \,|\, \boldsymbol{x})$, because we can then in principle work out the function value $g^*(\boldsymbol{x})$.

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 $f(\boldsymbol{x})$. In this case the guess $g(\boldsymbol{x})$ is an approximation of $f(\boldsymbol{x})$ and the risk is of the form

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

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 $\boldsymbol{x} \in \{0,1\}^{100}$ 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 $k \in \{1, \ldots, 100\}$ and a 0 otherwise). Based on a training set $\tau = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$, 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.

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

☞ 121

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.

☞ 167
☞ 251

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

$$\ell_{\mathcal{T}}(g) = \frac{1}{n}\sum_{i=1}^{n} \text{Loss}(Y_i, g(\boldsymbol{X}_i)), \tag{2.3}$$

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,

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

For example, the simplest and most useful $\mathcal{G}$ is the set of *linear* functions of $\boldsymbol{x}$; that is, the set of all functions $g : \boldsymbol{x} \mapsto \boldsymbol{\beta}^\top \boldsymbol{x}$ 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 $g(\boldsymbol{X}_i) = Y_i$ 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

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

where $(\boldsymbol{X}, Y)$ is distributed according to $f(\boldsymbol{x}, y)$. In the discrete case the generalization risk is therefore: $\ell(g_\tau^{\mathcal{G}}) = \sum_{\boldsymbol{x},y} \text{Loss}(y, g_\tau^{\mathcal{G}}(\boldsymbol{x}))f(\boldsymbol{x}, y)$ (replace the sum with an integral for the continuous case). The situation is illustrated in Figure 2.1, where the distribution of $(\boldsymbol{X}, Y)$ 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 $(\boldsymbol{X}, Y)$ 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 $(\boldsymbol{x}, y)$, weighted by the corresponding $f(\boldsymbol{x}, y)$.

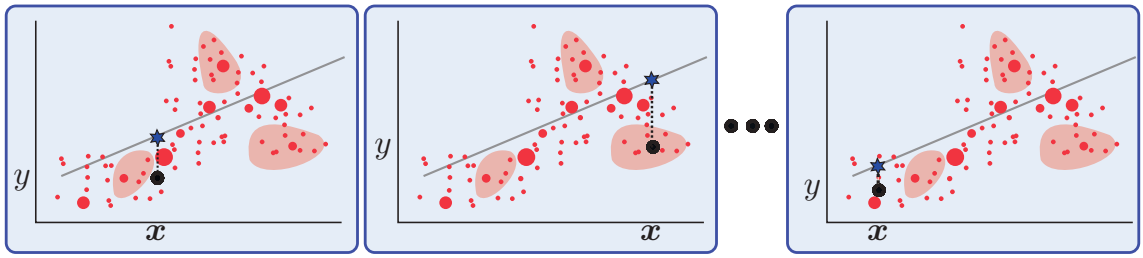


Figure 2.1: The generalization risk for a fixed training set is the weighted-average loss over all possible pairs $(\boldsymbol{x}, y)$.

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

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

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

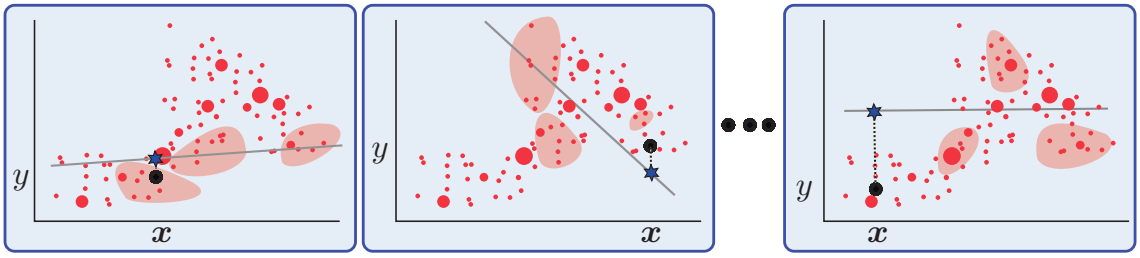


Figure 2.2: The expected generalization risk is the weighted-average loss over all possible pairs $(\boldsymbol{x}, y)$ and over all training sets.

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

$$\ell_{\mathcal{T}'}(g_{\tau}^{\mathcal{G}}) := \frac{1}{n'}\sum_{i=1}^{n'} \mathrm{Loss}(Y_i', g_{\tau}^{\mathcal{G}}(\boldsymbol{X}_i')), \tag{2.7}$$

TEST SAMPLE

TEST LOSS

where $\{(\boldsymbol{X}_1', Y_1'), \ldots, (\boldsymbol{X}_{n'}', Y_{n'}')\} =: \mathcal{T}'$ 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 $f(\boldsymbol{x}, y)$, 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.

$\boldsymbol{x}$	Fixed explanatory (feature) vector.
$\boldsymbol{X}$	Random explanatory (feature) vector.
y	Fixed (real-valued) response.
Y	Random response.
$f(\boldsymbol{x}, y)$	Joint pdf of $\boldsymbol{X}$ and Y, evaluated at $(\boldsymbol{x}, y)$.
$f(y \mid \boldsymbol{x})$	Conditional pdf of Y given $\boldsymbol{X} = \boldsymbol{x}$, evaluated at y.
τ or τ_n	Fixed training data $\{(\boldsymbol{x}_i, y_i), i = 1, \ldots, n\}$.
$\mathcal{T}$ or $\mathcal{T}_n$	Random training data $\{(\boldsymbol{X}_i, Y_i), i = 1, \ldots, n\}$.
$\mathbf{X}$	Matrix of explanatory variables, with n rows $\boldsymbol{x}_i^\top, i = 1, \ldots, n$ and $\dim(\boldsymbol{x})$ feature columns; one of the features may be the constant 1.
$\boldsymbol{y}$	Vector of response variables $(y_1, \ldots, y_n)^\top$.
g	Prediction (guess) function.
$\text{Loss}(y, \widehat{y})$	Loss incurred when predicting response y with $\widehat{y}$.
$\ell(g)$	Risk for prediction function g; that is, $\mathbb{E}\,\text{Loss}(Y, g(\boldsymbol{X}))$.
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 *learner*: $\text{argmin}_{g \in \mathcal{G}}\, \ell_\tau(g)$. 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 $g_\tau^{\mathcal{G}_1}, g_\tau^{\mathcal{G}_2}, \ldots$, 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

☞ 37

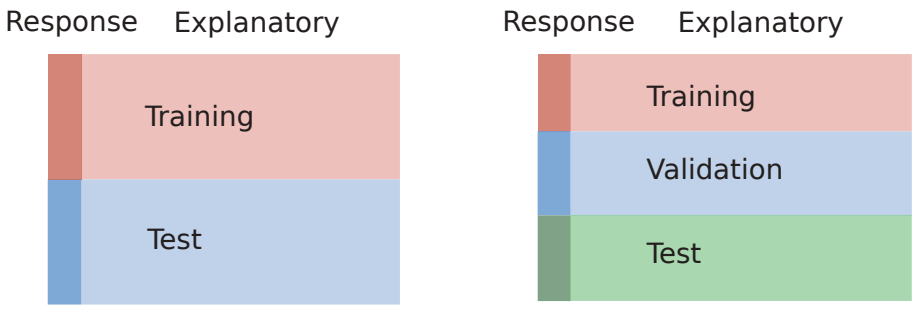


Figure 2.3: Statistical learning algorithms often require the data to be divided into training and test data. If the latter is used for model selection, a third set is needed for testing the performance of the selected model.

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 $\boldsymbol{x}, g, \mathcal{G}$ 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 $(u_i, y_i), i = 1, \ldots, n$ drawn from iid random points $(U_i, Y_i), i = 1, \ldots, n$, 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 $h^*(u) = \mathbb{E}[Y \mid U = u]$ is thus

POLYNOMIAL REGRESSION MODEL

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

which is depicted by the dashed curve in Figure 2.4.

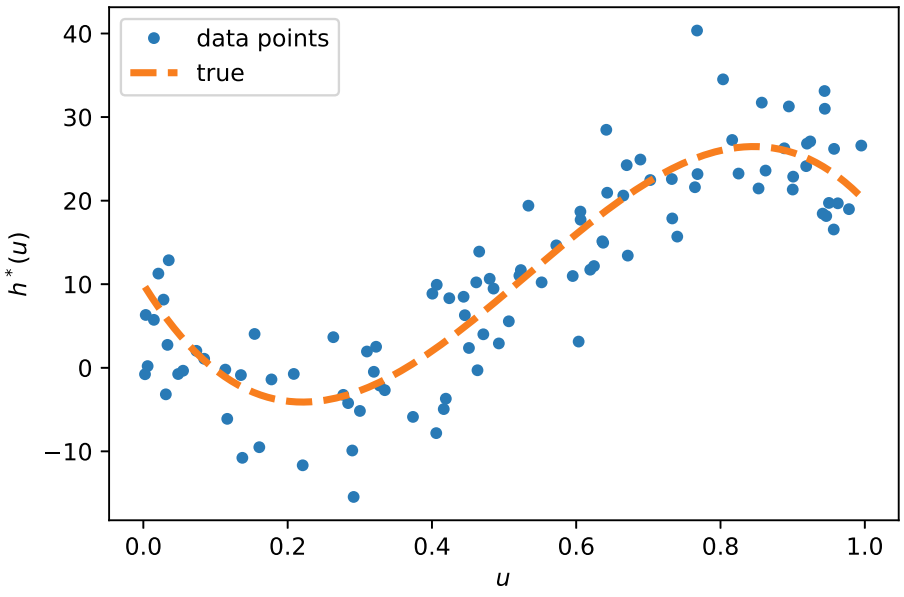


Figure 2.4: Training data and the optimal polynomial prediction function h^*.

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

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

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

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

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

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

which is linear in $\boldsymbol{x}$ (as well as $\boldsymbol{\beta}$). The optimal $h^*(u)$ in $\mathcal{H}_p$ for $p \geqslant 4$ then corresponds to the function $g^*(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{\beta}^*$ in the set $\mathcal{G}_p$ of linear functions from $\mathbb{R}^p$ to $\mathbb{R}$, where $\boldsymbol{\beta}^* = [10, -140, 400, -250, 0, \ldots, 0]^\top$. 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 $\boldsymbol{x}_i = [1, u_i, u_i^2, \ldots, u_i^{p-1}]^\top$, $i = 1, \ldots, n$. It will be convenient to arrange these feature vectors into a matrix $\mathbf{X}$ with rows $\boldsymbol{x}_1^\top, \ldots, \boldsymbol{x}_n^\top$:

$$\mathbf{X} = \begin{bmatrix} 1 & u_1 & u_1^2 & \cdots & u_1^{p-1} \\ 1 & u_2 & u_2^2 & \cdots & u_2^{p-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & u_n & u_n^2 & \cdots & u_n^{p-1} \end{bmatrix}. \tag{2.10}$$

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

$$\frac{1}{n} \|\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}\|^2. \tag{2.11}$$

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

$$\widehat{\boldsymbol{\beta}} = \operatorname*{argmin}_{\boldsymbol{\beta}} \|\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}\|^2, \tag{2.12}$$

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

ORDINARY LEAST-SQUARES

PROJECTION MATRIX

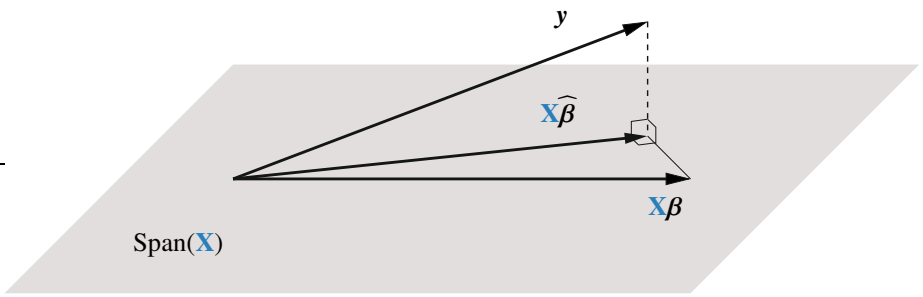


Figure 2.5: $\mathbf{X}\widehat{\boldsymbol{\beta}}$ is the orthogonal projection of $\boldsymbol{y}$ onto the linear space spanned by the columns of the matrix $\mathbf{X}$.

☞ 362

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

$$\mathbf{P} = \mathbf{X}\,\mathbf{X}^{+}, \tag{2.13}$$

☞ 360

PSEUDO-INVERSE

☞ 356

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 $\mathbf{X}\widehat{\boldsymbol{\beta}} = \mathbf{P}\boldsymbol{y}$ and $\mathbf{P}\mathbf{X} = \mathbf{X}$, we can see that $\widehat{\boldsymbol{\beta}}$ satisfies the *normal equations*:

NORMAL EQUATIONS

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

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

$$\widehat{\boldsymbol{\beta}} = \mathbf{X}^{+}\boldsymbol{y}. \tag{2.15}$$

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

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

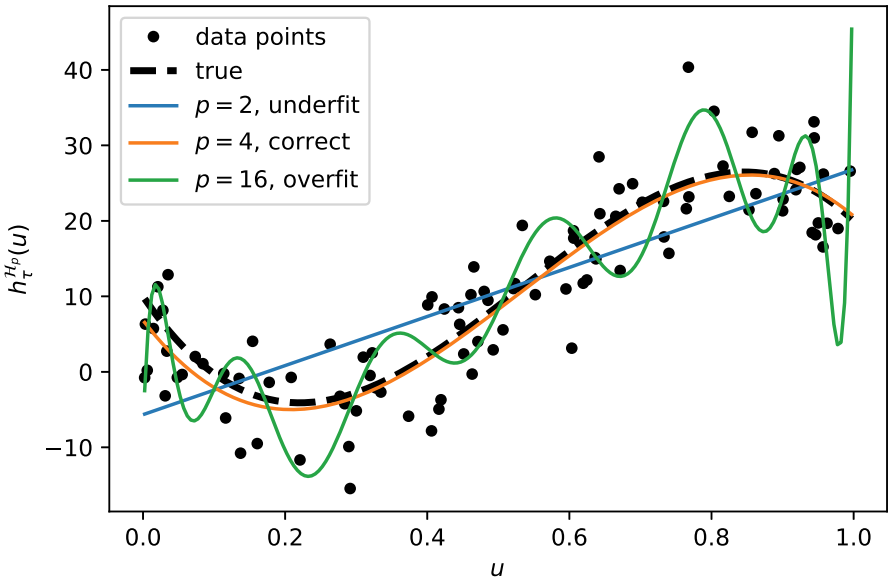


Figure 2.6: Training data with fitted curves for $p = 2, 4$, and 16. The true cubic polynomial curve for $p = 4$ is also plotted (dashed line).

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 $g_\tau^{\mathcal{G}_p}$, $p = 1, 2, \ldots$. 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 $\boldsymbol{y}'$, then, similar to (2.11), the test loss can be written compactly as

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

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 $\ell^* = 25$.

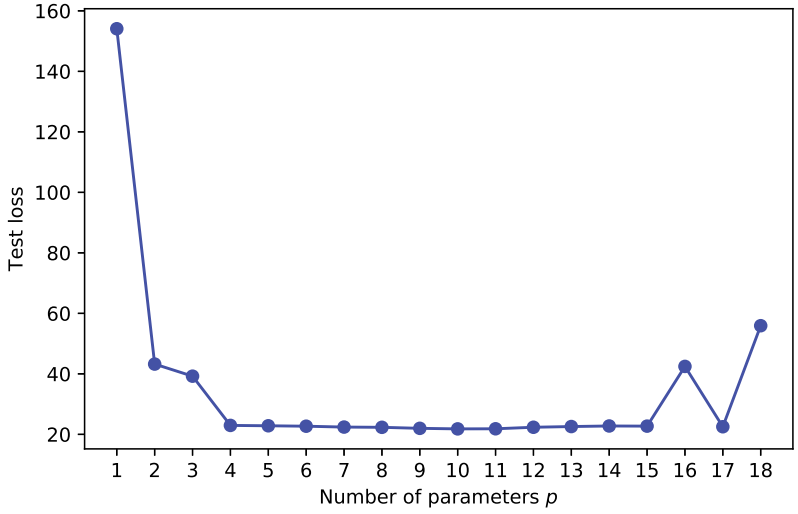


Figure 2.7: Test loss as function of the number of parameters p of the model.

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 $p = 1, \ldots, K = 18$ 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:

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

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.

☞ 439

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 $\ell(g_\tau^{\mathcal{G}}) = \mathbb{E}(Y - g_\tau^{\mathcal{G}}(\boldsymbol{X}))^2$; that is, the expected squared error[1] between the predicted value $g_\tau^{\mathcal{G}}(\boldsymbol{X})$ and the response Y. Recall that in this case the optimal prediction function is given by $g^*(\boldsymbol{x}) = \mathbb{E}[Y \mid \boldsymbol{X} = \boldsymbol{x}]$. The decomposition (2.16) can now be interpreted as follows.

1. The first component, $\ell^* = \mathbb{E}(Y - g^*(\boldsymbol{X}))^2$, 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 $\mathbb{E}(g^{\mathcal{G}}(\boldsymbol{X}) - g^*(\boldsymbol{X}))^2$. 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, $g(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{\beta}$ for some vector $\boldsymbol{\beta}$. In this case we can write (see Exercise 3) the statistical error as $\ell(g_\tau^{\mathcal{G}}) - \ell(g^{\mathcal{G}}) = \mathbb{E}(g_\tau^{\mathcal{G}}(\boldsymbol{X}) - g^{\mathcal{G}}(\boldsymbol{X}))^2$.

[1] Colloquially called *mean squared error*.

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

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

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 $\boldsymbol{x} = [1, u, u^2, \ldots, u^{p-1}]^\top$, and $g^*(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{\beta}^*$. Conditional on $\boldsymbol{X} = \boldsymbol{x}$ we have that $Y = g^*(\boldsymbol{x}) + \varepsilon(\boldsymbol{x})$, with $\varepsilon(\boldsymbol{x}) \sim \mathcal{N}(0, \ell^*)$, where $\ell^* = \mathbb{E}(Y - g^*(\boldsymbol{X}))^2 = 25$ 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

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

and so $g(\boldsymbol{X})$ is distributed as $[1, U, \ldots, U^{p-1}]\boldsymbol{\beta}$, where $U \sim \mathcal{U}(0,1)$. Similarly, $g^*(\boldsymbol{X})$ is distributed as $[1, U, U^2, U^3]\boldsymbol{\beta}^*$. It follows that an expression for the approximation error is: $\int_0^1 \left([1, u, \ldots, u^{p-1}]\boldsymbol{\beta} - [1, u, u^2, u^3]\boldsymbol{\beta}^*\right)^2 \mathrm{d}u$. To minimize this error, we set the gradient with respect to $\boldsymbol{\beta}$ to zero and obtain the p linear equations ☞ 397

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

Let

$$\mathbf{H}_p = \int_0^1 [1, u, \ldots, u^{p-1}][1, u, \ldots, u^{p-1}]^\top \mathrm{d}u$$

be the $p \times p$ *Hilbert matrix*, which has (i, j)-th entry given by $\int_0^1 u^{i+j-2}\,\mathrm{d}u = 1/(i + j - 1)$. Then, the above system of linear equations can be written as $\mathbf{H}_p\boldsymbol{\beta} = \widetilde{\mathbf{H}}\boldsymbol{\beta}^*$, where $\widetilde{\mathbf{H}}$ is the $p \times 4$ upper left sub-block of $\mathbf{H}_{\widetilde{p}}$ and $\widetilde{p} = \max\{p, 4\}$. The solution, which we denote by $\boldsymbol{\beta}_p$, is: HILBERT MATRIX

$$\boldsymbol{\beta}_p = \begin{cases} \frac{65}{6}, & p = 1,\\ [-\frac{20}{3}, 35]^\top, & p = 2,\\ [-\frac{5}{2}, 10, 25]^\top, & p = 3,\\ [10, -140, 400, -250, 0, \ldots, 0]^\top, & p \geqslant 4. \end{cases} \tag{2.18}$$

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

$$\int_0^1 \left([1, u, \ldots, u^{p-1}]\boldsymbol{\beta}_p - [1, u, u^2, u^3]\boldsymbol{\beta}^*\right)^2 \mathrm{d}u = \begin{cases} \frac{32225}{252} \approx 127.9, & p = 1,\\ \frac{1625}{63} \approx 25.8, & p = 2,\\ \frac{625}{28} \approx 22.3, & p = 3,\\ 0, & p \geqslant 4. \end{cases} \tag{2.19}$$

Notice how the approximation error becomes smaller as p increases. In this particular example the approximation error is in fact zero for $p \geqslant 4$. 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 $g_\tau(\boldsymbol{x}) = \boldsymbol{x}^\top \widehat{\boldsymbol{\beta}}$, the statistical error can be written as

$$\int_0^1 \left([1, \ldots, u^{p-1}](\widehat{\boldsymbol{\beta}} - \boldsymbol{\beta}_p)\right)^2 \mathrm{d}u = (\widehat{\boldsymbol{\beta}} - \boldsymbol{\beta}_p)^\top \mathbf{H}_p (\widehat{\boldsymbol{\beta}} - \boldsymbol{\beta}_p). \tag{2.20}$$

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.

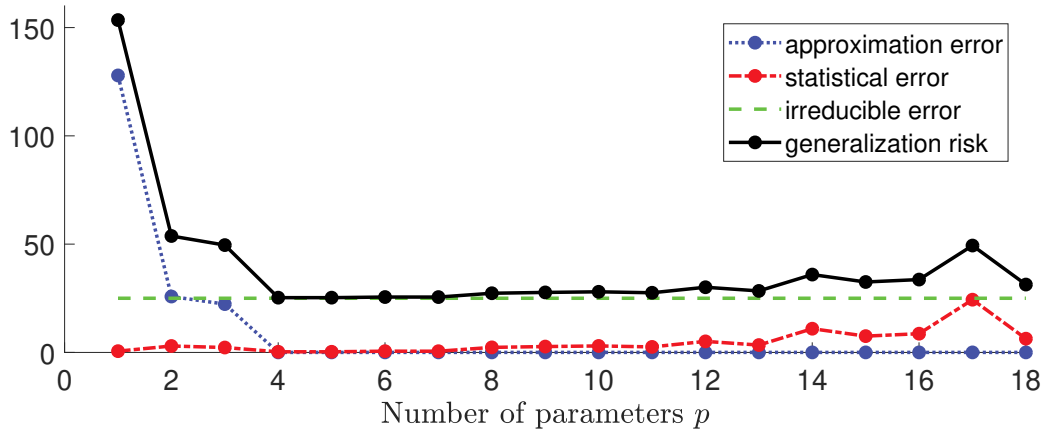


Figure 2.8: The generalization risk for a particular training set is the sum of the irreducible error, the approximation error, and the statistical error. The approximation error decreases to zero as p increases, whereas the statistical error has a tendency to increase after $p = 4$.

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

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

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

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

If we view the learner $g_{\mathcal{T}}^{\mathcal{G}}(\boldsymbol{x})$ as a function of a random training set, then the *pointwise squared bias* term is a measure for how close $g_{\mathcal{T}}^{\mathcal{G}}(\boldsymbol{x})$ is on average to the true $g^*(\boldsymbol{x})$, whereas the *pointwise variance* term measures the deviation of $g_{\mathcal{T}}^{\mathcal{G}}(\boldsymbol{x})$ from its expected value $\mathbb{E}g_{\mathcal{T}}^{\mathcal{G}}(\boldsymbol{x})$. 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

BIAS–VARIANCE TRADEOFF

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

$$\mathbb{E}\,\ell(g_{\mathcal{T}}^{\mathcal{G}}) = \ell^* + \underbrace{\mathbb{E}\,(\mathbb{E}[g_{\mathcal{T}}^{\mathcal{G}}(\boldsymbol{X}) \,|\, \boldsymbol{X}] - g^*(\boldsymbol{X}))^2}_{\text{expected squared bias}} + \underbrace{\mathbb{E}[\mathbb{V}\mathrm{ar}[g_{\mathcal{T}}^{\mathcal{G}}(\boldsymbol{X}) \,|\, \boldsymbol{X}]]}_{\text{expected variance}} . \tag{2.22}$$

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 $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ (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

$$\ell_{\text{in}}(g_\tau) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}\,\mathrm{Loss}(Y_i', g_\tau(\boldsymbol{x}_i)), \tag{2.23}$$

where each response Y_i' is drawn from $f(y \,|\, \boldsymbol{x}_i)$, 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 $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, is to draw new response values $Y_i' \sim f(y \,|\, \boldsymbol{x}_i)$, $i = 1, \ldots, n$, that are independent from the responses $y_1, \ldots, y_n$ in the training data, and then estimate the in-sample risk of g_τ via

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

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

$$\mathbb{E}[\text{op}_\mathcal{T} \,|\, \boldsymbol{X}_1 = \boldsymbol{x}_1, \ldots, \boldsymbol{X}_n = \boldsymbol{x}_n] =: \mathbb{E}_\mathbf{X} \, \text{op}_\mathcal{T},$$

where the expectation is taken over a random training set $\mathcal{T}$, conditional on $\boldsymbol{X}_i = \boldsymbol{x}_i$, $i = 1, \ldots, n$. For ease of notation, we have abbreviated the expected optimism to $\mathbb{E}_\mathbf{X} \, \text{op}_\mathcal{T}$, where $\mathbb{E}_\mathbf{X}$ denotes the expectation operator conditional on $\boldsymbol{X}_i = \boldsymbol{x}_i, i = 1, \ldots, n$. 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

$$\mathbb{E}_\mathbf{X} \, \text{op}_\mathcal{T} = \frac{2}{n} \sum_{i=1}^{n} \mathbb{C}\text{ov}_\mathbf{X}(g_\mathcal{T}(\boldsymbol{x}_i), Y_i). \tag{2.24}$$

Proof: In what follows, all expectations are taken conditional on $\boldsymbol{X}_1 = \boldsymbol{x}_1, \ldots, \boldsymbol{X}_n = \boldsymbol{x}_n$. Let Y_i be the response for $\boldsymbol{x}_i$ and let $\widehat{Y}_i = g_\mathcal{T}(\boldsymbol{x}_i)$ be the predicted value. Note that the latter depends on $Y_1, \ldots, Y_n$. Also, let Y_i' be an independent copy of Y_i for the *same* $\boldsymbol{x}_i$, 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

$$\begin{aligned}
\mathbb{E}_\mathbf{X} \, \text{op}_\mathcal{T} &= \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}_\mathbf{X} \left[(Y_i' - \widehat{Y}_i)^2 - (Y_i - \widehat{Y}_i)^2 \right] = \frac{2}{n} \sum_{i=1}^{n} \mathbb{E}_\mathbf{X} \left[(Y_i - Y_i') \widehat{Y}_i \right] \\
&= \frac{2}{n} \sum_{i=1}^{n} \left(\mathbb{E}_\mathbf{X}[Y_i \widehat{Y}_i] - \mathbb{E}_\mathbf{X} Y_i \, \mathbb{E}_\mathbf{X} \widehat{Y}_i \right) = \frac{2}{n} \sum_{i=1}^{n} \mathbb{C}\text{ov}_\mathbf{X}(\widehat{Y}_i, Y_i).
\end{aligned}$$

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

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 $g_\mathcal{T}$ is statistically independent from the responses $Y_1, \ldots, Y_n$.

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

$$\frac{2}{n}\sum_{i=1}^{n} \mathbb{C}\mathrm{ov}_{\mathbf{X}}\left(\boldsymbol{x}_i^\top \widehat{\boldsymbol{\beta}}, Y_i\right) = \frac{2}{n}\mathrm{tr}\left(\mathbb{C}\mathrm{ov}_{\mathbf{X}}\left(\mathbf{X}\widehat{\boldsymbol{\beta}}, \boldsymbol{Y}\right)\right) = \frac{2}{n}\mathrm{tr}\left(\mathbb{C}\mathrm{ov}_{\mathbf{X}}\left(\mathbf{X}\mathbf{X}^+\boldsymbol{Y}, \boldsymbol{Y}\right)\right)$$
$$= \frac{2\mathrm{tr}\left(\mathbf{X}\mathbf{X}^+\mathbb{C}\mathrm{ov}_{\mathbf{X}}\left(\boldsymbol{Y}, \boldsymbol{Y}\right)\right)}{n} = \frac{2\ell^*\mathrm{tr}\left(\mathbf{X}\mathbf{X}^+\right)}{n} = \frac{2\ell^* p}{n}.$$

In the last equation we used the cyclic property of the trace (Theorem A.1): $\mathrm{tr}(\mathbf{X}\mathbf{X}^+) = \mathrm{tr}(\mathbf{X}^+\mathbf{X}) = \mathrm{tr}(\mathbf{I}_p)$, assuming that $\mathrm{rank}(\mathbf{X}) = p$. Therefore, an estimate for the in-sample risk (2.23) is: ☞ 357

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

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.

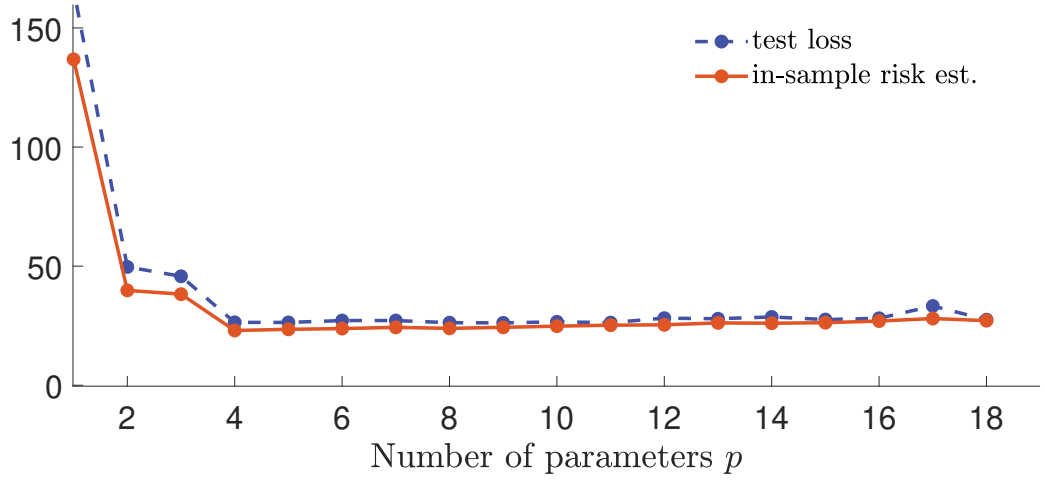


Figure 2.9: In-sample risk estimate $\widehat{\ell}_{\mathrm{in}}(g_\tau)$ as a function of the number of parameters p of the model. The test loss is superimposed as a blue dashed curve.

■

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*. ☞ 24

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.

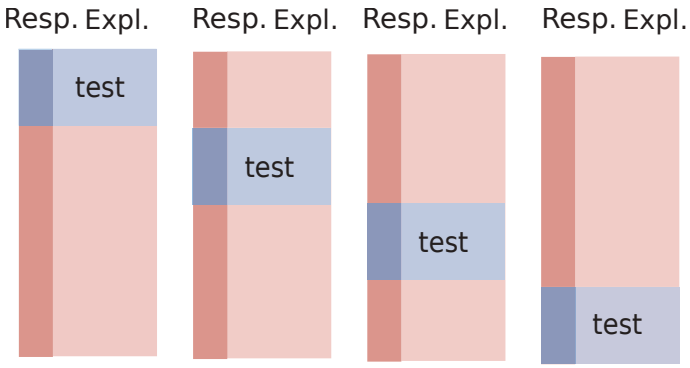


Figure 2.10: An illustration of four-fold cross-validation, representing four copies of the same data set. The data in each copy is partitioned into a training set (pink) and a test set (blue). The darker columns represent the response variable and the lighter ones the explanatory variables.

FOLDS

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

Let $\ell_{\mathcal{T}_{-k}}$ be the test loss when using C_k as test data and all remaining data, denoted $\mathcal{T}_{-k}$, as training data. Each $\ell_{\mathcal{T}_{-k}}$ 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:

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

where the function $\kappa : \{1, \ldots, n\} \mapsto \{1, \ldots, K\}$ 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 $\ell(g_\tau)$ for the particular training set τ.

■ **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()
```

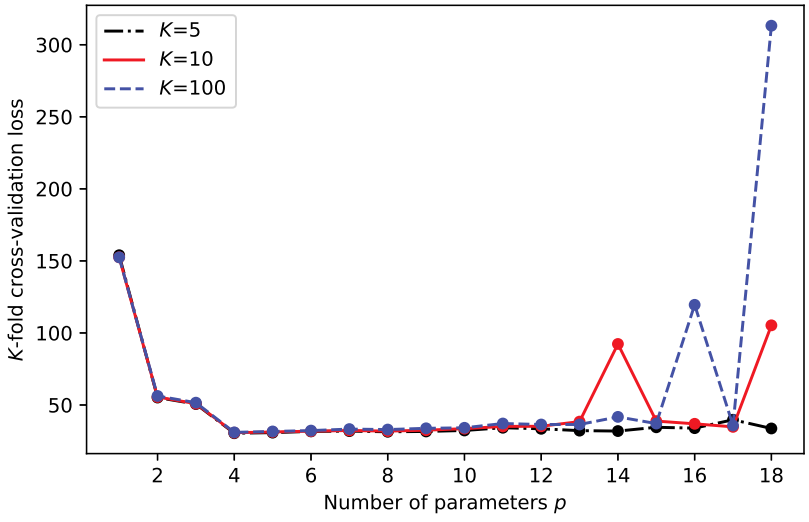


Figure 2.11: K-fold cross-validation for the polynomial regression example.

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. ■

LEAVE-ONE-OUT CROSS-VALIDATION

☞ 174

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 $\boldsymbol{x} = [x_1, \ldots, x_p]^\top$, a very general model is to assume that $\boldsymbol{x}$ is the outcome of a random vector $\boldsymbol{X} = [X_1, \ldots, X_p]^\top$ with some unknown pdf f. The model can then be refined by assuming a specific form of f.

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

☞ 429

$$\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n \overset{\text{iid}}{\sim} f \quad \text{or} \quad \boldsymbol{X}_1, \ldots, \boldsymbol{X}_n \overset{\text{iid}}{\sim} \mathsf{Dist},$$

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 $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$ is simply the *product* of the marginal ones; see Theorem C.1. Specifically,

☞ 429

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

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, $g(\boldsymbol{x})$ is of the form $g(\boldsymbol{x} \,|\, \boldsymbol{\beta})$ which is known up to some parameter vector $\boldsymbol{\beta}$. Examples for the one-dimensional case ($p = 1$) include the $\mathcal{N}(\mu, \sigma^2)$, $\mathsf{Bin}(n, p)$, and $\mathsf{Exp}(\lambda)$ distributions. See Tables C.1 and C.2 for other common sampling distributions.

☞ 425

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

☞ 11

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

$$f_{\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) = f_{\boldsymbol{X}_{\pi_1}, \ldots, \boldsymbol{X}_{\pi_n}}(\boldsymbol{x}_{\pi_1}, \ldots, \boldsymbol{x}_{\pi_n}) \tag{2.26}$$

for any permutation $\pi_1, \ldots, \pi_n$ of the integers $1, \ldots, n$. We say that the infinite sequence $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots$ 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 $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$ 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.

EXCHANGEABLE

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.

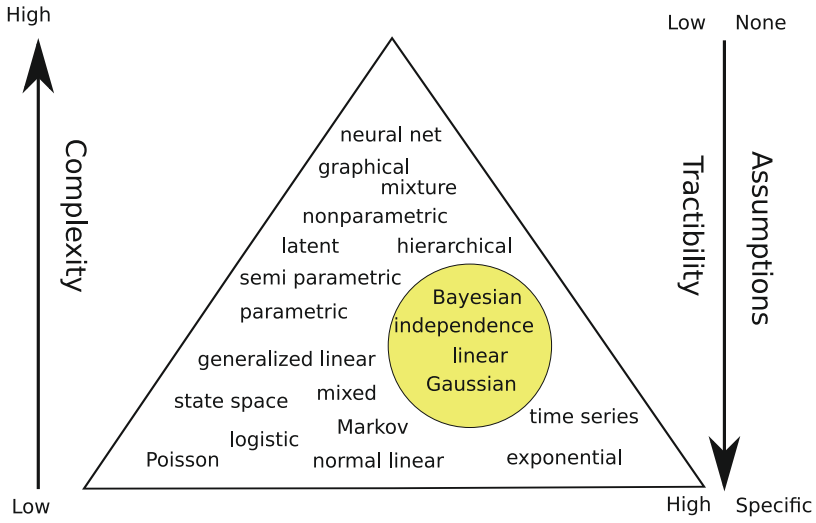


Figure 2.12: Illustration of the modeling dilemma. Complex models are more generally applicable, but may be difficult to analyze. Simple models may be highly tractable, but may not describe the data accurately. The triangular shape signifies that there are a great many specific models but not so many 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 $\tau = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ that is viewed as the outcome of n iid random variables $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$ 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 $\mathcal{G}_p := \{g(\cdot \,|\, \boldsymbol{\theta}), \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$, where $\boldsymbol{\theta}$ is a parameter in some subset $\boldsymbol{\Theta}$ 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 $g(\boldsymbol{x})$ 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 $\boldsymbol{x}$ we measure the discrepancy between the true model $f(\boldsymbol{x})$ and the hypothesized model $g(\boldsymbol{x} \,|\, \boldsymbol{\theta})$ using the loss function

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

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

$$\ell(g) = \mathbb{E} \ln \frac{f(\boldsymbol{X})}{g(\boldsymbol{X} \mid \boldsymbol{\theta})} = \int f(\boldsymbol{x}) \ln \frac{f(\boldsymbol{x})}{g(\boldsymbol{x} \mid \boldsymbol{\theta})} \, \mathrm{d}\boldsymbol{x}. \tag{2.27}$$

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 $g(\cdot \mid \boldsymbol{\theta})$. Note that the KL divergence is not symmetric in f and $g(\cdot \mid \boldsymbol{\theta})$. Moreover, it is always greater than or equal to 0 (see Exercise 15) and equal to 0 when $f = g(\cdot \mid \boldsymbol{\theta})$.

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

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

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

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

giving:

$$\widehat{\boldsymbol{\theta}}_n := \operatorname*{argmax}_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} \ln g(\boldsymbol{x}_i \mid \boldsymbol{\theta}). \tag{2.28}$$

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

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

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

MAXIMUM LIKELIHOOD ESTIMATE

☞ 456

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

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

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

SCORE

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

[2] Sometimes 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

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

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

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

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 $\boldsymbol{X} = \boldsymbol{x}$, the error term ε does not depend on $\boldsymbol{x}$, that is, $\mathbb{E}\,\varepsilon = 0$ and $\mathbb{V}\text{ar}\,\varepsilon = \sigma^2$, 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 $\boldsymbol{x} = [x_1, \ldots, x_p]^\top$ via the linear relationship

LINEAR MODEL

$$Y = \boldsymbol{x}^\top \boldsymbol{\beta} + \varepsilon, \tag{2.29}$$

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

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

$$\boldsymbol{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}, \tag{2.30}$$

where $\boldsymbol{\varepsilon} = [\varepsilon_1, \ldots, \varepsilon_n]^\top$ is a vector of iid copies of ε and $\mathbf{X}$ is the so-called *model matrix*, with rows $\boldsymbol{x}_1^\top, \ldots, \boldsymbol{x}_n^\top$. 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 $x_1, \ldots, x_n$ (forming, e.g., a column in a spreadsheet or data frame) is that they are the outcomes of iid normal random variables

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

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 $Z_1, \ldots, Z_n \overset{\text{iid}}{\sim} \mathcal{N}(0, 1)$. The pdf of $\boldsymbol{Z} = [Z_1, \ldots, Z_n]^\top$ (that is, the joint pdf of $Z_1, \ldots, Z_n$) is given by

☞ 434

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

We write $\boldsymbol{Z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_n)$ and say that $\boldsymbol{Z}$ has a standard normal distribution in $\mathbb{R}^n$. Let

$$\boldsymbol{X} = \boldsymbol{\mu} + \mathbf{B}\,\boldsymbol{Z} \tag{2.32}$$

for some $m \times n$ matrix $\mathbf{B}$ and m-dimensional vector $\boldsymbol{\mu}$. Then $\boldsymbol{X}$ has expectation vector $\boldsymbol{\mu}$ and covariance matrix $\Sigma = \mathbf{B}\mathbf{B}^\top$; see (C.20) and (C.21). This leads to the following definition. ☞ 432

Definition 2.2: Multivariate Normal Distribution

An m-dimensional random vector $\boldsymbol{X}$ that can be written in the form (2.32) for some m-dimensional vector $\boldsymbol{\mu}$ and $m \times n$ matrix $\mathbf{B}$, with $\boldsymbol{Z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_n)$, is said to have a *multivariate normal* or *multivariate Gaussian* distribution with mean vector $\boldsymbol{\mu}$ and covariance matrix $\Sigma = \mathbf{B}\mathbf{B}^\top$. We write $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$.

MULTIVARIATE NORMAL

The m-dimensional density of a multivariate normal distribution has a very similar form to the density of the one-dimensional normal distribution and is given in the next theorem. We leave the proof as an exercise; see Exercise 5. ☞ 59

Theorem 2.3: Density of a Multivariate Random Vector

Let $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$, where the $m \times m$ covariance matrix Σ is invertible. Then $\boldsymbol{X}$ has pdf

$$f_{\boldsymbol{X}}(\boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^m\,|\Sigma|}}\, \mathrm{e}^{-\frac{1}{2}\,(\boldsymbol{x}-\boldsymbol{\mu})^\top \Sigma^{-1} (\boldsymbol{x}-\boldsymbol{\mu})}, \quad \boldsymbol{x} \in \mathbb{R}^m. \tag{2.33}$$

Figure 2.13 shows the pdfs of two bivariate (that is, two-dimensional) normal distributions. In both cases the mean vector is $\boldsymbol{\mu} = [0, 0]^\top$ and the variances (the diagonal elements of Σ) are 1. The correlation coefficients (or, equivalently here, the covariances) are respectively $\varrho = 0$ and $\varrho = 0.8$.

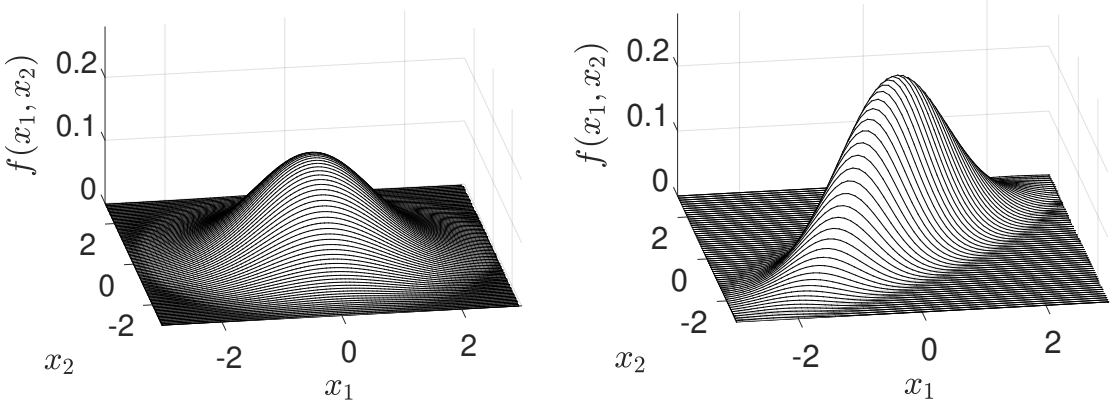


Figure 2.13: Pdfs of bivariate normal distributions with means zero, variances 1, and correlation coefficients 0 (left) and 0.8 (right).

The main reason why the multivariate normal distribution plays an important role in data science and machine learning is that it satisfies the following properties, the details and proofs of which can be found in Appendix C.7:

☞ 434

1. Affine combinations are normal.
2. Marginal distributions are normal.
3. Conditional distributions are normal.

2.8 Normal Linear Models

Normal linear models combine the simplicity of the linear model with the tractability of the Gaussian distribution. They are the principal model for traditional statistics, and include the classic linear regression and analysis of variance models.

Definition 2.3: Normal Linear Model

NORMAL LINEAR MODEL

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

$$Y = \boldsymbol{x}^\top \boldsymbol{\beta} + \varepsilon, \tag{2.34}$$

where $\varepsilon \sim \mathcal{N}(0, \sigma^2)$.

Thus, a normal linear model is a linear model (in the sense of Definition 2.1) with normal error terms. Similar to (2.30), the corresponding normal linear model for the whole training set $\{(\boldsymbol{x}_i, Y_i)\}$ has the form

$$\boldsymbol{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}, \tag{2.35}$$

where $\mathbf{X}$ is the model matrix comprised of rows $\boldsymbol{x}_1^\top, \ldots, \boldsymbol{x}_n^\top$ and $\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I}_n)$. Consequently, $\boldsymbol{Y}$ can be written as $\boldsymbol{Y} = \mathbf{X}\boldsymbol{\beta} + \sigma \boldsymbol{Z}$, where $\boldsymbol{Z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_n)$, so that $\boldsymbol{Y} \sim \mathcal{N}(\mathbf{X}\boldsymbol{\beta}, \sigma^2 \mathbf{I}_n)$. It follows from (2.33) that its joint density is given by

☞ 45

$$g(\boldsymbol{y} \,|\, \boldsymbol{\beta}, \sigma^2, \mathbf{X}) = (2\pi\sigma^2)^{-\frac{n}{2}} \, \mathrm{e}^{-\frac{1}{2\sigma^2}\|\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}\|^2}. \tag{2.36}$$

Estimation of the parameter $\boldsymbol{\beta}$ can be performed via the least-squares method, as discussed in Example 2.1. An estimate can also be obtained via the maximum likelihood method. This simply means finding the parameters σ^2 and $\boldsymbol{\beta}$ that maximize the likelihood of the outcome $\boldsymbol{y}$, given by the right-hand side of (2.36). It is clear that for every value of σ^2 the likelihood is maximal when $\|\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}\|^2$ is minimal. As a consequence, the maximum likelihood estimate for $\boldsymbol{\beta}$ is the same as the least-squares estimate (2.15). We leave it as an exercise (see Exercise 18) to show that the maximum likelihood estimate of σ^2 is equal to

☞ 63

$$\widehat{\sigma^2} = \frac{\|\boldsymbol{y} - \mathbf{X}\widehat{\boldsymbol{\beta}}\|^2}{n}, \tag{2.37}$$

where $\widehat{\boldsymbol{\beta}}$ is the maximum likelihood estimate (least squares estimate in this case) of $\boldsymbol{\beta}$.

2.9 Bayesian Learning

In Bayesian unsupervised learning, we seek to approximate the unknown joint density $f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ of the training data $\mathcal{T}_n = \{\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\}$ via a joint pdf of the form

$$\int \left(\prod_{i=1}^{n} g(\boldsymbol{x}_i \,|\, \boldsymbol{\theta}) \right) w(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}, \tag{2.38}$$

where $g(\cdot \,|\, \boldsymbol{\theta})$ belongs to a family of parametric densities $\mathcal{G}_p := \{g(\cdot \,|\, \boldsymbol{\theta}),\ \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$ (viewed as a family of pdfs conditional on a parameter $\boldsymbol{\theta}$ in some set $\boldsymbol{\Theta} \subset \mathbb{R}^p$) and $w(\boldsymbol{\theta})$ is a pdf that belongs to a (possibly different) family of densities $\mathcal{W}_p$. Note how the joint pdf (2.38) satisfies the permutational invariance (2.26) and can thus be useful as a model for training data which is part of an exchangeable sequence of random variables.

Following standard practice in a Bayesian context, instead of writing $f_X(x)$ and $f_{X|Y}(x \,|\, y)$ for the pdf of X and the conditional pdf of X given Y, one simply writes $f(x)$ and $f(x \,|\, y)$. If Y is a different random variable, its pdf (at y) is thus denoted by $f(y)$.

Thus, we will use the same symbol g for different (conditional) approximating probability densities and f for the different (conditional) true and unknown probability densities. Using Bayesian notation, we can write $g(\tau \,|\, \boldsymbol{\theta}) = \prod_{i=1}^{n} g(\boldsymbol{x}_i \,|\, \boldsymbol{\theta})$ and thus the approximating joint pdf (2.38) can then be written as $\int g(\tau \,|\, \boldsymbol{\theta}) \, w(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}$ and the true unknown joint pdf as $f(\tau) = f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$.

Once $\mathcal{G}_p$ and $\mathcal{W}_p$ are specified, selecting an approximating function $g(\boldsymbol{x})$ of the form

$$g(\boldsymbol{x}) = \int g(\boldsymbol{x} \,|\, \boldsymbol{\theta}) \, w(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}$$

is equivalent to selecting a suitable w from $\mathcal{W}_p$. Similar to (2.27), we can use the Kullback–Leibler risk to measure the discrepancy between the proposed approximation (2.38) and the true $f(\tau)$:

$$\ell(g) = \mathbb{E} \ln \frac{f(\mathcal{T})}{\int g(\mathcal{T} \,|\, \boldsymbol{\theta}) \, w(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}} = \int f(\tau) \ln \frac{f(\tau)}{\int g(\tau \,|\, \boldsymbol{\theta}) \, w(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}} \, \mathrm{d}\tau. \tag{2.39}$$

The main difference with (2.27) is that since the training data is not necessarily iid (it may be exchangeable, for example), the expectation must be with respect to the joint density of $\mathcal{T}$, not with respect to the marginal $f(\boldsymbol{x})$ (as in the iid case). ☞ 40

Minimizing the training loss is equivalent to maximizing the likelihood of the training data τ; that is, solving the optimization problem

$$\max_{w \in \mathcal{W}_p} \int g(\tau \,|\, \boldsymbol{\theta}) \, w(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta},$$

where the maximization is over an appropriate class $\mathcal{W}_p$ of density functions that is believed to result in the smallest KL risk.

Suppose that we have a rough guess, denoted $w_0(\boldsymbol{\theta})$, for the best $w \in \mathcal{W}_p$ that minimizes the Kullback–Leibler risk. We can always increase the resulting likelihood $L_0 := \int g(\tau \,|\, \boldsymbol{\theta})\, w_0(\boldsymbol{\theta})\, \mathrm{d}\boldsymbol{\theta}$ by instead using the density $w_1(\boldsymbol{\theta}) := w_0(\boldsymbol{\theta})\, g(\tau \,|\, \boldsymbol{\theta})/L_0$, giving a likelihood $L_1 := \int g(\tau \,|\, \boldsymbol{\theta})\, w_1(\boldsymbol{\theta})\, \mathrm{d}\boldsymbol{\theta}$. To see this, write L_0 and L_1 as expectations with respect to w_0. In particular, we can write

$$L_0 = \mathbb{E}_{w_0}\, g(\tau \,|\, \boldsymbol{\theta}) \quad \text{and} \quad L_1 = \mathbb{E}_{w_1}\, g(\tau \,|\, \boldsymbol{\theta}) = \mathbb{E}_{w_0} g^2(\tau \,|\, \boldsymbol{\theta})/L_0.$$

It follows that

$$L_1 - L_0 = \frac{1}{L_0}\mathbb{E}_{w_0}\left[g^2(\tau \,|\, \boldsymbol{\theta}) - L_0^2\right] = \frac{1}{L_0}\mathbb{V}\mathrm{ar}_{w_0}[g(\tau \,|\, \boldsymbol{\theta})] \geqslant 0. \tag{2.40}$$

We may thus expect to obtain better predictions using w_1 instead of w_0, because w_1 has taken into account the observed data τ and increased the likelihood of the model. In fact, if we iterate this process (see Exercise 20) and create a sequence of densities $w_1, w_2, \ldots$ such that $w_t(\boldsymbol{\theta}) \propto w_{t-1}(\boldsymbol{\theta})\, g(\tau \,|\, \boldsymbol{\theta})$, then $w_t(\boldsymbol{\theta})$ concentrates more and more of its probability mass at the maximum likelihood estimator $\widehat{\boldsymbol{\theta}}$ (see (2.28)) and in the limit equals a (degenerate) point-mass pdf at $\widehat{\boldsymbol{\theta}}$. In other words, in the limit we recover the maximum likelihood method: $g_\tau(\boldsymbol{x}) = g(\boldsymbol{x} \,|\, \widehat{\boldsymbol{\theta}})$. Thus, unless the class of densities $\mathcal{W}_p$ is restricted to be non-degenerate, maximizing the likelihood as much as possible leads to a degenerate choice for $w(\boldsymbol{\theta})$.

In many situations, the maximum likelihood estimate $g(\tau \,|\, \widehat{\boldsymbol{\theta}})$ is either not an appropriate approximation to $f(\tau)$ (see Example 2.9), or simply fails to exist (see Exercise 10 in Chapter 4). In such cases, given an initial non-degenerate guess $w_0(\boldsymbol{\theta}) = g(\boldsymbol{\theta})$, one can obtain a more appropriate and non-degenerate approximation to $f(\tau)$ by taking $w(\boldsymbol{\theta}) = w_1(\boldsymbol{\theta}) \propto g(\tau \,|\, \boldsymbol{\theta})\, g(\boldsymbol{\theta})$ in (2.38), giving the following Bayesian learner of $f(\boldsymbol{x})$:

☞ 161

$$g_\tau(\boldsymbol{x}) := \int g(\boldsymbol{x} \,|\, \boldsymbol{\theta}) \frac{g(\tau \,|\, \boldsymbol{\theta})\, g(\boldsymbol{\theta})}{\int g(\tau \,|\, \boldsymbol{\vartheta})\, g(\boldsymbol{\vartheta})\, \mathrm{d}\boldsymbol{\vartheta}}\, \mathrm{d}\boldsymbol{\theta}, \tag{2.41}$$

where $\int g(\tau \,|\, \boldsymbol{\vartheta})\, g(\boldsymbol{\vartheta})\, \mathrm{d}\boldsymbol{\vartheta} = g(\tau)$. Using Bayes' formula for probability densities,

☞ 428

$$g(\boldsymbol{\theta} \,|\, \tau) = \frac{g(\tau \,|\, \boldsymbol{\theta})\, g(\boldsymbol{\theta})}{g(\tau)}, \tag{2.42}$$

we can write $w_1(\boldsymbol{\theta}) = g(\boldsymbol{\theta} \,|\, \tau)$. With this notation, we have the following definitions.

Definition 2.4: Prior, Likelihood, and Posterior

Let τ and $\mathcal{G}_p := \{g(\cdot \,|\, \boldsymbol{\theta}), \boldsymbol{\theta} \in \boldsymbol{\Theta}\}$ be the training set and family of approximating functions.

PRIOR
- A pdf $g(\boldsymbol{\theta})$ that reflects our *a priori* beliefs about $\boldsymbol{\theta}$ is called the *prior* pdf.

LIKELIHOOD
- The conditional pdf $g(\tau \,|\, \boldsymbol{\theta})$ is called the *likelihood*.

POSTERIOR
- Inference about $\boldsymbol{\theta}$ is given by the *posterior* pdf $g(\boldsymbol{\theta} \,|\, \tau)$, which is proportional to the product of the prior and the likelihood:

$$g(\boldsymbol{\theta} \,|\, \tau) \propto g(\tau \,|\, \boldsymbol{\theta})\, g(\boldsymbol{\theta}).$$

■ **Remark 2.1 (Early Stopping)** Bayes iteration is an example of an "early stopping" heuristic for maximum likelihood optimization, where we exit after only one step. As observed above, if we keep iterating, we obtain the maximum likelihood estimate (MLE). In a sense the Bayes rule provides a regularization of the MLE. Regularization is discussed in more detail in Chapter 6; see also Example 2.9. The early stopping rule is also of benefit in regularization; see Exercise 20 in Chapter 6. ■

On the one hand, the initial guess $g(\boldsymbol{\theta})$ conveys the *a priori* (prior to training the Bayesian learner) information about the optimal density in $\mathcal{W}_p$ that minimizes the KL risk. Using this prior $g(\boldsymbol{\theta})$, the Bayesian approximation to $f(\boldsymbol{x})$ is the *prior predictive density*:

PRIOR PREDICTIVE DENSITY

$$g(\boldsymbol{x}) = \int g(\boldsymbol{x} \,|\, \boldsymbol{\theta})\, g(\boldsymbol{\theta})\, \mathrm{d}\boldsymbol{\theta}.$$

On the other hand, the posterior pdf conveys improved knowledge about this optimal density in $\mathcal{W}_p$ after training with τ. Using the posterior $g(\boldsymbol{\theta} \,|\, \tau)$, the Bayesian learner of $f(\boldsymbol{x})$ is the *posterior predictive density*:

POSTERIOR PREDICTIVE DENSITY

$$g_\tau(\boldsymbol{x}) = g(\boldsymbol{x} \,|\, \tau) = \int g(\boldsymbol{x} \,|\, \boldsymbol{\theta})\, g(\boldsymbol{\theta} \,|\, \tau)\, \mathrm{d}\boldsymbol{\theta},$$

where we have assumed that $g(\boldsymbol{x} \,|\, \boldsymbol{\theta}, \tau) = g(\boldsymbol{x} \,|\, \boldsymbol{\theta})$; that is, the likelihood depends on τ only through the parameter $\boldsymbol{\theta}$.

The choice of the prior is typically governed by two considerations:

1. the prior should be simple enough to facilitate the computation or simulation of the posterior pdf;
2. the prior should be general enough to model ignorance of the parameter of interest.

Priors that do not convey much knowledge of the parameter are said to be *uninformative*. The uniform or *flat* prior in Example 2.9 (to follow) is frequently used.

UNINFORMATIVE PRIOR

For the purpose of analytical and numerical computations, we can view $\boldsymbol{\theta}$ as a random vector with prior density $g(\boldsymbol{\theta})$, which after training is updated to the posterior density $g(\boldsymbol{\theta} \,|\, \tau)$.

The above thinking allows us to write $g(\boldsymbol{x} \,|\, \tau) \propto \int g(\boldsymbol{x} \,|\, \boldsymbol{\theta})\, g(\tau \,|\, \boldsymbol{\theta})\, g(\boldsymbol{\theta})\, \mathrm{d}\boldsymbol{\theta}$, for example, thus ignoring any constants that do not depend on the argument of the densities.

■ **Example 2.7 (Normal Model)** Suppose that the training data $\mathcal{T} = \{X_1, \ldots, X_n\}$ is modeled using the likelihood $g(x \,|\, \boldsymbol{\theta})$ that is the pdf of

$$X \,|\, \boldsymbol{\theta} \sim \mathcal{N}(\mu, \sigma^2),$$

where $\boldsymbol{\theta} := [\mu, \sigma^2]^\top$. Next, we need to specify the prior distribution of $\boldsymbol{\theta}$ to complete the model. We can specify prior distributions for μ and σ^2 separately and then take their product to obtain the prior for vector $\boldsymbol{\theta}$ (assuming independence). A possible prior distribution for μ is

$$\mu \sim \mathcal{N}(\nu, \phi^2). \tag{2.43}$$

HYPERPARAMETERS

It is typical to refer to any parameters of the prior density as *hyperparameters* of the Bayesian model. Instead of giving directly a prior for σ^2 (or σ), it turns out to be convenient to give the following prior distribution to $1/\sigma^2$:

$$\frac{1}{\sigma^2} \sim \mathsf{Gamma}(\alpha, \beta). \tag{2.44}$$

INVERSE GAMMA

☞ 63

The smaller α and β are, the less informative is the prior. Under this prior, σ^2 is said to have an *inverse gamma*[3] distribution. If $1/Z \sim \mathsf{Gamma}(\alpha, \beta)$, then the pdf of Z is proportional to $\exp(-\beta/z)/z^{\alpha+1}$ (Exercise 19). The Bayesian posterior is then given by:

$$\begin{aligned} g(\mu, \sigma^2 \,|\, \tau) &\propto g(\mu) \times g(\sigma^2) \times g(\tau \,|\, \mu, \sigma^2) \\ &\propto \exp\left\{-\frac{(\mu-\nu)^2}{2\phi^2}\right\} \times \frac{\exp\left\{-\beta/\sigma^2\right\}}{(\sigma^2)^{\alpha+1}} \times \frac{\exp\left\{-\sum_i (x_i-\mu)^2/(2\sigma^2)\right\}}{(\sigma^2)^{n/2}} \\ &\propto (\sigma^2)^{-n/2-\alpha-1} \exp\left\{-\frac{(\mu-\nu)^2}{2\phi^2} - \frac{\beta}{\sigma^2} - \frac{(\mu-\overline{x}_n)^2 + S_n^2}{2\sigma^2/n}\right\}, \end{aligned}$$

where $S_n^2 := \frac{1}{n}\sum_i x_i^2 - \overline{x}_n^2 = \frac{1}{n}\sum_i (x_i - \overline{x}_n)^2$ is the (scaled) sample variance. All inference about (μ, σ^2) is then represented by the posterior pdf. To facilitate computations it is helpful to find out if the posterior belongs to a recognizable family of distributions. For example, the conditional pdf of μ given σ^2 and τ is

$$g(\mu \,|\, \sigma^2, \tau) \propto \exp\left\{-\frac{(\mu-\nu)^2}{2\phi^2} - \frac{(\mu-\overline{x}_n)^2}{2\sigma^2/n}\right\},$$

which after simplification can be recognized as the pdf of

$$(\mu \,|\, \sigma^2, \tau) \sim \mathcal{N}\left(\gamma_n \overline{x}_n + (1-\gamma_n)\nu,\ \gamma_n \sigma^2/n\right), \tag{2.45}$$

where we have defined the weight parameter: $\gamma_n := \frac{n}{\sigma^2} \Big/ \left(\frac{1}{\phi^2} + \frac{n}{\sigma^2}\right)$. We can then see that the posterior mean $\mathbb{E}[\mu \,|\, \sigma^2, \tau] = \gamma_n \overline{x}_n + (1-\gamma_n)\nu$ is a weighted linear combination of the prior mean ν and the sample average $\overline{x}_n$. Further, as $n \to \infty$, the weight $\gamma_n \to 1$ and thus the posterior mean approaches the maximum likelihood estimate $\overline{x}_n$. ■

IMPROPER PRIOR

It is sometimes possible to use a prior $g(\boldsymbol{\theta})$ that is not a *bona fide* probability density, in the sense that $\int g(\boldsymbol{\theta})\, \mathrm{d}\boldsymbol{\theta} = \infty$, as long as the resulting posterior $g(\boldsymbol{\theta} \,|\, \tau) \propto g(\tau \,|\, \boldsymbol{\theta}) g(\boldsymbol{\theta})$ is a proper pdf. Such a prior is called an *improper prior*.

■ **Example 2.8 (Normal Model (cont.))** An example of an improper prior is obtained from (2.43) when we let $\phi \to \infty$ (the larger ϕ is, the more uninformative is the prior). Then, $g(\mu) \propto 1$ is a flat prior, but $\int g(\mu)\, \mathrm{d}\mu = \infty$, making it an improper prior. Nevertheless, the posterior is a proper density, and in particular the conditional posterior of $(\mu \,|\, \sigma^2, \tau)$ simplifies to

$$(\mu \,|\, \sigma^2, \tau) \sim \mathcal{N}\left(\overline{x}_n, \sigma^2/n\right),$$

[3] Reciprocal gamma distribution would have been a better name.

because the weight parameter γ_n goes to 1 as $\phi \to \infty$. The improper prior $g(\mu) \propto 1$ also allows us to simplify the posterior marginal for σ^2:

$$g(\sigma^2 \,|\, \tau) = \int g(\mu, \sigma^2 \,|\, \tau)\, \mathrm{d}\mu \propto (\sigma^2)^{-(n-1)/2-\alpha-1} \exp\left\{-\frac{\beta + nS_n^2/2}{\sigma^2}\right\},$$

which we recognize as the density corresponding to

$$\frac{1}{\sigma^2} \,\Big|\, \tau \sim \mathsf{Gamma}\left(\alpha + \frac{n-1}{2},\, \beta + \frac{n}{2} S_n^2\right).$$

In addition to $g(\mu) \propto 1$, we can also use an improper prior for σ^2. If we take the limit $\alpha \to 0$ and $\beta \to 0$ in (2.44), then we also obtain the improper prior $g(\sigma^2) \propto 1/\sigma^2$ (or equivalently $g(1/\sigma^2) \propto 1/\sigma^2$). In this case, the posterior marginal density for σ^2 implies that:

$$\frac{nS_n^2}{\sigma^2} \,\Big|\, \tau \sim \chi^2_{n-1}$$

and the posterior marginal density for μ implies that:

$$\frac{\mu - \bar{x}_n}{S_n/\sqrt{n-1}} \,\Big|\, \tau \sim \mathsf{t}_{n-1}. \tag{2.46}$$

In general, deriving a simple formula for the posterior density of $\boldsymbol{\theta}$ is either impossible or too tedious. Instead, the Monte Carlo methods in Chapter 3 can be used to simulate (approximately) from the posterior for the purposes of inference and prediction. ■

One way in which a distributional result such as (2.46) can be useful is in the construction of a 95% *credible interval* $\mathcal{I}$ for the parameter μ; that is, an interval $\mathcal{I}$ such that the probability $\mathbb{P}[\mu \in \mathcal{I} \,|\, \tau]$ is equal to 0.95. For example, the symmetric 95% credible interval is

CREDIBLE INTERVAL

$$\mathcal{I} = \left[\bar{x}_n - \frac{S_n}{\sqrt{n-1}}\gamma,\ \bar{x}_n + \frac{S_n}{\sqrt{n-1}}\gamma\right],$$

where γ is the 0.975-quantile of the t_{n-1} distribution. Note that the credible interval is not a random object and that the parameter μ is interpreted as a random variable with a distribution. This is unlike the case of classical confidence intervals, where the parameter is nonrandom, but the interval is (the outcome of) a random object.

☞ 457

As a generalization of the 95% Bayesian credible interval we can define a $1-\alpha$ *credible region*, which is any set $\mathcal{R}$ satisfying

CREDIBLE REGION

$$\mathbb{P}[\boldsymbol{\theta} \in \mathcal{R} \,|\, \tau] = \int_{\boldsymbol{\theta} \in \mathcal{R}} g(\boldsymbol{\theta} \,|\, \tau)\, \mathrm{d}\boldsymbol{\theta} \geqslant 1 - \alpha. \tag{2.47}$$

■ **Example 2.9 (Bayesian Regularization of Maximum Likelihood)** Consider modeling the number of deaths during birth in a maternity ward. Suppose that the hospital data consists of $\tau = \{x_1, \ldots, x_n\}$, with $x_i = 1$ if the i-th baby has died during birth and $x_i = 0$ otherwise, for $i = 1, \ldots, n$. A possible Bayesian model for the data is $\theta \sim \mathcal{U}(0,1)$ (uniform prior) with $(X_1, \ldots, X_n \,|\, \theta) \overset{\text{iid}}{\sim} \mathsf{Ber}(\theta)$. The likelihood is therefore

$$g(\tau \,|\, \theta) = \prod_{i=1}^{n} \theta^{x_i} (1-\theta)^{1-x_i} = \theta^s \, (1-\theta)^{n-s},$$

where $s = x_1 + \cdots + x_n$ is the total number of deaths. Since $g(\theta) = 1$, the posterior pdf is

$$g(\theta \,|\, \tau) \propto \theta^s \, (1-\theta)^{n-s}, \quad \theta \in [0,1],$$

which is the pdf of the $\mathsf{Beta}(s+1, n-s+1)$ distribution. The normalization constant is $(n+1)\binom{n}{s}$. The posterior pdf is shown in Figure 2.14 for $(s,n) = (0,100)$. It is not difficult

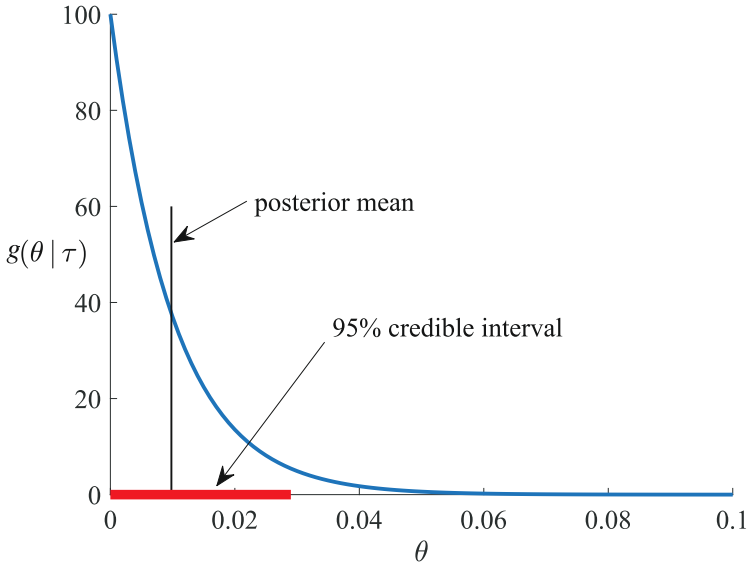


Figure 2.14: Posterior pdf for θ, with $n = 100$ and $s = 0$.

MAXIMUM A POSTERIORI

to see that the *maximum a posteriori* (MAP) estimate of θ (the mode or maximizer of the posterior density) is

$$\underset{\theta}{\text{argmax}} \, g(\theta \,|\, \tau) = \frac{s}{n},$$

which agrees with the maximum likelihood estimate. Figure 2.14 also shows that the left one-sided 95% credible interval for θ is $[0, 0.0292]$, where 0.0292 is the 0.95 quantile (rounded) of the $\mathsf{Beta}(1, 101)$ distribution.

Observe that when $(s,n) = (0,100)$ the maximum likelihood estimate $\widehat{\theta} = 0$ infers that deaths at birth are not possible. We know that this inference is wrong — the probability of death can never be zero, it is simply (and fortunately) too small to be inferred accurately from a sample size of $n = 100$. In contrast to the maximum likelihood estimate, the posterior mean $\mathbb{E}[\theta \,|\, \tau] = (s+1)/(n+2)$ is not zero for $(s,n) = (0,100)$ and provides the more reasonable point estimate of 0.0098 for the probability of death.

In addition, while computing a Bayesian credible interval poses no conceptual difficulties, it is not simple to derive a confidence interval for the maximum likelihood estimate of $\widehat{\theta}$, because the likelihood as a function of θ is not differentiable at $\theta = 0$. As a result of this lack of smoothness, the usual confidence intervals based on the normal approximation cannot be used. ■

We now return to the unsupervised learning setting of Section 2.6, but consider this from a Bayesian perspective. Recall from (2.39) that the Kullback–Leibler risk for an approximating function g is

$$\ell(g) = \int f(\tau'_n)[\ln f(\tau'_n) - \ln g(\tau'_n)]\,\mathrm{d}\tau'_n,$$

where τ' denotes the test data. Since $\int f(\tau'_n) \ln f(\tau'_n)\,\mathrm{d}\tau'_n$ plays no role in minimizing the risk, we consider instead the *cross-entropy risk*, defined as ☞ 122

$$\ell(g) = -\int f(\tau'_n) \ln g(\tau'_n)\,\mathrm{d}\tau'_n.$$

Note that the smallest possible cross-entropy risk is $\ell_n^* = -\int f(\tau'_n) \ln f(\tau'_n)\,\mathrm{d}\tau'_n$. The expected generalization risk of the Bayesian learner can then be decomposed as

$$\mathbb{E}\,\ell(g_{\mathcal{T}_n}) = \ell_n^* + \underbrace{\int f(\tau'_n) \ln \frac{f(\tau'_n)}{\mathbb{E}\,g(\tau'_n \,|\, \mathcal{T}_n)}\,\mathrm{d}\tau'_n}_{\text{“bias” component}} + \underbrace{\mathbb{E} \int f(\tau'_n) \ln \frac{\mathbb{E}\,g(\tau'_n \,|\, \mathcal{T}_n)}{g(\tau'_n \,|\, \mathcal{T}_n)}\,\mathrm{d}\tau'_n}_{\text{“variance” component}},$$

where $g_{\mathcal{T}_n}(\tau') = g(\tau' \,|\, \mathcal{T}_n) = \int g(\tau' \,|\, \boldsymbol{\theta})\, g(\boldsymbol{\theta} \,|\, \mathcal{T}_n)\,\mathrm{d}\boldsymbol{\theta}$ is the posterior predictive density after observing $\mathcal{T}_n$.

Assuming that the sets $\mathcal{T}_n$ and $\mathcal{T}'_n$ are comprised of $2n$ iid random variables with density f, we can show (Exercise 23) that the expected generalization risk simplifies to

$$\mathbb{E}\,\ell(g_{\mathcal{T}_n}) = \mathbb{E} \ln g(\mathcal{T}_n) - \mathbb{E} \ln g(\mathcal{T}_{2n}), \tag{2.48}$$

where $g(\tau_n)$ and $g(\tau_{2n})$ are the prior predictive densities of τ_n and τ_{2n}, respectively.

Let $\overline{\boldsymbol{\theta}}_n = \operatorname{argmax}_{\boldsymbol{\theta}} g(\boldsymbol{\theta} \,|\, \mathcal{T}_n)$ be the MAP estimator of $\boldsymbol{\theta}^* := \operatorname{argmax}_{\boldsymbol{\theta}} \mathbb{E} \ln g(\boldsymbol{X} \,|\, \boldsymbol{\theta})$. Assuming that $\overline{\boldsymbol{\theta}}_n$ converges to $\boldsymbol{\theta}^*$ (with probability one) and $\frac{1}{n}\mathbb{E} \ln g(\mathcal{T}_n \,|\, \overline{\boldsymbol{\theta}}_n) = \mathbb{E} \ln g(\boldsymbol{X} \,|\, \boldsymbol{\theta}^*) + O(1/n)$, we can use the following large-sample approximation of the expected generalization risk.

Theorem 2.4: Approximating the Bayesian Cross-Entropy Risk

For $n \to \infty$, the expected cross-entropy generalization risk satisfies:

$$\mathbb{E}\ell(g_{\mathcal{T}_n}) \simeq -\mathbb{E} \ln g(\mathcal{T}_n) - \frac{p}{2} \ln n, \tag{2.49}$$

where (with p the dimension of the parameter vector $\boldsymbol{\theta}$ and $\overline{\boldsymbol{\theta}}_n$ the MAP estimator):

$$\mathbb{E} \ln g(\mathcal{T}_n) \simeq \mathbb{E} \ln g(\mathcal{T}_n \,|\, \overline{\boldsymbol{\theta}}_n) - \frac{p}{2} \ln n. \tag{2.50}$$

☞ 450

Proof: To show (2.50), we apply Theorem C.21 to $\ln \int \mathrm{e}^{-nr_n(\boldsymbol{\theta})} g(\boldsymbol{\theta})\,\mathrm{d}\boldsymbol{\theta}$, where

$$r_n(\boldsymbol{\theta}) := -\frac{1}{n} \ln g(\mathcal{T}_n \,|\, \boldsymbol{\theta}) = -\frac{1}{n} \sum_{i=1}^{n} \ln g(\boldsymbol{X}_i \,|\, \boldsymbol{\theta}) \xrightarrow{\text{a.s.}} -\mathbb{E} \ln g(\boldsymbol{X} \,|\, \boldsymbol{\theta}) =: r(\boldsymbol{\theta}) < \infty.$$

This gives (with probability one)

$$\ln \int g(\mathcal{T}_n \,|\, \boldsymbol{\theta})\, g(\boldsymbol{\theta})\,\mathrm{d}\boldsymbol{\theta} \simeq -nr(\boldsymbol{\theta}^*) - \frac{p}{2} \ln(n).$$

Taking expectations on both sides and using $nr(\boldsymbol{\theta}^*) = n\mathbb{E}[r_n(\overline{\boldsymbol{\theta}}_n)] + O(1)$, we deduce (2.50). To demonstrate (2.49), we derive the asymptotic approximation of $\mathbb{E} \ln g(\mathcal{T}_{2n})$ by repeating the argument for (2.50), but replacing n with $2n$, where necessary. Thus, we obtain:

$$\mathbb{E} \ln g(\mathcal{T}_{2n}) \simeq -2nr(\boldsymbol{\theta}^*) - \frac{p}{2} \ln(2n).$$

Then, (2.49) follows from the identity (2.48). □

The results of Theorem 2.4 have two major implications for model selection and assessment. First, (2.49) suggests that $-\ln g(\mathcal{T}_n)$ can be used as a crude (leading-order) asymptotic approximation to the expected generalization risk for large n and fixed p. In this context, the prior predictive density $g(\mathcal{T}_n)$ is usually called the *model evidence* or *marginal likelihood* for the class $\mathcal{G}_p$. Since the integral $\int g(\mathcal{T}_n \,|\, \boldsymbol{\theta})\, g(\boldsymbol{\theta})\,\mathrm{d}\boldsymbol{\theta}$ is rarely available in closed form, the exact computation of the model evidence is typically not feasible and may require Monte Carlo estimation methods.

MODEL EVIDENCE

☞ 78

Second, when the model evidence is difficult to compute via Monte Carlo methods or otherwise, (2.50) suggests that we can use the following large-sample approximation:

$$-2\mathbb{E} \ln g(\mathcal{T}_n) \simeq -2 \ln g(\mathcal{T}_n \,|\, \overline{\boldsymbol{\theta}}_n) + p \ln(n). \tag{2.51}$$

The asymptotic approximation on the right-hand side of (2.51) is called the *Bayesian information criterion* (BIC). We prefer the class $\mathcal{G}_p$ with the smallest BIC. The BIC is typically used when the model evidence is difficult to compute and n is sufficiently larger than p. For a fixed p, and as n becomes larger and larger, the BIC becomes a more and more accurate estimator of $-2\mathbb{E} \ln g(\mathcal{T}_n)$. Note that the BIC approximation is valid even when the true density $f \notin \mathcal{G}_p$. The BIC provides an alternative to the *Akaike information criterion* (AIC) for model selection. However, while the BIC approximation does not assume that the true model f belongs to the parametric class under consideration, the AIC assumes that $f \in \mathcal{G}_p$. Thus, the AIC is merely a *heuristic* approximation based on the asymptotic approximations in Theorem 4.1.

BAYESIAN INFORMATION CRITERION

☞ 126

Although the above Bayesian theory has been presented in an unsupervised learning setting, it can be readily extended to the supervised case. We only need to relabel the training set $\mathcal{T}_n$. In particular, when (as is typical for regression models) the training responses $Y_1, \ldots, Y_n$ are considered as random variables but the corresponding feature vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ are viewed as being fixed, then $\mathcal{T}_n$ is the collection of random responses $\{Y_1, \ldots, Y_n\}$. Alternatively, we can simply identify $\mathcal{T}_n$ with the response vector $\boldsymbol{Y} = [Y_1, \ldots, Y_n]^\top$. We will adopt this notation in the next example.

■ **Example 2.10 (Polynomial Regression (cont.))** Consider Example 2.2 once again, but now in a Bayesian framework, where the prior knowledge on $(\sigma^2, \boldsymbol{\beta})$ is specified by $g(\sigma^2) = 1/\sigma^2$ and $\boldsymbol{\beta} \,|\, \sigma^2 \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{D})$, and $\mathbf{D}$ is a (matrix) hyperparameter. Let $\boldsymbol{\Sigma} := (\mathbf{X}^\top \mathbf{X} + \mathbf{D}^{-1})^{-1}$. Then the posterior can be written as:

$$
\begin{aligned}
g(\boldsymbol{\beta}, \sigma^2 \,|\, \boldsymbol{y}) &= \frac{\exp\left(-\frac{\|\boldsymbol{y}-\mathbf{X}\boldsymbol{\beta}\|^2}{2\sigma^2}\right)}{(2\pi\sigma^2)^{n/2}} \times \frac{\exp\left(-\frac{\boldsymbol{\beta}^\top \mathbf{D}^{-1}\boldsymbol{\beta}}{2\sigma^2}\right)}{(2\pi\sigma^2)^{p/2}\,|\mathbf{D}|^{1/2}} \times \frac{1}{\sigma^2} \Bigg| g(\boldsymbol{y}) \\
&= \frac{(\sigma^2)^{-(n+p)/2-1}}{(2\pi)^{(n+p)/2}\,|\mathbf{D}|^{1/2}} \exp\left(-\frac{\|\boldsymbol{\Sigma}^{-1/2}(\boldsymbol{\beta}-\overline{\boldsymbol{\beta}})\|^2}{2\sigma^2} - \frac{(n+p+2)\,\overline{\sigma}^2}{2\sigma^2}\right) \Bigg| g(\boldsymbol{y}),
\end{aligned}
$$

where $\overline{\boldsymbol{\beta}} := \boldsymbol{\Sigma}\mathbf{X}^\top \boldsymbol{y}$ and $\overline{\sigma}^2 := \boldsymbol{y}^\top(\mathbf{I} - \mathbf{X}\boldsymbol{\Sigma}\mathbf{X}^\top)\boldsymbol{y}/(n+p+2)$ are the MAP estimates of $\boldsymbol{\beta}$ and σ^2, and $g(\boldsymbol{y})$ is the model evidence for $\mathcal{G}_p$:

$$
\begin{aligned}
g(\boldsymbol{y}) &= \iint g(\boldsymbol{\beta}, \sigma^2, \boldsymbol{y})\, \mathrm{d}\boldsymbol{\beta}\, \mathrm{d}\sigma^2 \\
&= \frac{|\boldsymbol{\Sigma}|^{1/2}}{(2\pi)^{n/2}|\mathbf{D}|^{1/2}} \int_0^\infty \frac{\exp\left(-\frac{(n+p+2)\,\overline{\sigma}^2}{2\sigma^2}\right)}{(\sigma^2)^{n/2+1}}\, \mathrm{d}\sigma^2 \\
&= \frac{|\boldsymbol{\Sigma}|^{1/2}\Gamma(n/2)}{|\mathbf{D}|^{1/2}(\pi(n+p+2)\,\overline{\sigma}^2)^{n/2}}.
\end{aligned}
$$

Therefore, based on (2.49), we have

$$
2\mathbb{E}\ell(g_{\mathcal{T}_n}) \simeq -2\ln g(\boldsymbol{y}) = n\ln\left[\pi(n+p+2)\,\overline{\sigma}^2\right] - 2\ln\Gamma(n/2) + \ln|\mathbf{D}| - \ln|\boldsymbol{\Sigma}|.
$$

On the other hand, the minus of the log-likelihood of $\boldsymbol{Y}$ can be written as

$$
\begin{aligned}
-\ln g(\boldsymbol{y} \,|\, \boldsymbol{\beta}, \sigma^2) &= \frac{\|\boldsymbol{y} - \mathbf{X}\boldsymbol{\beta}\|^2}{2\sigma^2} + \frac{n}{2}\ln(2\pi\sigma^2) \\
&= \frac{\|\boldsymbol{\Sigma}^{-1/2}(\boldsymbol{\beta}-\overline{\boldsymbol{\beta}})\|^2}{2\sigma^2} + \frac{(n+p+2)\,\overline{\sigma}^2}{2\sigma^2} + \frac{n}{2}\ln(2\pi\sigma^2).
\end{aligned}
$$

Therefore, the BIC approximation (2.51) is

$$
-2\ln g(\boldsymbol{y} \,|\, \overline{\boldsymbol{\beta}}, \overline{\sigma}^2) + (p+1)\ln(n) = n[\ln(2\pi\,\overline{\sigma}^2) + 1] + (p+1)\ln(n) + (p+2), \tag{2.52}
$$

where the extra $\ln(n)$ term in $(p+1)\ln(n)$ is due to the inclusion of σ^2 in $\boldsymbol{\theta} = (\sigma^2, \boldsymbol{\beta})$. Figure 2.15 shows the model evidence and its BIC approximation, where we used a hyperparameter $\mathbf{D} = 10^4 \times \mathbf{I}_p$ for the prior density of $\boldsymbol{\beta}$. We can see that both approximations exhibit a pronounced minimum at $p = 4$, thus identifying the true polynomial regression model. Compare the overall qualitative shape of the cross-entropy risk estimate with the shape of the square-error risk estimate in Figure 2.11.

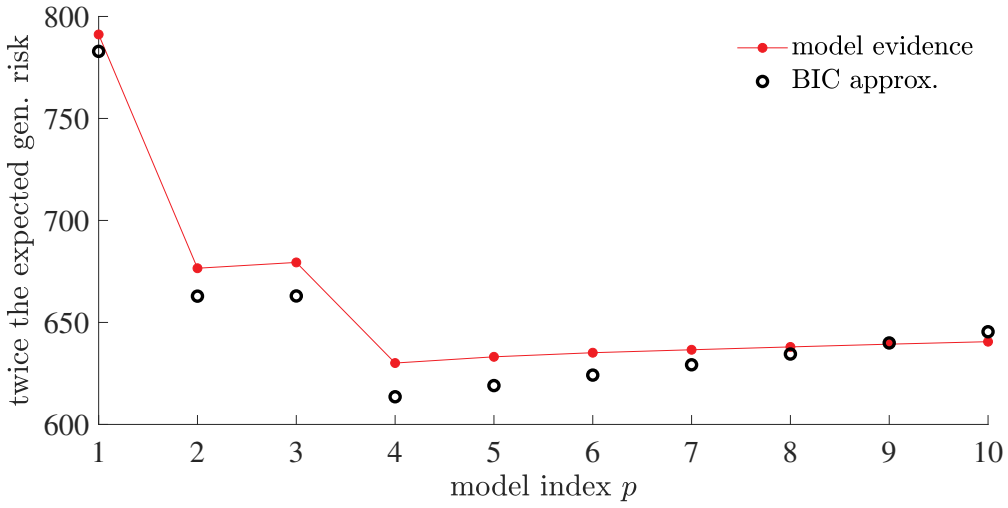


Figure 2.15: The BIC and marginal likelihood used for model selection.

■

It is possible to give the model complexity parameter p a Bayesian treatment, in which we define a prior density on the set of all models under consideration. For example, let $g(p)$, $p = 1, \ldots, m$ be a prior density on m candidate models. Treating the model complexity index p as an additional parameter to $\boldsymbol{\theta} \in \mathbb{R}^p$, and applying Bayes' formula, the posterior for $(\boldsymbol{\theta}, p)$ can be written as:

$$\begin{aligned} g(\boldsymbol{\theta}, p \,|\, \tau) &= g(\boldsymbol{\theta} \,|\, p, \tau) \times g(p \,|\, \tau) \\ &= \underbrace{\frac{g(\tau \,|\, \boldsymbol{\theta}, p)\, g(\boldsymbol{\theta} \,|\, p)}{g(\tau \,|\, p)}}_{\text{posterior of } \boldsymbol{\theta} \text{ given model } p} \times \underbrace{\frac{g(\tau \,|\, p)\, g(p)}{g(\tau)}}_{\text{posterior of model } p}. \end{aligned}$$

The model evidence for a fixed p is now interpreted as the prior predictive density of τ, conditional on the model p:

$$g(\tau \,|\, p) = \int g(\tau \,|\, \boldsymbol{\theta}, p)\, g(\boldsymbol{\theta} \,|\, p)\, \mathrm{d}\boldsymbol{\theta},$$

and the quantity $g(\tau) = \sum_{p=1}^{m} g(\tau \,|\, p)\, g(p)$ is interpreted as the marginal likelihood of all the m candidate models. Finally, a simple method for model selection is to pick the index $\widehat{p}$ with the largest posterior probability:

$$\widehat{p} = \operatorname*{argmax}_p g(p \,|\, \tau) = \operatorname*{argmax}_p g(\tau \,|\, p)\, g(p).$$

■ **Example 2.11 (Polynomial Regression (cont.))** Let us revisit Example 2.10 by giving the parameter $p = 1, \ldots, m$, with $m = 10$, a Bayesian treatment. Recall that we used the notation $\tau = \boldsymbol{y}$ in that example. We assume that the prior $g(p) = 1/m$ is flat and uninformative so that the posterior is given by

$$g(p \,|\, \boldsymbol{y}) \propto g(\boldsymbol{y} \,|\, p) = \frac{|\boldsymbol{\Sigma}|^{1/2}\, \Gamma(n/2)}{|\mathbf{D}|^{1/2} (\pi(n + p + 2)\, \overline{\sigma}^2)^{n/2}},$$

where all quantities in $g(\boldsymbol{y} \mid p)$ are computed using the first p columns of $\mathbf{X}$. Figure 2.16 shows the resulting posterior distribution $g(p \mid \boldsymbol{y})$. The figure also shows the posterior density $\widehat{g}(\boldsymbol{y} \mid p) / \sum_{p=1}^{10} \widehat{g}(\boldsymbol{y} \mid p)$, where

$$\widehat{g}(\boldsymbol{y} \mid p) := \exp\left(-\frac{n[\ln(2\pi\,\overline{\sigma}^2) + 1] + (p+1)\ln(n) + (p+2)}{2}\right)$$

is derived from the BIC approximation (2.52). In both cases, there is a clear maximum at $p = 4$, suggesting that a third-degree polynomial is the most appropriate model for the data.

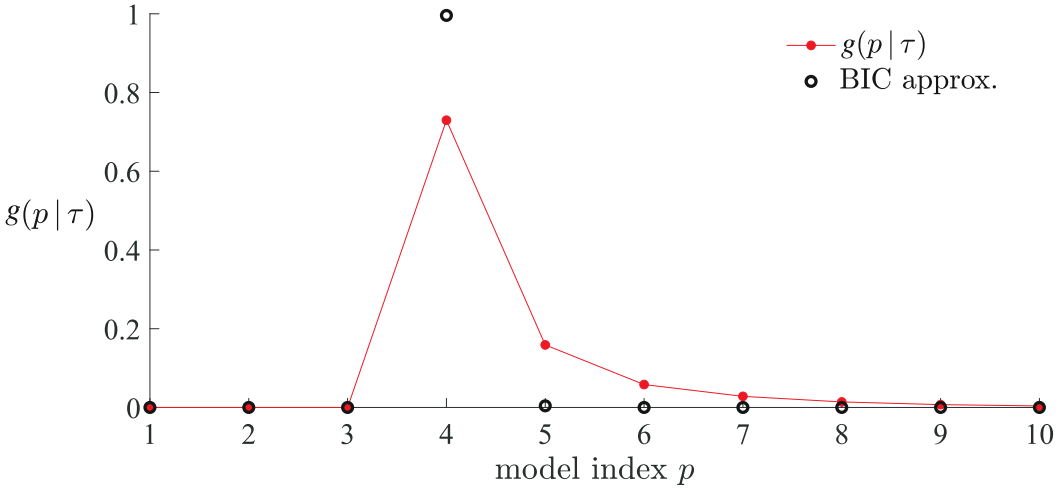


Figure 2.16: Posterior probabilities for each polynomial model of degree $p - 1$.

Suppose that we wish to compare two models, say model $p = 1$ and model $p = 2$. Instead of computing the posterior $g(p \mid \tau)$ explicitly, we can compare the posterior odds ratio:

$$\frac{g(p=1 \mid \tau)}{g(p=2 \mid \tau)} = \frac{g(p=1)}{g(p=2)} \times \underbrace{\frac{g(\tau \mid p=1)}{g(\tau \mid p=2)}}_{\text{Bayes factor } B_{1\mid 2}}.$$

This gives rise to the *Bayes factor* $B_{i\mid j}$, whose value signifies the strength of the evidence in favor of model i over model j. In particular $B_{i\mid j} > 1$ means that the evidence in favor for model i is larger.

BAYES FACTOR

■ **Example 2.12 (Savage–Dickey Ratio)** Suppose that we have two models. Model $p = 2$ has a likelihood $g(\tau \mid \mu, \nu, p = 2)$, depending on two parameters. Model $p = 1$ has the same functional form for the likelihood but now ν is fixed to some (known) ν_0; that is, $g(\tau \mid \mu, p = 1) = g(\tau \mid \mu, \nu = \nu_0, p = 2)$. We also assume that the prior information on μ for model 1 is the same as that for model 2, conditioned on $\nu = \nu_0$. That is, we assume

$g(\mu \,|\, p=1) = g(\mu \,|\, \nu=\nu_0, p=2)$. As model 2 contains model 1 as a special case, the latter is said to be *nested* inside model 2. We can formally write (see also Exercise 26):

$$\begin{aligned} g(\tau \,|\, p=1) &= \int g(\tau \,|\, \mu, p=1)\, g(\mu \,|\, p=1)\, \mathrm{d}\mu \\ &= \int g(\tau \,|\, \mu, \nu=\nu_0, p=2)\, g(\mu \,|\, \nu=\nu_0, p=2)\, \mathrm{d}\mu \\ &= g(\tau \,|\, \nu=\nu_0, p=2) = \frac{g(\tau, \nu=\nu_0 \,|\, p=2)}{g(\nu=\nu_0 \,|\, p=2)}. \end{aligned}$$

Hence, the Bayes factor simplifies to

$$B_{1|2} = \frac{g(\tau \,|\, p=1)}{g(\tau \,|\, p=2)} = \frac{g(\tau, \nu=\nu_0 \,|\, p=2)}{g(\nu=\nu_0 \,|\, p=2)} \Big/ g(\tau \,|\, p=2) = \frac{g(\nu=\nu_0 \,|\, \tau, p=2)}{g(\nu=\nu_0 \,|\, p=2)}.$$

In other words, $B_{1|2}$ is the ratio of the posterior density to the prior density of ν, evaluated at $\nu=\nu_0$ and both under the unrestricted model $p=2$. This ratio of posterior to prior densities is called the *Savage–Dickey density ratio*. ■

SAVAGE–DICKEY DENSITY RATIO

Whether to use a classical (frequentist) or Bayesian model is largely a question of convenience. Classical inference is useful because it comes with a huge repository of ready-to-use results, and requires no (subjective) prior information on the parameters. Bayesian models are useful because the whole theory is based on the elegant Bayes' formula, and uncertainty in the inference (e.g., confidence intervals) can be quantified much more naturally (e.g., credible intervals). A usual practice is to "Bayesify" a classical model, simply by adding some prior information on the parameters.

Further Reading

A popular textbook on statistical learning is [55]. Accessible treatments of mathematical statistics can be found, for example, in [69], [74], and [124]. More advanced treatments are given in [10], [25], and [78]. A good overview of modern-day statistical inference is given in [36]. Classical references on pattern classification and machine learning are [12] and [35]. For advanced learning theory including information theory and Rademacher complexity, we refer to [28] and [109]. An applied reference for Bayesian inference is [46]. For a survey of numerical techniques relevant to computational statistics, see [90].

Exercises

1. Suppose that the loss function is the piecewise linear function

$$\mathrm{Loss}(y, \widehat{y}) = \alpha\, (\widehat{y} - y)_+ + \beta\, (y - \widehat{y})_+, \quad \alpha, \beta > 0,$$

where c_+ is equal to c if $c > 0$, and zero otherwise. Show that the minimizer of the risk $\ell(g) = \mathbb{E}\,\mathrm{Loss}(Y, g(\boldsymbol{X}))$ satisfies

$$\mathbb{P}[Y < g^*(\boldsymbol{x}) \,|\, \boldsymbol{X} = \boldsymbol{x}] = \frac{\beta}{\alpha + \beta}.$$

In other words, $g^*(\boldsymbol{x})$ is the $\beta/(\alpha+\beta)$ quantile of Y, conditional on $\boldsymbol{X} = \boldsymbol{x}$.

2. Show that, for the squared-error loss, the approximation error $\ell(g^{\mathcal{G}}) - \ell(g^*)$ in (2.16), is equal to $\mathbb{E}(g^{\mathcal{G}}(\boldsymbol{X}) - g^*(\boldsymbol{X}))^2$. [Hint: expand $\ell(g^{\mathcal{G}}) = \mathbb{E}(Y - g^*(\boldsymbol{X}) + g^*(\boldsymbol{X}) - g^{\mathcal{G}}(\boldsymbol{X}))^2$.]

3. Suppose $\mathcal{G}$ is the class of *linear* functions. A linear function evaluated at a feature $\boldsymbol{x}$ can be described as $g(\boldsymbol{x}) = \boldsymbol{\beta}^\top \boldsymbol{x}$ for some parameter vector $\boldsymbol{\beta}$ of appropriate dimension. Denote $g^{\mathcal{G}}(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{\beta}^{\mathcal{G}}$ and $g_\tau^{\mathcal{G}}(\boldsymbol{x}) = \boldsymbol{x}^\top \widehat{\boldsymbol{\beta}}$. Show that

$$\mathbb{E}\left(g_\tau^{\mathcal{G}}(\boldsymbol{X}) - g^*(\boldsymbol{X})\right)^2 = \mathbb{E}\left(\boldsymbol{X}^\top \widehat{\boldsymbol{\beta}} - \boldsymbol{X}^\top \boldsymbol{\beta}^{\mathcal{G}}\right)^2 + \mathbb{E}\left(\boldsymbol{X}^\top \boldsymbol{\beta}^{\mathcal{G}} - g^*(\boldsymbol{X})\right)^2.$$

Hence, deduce that the statistical error in (2.16) is $\ell(g_\tau^{\mathcal{G}}) - \ell(g^{\mathcal{G}}) = \mathbb{E}\,(g_\tau^{\mathcal{G}}(\boldsymbol{X}) - g^{\mathcal{G}}(\boldsymbol{X}))^2$.

4. Show that formula (2.24) holds for the 0–1 loss with 0–1 response.

5. Let $\boldsymbol{X}$ be an n-dimensional normal random vector with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$, where the determinant of $\boldsymbol{\Sigma}$ is non-zero. Show that $\boldsymbol{X}$ has joint probability density

$$f_X(\boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^n\,|\boldsymbol{\Sigma}|}}\, \mathrm{e}^{-\frac{1}{2}\,(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})}, \quad \boldsymbol{x} \in \mathbb{R}^n.$$

6. Let $\widehat{\boldsymbol{\beta}} = \mathbf{A}^+ \boldsymbol{y}$. Using the defining properties of the pseudo-inverse, show that for any $\boldsymbol{\beta} \in \mathbb{R}^p$, ☞ 360

$$\|\mathbf{A}\widehat{\boldsymbol{\beta}} - \boldsymbol{y}\| \leqslant \|\mathbf{A}\boldsymbol{\beta} - \boldsymbol{y}\|.$$

7. Suppose that in the polynomial regression Example 2.1 we select the linear class of functions $\mathcal{G}_p$ with $p \geqslant 4$. Then, $g^* \in \mathcal{G}_p$ and the approximation error is zero, because $g^{\mathcal{G}_p}(\boldsymbol{x}) = g^*(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{\beta}$, where $\boldsymbol{\beta} = [10, -140, 400, -250, 0, \ldots, 0]^\top \in \mathbb{R}^p$. Use the tower property to show that the learner $g_\tau(\boldsymbol{x}) = \boldsymbol{x}^\top \widehat{\boldsymbol{\beta}}$ with $\widehat{\boldsymbol{\beta}} = \mathbf{X}^+ \boldsymbol{y}$, assuming $\operatorname{rank}(\mathbf{X}) \geqslant 4$, is *unbiased*: ☞ 431 UNBIASED

$$\mathbb{E}\, g_{\mathcal{T}}(\boldsymbol{x}) = g^*(\boldsymbol{x}).$$

8. (Exercise 7 continued.) Observe that the learner $g_{\mathcal{T}}$ can be written as a linear combination of the response variable: $g_{\mathcal{T}}(\boldsymbol{x}) = \boldsymbol{x}^\top \mathbf{X}^+ \boldsymbol{Y}$. Prove that for any learner of the form $\boldsymbol{x}^\top \mathbf{A}\boldsymbol{y}$, where $\mathbf{A} \in \mathbb{R}^{p\times n}$ is some matrix and that satisfies $\mathbb{E}_{\mathbf{X}}[\boldsymbol{x}^\top \mathbf{A}\boldsymbol{Y}] = g^*(\boldsymbol{x})$, we have

$$\mathbb{V}\mathrm{ar}_{\mathbf{X}}[\boldsymbol{x}^\top \mathbf{X}^+ \boldsymbol{Y}] \leqslant \mathbb{V}\mathrm{ar}_{\mathbf{X}}[\boldsymbol{x}^\top \mathbf{A}\boldsymbol{Y}],$$

where the equality is achieved for $\mathbf{A} = \mathbf{X}^+$. This is called the *Gauss–Markov inequality*. Hence, using the Gauss–Markov inequality deduce that for the unconditional variance: GAUSS–MARKOV INEQUALITY

$$\mathbb{V}\mathrm{ar}\, g_{\mathcal{T}}(\boldsymbol{x}) \leqslant \mathbb{V}\mathrm{ar}[\boldsymbol{x}^\top \mathbf{A}\boldsymbol{Y}].$$

Deduce that $\mathbf{A} = \mathbf{X}^+$ also minimizes the expected generalization risk.

9. Consider again the polynomial regression Example 2.1. Use the fact that $\mathbb{E}_{\mathbf{X}}\, \widehat{\boldsymbol{\beta}} = \mathbf{X}^+ \boldsymbol{h}^*(\boldsymbol{u})$, where $\boldsymbol{h}^*(\boldsymbol{u}) = \mathbb{E}[\boldsymbol{Y} \,|\, \boldsymbol{U} = \boldsymbol{u}] = [h^*(u_1), \ldots, h^*(u_n)]^\top$, to show that the expected in-sample risk is:

$$\mathbb{E}_{\mathbf{X}}\, \ell_{\mathrm{in}}(g_{\mathcal{T}}) = \ell^* + \frac{\|\boldsymbol{h}^*(\boldsymbol{u})\|^2 - \|\mathbf{X}\mathbf{X}^+ \boldsymbol{h}^*(\boldsymbol{u})\|^2}{n} + \frac{\ell^* p}{n}.$$

Also, use Theorem C.2 to show that the expected statistical error is: ☞ 430

$$\mathbb{E}_{\mathbf{X}}\, (\widehat{\boldsymbol{\beta}} - \boldsymbol{\beta})^\top \mathbf{H}_p (\widehat{\boldsymbol{\beta}} - \boldsymbol{\beta}) = \ell^* \mathrm{tr}(\mathbf{X}^+ (\mathbf{X}^+)^\top \mathbf{H}_p) + (\mathbf{X}^+ \boldsymbol{h}^*(\boldsymbol{u}) - \boldsymbol{\beta})^\top \mathbf{H}_p (\mathbf{X}^+ \boldsymbol{h}^*(\boldsymbol{u}) - \boldsymbol{\beta}).$$

10. Consider the setting of the polynomial regression in Example 2.2. Use Theorem C.19 to prove that

☞ 449

$$\sqrt{n}\,(\widehat{\boldsymbol{\beta}}_n - \boldsymbol{\beta}_p) \xrightarrow{\mathrm{d}} \mathcal{N}\left(\mathbf{0},\, \ell^* \mathbf{H}_p^{-1} + \mathbf{H}_p^{-1}\mathbf{M}_p\mathbf{H}_p^{-1}\right), \tag{2.53}$$

where $\mathbf{M}_p := \mathbb{E}[\boldsymbol{X}\boldsymbol{X}^\top (g^*(\boldsymbol{X}) - g^{\mathcal{G}_p}(\boldsymbol{X}))^2]$ is the matrix with (i,j)-th entry:

$$\int_0^1 u^{i+j-2} (h^{\mathcal{H}_p}(u) - h^*(u))^2 \,\mathrm{d}u,$$

INVERSE HILBERT MATRIX

and $\mathbf{H}_p^{-1}$ is the $p \times p$ *inverse Hilbert matrix* with (i,j)-th entry:

$$(-1)^{i+j}(i+j-1)\binom{p+i-1}{p-j}\binom{p+j-1}{p-i}\binom{i+j-2}{i-1}^2.$$

Observe that $\mathbf{M}_p = \mathbf{0}$ for $p \geqslant 4$, so that the matrix $\mathbf{M}_p$ term is due to choosing a restrictive class $\mathcal{G}_p$ that does not contain the true prediction function.

11. In Example 2.2 we saw that the statistical error can be expressed (see (2.20)) as

$$\int_0^1 \left([1, \ldots, u^{p-1}](\widehat{\boldsymbol{\beta}} - \boldsymbol{\beta}_p)\right)^2 \mathrm{d}u = (\widehat{\boldsymbol{\beta}} - \boldsymbol{\beta}_p)^\top \mathbf{H}_p (\widehat{\boldsymbol{\beta}} - \boldsymbol{\beta}_p).$$

By Exercise 10 the random vector $\boldsymbol{Z}_n := \sqrt{n}(\widehat{\boldsymbol{\beta}}_n - \boldsymbol{\beta}_p)$ has asymptotically a multivariate normal distribution with mean vector $\mathbf{0}$ and covariance matrix $\mathbf{V} := \ell^* \mathbf{H}_p^{-1} + \mathbf{H}_p^{-1}\mathbf{M}_p\mathbf{H}_p^{-1}$. Use Theorem C.2 to show that the *expected* statistical error is asymptotically

☞ 430

$$\mathbb{E}\,(\widehat{\boldsymbol{\beta}} - \boldsymbol{\beta}_p)^\top \mathbf{H}_p (\widehat{\boldsymbol{\beta}} - \boldsymbol{\beta}_p) \simeq \frac{\ell^* p}{n} + \frac{\mathrm{tr}(\mathbf{M}_p \mathbf{H}_p^{-1})}{n}, \quad n \to \infty. \tag{2.54}$$

Plot this large-sample approximation of the expected statistical error and compare it with the outcome of the statistical error.

We note a subtle technical detail: In general, convergence in distribution does not imply convergence in L_p-norm (see Example C.6), and so here we have implicitly assumed that $\|\boldsymbol{Z}_n\| \xrightarrow{\mathrm{d}} \mathsf{Dist.} \Rightarrow \|\boldsymbol{Z}_n\| \xrightarrow{L_2} \text{constant} := \lim_{n\uparrow\infty} \mathbb{E}\|\boldsymbol{Z}_n\|$.

☞ 442

12. Consider again Example 2.2. The result in (2.53) suggests that $\mathbb{E}\widehat{\boldsymbol{\beta}} \to \boldsymbol{\beta}_p$ as $n \to \infty$, where $\boldsymbol{\beta}_p$ is the solution in the class $\mathcal{G}_p$ given in (2.18). Thus, the large-sample approximation of the pointwise bias of the learner $g_{\mathcal{T}}^{\mathcal{G}_p}(\boldsymbol{x}) = \boldsymbol{x}^\top \widehat{\boldsymbol{\beta}}$ at $\boldsymbol{x} = [1, \ldots, u^{p-1}]^\top$ is

$$\mathbb{E}\, g_{\mathcal{T}}^{\mathcal{G}_p}(\boldsymbol{x}) - g^*(\boldsymbol{x}) \simeq [1, \ldots, u^{p-1}]\boldsymbol{\beta}_p - [1, u, u^2, u^3]\boldsymbol{\beta}^*, \quad n \to \infty.$$

Use Python to reproduce Figure 2.17, which shows the (large-sample) pointwise squared bias of the learner for $p \in \{1, 2, 3\}$. Note how the bias is larger near the endpoints $u = 0$ and $u = 1$. Explain why the areas under the curves correspond to the approximation errors.

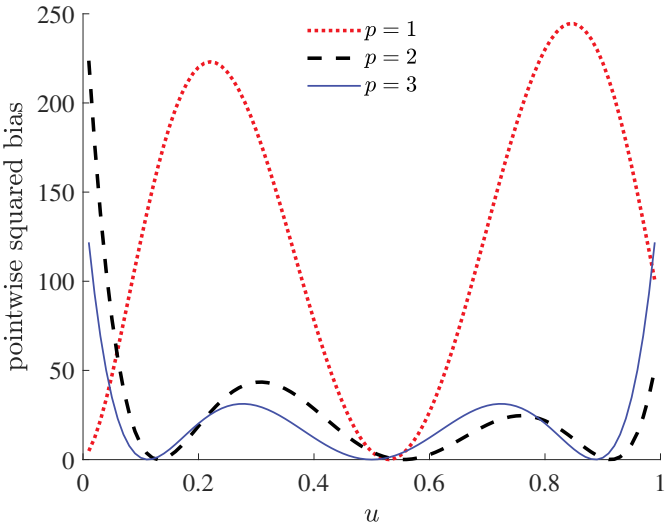


Figure 2.17: The large-sample pointwise squared bias of the learner for $p = 1, 2, 3$. The bias is zero for $p \geqslant 4$.

13. For our running Example 2.2 we can use (2.53) to derive a large-sample approximation of the pointwise variance of the learner $g_{\mathcal{T}}(\boldsymbol{x}) = \boldsymbol{x}^\top \widehat{\boldsymbol{\beta}}_n$. In particular, show that for large n

$$\mathbb{V}\mathrm{ar}\, g_{\mathcal{T}}(\boldsymbol{x}) \simeq \frac{\ell^* \, \boldsymbol{x}^\top \mathbf{H}_p^{-1} \boldsymbol{x}}{n} + \frac{\boldsymbol{x}^\top \mathbf{H}_p^{-1} \mathbf{M}_p \mathbf{H}_p^{-1} \boldsymbol{x}}{n}, \quad n \to \infty. \tag{2.55}$$

Figure 2.18 shows this (large-sample) variance of the learner for different values of the predictor u and model index p. Observe that the variance ultimately increases in p and that it is smaller at $u = 1/2$ than closer to the endpoints $u = 0$ or $u = 1$. Since the bias is also

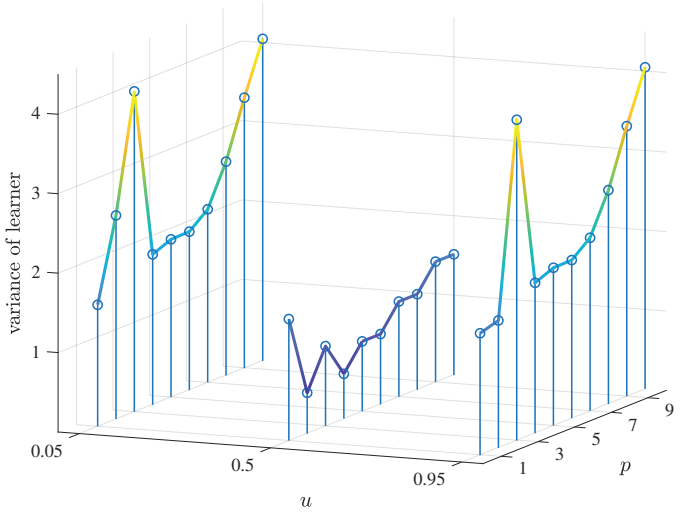


Figure 2.18: The pointwise variance of the learner for various pairs of p and u.

larger near the endpoints, we deduce that the pointwise mean squared error (2.21) is larger near the endpoints of the interval $[0, 1]$ than near its middle. In other words, the error is much smaller in the center of the data cloud than near its periphery.

☞ 403

JENSEN'S INEQUALITY

14. Let $h : \boldsymbol{x} \mapsto \mathbb{R}$ be a convex function and let $\boldsymbol{X}$ be a random variable. Use the subgradient definition of convexity to prove *Jensen's inequality*:

$$\mathbb{E}\, h(\boldsymbol{X}) \geqslant h(\mathbb{E}\boldsymbol{X}). \tag{2.56}$$

15. Using Jensen's inequality, show that the Kullback–Leibler divergence between probability densities f and g is always positive; that is,

$$\mathbb{E} \ln \frac{f(\boldsymbol{X})}{g(\boldsymbol{X})} \geqslant 0,$$

where $\boldsymbol{X} \sim f$.

VAPNIK–CHERNOVENKIS BOUND

16. The purpose of this exercise is to prove the following *Vapnik–Chernovenkis bound*: for any *finite* class $\mathcal{G}$ (containing only a finite number $|\mathcal{G}|$ of possible functions) and a general *bounded* loss function, $l \leqslant \text{Loss} \leqslant u$, the expected statistical error is bounded from above according to:

$$\mathbb{E}\, \ell(g^{\mathcal{G}}_{\mathcal{T}_n}) - \ell(g^{\mathcal{G}}) \leqslant \frac{(u-l)\sqrt{2\ln(2|\mathcal{G}|)}}{\sqrt{n}}. \tag{2.57}$$

Note how this bound conveniently does not depend on the distribution of the training set $\mathcal{T}_n$ (which is typically unknown), but only on the complexity (i.e., cardinality) of the class $\mathcal{G}$. We can break up the proof of (2.57) into the following four parts:

(a) For a general function class $\mathcal{G}$, training set $\mathcal{T}$, risk function ℓ, and training loss $\ell_{\mathcal{T}}$, we have, by definition, $\ell(g^{\mathcal{G}}) \leqslant \ell(g)$ and $\ell_{\mathcal{T}}(g^{\mathcal{G}}_{\mathcal{T}}) \leqslant \ell_{\mathcal{T}}(g)$ for all $g \in \mathcal{G}$. Show that

$$\ell(g^{\mathcal{G}}_{\mathcal{T}}) - \ell(g^{\mathcal{G}}) \leqslant \sup_{g\in\mathcal{G}} |\ell_{\mathcal{T}}(g) - \ell(g)| + \ell_{\mathcal{T}}(g^{\mathcal{G}}) - \ell(g^{\mathcal{G}}),$$

where we used the notation sup (supremum) for the least upper bound. Since $\mathbb{E}\ell_{\mathcal{T}}(g) = \mathbb{E}\ell(g)$, we obtain, after taking expectations on both sides of the inequality above:

$$\mathbb{E}\, \ell(g^{\mathcal{G}}_{\mathcal{T}}) - \ell(g^{\mathcal{G}}) \leqslant \mathbb{E} \sup_{g\in\mathcal{G}} |\ell_{\mathcal{T}}(g) - \ell(g)|.$$

HOEFFDING'S INEQUALITY

(b) If X is a zero-mean random variable taking values in the interval $[l, u]$, then the following *Hoeffding's inequality* states that the moment generating function satisfies

$$\mathbb{E}\, \mathrm{e}^{tX} \leqslant \exp\left(\frac{t^2(u-l)^2}{8}\right), \quad t \in \mathbb{R}. \tag{2.58}$$

Prove this result by using the fact that the line segment joining points $(l, \exp(tl))$ and $(u, \exp(tu))$ bounds the convex function $x \mapsto \exp(tx)$ for $x \in [l, u]$; that is:

$$\mathrm{e}^{tx} \leqslant \mathrm{e}^{tl}\frac{u-x}{u-l} + \mathrm{e}^{tu}\frac{x-l}{u-l}, \quad x \in [l, u].$$

(c) Let $Z_1, \ldots, Z_n$ be (possibly dependent and non-identically distributed) zero-mean random variables with moment generating functions that satisfy $\mathbb{E}\exp(tZ_k) \leqslant \exp(t^2\eta^2/2)$ for all k and some parameter η. Use Jensen's inequality (2.56) to prove that for any

☞ 427

$t > 0$,

$$\mathbb{E} \max_k Z_k = \frac{1}{t} \mathbb{E} \ln \max_k \mathrm{e}^{tZ_k} \leqslant \frac{1}{t} \ln n + \frac{t\eta^2}{2}.$$

From this derive that

$$\mathbb{E} \max_k Z_k \leqslant \eta \sqrt{2 \ln n}.$$

Finally, show that this last inequality implies that

$$\mathbb{E} \max_k |Z_k| \leqslant \eta \sqrt{2 \ln(2n)}. \tag{2.59}$$

(d) Returning to the objective of this exercise, denote the elements of $\mathcal{G}$ by $g_1, \ldots, g_{|\mathcal{G}|}$, and let $Z_k = \ell_{\mathcal{T}_n}(g_k) - \ell(g_k)$. By part (a) it is sufficient to bound $\mathbb{E} \max_k |Z_k|$. Show that the $\{Z_k\}$ satisfy the conditions of (c) with $\eta = (u - l)/\sqrt{n}$. For this you will need to apply part (b) to the random variable $\mathrm{Loss}(g(\boldsymbol{X}), Y) - \ell(g)$, where $(\boldsymbol{X}, Y)$ is a generic data point. Now complete the proof of (2.57).

17. Consider the problem in Exercise 16a above. Show that

$$|\ell_{\mathcal{T}}(g_{\mathcal{T}}^{\mathcal{G}}) - \ell(g^{\mathcal{G}})| \leqslant 2 \sup_{g \in \mathcal{G}} |\ell_{\mathcal{T}}(g) - \ell(g)| + \ell_{\mathcal{T}}(g^{\mathcal{G}}) - \ell(g^{\mathcal{G}}).$$

From this, conclude:

$$\mathbb{E} \, |\ell_{\mathcal{T}}(g_{\mathcal{T}}^{\mathcal{G}}) - \ell(g^{\mathcal{G}})| \leqslant 2 \mathbb{E} \sup_{g \in \mathcal{G}} |\ell_{\mathcal{T}}(g) - \ell(g)|.$$

The last bound allows us to assess how close the training loss $\ell_{\mathcal{T}}(g_{\mathcal{T}}^{\mathcal{G}})$ is to the optimal risk $\ell(g^{\mathcal{G}})$ within class $\mathcal{G}$.

18. Show that for the normal linear model $\boldsymbol{Y} \sim \mathcal{N}(\mathbf{X}\boldsymbol{\beta}, \sigma^2 \mathbf{I}_n)$, the maximum likelihood estimator of σ^2 is identical to the method of moments estimator (2.37).

19. Let $X \sim \mathsf{Gamma}(\alpha, \lambda)$. Show that the pdf of $Z = 1/X$ is equal to

$$\frac{\lambda^\alpha (z)^{-\alpha-1} \mathrm{e}^{-\lambda (z)^{-1}}}{\Gamma(\alpha)}, \quad z > 0.$$

20. Consider the sequence $w_0, w_1, \ldots$, where $w_0 = g(\boldsymbol{\theta})$ is a non-degenerate initial guess and $w_t(\boldsymbol{\theta}) \propto w_{t-1}(\boldsymbol{\theta}) g(\tau \,|\, \boldsymbol{\theta})$, $t > 1$. We assume that $g(\tau \,|\, \boldsymbol{\theta})$ is not the constant function (with respect to $\boldsymbol{\theta}$) and that the maximum likelihood value

$$g(\tau \,|\, \widehat{\boldsymbol{\theta}}) = \max_{\boldsymbol{\theta}} g(\tau \,|\, \boldsymbol{\theta}) < \infty$$

exists (is bounded). Let

$$l_t := \int g(\tau \,|\, \boldsymbol{\theta}) w_t(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}.$$

Show that $\{l_t\}$ is a strictly increasing and bounded sequence. Hence, conclude that its limit is $g(\tau \,|\, \widehat{\boldsymbol{\theta}})$.

21. Consider the Bayesian model for $\tau = \{x_1, \ldots, x_n\}$ with likelihood $g(\tau \,|\, \mu)$ such that $(X_1, \ldots, X_n \,|\, \mu) \sim_{\text{iid}} \mathcal{N}(\mu, 1)$ and prior pdf $g(\mu)$ such that $\mu \sim \mathcal{N}(\nu, 1)$ for some hyperparameter ν. Define a sequence of densities $w_t(\mu), t \geqslant 2$ via $w_t(\mu) \propto w_{t-1}(\mu)\, g(\tau \,|\, \mu)$, starting with $w_1(\mu) = g(\mu)$. Let a_t and b_t denote the mean and precision[4] of μ under the posterior $g_t(\mu \,|\, \tau) \propto g(\tau \,|\, \mu) w_t(\mu)$. Show that $g_t(\mu \,|\, \tau)$ is a normal density with precision $b_t = b_{t-1} + n,\ b_0 = 1$ and mean $a_t = (1 - \gamma_t)a_{t-1} + \gamma_t \overline{x}_n,\ a_0 = \nu$, where $\gamma_t := n/(b_{t-1} + n)$. Hence, deduce that $g_t(\mu \,|\, \tau)$ converges to a degenerate density with a point-mass at $\overline{x}_n$.

22. Consider again Example 2.8, where we have a normal model with improper prior $g(\boldsymbol{\theta}) = g(\mu, \sigma^2) \propto 1/\sigma^2$. Show that the prior predictive pdf is an improper density $g(x) \propto 1$, but that the posterior predictive density is

$$g(x \,|\, \tau) \propto \left(1 + \frac{(x - \overline{x}_n)^2}{(n+1)S_n^2}\right)^{-n/2}.$$

Deduce that $\frac{X - \overline{x}_n}{S_n \sqrt{(n+1)/(n-1)}} \sim \mathsf{t}_{n-1}$.

23. Assuming that $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n \overset{\text{iid}}{\sim} f$, show that (2.48) holds and that $\ell_n^* = -n\, \mathbb{E} \ln f(\boldsymbol{X})$.

24. Suppose that $\tau = \{x_1, \ldots, x_n\}$ are observations of iid continuous and strictly positive random variables, and that there are two possible models for their pdf. The first model $p = 1$ is

$$g(x \,|\, \theta, p = 1) = \theta \exp(-\theta x)$$

and the second $p = 2$ is

$$g(x \,|\, \theta, p = 2) = \left(\frac{2\theta}{\pi}\right)^{1/2} \exp\left(-\frac{\theta x^2}{2}\right).$$

For both models, assume that the prior for θ is a gamma density

$$g(\theta) = \frac{b^t}{\Gamma(t)} \theta^{t-1} \exp(-b\theta),$$

with the same hyperparameters b and t. Find a formula for the Bayes factor, $g(\tau \,|\, p = 1)/g(\tau \,|\, p = 2)$, for comparing these models.

25. Suppose that we have a total of m possible models with prior probabilities $g(p), p = 1, \ldots, m$. Show that the posterior probability of model $g(p \,|\, \tau)$ can be expressed in terms of all the $p(p-1)$ Bayes factors:

$$g(p = i \,|\, \tau) = \left(1 + \sum_{j \neq i} \frac{g(p = j)}{g(p = i)} B_{j|i}\right)^{-1}.$$

[4] The precision is the reciprocal of the variance.

26. Given the data $\tau = \{x_1, \ldots, x_n\}$, suppose that we use the likelihood $(X \,|\, \boldsymbol{\theta}) \sim \mathcal{N}(\mu, \sigma^2)$ with parameter $\boldsymbol{\theta} = (\mu, \sigma^2)^\top$ and wish to compare the following two nested models.

(a) Model $p = 1$, where $\sigma^2 = \sigma_0^2$ is known and this is incorporated via the prior

$$g(\boldsymbol{\theta} \,|\, p = 1) = g(\mu \,|\, \sigma^2, p = 1)\, g(\sigma^2 \,|\, p = 1) = \frac{1}{\sqrt{2\pi}\sigma} \mathrm{e}^{-\frac{(\mu - x_0)^2}{2\sigma^2}} \times \delta(\sigma^2 - \sigma_0^2).$$

(b) Model $p = 2$, where both mean and variance are unknown with prior

$$g(\boldsymbol{\theta} \,|\, p = 2) = g(\mu \,|\, \sigma^2)\, g(\sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \mathrm{e}^{-\frac{(\mu - x_0)^2}{2\sigma^2}} \times \frac{b^t (\sigma^2)^{-t-1} \mathrm{e}^{-b/\sigma^2}}{\Gamma(t)}.$$

Show that the prior $g(\boldsymbol{\theta} \,|\, p = 1)$ can be viewed as the limit of the prior $g(\boldsymbol{\theta} \,|\, p = 2)$ when $t \to \infty$ and $b = t\sigma_0^2$. Hence, conclude that

$$g(\tau \,|\, p = 1) = \lim_{\substack{t \to \infty \\ b = t\sigma_0^2}} g(\tau \,|\, p = 2)$$

and use this result to calculate $B_{1\,|\,2}$. Check that the formula for $B_{1\,|\,2}$ agrees with the Savage–Dickey density ratio:

$$\frac{g(\tau \,|\, p = 1)}{g(\tau \,|\, p = 2)} = \frac{g(\sigma^2 = \sigma_0^2 \,|\, \tau)}{g(\sigma^2 = \sigma_0^2)},$$

where $g(\sigma^2 \,|\, \tau)$ and $g(\sigma^2)$ are the posterior and prior, respectively, under model $p = 2$.


Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 3

MONTE CARLO METHODS

Many algorithms in machine learning and data science make use of Monte Carlo techniques. This chapter gives an introduction to the three main uses of Monte Carlo simulation: to (1) simulate random objects and processes in order to observe their behavior, (2) estimate numerical quantities by repeated sampling, and (3) solve complicated optimization problems through randomized algorithms.

3.1 Introduction

MONTE CARLO SIMULATION

Briefly put, *Monte Carlo simulation* is the generation of random data by means of a computer. These data could arise from simple models, such as those described in Chapter 2, or from very complicated models describing real-life systems, such as the positions of vehicles on a complex road network, or the evolution of security prices in the stock market. In many cases, Monte Carlo simulation simply involves random sampling from certain probability distributions. The idea is to repeat the random experiment that is described by the model many times to obtain a large quantity of data that can be used to answer questions about the model. The three main uses of Monte Carlo simulation are:

Sampling. Here the objective is to gather information about a random object by observing many realizations of it. For instance, this could be a random process that mimics the behavior of some real-life system such as a production line or telecommunications network. Another usage is found in Bayesian statistics, where Markov chains are often used to sample from a posterior distribution. ☞ 48

Estimation. In this case the emphasis is on estimating certain numerical quantities related to a simulation model. An example is the evaluation of multidimensional integrals via Monte Carlo techniques. This is achieved by writing the integral as the expectation of a random variable, which is then approximated by the sample mean. Appealing to the Law of Large Numbers guarantees that this approximation will eventually converge when the sample size becomes large. ☞ 446

Optimization. Monte Carlo simulation is a powerful tool for the optimization of complicated objective functions. In many applications these functions are deterministic and

randomness is introduced artificially in order to more efficiently search the domain of the objective function. Monte Carlo techniques are also used to optimize noisy functions, where the function itself is random; for example, when the objective function is the output of a Monte Carlo simulation.

The Monte Carlo method dramatically changed the way in which statistics is used in today's analysis of data. The ever-increasing complexity of data requires radically different statistical models and analysis techniques from those that were used 20 to 100 years ago. By using Monte Carlo techniques, the data analyst is no longer restricted to using basic (and often inappropriate) models to describe data. Now, any probabilistic model that can be simulated on a computer can serve as the basis for statistical analysis. This Monte Carlo revolution has had an impact on both Bayesian and frequentist statistics. In particular, in frequentist statistics, Monte Carlo methods are often referred to as resampling techniques. An important example is the well-known bootstrap method [37], where statistical quantities such as confidence intervals and P-values for statistical tests can simply be determined by simulation without the need of a sophisticated analysis of the underlying probability distributions; see, for example, [69] for basic applications. The impact on Bayesian statistics has been even more profound, through the use of Markov chain Monte Carlo (MCMC) techniques [87, 48]. MCMC samplers construct a Markov process which converges in distribution to a desired (often high-dimensional) density. This convergence in distribution justifies using a finite run of the Markov process as an approximate random realization from the target density. The MCMC approach has rapidly gained popularity as a versatile heuristic approximation, partly due to its simple computer implementation and inbuilt mechanism to tradeoff between computational cost and accuracy; namely, the longer one runs the Markov process, the better the approximation. Nowadays, MCMC methods are indispensable for analyzing posterior distributions for inference and model selection; see also [50, 99].

The following three sections elaborate on these three uses of Monte Carlo simulation in turn.

3.2 Monte Carlo Sampling

In this section we describe a variety of Monte Carlo sampling methods, from the building block of simulating uniform random numbers to MCMC samplers.

3.2.1 Generating Random Numbers

RANDOM NUMBER GENERATOR

At the heart of any Monte Carlo method is a *random number generator*: a procedure that produces a stream of uniform random numbers on the interval (0,1). Since such numbers are usually produced via deterministic algorithms, they are not truly random. However, for most applications all that is required is that such pseudo-random numbers are statistically indistinguishable from genuine random numbers $U_1, U_2, \ldots$ that are uniformly distributed on the interval (0,1) and are independent of each other; we write $U_1, U_2, \ldots \sim_{\text{iid}} \mathcal{U}(0,1)$. For example, in Python the `rand` method of the `numpy.random` module is widely used for this purpose.

Most random number generators at present are based on linear recurrence relations. One of the most important random number generators is the *multiple-recursive generator* (MRG) of *order* k, which generates a sequence of integers $X_k, X_{k+1}, \ldots$ via the linear recurrence

MULTIPLE-RECURSIVE GENERATOR

$$X_t = (a_1 X_{t-1} + \cdots + a_k X_{t-k}) \bmod m, \quad t = k, k+1, \ldots \tag{3.1}$$

for some *modulus* m and *multipliers* $\{a_i, i = 1, \ldots, k\}$. Here "mod" refers to the modulo operation: $n \bmod m$ is the remainder when n is divided by m. The recurrence is initialized by specifying k "seeds", $X_0, \ldots, X_{k-1}$. To yield fast algorithms, all but a few of the multipliers should be 0. When m is a large integer, one can obtain a stream of pseudo-random numbers $U_k, U_{k+1}, \ldots$ between 0 and 1 from the sequence $X_k, X_{k+1}, \ldots$, simply by setting $U_t = X_t/m$. It is also possible to set a small modulus, in particular $m = 2$. The output function for such *modulo 2 generators* is then typically of the form

MODULUS
MULTIPLIERS

MODULO 2 GENERATORS

$$U_t = \sum_{i=1}^{w} X_{tw+i-1} 2^{-i}$$

for some $w \leqslant k$, e.g., $w = 32$ or 64. Examples of modulo 2 generators are the *feedback shift register* generators, the most popular of which are the *Mersenne twisters*; see, for example, [79] and [83]. MRGs with excellent statistical properties can be implemented efficiently by combining several simpler MRGs and carefully choosing their respective moduli and multipliers. One of the most successful is L'Ecuyer's `MRG32k3a` generator; see [77]. From now on, we assume that the reader has a sound random number generator available.

FEEDBACK SHIFT REGISTER

MERSENNE TWISTERS

3.2.2 Simulating Random Variables

Simulating a random variable X from an arbitrary (that is, not necessarily uniform) distribution invariably involves the following two steps:

1. Simulate uniform random numbers $U_1, \ldots, U_k$ on $(0, 1)$ for some $k = 1, 2, \ldots$.
2. Return $X = g(U_1, \ldots, U_k)$, where g is some real-valued function.

The construction of suitable functions g is as much of an art as a science. Many simulation methods may be found, for example, in [71] and the accompanying website `www.montecarlohandbook.org`. Two of the most useful general procedures for generating random variables are the *inverse-transform* method and the *acceptance–rejection* method. Before we discuss these, we show one possible way to simulate standard normal random variables. In Python we can generate standard normal random variables via the `randn` method of the `numpy.random` module.

■ **Example 3.1 (Simulating Standard Normal Random Variables)** If X and Y are independent standard normally distributed random variables (that is, $X, Y \sim_{\text{iid}} \mathcal{N}(0, 1)$), then their joint pdf is

$$f(x, y) = \frac{1}{2\pi} \mathrm{e}^{-\frac{1}{2}(x^2+y^2)}, \quad (x, y) \in \mathbb{R}^2,$$

which is a radially symmetric function. In Example C.2 we see that, in polar coordinates, the angle Θ that the random vector $[X, Y]^\top$ makes with the positive x-axis is $\mathcal{U}(0, 2\pi)$

☞ 433

distributed (as would be expected from the radial symmetry) and the radius R has pdf $f_R(r) = r\,\mathrm{e}^{-r^2/2}, r > 0$. Moreover, R and Θ are independent. We will see shortly, in Example 3.4, that R has the same distribution as $\sqrt{-2\ln U}$ with $U \sim \mathcal{U}(0,1)$. So, to simulate $X, Y \sim_{\text{iid}} \mathcal{N}(0,1)$, the idea is to first simulate R and Θ independently and then return $X = R\cos(\Theta)$ and $Y = R\sin(\Theta)$ as a pair of independent standard normal random variables. This leads to the Box–Muller approach for generating standard normal random variables.

☞ 72

Algorithm 3.2.1: Normal Random Variable Simulation: Box–Muller Approach

output: Independent standard normal random variables X and Y.

1 Simulate two independent random variables, U_1 and U_2, from $\mathcal{U}(0,1)$.
2 $X \leftarrow (-2\ln U_1)^{1/2}\cos(2\pi U_2)$
3 $Y \leftarrow (-2\ln U_1)^{1/2}\sin(2\pi U_2)$
4 **return** X, Y

■

Once a standard normal number generator is available, simulation from any n-dimensional normal distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is relatively straightforward. The first step is to find an $n \times n$ matrix $\mathbf{B}$ that decomposes $\boldsymbol{\Sigma}$ into the matrix product $\mathbf{B}\mathbf{B}^\top$. In fact there exist many such decompositions. One of the more important ones is the *Cholesky decomposition*, which is a special case of the LU decomposition; see Section A.6.1 for more information on such decompositions. In Python, the function `cholesky` of `numpy.linalg` can be used to produce such a matrix $\mathbf{B}$.

CHOLESKY DECOMPOSITION

☞ 368

Once the Cholesky factorization is determined, it is easy to simulate $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ as, by definition, it is the affine transformation $\boldsymbol{\mu} + \mathbf{B}\boldsymbol{Z}$ of an n-dimensional standard normal random vector.

Algorithm 3.2.2: Normal Random Vector Simulation

input: $\boldsymbol{\mu}, \boldsymbol{\Sigma}$
output: $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$

1 Determine the Cholesky factorization $\boldsymbol{\Sigma} = \mathbf{B}\mathbf{B}^\top$.
2 Simulate $\boldsymbol{Z} = [Z_1, \ldots, Z_n]^\top$ by drawing $Z_1, \ldots, Z_n \sim_{\text{iid}} \mathcal{N}(0,1)$.
3 $\boldsymbol{X} \leftarrow \boldsymbol{\mu} + \mathbf{B}\boldsymbol{Z}$
4 **return** $\boldsymbol{X}$

■ **Example 3.2 (Simulating from a Bivariate Normal Distribution)** The Python code below draws $N = 1000$ iid samples from the two bivariate ($n = 2$) normal pdfs in Figure 2.13. The resulting point clouds are given in Figure 3.1.

☞ 45

bvnormal.py

```
import numpy as np
from numpy.random import randn
import matplotlib.pyplot as plt

N = 1000
r = 0.0   #change to 0.8 for other plot
Sigma = np.array([[1, r], [r, 1]])
```

```
B = np.linalg.cholesky(Sigma)
x = B @ randn(2,N)
plt.scatter([x[0,:]],[x[1,:]], alpha =0.4, s = 4)
```

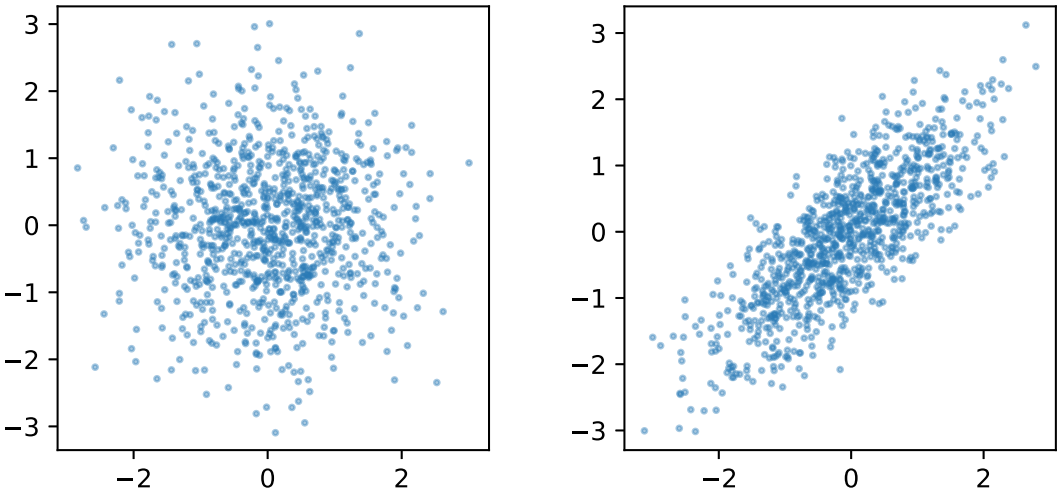


Figure 3.1: 1000 realizations of bivariate normal distributions with means zero, variances 1, and correlation coefficients 0 (left) and 0.8 (right).

■

In some cases, the covariance matrix $\boldsymbol{\Sigma}$ has special structure which can be exploited to create even faster generation algorithms, as illustrated in the following example.

■ **Example 3.3 (Simulating Normal Vectors in $O(n^2)$ Time)** Suppose that the random vector $\boldsymbol{X} = [X_1, \ldots, X_n]^\top$ represents the values at times $t_0 + k\delta$, $k = 0, \ldots, n-1$ of a zero-mean *Gaussian process* $(X(t), t \geqslant 0)$ that is *weakly stationary*, meaning that $\mathbb{C}\mathrm{ov}(X(s), X(t))$ depends only on $t-s$. Then clearly the covariance matrix of $\boldsymbol{X}$, say $\mathbf{A}_n$, is a symmetric Toeplitz matrix. Suppose for simplicity that $\mathbb{V}\mathrm{ar}\, X(t) = 1$. Then the covariance matrix is in fact a correlation matrix, and will have the following structure:

☞ 238
☞ 379

$$\mathbf{A}_n := \begin{bmatrix} 1 & a_1 & \ldots & a_{n-2} & a_{n-1} \\ a_1 & 1 & \ddots & & a_{n-2} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ a_{n-2} & & \ddots & \ddots & a_1 \\ a_{n-1} & a_{n-2} & \cdots & a_1 & 1 \end{bmatrix}.$$

Using the Levinson–Durbin algorithm we can compute a lower diagonal matrix $\mathbf{L}_n$ and a diagonal matrix $\mathbf{D}_n$ in $O(n^2)$ time such that $\mathbf{L}_n \mathbf{A}_n \mathbf{L}_n^\top = \mathbf{D}_n$; see Theorem A.14. If we simulate $\boldsymbol{Z}_n \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_n)$, then the solution $\boldsymbol{X}$ of the linear system:

☞ 383

$$\mathbf{L}_n \boldsymbol{X} = \mathbf{D}_n^{1/2} \boldsymbol{Z}_n$$

has the desired distribution $\mathcal{N}(\mathbf{0}, \mathbf{A}_n)$. The linear system is solved in $O(n^2)$ time via forward substitution. ■

3.2.2.1 Inverse-Transform Method

Let X be a random variable with cumulative distribution function (cdf) F. Let F^{-1} denote the inverse[1] of F and $U \sim \mathcal{U}(0,1)$. Then,

$$\mathbb{P}[F^{-1}(U) \leqslant x] = \mathbb{P}[U \leqslant F(x)] = F(x). \tag{3.2}$$

This leads to the following method to simulate a random variable X with cdf F:

Algorithm 3.2.3: Inverse-Transform Method

input: Cumulative distribution function F.
output: Random variable X distributed according to F.

1 Generate U from $\mathcal{U}(0,1)$.
2 $X \leftarrow F^{-1}(U)$
3 **return** X

The inverse-transform method works both for continuous and discrete distributions. After importing `numpy` as `np`, simulating numbers $0,\ldots,k-1$ according to probabilities $p_0,\ldots,p_{k-1}$ can be done via `np.min(np.where(np.cumsum(p) > np.random.rand()))`, where `p` is the vector of the probabilities.

■ **Example 3.4 (Example 3.1 (cont.))** One remaining issue in Example 3.1 was how to simulate the radius R when we only know its density $f_R(r) = r\,\mathrm{e}^{-r^2/2}, r > 0$. We can use the inverse-transform method for this, but first we need to determine its cdf. The cdf of R is, by integration of the pdf,

$$F_R(r) = 1 - \mathrm{e}^{-\frac{1}{2}r^2}, \quad r > 0,$$

and its inverse is found by solving $u = F_R(r)$ in terms of r, giving

$$F_R^{-1}(u) = \sqrt{-2\ln(1-u)}, \quad u \in (0,1).$$

Thus R has the same distribution as $\sqrt{-2\ln(1-U)}$, with $U \sim \mathcal{U}(0,1)$. Since $1-U$ also has a $\mathcal{U}(0,1)$ distribution, R has also the same distribution as $\sqrt{-2\ln U}$. ■

3.2.2.2 Acceptance–Rejection Method

The acceptance–rejection method is used to sample from a "difficult" probability density function (pdf) $f(x)$ by generating instead from an "easy" pdf $g(x)$ satisfying $f(x) \leqslant C\,g(x)$ for some constant $C \geqslant 1$ (for example, via the inverse-transform method), and then accepting or rejecting the drawn sample with a certain probability. Algorithm 3.2.4 gives the pseudo-code.

The idea of the algorithm is to generate uniformly a point (X, Y) under the graph of the function Cg, by first drawing $X \sim g$ and then $Y \sim \mathcal{U}(0, Cg(X))$. If this point lies under the graph of f, then we accept X as a sample from f; otherwise, we try again. The efficiency of the acceptance–rejection method is usually expressed in terms of the the probability of acceptance, which is $1/C$.

[1] Every cdf has a unique inverse function defined by $F^{-1}(u) = \inf\{x : F(x) \geqslant u\}$. If, for each u, the equation $F(x) = u$ has a unique solution x, this definition coincides with the usual interpretation of the inverse function.

Algorithm 3.2.4: Acceptance–Rejection Method

input: Pdf g and constant C such that $Cg(x) \geqslant f(x)$ for all x.
output: Random variable X distributed according to pdf f.

1 found $\leftarrow$ **false**
2 **while not** found **do**
3 Generate X from g.
4 Generate U from $\mathcal{U}(0,1)$ independently of X.
5 $Y \leftarrow UCg(X)$
6 **if** $Y \leqslant f(X)$ **then** found $\leftarrow$ **true**
7 **return** X

■ **Example 3.5 (Simulating Gamma Random Variables)** Simulating random variables from a Gamma(α, λ) distribution is generally done via the acceptance–rejection method. Consider, for example, the Gamma distribution with $\alpha = 1.3$ and $\lambda = 5.6$. Its pdf, ☞ 425

$$f(x) = \frac{\lambda^\alpha x^{\alpha-1} \mathrm{e}^{-\lambda x}}{\Gamma(\alpha)}, \quad x \geqslant 0,$$

where Γ is the gamma function $\Gamma(\alpha) := \int_0^\infty \mathrm{e}^{-x} x^{\alpha-1}\, \mathrm{d}x$, $\alpha > 0$, is depicted by the blue solid curve in Figure 3.2.

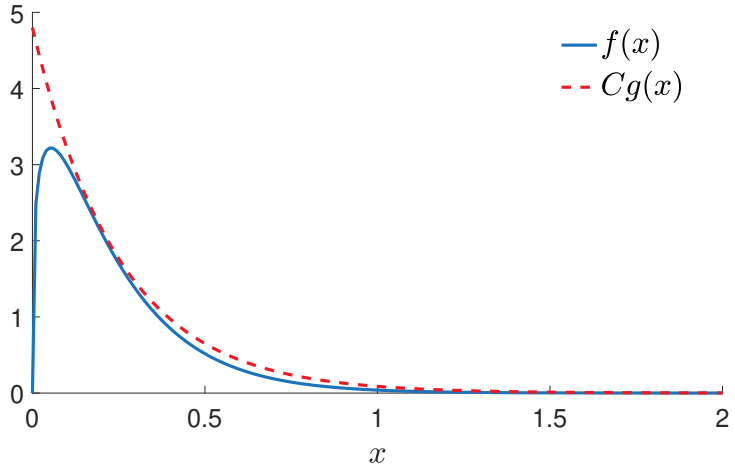


Figure 3.2: The pdf g of the Exp(4) distribution multiplied by $C = 1.2$ dominates the pdf f of the Gamma(1.3, 5.6) distribution.

This pdf happens to lie completely under the graph of $Cg(x)$, where $C = 1.2$ and $g(x) = 4\exp(-4x), x \geqslant 0$ is the pdf of the exponential distribution Exp(4). Hence, we can simulate from this particular Gamma distribution by accepting or rejecting a sample from the Exp(4) distribution according to Step 6 of Algorithm 3.2.4. Simulating from the Exp(4) distribution can be done via the inverse-transform method: simulate $U \sim \mathcal{U}(0,1)$ and return $X = -\ln(U)/4$. The following Python code implements Algorithm 3.2.4 for this example. ☞ 425

accrejgamma.py

```
from math import exp, gamma, log
from numpy.random import rand

alpha = 1.3
lam = 5.6
f = lambda x: lam**alpha * x**(alpha-1) * exp(-lam*x)/gamma(alpha)
g = lambda x: lam*exp(-lam*x)
C = 1.2

found = False
while not found:
    x = - log(rand())/lam
    if C*g(x)*rand() <= f(x):
        found = True

print(x)
```

3.2.3 Simulating Random Vectors and Processes

Techniques for generating random vectors and processes are as diverse as the class of random processes themselves; see, for example, [71]. We highlight a few general scenarios.

When $X_1, \ldots, X_n$ are *independent* random variables with pdfs f_i, $i = 1, \ldots, n$, so that their joint pdf is $f(\boldsymbol{x}) = f_1(x_1) \cdots f_n(x_n)$, the random vector $\boldsymbol{X} = [X_1, \ldots, X_n]^\top$ can be simply simulated by drawing each component $X_i \sim f_i$ individually — for example, via the inverse-transform method or acceptance–rejection.

☞ 429

For *dependent* components $X_1, \ldots, X_n$, we can, as a consequence of the *product rule* of probability, represent the joint pdf $f(\boldsymbol{x})$ as

☞ 431

$$f(\boldsymbol{x}) = f(x_1, \ldots, x_n) = f_1(x_1)\, f_2(x_2 \,|\, x_1) \cdots f_n(x_n \,|\, x_1, \ldots, x_{n-1}), \tag{3.3}$$

where $f_1(x_1)$ is the marginal pdf of X_1 and $f_k(x_k \,|\, x_1, \ldots, x_{k-1})$ is the conditional pdf of X_k given $X_1 = x_1, X_2 = x_2, \ldots, X_{k-1} = x_{k-1}$. Provided the conditional pdfs are known, one can generate $\boldsymbol{X}$ by first generating X_1, then, given $X_1 = x_1$, generate X_2 from $f_2(x_2 \,|\, x_1)$, and so on, until generating X_n from $f_n(x_n \,|\, x_1, \ldots, x_{n-1})$.

The latter method is particularly applicable for generating Markov chains. Recall from Section C.10 that a *Markov chain* is a stochastic process $\{X_t, t = 0, 1, 2, \ldots\}$ that satisfies the *Markov property*; meaning that for all t and s the conditional distribution of X_{t+s} given $X_u, u \leqslant t$, is the same as that of X_{t+s} given only X_t. As a result, each conditional density $f_t(x_t \,|\, x_1, \ldots, x_{t-1})$ can be written as a one-step *transition density* $q_t(x_t \,|\, x_{t-1})$; that is, the probability density to go from state x to state y in one step. In many cases of interest the chain is *time-homogeneous*, meaning that the transition density q_t does not depend on t. Such Markov chains can be generated *sequentially*, as given in Algorithm 3.2.5.

☞ 451

MARKOV CHAIN

Algorithm 3.2.5: Simulate a Markov Chain

input: Number of steps N, initial pdf f_0, transition density q.

1 Draw X_0 from the initial pdf f_0.
2 **for** $t = 1$ **to** N **do**
3 Draw X_t from the distribution corresponding to the density $q(\cdot \,|\, X_{t-1})$
4 **return** $X_0, \ldots, X_N$

■ **Example 3.6 (Markov Chain Simulation)** For time-homogeneous Markov chains with a discrete state space, we can visualize the one-step transitions by means of a *transition graph*, where arrows indicate possible transitions between states and the labels describe the corresponding probabilities. Figure 3.3 shows (on the left) the transition graph of the Markov chain $\{X_t, t = 0, 1, 2, \ldots\}$ with state space $\{1, 2, 3, 4\}$ and one-step transition matrix

TRANSITION GRAPH

$$\mathbf{P} = \begin{bmatrix} 0 & 0.2 & 0.5 & 0.3 \\ 0.5 & 0 & 0.5 & 0 \\ 0.3 & 0.7 & 0 & 0 \\ 0.1 & 0 & 0 & 0.9 \end{bmatrix}.$$

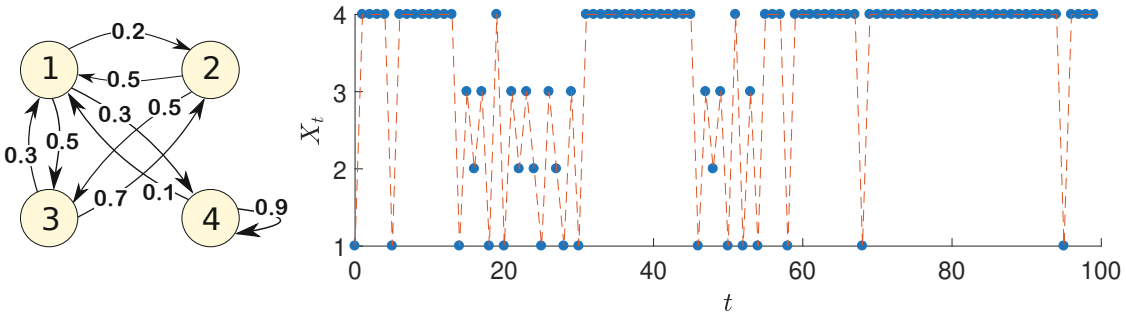


Figure 3.3: The transition graph (left) and a typical path (right) of the Markov chain.

In the same figure (on the right) a typical outcome (path) of the Markov chain is shown. The path was simulated using the Python program below. In this implementation the Markov chain always starts in state 1. We will revisit Markov chains, and in particular Markov chains with continuous state spaces, in Section 3.2.5. ☞ 78

MCsim.py

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

n = 101
P = np.array([[0, 0.2, 0.5, 0.3],
              [0.5, 0, 0.5, 0],
              [0.3, 0.7, 0, 0],
              [0.1, 0, 0, 0.9]])
x = np.array(np.ones(n, dtype=int))
x[0] = 0
for t in range(0,n-1):
```

```
    x[t+1] = np.min(np.where(np.cumsum(P[x[t],:]) >
                     np.random.rand()))
x = x + 1  #add 1 to all elements of the vector x
plt.plot(np.array(range(0,n)),x, 'o')
plt.plot(np.array(range(0,n)),x, '--')
plt.show()
```

■

3.2.4 Resampling

RESAMPLING

The idea behind *resampling* is very simple: an iid sample $\tau := \{x_1, \ldots, x_n\}$ from some unknown cdf F represents our best knowledge of F if we make no further *a priori* assumptions about it. If it is not possible to simulate more samples from F, the best way to "repeat" the experiment is to *resample* from the original data by drawing from the empirical cdf F_n; see (1.2). That is, we draw each x_i with equal probability and repeat this N times, according to Algorithm 3.2.6 below. As we draw here "with replacement", multiple instances of the original data points may occur in the resampled data.

☞ 11

Algorithm 3.2.6: Sampling from an Empirical Cdf.

input: Original iid sample $x_1, \ldots, x_n$ and sample size N.
output: Iid sample $X_1^*, \ldots, X_N^*$ from the empirical cdf.

1 **for** $t = 1$ **to** N **do**
2 Draw $U \sim \mathcal{U}(0, 1)$
3 Set $I \leftarrow \lceil nU \rceil$
4 Set $X_t^* \leftarrow x_I$
5 **return** $X_1^*, \ldots, X_N^*$

In Step 3, $\lceil nU \rceil$ returns the *ceiling* of nU; that is, it is the smallest integer larger than or equal to nU. Consequently, I is drawn uniformly at random from the set of indices $\{1, \ldots, n\}$.

By sampling from the empirical cdf we can thus (approximately) repeat the experiment that gave us the original data as many times as we like. This is useful if we want to assess the properties of certain statistics obtained from the data. For example, suppose that the original data τ gave the statistic $t(\tau)$. By resampling we can gain information about the *distribution* of the corresponding random variable $t(\mathcal{T})$.

■ **Example 3.7 (Quotient of Uniforms)** Let $U_1, \ldots, U_n, V_1, \ldots, V_n$ be iid $\mathcal{U}(0, 1)$ random variables and define $X_i = U_i/V_i$, $i = 1, \ldots, n$. Suppose we wish to investigate the distribution of the sample median $\widetilde{X}$ and sample mean $\overline{X}$ of the (random) data $\mathcal{T} := \{X_1, \ldots, X_n\}$. Since we know the model for $\mathcal{T}$ exactly, we can generate a large number, N say, of independent copies of it, and for each of these copies evaluate the sample medians $\widetilde{X}_1, \ldots, \widetilde{X}_N$ and sample means $\overline{X}_1, \ldots, \overline{X}_N$. For $n = 100$ and $N = 1000$ the empirical cdfs might look like the left and right curves in Figure 3.4, respectively. Contrary to what you might have expected, the distributions of the sample median and sample mean do not match at all. The sample median is quite concentrated around 1, whereas the distribution of the sample mean is much more spread out.

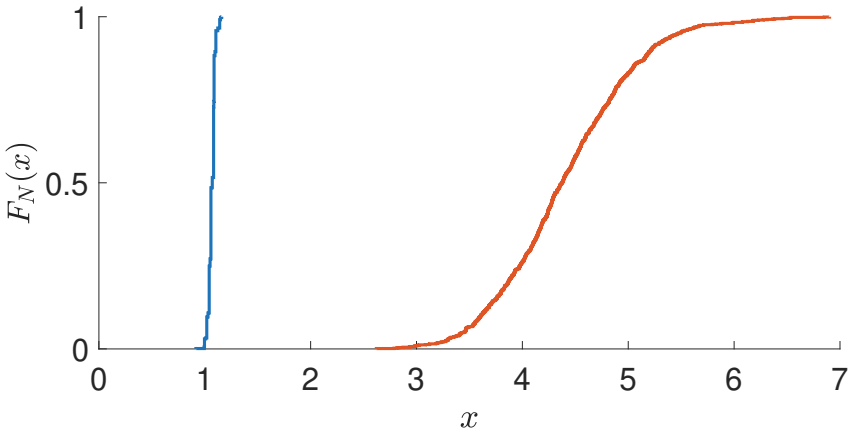


Figure 3.4: Empirical cdfs of the medians of the resampled data (left curve) and sample means (right curve) of the resampled data.

Instead of sampling completely new data, we could also *reuse* the original data by resampling them via Algorithm 3.2.6. This gives independent copies $\widetilde{X}_1^*, \ldots, \widetilde{X}_N^*$ and $\overline{X}_1^*, \ldots, \overline{X}_N^*$, for which we can again plot the empirical cdf. The results will be similar to the previous case. In fact, in Figure 3.4 the cdf of the *resampled* sample medians and sample means are plotted. The corresponding Python code is given below. The essential point of this example is that resampling of data can greatly add to the understanding of the probabilistic properties of certain measurements on the data, *even if the underlying model is not known*. See Exercise 12 for a further investigation of this example.

☞ 116

quotunif.py

```
import numpy as np
from numpy.random import rand, choice
import matplotlib.pyplot as plt
from statsmodels.distributions.empirical_distribution import ECDF

n = 100
N = 1000
x = rand(n)/rand(n)  # data
med = np.zeros(N)
ave = np.zeros(N)
for i in range(0,N):
    s = choice(x, n, replace=True) # resampled data
    med[i] = np.median(s)
    ave[i] = np.mean(s)

med_cdf = ECDF(med)
ave_cdf = ECDF(ave)
plt.plot(med_cdf.x, med_cdf.y)
plt.plot(ave_cdf.x, ave_cdf.y)
plt.show()
```

3.2.5 Markov Chain Monte Carlo

MARKOV CHAIN MONTE CARLO

TARGET

Markov chain Monte Carlo (MCMC) is a Monte Carlo sampling technique for (approximately) generating samples from an arbitrary distribution — often referred to as the *target* distribution. The basic idea is to run a Markov chain long enough such that its limiting distribution is close to the target distribution. Often such a Markov chain is constructed to be reversible, so that the detailed balance equations (C.43) can be used. Depending on the starting position of the Markov chain, the initial random variables in the Markov chain may have a distribution that is significantly different from the target (limiting) distribution. The random variables that are generated during this *burn-in period* are often discarded. The remaining random variables form an *approximate* and *dependent* sample from the target distribution.

☞ 453

BURN-IN PERIOD

In the next two sections we discuss two popular MCMC samplers: the Metropolis–Hastings sampler and the Gibbs sampler.

3.2.5.1 Metropolis–Hastings Sampler

☞ 72

The Metropolis–Hastings sampler [87] is similar to the acceptance–rejection method in that it simulates a trial state, which is then accepted or rejected according to some random mechanism. Specifically, suppose we wish to sample from a target pdf $f(\boldsymbol{x})$, where $\boldsymbol{x}$ takes values in some d-dimensional set. The aim is to construct a Markov chain $\{\boldsymbol{X}_t, t = 0, 1, \ldots\}$ in such a way that its limiting pdf is f. Suppose the Markov chain is in state $\boldsymbol{x}$ at time t. A transition of the Markov chain from state $\boldsymbol{x}$ is carried out in two phases. First a *proposal* state $\boldsymbol{Y}$ is drawn from a transition density $q(\cdot \,|\, \boldsymbol{x})$. This state is accepted as the new state, with *acceptance probability*

PROPOSAL

ACCEPTANCE PROBABILITY

$$\alpha(\boldsymbol{x}, \boldsymbol{y}) = \min\left\{\frac{f(\boldsymbol{y})\, q(\boldsymbol{x} \,|\, \boldsymbol{y})}{f(\boldsymbol{x})\, q(\boldsymbol{y} \,|\, \boldsymbol{x})}, 1\right\}, \tag{3.4}$$

or rejected otherwise. In the latter case the chain remains in state $\boldsymbol{x}$. The algorithm just described can be summarized as follows.

Algorithm 3.2.7: Metropolis–Hastings Sampler

input: Initial state $\boldsymbol{X}_0$, sample size N, target pdf $f(\boldsymbol{x})$, proposal function $q(\boldsymbol{x}, \boldsymbol{y})$.
output: $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N$ (dependent), approximately distributed according to $f(\boldsymbol{x})$.

1 **for** $t = 0$ **to** $N - 1$ **do**
2 Draw $\boldsymbol{Y} \sim q(\boldsymbol{y} \,|\, \boldsymbol{X}_t)$ `// draw a proposal`
3 $\alpha \leftarrow \alpha(\boldsymbol{X}_t, \boldsymbol{Y})$ `// acceptance probability as in (3.4)`
4 Draw $U \sim \mathcal{U}(0, 1)$
5 **if** $U \leqslant \alpha$ **then** $\boldsymbol{X}_{t+1} \leftarrow \boldsymbol{Y}$
6 **else** $\boldsymbol{X}_{t+1} \leftarrow \boldsymbol{X}_t$
7 **return** $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N$

The fact that the limiting distribution of the Metropolis–Hastings Markov chain is equal to the target distribution (under general conditions) is a consequence of the following result.

Theorem 3.1: Local Balance for the Metropolis–Hastings Sampler

The transition density of the Metropolis–Hastings Markov chain satisfies the detailed balance equations.

☞ 453

Proof: We prove the theorem for the discrete case only. Because a transition of the Metropolis–Hastings Markov chain consists of two steps, the one-step transition probability to go from $\boldsymbol{x}$ to $\boldsymbol{y}$ is not $q(\boldsymbol{y} \,|\, \boldsymbol{x})$ but

$$\widetilde{q}(\boldsymbol{y} \,|\, \boldsymbol{x}) = \begin{cases} q(\boldsymbol{y} \,|\, \boldsymbol{x})\, \alpha(\boldsymbol{x}, \boldsymbol{y}), & \text{if } \boldsymbol{y} \neq \boldsymbol{x}, \\ 1 - \sum_{z \neq x} q(\boldsymbol{z} \,|\, \boldsymbol{x})\, \alpha(\boldsymbol{x}, \boldsymbol{z}), & \text{if } \boldsymbol{y} = \boldsymbol{x}. \end{cases} \tag{3.5}$$

We thus need to show that

$$f(\boldsymbol{x})\, \widetilde{q}(\boldsymbol{y} \,|\, \boldsymbol{x}) = f(\boldsymbol{y})\, \widetilde{q}(\boldsymbol{x} \,|\, \boldsymbol{y}) \quad \text{for all } \boldsymbol{x}, \boldsymbol{y}. \tag{3.6}$$

With the acceptance probability as in (3.4), we need to check (3.6) for three cases:

(a) $\boldsymbol{x} = \boldsymbol{y}$,

(b) $\boldsymbol{x} \neq \boldsymbol{y}$ and $f(\boldsymbol{y}) q(\boldsymbol{x} \,|\, \boldsymbol{y}) \leqslant f(\boldsymbol{x}) q(\boldsymbol{y} \,|\, \boldsymbol{x})$, and

(c) $\boldsymbol{x} \neq \boldsymbol{y}$ and $f(\boldsymbol{y}) q(\boldsymbol{x} \,|\, \boldsymbol{y}) > f(\boldsymbol{x}) q(\boldsymbol{y} \,|\, \boldsymbol{x})$.

Case (a) holds trivially. For case (b), $\alpha(\boldsymbol{x}, \boldsymbol{y}) = f(\boldsymbol{y}) q(\boldsymbol{x} \,|\, \boldsymbol{y}) / (f(\boldsymbol{x}) q(\boldsymbol{y} \,|\, \boldsymbol{x}))$ and $\alpha(\boldsymbol{y}, \boldsymbol{x}) = 1$. Consequently,

$$\widetilde{q}(\boldsymbol{y} \,|\, \boldsymbol{x}) = f(\boldsymbol{y}) q(\boldsymbol{x} \,|\, \boldsymbol{y}) / f(\boldsymbol{x}) \quad \text{and} \quad \widetilde{q}(\boldsymbol{x} \,|\, \boldsymbol{y}) = q(\boldsymbol{x} \,|\, \boldsymbol{y}),$$

so that (3.6) holds. Similarly, for case (c) we have $\alpha(\boldsymbol{x}, \boldsymbol{y}) = 1$ and $\alpha(\boldsymbol{y}, \boldsymbol{x}) = f(\boldsymbol{x}) q(\boldsymbol{y} \,|\, \boldsymbol{x}) / (f(\boldsymbol{y}) q(\boldsymbol{x} \,|\, \boldsymbol{y}))$. It follows that,

$$\widetilde{q}(\boldsymbol{y} \,|\, \boldsymbol{x}) = q(\boldsymbol{y} \,|\, \boldsymbol{x}) \quad \text{and} \quad \widetilde{q}(\boldsymbol{x} \,|\, \boldsymbol{y}) = f(\boldsymbol{x}) q(\boldsymbol{y} \,|\, \boldsymbol{x}) / f(\boldsymbol{y}),$$

so that (3.6) holds again. □

Thus if the Metropolis–Hastings Markov chain is ergodic, then its limiting pdf is $f(\boldsymbol{x})$. ☞ 452 A fortunate property of the algorithm, which is important in many applications, is that in order to evaluate the acceptance probability $\alpha(\boldsymbol{x}, \boldsymbol{y})$ in (3.4), one only needs to know the target pdf $f(\boldsymbol{x})$ *up to a constant*; that is $f(\boldsymbol{x}) = c\, \overline{f}(\boldsymbol{x})$ for some known function $\overline{f}(\boldsymbol{x})$ but unknown constant c.

The efficiency of the algorithm depends of course on the choice of the proposal transition density $q(\boldsymbol{y} \,|\, \boldsymbol{x})$. Ideally, we would like $q(\boldsymbol{y} \,|\, \boldsymbol{x})$ to be "close" to the target $f(\boldsymbol{y})$, irrespective of $\boldsymbol{x}$. We discuss two common approaches.

1. Choose the proposal transition density $q(\boldsymbol{y} \,|\, \boldsymbol{x})$ independent of $\boldsymbol{x}$; that is, $q(\boldsymbol{y} \,|\, \boldsymbol{x}) = g(\boldsymbol{y})$ for some pdf $g(\boldsymbol{y})$. An MCMC sampler of this type is called an *independence sampler*. The acceptance probability is thus

INDEPENDENCE SAMPLER

$$\alpha(\boldsymbol{x}, \boldsymbol{y}) = \min \left\{ \frac{f(\boldsymbol{y})\, g(\boldsymbol{x})}{f(\boldsymbol{x})\, g(\boldsymbol{y})},\ 1 \right\}.$$

2. If the proposal transition density is symmetric (that is, $q(\boldsymbol{y}\,|\,\boldsymbol{x}) = q(\boldsymbol{x}\,|\,\boldsymbol{y})$), then the acceptance probability has the simple form

$$\alpha(\boldsymbol{x}, \boldsymbol{y}) = \min\left\{\frac{f(\boldsymbol{y})}{f(\boldsymbol{x})},\ 1\right\}, \tag{3.7}$$

RANDOM WALK SAMPLER

and the MCMC algorithm is called a *random walk sampler*. A typical example is when, for a given current state $\boldsymbol{x}$, the proposal state $\boldsymbol{Y}$ is of the form $\boldsymbol{Y} = \boldsymbol{x} + \boldsymbol{Z}$, where $\boldsymbol{Z}$ is generated from some spherically symmetric distribution, such as $\mathcal{N}(\mathbf{0}, \mathbf{I})$.

We now give an example illustrating the second approach.

■ **Example 3.8 (Random Walk Sampler)** Consider the two-dimensional pdf

$$f(x_1, x_2) = c\, \mathrm{e}^{-\frac{1}{4}\sqrt{x_1^2+x_2^2}}\left(\sin\left(2\sqrt{x_1^2+x_2^2}\right)+1\right), \quad -2\pi < x_1 < 2\pi,\ -2\pi < x_2 < 2\pi, \tag{3.8}$$

where c is an unknown normalization constant. The graph of this pdf (unnormalized) is depicted in the left panel of Figure 3.5.

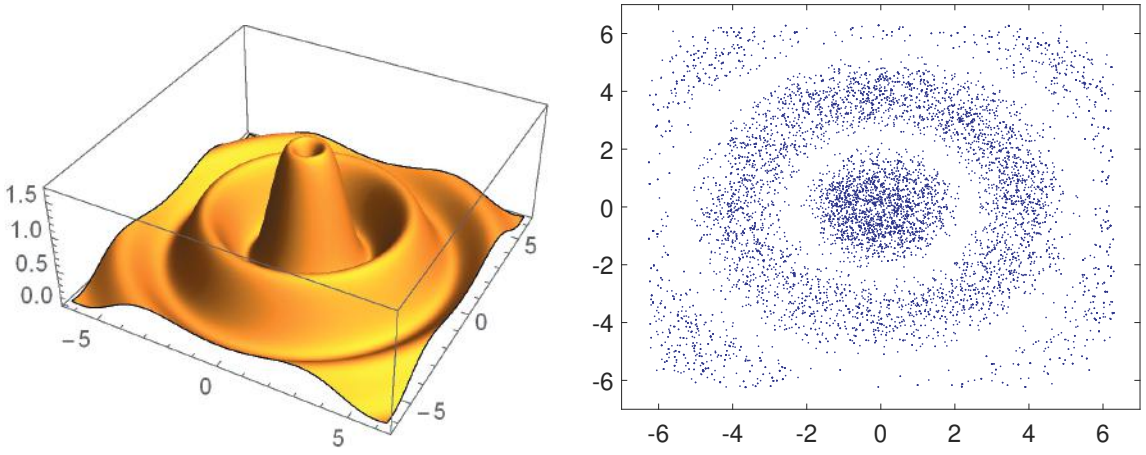


Figure 3.5: Left panel: the two-dimensional target pdf. Right panel: points from the random walk sampler are approximately distributed according to the target pdf.

The following Python program implements a random walk sampler to (approximately) draw $N = 10^4$ dependent samples from the pdf f. At each step, given a current state $\boldsymbol{x}$, a proposal $\boldsymbol{Y}$ is drawn from the $\mathcal{N}(\boldsymbol{x}, \mathbf{I})$ distribution. That is, $\boldsymbol{Y} = \boldsymbol{x} + \boldsymbol{Z}$, with $\boldsymbol{Z}$ bivariate standard normal. We see in the right panel of Figure 3.5 that the sampler works correctly. The starting point for the Markov chain is chosen as $(0, 0)$. Note that the normalization constant c is never required to be specified in the program.

rwsamp.py

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import pi, exp, sqrt, sin
from numpy.random import rand, randn

N = 10000
```

```
a = lambda x: -2*pi < x
b = lambda x: x < 2*pi
f = lambda x1, x2: exp(-sqrt(x1**2+x2**2)/4)*(
        sin(2*sqrt(x1**2+x2**2))+1)*a(x1)*b(x1)*a(x2)*b(x2)

xx = np.zeros((N,2))
x = np.zeros((1,2))
for i in range(1,N):
    y = x + randn(1,2)
    alpha = np.amin((f(y[0][0],y[0][1])/f(x[0][0],x[0][1]),1))
    r = rand() < alpha
    x = r*y + (1-r)*x
    xx[i,:] = x

plt.scatter(xx[:,0], xx[:,1], alpha =0.4,s =2)
plt.axis('equal')
plt.show()
```

3.2.5.2 Gibbs Sampler

GIBBS SAMPLER

The *Gibbs sampler* [48] uses a somewhat different methodology from the Metropolis–Hastings algorithm and is particularly useful for generating n-dimensional random vectors. The key idea of the Gibbs sampler is to update the components of the random vector one at a time, by sampling them from *conditional* pdfs. Thus, Gibbs sampling can be advantageous if it is easier to sample from the conditional distributions than from the joint distribution.

Specifically, suppose that we wish to sample a random vector $\boldsymbol{X} = [X_1, \ldots, X_n]^\top$ according to a target pdf $f(\boldsymbol{x})$. Let $f(x_i \,|\, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ represent the conditional pdf[2] of the i-th component, X_i, given the other components $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$. The Gibbs sampling algorithm is as follows.

Algorithm 3.2.8: Gibbs Sampler

input: Initial point $\boldsymbol{X}_0$, sample size N, and target pdf f.
output: $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N$ approximately distributed according to f.

1 **for** $t = 0$ **to** $N - 1$ **do**
2 Draw Y_1 from the conditional pdf $f(y_1 \,|\, X_{t,2}, \ldots, X_{t,n})$.
3 **for** $i = 2$ **to** n **do**
4 Draw Y_i from the conditional pdf $f(y_i \,|\, Y_1, \ldots, Y_{i-1}, X_{t,i+1}, \ldots, X_{t,n})$.
5 $\boldsymbol{X}_{t+1} \leftarrow \boldsymbol{Y}$
6 **return** $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N$

There exist many variants of the Gibbs sampler, depending on the steps required to update $\boldsymbol{X}_t$ to $\boldsymbol{X}_{t+1}$ — called the *cycle* of the Gibbs algorithm. In the algorithm above, the cycle consists of Steps 2–5, in which the components are updated in a fixed order $1 \to 2 \to \cdots \to n$. For this reason Algorithm 3.2.8 is also called the *systematic Gibbs sampler*.

CYCLE

SYSTEMATIC GIBBS SAMPLER

[2] In this section we employ a Bayesian notation style, using the same letter f for different (conditional) densities.

RANDOM-ORDER GIBBS SAMPLER
☞ 115

RANDOM GIBBS SAMPLER
REVERSIBLE GIBBS SAMPLER
☞ 452

In the *random-order Gibbs sampler*, the order in which the components are updated in each cycle is a random permutation of $\{1, \ldots, n\}$ (see Exercise 9). Other modifications are to update the components in blocks (i.e., several at the same time), or to update only a random selection of components. The variant where in each cycle only a single random component is updated is called the *random Gibbs sampler*. In the *reversible Gibbs sampler* a cycle consists of the coordinate-wise updating $1 \to 2 \to \cdots \to n-1 \to n \to n-1 \to \cdots \to 2 \to 1$. In all cases, except for the systematic Gibbs sampler, the resulting Markov chain $\{\boldsymbol{X}_t, t = 1, 2, \ldots\}$ is *reversible* and hence its limiting distribution is precisely $f(\boldsymbol{x})$.

Unfortunately, the systematic Gibbs Markov chain is not reversible and so the detailed balance equations are not satisfied. However, a similar result holds, due to Hammersley and Clifford, under the so-called *positivity condition*: if at a point $\boldsymbol{x} = (x_1, \ldots, x_n)$ all marginal densities $f(x_i) > 0, i = 1, \ldots, n$, then the joint density $f(\boldsymbol{x}) > 0$.

Theorem 3.2: Hammersley–Clifford Balance for the Gibbs Sampler

Let $q_{1\to n}(\boldsymbol{y} \,|\, \boldsymbol{x})$ denote the transition density of the systematic Gibbs sampler, and let $q_{n\to 1}(\boldsymbol{x} \,|\, \boldsymbol{y})$ be the transition density of the reverse move, in the order $n \to n-1 \to \cdots \to 1$. Then, if the positivity condition holds,

$$f(\boldsymbol{x})\, q_{1\to n}(\boldsymbol{y} \,|\, \boldsymbol{x}) = f(\boldsymbol{y})\, q_{n\to 1}(\boldsymbol{x} \,|\, \boldsymbol{y}). \tag{3.9}$$

Proof: For the forward move we have:

$$q_{1\to n}(\boldsymbol{y} \,|\, \boldsymbol{x}) = f(y_1 \,|\, x_2, \ldots, x_n) f(y_2 \,|\, y_1, x_3, \ldots, x_n) \cdots f(y_n \,|\, y_1, \ldots, y_{n-1}),$$

and for the reverse move:

$$q_{n\to 1}(\boldsymbol{x} \,|\, \boldsymbol{y}) = f(x_n \,|\, y_1, \ldots, y_{n-1}) f(x_{n-1} \,|\, y_1, \ldots, y_{n-2}, x_n) \cdots f(x_1 \,|\, x_2, \ldots, x_n).$$

Consequently,

$$\begin{aligned}
\frac{q_{1\to n}(\boldsymbol{y} \,|\, \boldsymbol{x})}{q_{n\to 1}(\boldsymbol{x} \,|\, \boldsymbol{y})} &= \prod_{i=1}^{n} \frac{f(y_i \,|\, y_1, \ldots, y_{i-1}, x_{i+1}, \ldots, x_n)}{f(x_i \,|\, y_1, \ldots, y_{i-1}, x_{i+1}, \ldots, x_n)} \\
&= \prod_{i=1}^{n} \frac{f(y_1, \ldots, y_i, x_{i+1}, \ldots, x_n)}{f(y_1, \ldots, y_{i-1}, x_i, \ldots, x_n)} \\
&= \frac{f(\boldsymbol{y}) \prod_{i=1}^{n-1} f(y_1, \ldots, y_i, x_{i+1}, \ldots, x_n)}{f(\boldsymbol{x}) \prod_{j=2}^{n} f(y_1, \ldots, y_{j-1}, x_j, \ldots, x_n)} \\
&= \frac{f(\boldsymbol{y}) \prod_{i=1}^{n-1} f(y_1, \ldots, y_i, x_{i+1}, \ldots, x_n)}{f(\boldsymbol{x}) \prod_{j=1}^{n-1} f(y_1, \ldots, y_j, x_{j+1}, \ldots, x_n)} = \frac{f(\boldsymbol{y})}{f(\boldsymbol{x})}.
\end{aligned}$$

The result follows by rearranging the last identity. The positivity condition ensures that we do not divide by 0 along the line. □

Intuitively, the long-run proportion of transitions $\boldsymbol{x} \to \boldsymbol{y}$ for the "forward move" chain is equal to the long-run proportion of transitions $\boldsymbol{y} \to \boldsymbol{x}$ for the "reverse move" chain. To verify that the Markov chain $\boldsymbol{X}_0, \boldsymbol{X}_1, \ldots$ for the systematic Gibbs sampler indeed has

limiting pdf $f(\boldsymbol{x})$, we need to check that the global balance equations (C.42) hold. By integrating (in the continuous case) both sides in (3.9) with respect to $\boldsymbol{x}$, we see that indeed ☞ 452

$$\int f(\boldsymbol{x})\, q_{1\to n}(\boldsymbol{y}\,|\,\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = f(\boldsymbol{y}).$$

■ **Example 3.9 (Gibbs Sampler for the Bayesian Normal Model)** Gibbs samplers are often applied in Bayesian statistics, to sample from the posterior pdf. Consider for instance the Bayesian normal model ☞ 50

$$\begin{aligned} f(\mu, \sigma^2) &= 1/\sigma^2 \\ (\boldsymbol{x}\,|\,\mu, \sigma^2) &\sim \mathcal{N}(\mu\mathbf{1}, \sigma^2\mathbf{I}). \end{aligned}$$

Here the prior for (μ, σ^2) is *improper*. That is, it is not a pdf in itself, but by obstinately applying Bayes' formula it does yield a proper posterior pdf. In some sense this prior conveys the least amount of information about μ and σ^2. Following the same procedure as in Example 2.8, we find the posterior pdf: IMPROPER PRIOR

$$f(\mu, \sigma^2\,|\,\boldsymbol{x}) \propto \left(\sigma^2\right)^{-n/2-1} \exp\left\{-\frac{1}{2}\frac{\sum_i (x_i-\mu)^2}{\sigma^2}\right\}. \tag{3.10}$$

Note that μ and σ^2 here are the "variables" and $\boldsymbol{x}$ is a fixed data vector. To simulate samples μ and σ^2 from (3.10) using the Gibbs sampler, we need the distributions of both $(\mu\,|\,\sigma^2, \boldsymbol{x})$ and $(\sigma^2\,|\,\mu, \boldsymbol{x})$. To find $f(\mu\,|\,\sigma^2, \boldsymbol{x})$, view the right-hand side of (3.10) as a function of μ only, regarding σ^2 as a constant. This gives

$$\begin{aligned} f(\mu\,|\,\sigma^2, \boldsymbol{x}) &\propto \exp\left\{-\frac{n\mu^2 - 2\mu\sum_i x_i}{2\sigma^2}\right\} = \exp\left\{-\frac{\mu^2 - 2\mu\bar{x}}{2(\sigma^2/n)}\right\} \\ &\propto \exp\left\{-\frac{1}{2}\frac{(\mu-\bar{x})^2}{\sigma^2/n}\right\}. \end{aligned} \tag{3.11}$$

This shows that $(\mu\,|\,\sigma^2, \boldsymbol{x})$ has a normal distribution with mean $\bar{x}$ and variance σ^2/n.

Similarly, to find $f(\sigma^2\,|\,\mu, \boldsymbol{x})$, view the right-hand side of (3.10) as a function of σ^2, regarding μ as a constant. This gives

$$f(\sigma^2\,|\,\mu, \boldsymbol{x}) \propto (\sigma^2)^{-n/2-1} \exp\left\{-\frac{1}{2}\sum_{i=1}^{n}(x_i-\mu)^2/\sigma^2\right\}, \tag{3.12}$$

showing that $(\sigma^2\,|\,\mu, \boldsymbol{x})$ has an inverse-gamma distribution with parameters $n/2$ and $\sum_{i=1}^n (x_i-\mu)^2/2$. The Gibbs sampler thus involves the repeated simulation of ☞ 425

$$(\mu\,|\,\sigma^2, \boldsymbol{x}) \sim \mathcal{N}\left(\bar{x},\, \sigma^2/n\right) \quad \text{and} \quad (\sigma^2\,|\,\mu, \boldsymbol{x}) \sim \mathsf{InvGamma}\left(n/2,\, \sum_{i=1}^{n}(x_i-\mu)^2/2\right).$$

Simulating $X \sim \mathsf{InvGamma}(\alpha, \lambda)$ is achieved by first generating $Z \sim \mathsf{Gamma}(\alpha, \lambda)$ and then returning $X = 1/Z$.

In our parameterization of the Gamma(α, λ) distribution, λ is the *rate* parameter. Many software packages instead use the *scale* parameter $c = 1/\lambda$. Be aware of this when simulating Gamma random variables.

The Python script below defines a small data set of size $n = 10$ (which was randomly simulated from a standard normal distribution), and implements the systematic Gibbs sampler to simulate from the posterior distribution, using $N = 10^5$ samples.

gibbsamp.py

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

x = np.array([[-0.9472, 0.5401, -0.2166, 1.1890, 1.3170,
               -0.4056, -0.4449, 1.3284, 0.8338, 0.6044]])
n=x.size
sample_mean = np.mean(x)
sample_var = np.var(x)
sig2 = np.var(x)
mu=sample_mean

N=10**5
gibbs_sample = np.array(np.zeros((N, 2)))
for k in range(N):
    mu=sample_mean + np.sqrt(sig2/n)*np.random.randn()
    V=np.sum((x-mu)**2)/2
    sig2 = 1/np.random.gamma(n/2, 1/V)
    gibbs_sample[k,:]= np.array([mu, sig2])
plt.scatter(gibbs_sample[:,0], gibbs_sample[:,1],alpha =0.1,s =1)
plt.plot(np.mean(x), np.var(x),'wo')
plt.show()
```

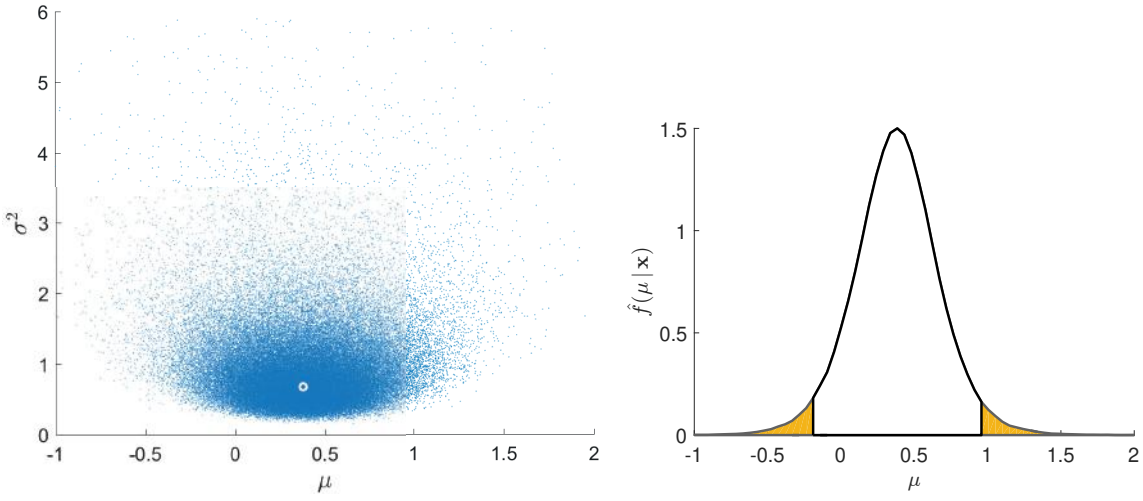


Figure 3.6: Left: approximate draws from the posterior pdf $f(\mu, \sigma^2 \,|\, \boldsymbol{x})$ obtained via the Gibbs sampler. Right: estimate of the posterior pdf $f(\mu \,|\, \boldsymbol{x})$.

The left panel of Figure 3.6 shows the (μ, σ^2) points generated by the Gibbs sampler. Also shown, via the white circle, is the point $(\overline{x}, s^2)$, where $\overline{x} = 0.3798$ is the sample mean and $s^2 = 0.6810$ the sample variance. This posterior point cloud visualizes the considerable uncertainty in the estimates. By projecting the (μ, σ^2) points onto the μ-axis — that is, by ignoring the σ^2 values — one obtains (approximate) samples from the posterior pdf of μ; that is, $f(\mu \,|\, \boldsymbol{x})$. The right panel of Figure 3.6 shows a kernel density estimate (see Section 4.4) of this pdf. The corresponding 0.05 and 0.95 sample quantiles were found to be -0.2054 and 0.9662, respectively, giving the 95% credible interval $(-0.2054, 0.9662)$ for μ, which contains the true expectation 0. Similarly, an estimated 95% credible interval for σ^2 is $(0.3218, 2.2485)$, which contains the true variance 1.

☞ 134

3.3 Monte Carlo Estimation

In this section we describe how Monte Carlo simulation can be used to estimate complicated integrals, probabilities, and expectations. A number of variance reduction techniques are introduced as well, including the recent cross-entropy method.

3.3.1 Crude Monte Carlo

The most common setting for Monte Carlo estimation is the following: Suppose we wish to compute the expectation $\mu = \mathbb{E}Y$ of some (say continuous) random variable Y with pdf f, but the integral $\mathbb{E}Y = \int y f(y)\, \mathrm{d}y$ is difficult to evaluate. For example, if Y is a complicated function of other random variables, it would be difficult to obtain an exact expression for $f(y)$. The idea of *crude Monte Carlo* — sometimes abbreviated as CMC — is to approximate μ by simulating many independent copies $Y_1, \ldots, Y_N$ of Y and then take their *sample mean* $\overline{Y}$ as an estimator of μ. All that is needed is an algorithm to simulate such copies.

CRUDE MONTE CARLO

By the Law of Large Numbers, $\overline{Y}$ converges to μ as $N \to \infty$, provided the expectation of Y exists. Moreover, by the Central Limit Theorem, $\overline{Y}$ approximately has a $\mathcal{N}(\mu, \sigma^2/N)$ distribution for large N, provided that the variance $\sigma^2 = \mathbb{V}\mathrm{ar}Y < \infty$. This enables the construction of an approximate $(1-\alpha)$ *confidence interval* for μ:

☞ 446
☞ 447

CONFIDENCE INTERVAL

$$\left(\overline{Y} - z_{1-\alpha/2} \frac{S}{\sqrt{N}}, \quad \overline{Y} + z_{1-\alpha/2} \frac{S}{\sqrt{N}} \right), \tag{3.13}$$

where S is the sample standard deviation of the $\{Y_i\}$ and z_γ denotes the γ-quantile of the $\mathcal{N}(0,1)$ distribution; see also Section C.13. Instead of specifying the confidence interval, one often reports only the sample mean and the *estimated standard error*: $S/\sqrt{N}$, or the *estimated relative error*: $S/(\overline{Y}\sqrt{N})$. The basic estimation procedure for independent data is summarized in Algorithm 3.3.1 below.

☞ 457

ESTIMATED STANDARD ERROR

ESTIMATED RELATIVE ERROR

It is often the case that the output Y is a function of some underlying random vector or stochastic process; that is, $Y = H(\boldsymbol{X})$, where H is a real-valued function and $\boldsymbol{X}$ is a random vector or process. The beauty of Monte Carlo for estimation is that (3.13) holds regardless of the dimension of $\boldsymbol{X}$.

Algorithm 3.3.1: Crude Monte Carlo for Independent Data

input: Simulation algorithm for $Y \sim f$, sample size N, confidence level $1-\alpha$.
output: Point estimate and approximate $(1-\alpha)$ confidence interval for $\mu = \mathbb{E}Y$.

1 Simulate $Y_1, \ldots, Y_N \overset{\text{iid}}{\sim} f$.
2 $\overline{Y} \leftarrow \frac{1}{N}\sum_{i=1}^{N} Y_i$
3 $S^2 \leftarrow \frac{1}{N-1}\sum_{i=1}^{N}(Y_i - \overline{Y})^2$
4 **return** $\overline{Y}$ and the interval (3.13).

MONTE CARLO INTEGRATION

■ **Example 3.10 (Monte Carlo Integration)** In *Monte Carlo integration*, simulation is used to evaluate complicated integrals. Consider, for example, the integral

$$\mu = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \sqrt{|x_1 + x_2 + x_3|}\, \mathrm{e}^{-(x_1^2+x_2^2+x_3^2)/2}\, \mathrm{d}x_1\, \mathrm{d}x_2\, \mathrm{d}x_3.$$

Defining $Y = |X_1 + X_2 + X_3|^{1/2}(2\pi)^{3/2}$, with $X_1, X_2, X_3 \overset{\text{iid}}{\sim} \mathcal{N}(0,1)$, we can write $\mu = \mathbb{E}Y$. Using the following Python program, with a sample size of $N = 10^6$, we obtained an estimate $\overline{Y} = 17.031$ with an approximate 95% confidence interval $(17.017, 17.046)$.

mcint.py

```
import numpy as np
from numpy import pi

c = (2*pi)**(3/2)
H = lambda x: c*np.sqrt(np.abs(np.sum(x,axis=1)))
N = 10**6
z = 1.96
x = np.random.randn(N,3)
y = H(x)
mY = np.mean(y)
sY = np.std(y)
RE = sY/mY/np.sqrt(N)
print('Estimate = {:3.3f}, CI = ({:3.3f},{:3.3f})'.format(
        mY, mY*(1-z*RE), mY*(1+z*RE)))
```

```
Estimate = 17.031, CI = (17.017,17.046)
```

■

■ **Example 3.11 (Example 2.1 (cont.))** We return to the bias–variance tradeoff in Example 2.1. Figure 2.7 gives estimates of the (squared-error) generalization risk (2.5) as a function of the number of parameters in the model. But how accurate are these estimates? Because we know in this case the exact model for the data, we can use Monte Carlo simulation to estimate the generalization risk (for a fixed training set) and the expected generalization risk (averaged over all training sets) precisely. All we need to do is repeat the data generation, fitting, and validation steps many times and then take averages of the results. The following Python code repeats 100 times:

☞ 26 ☞ 29 ☞ 23

1. Simulate the training set of size $n = 100$.
2. Fit models up to size $k = 8$.

3. Estimate the test loss using a test set with the same sample size $n = 100$.

Figure 3.7 shows that there is some variation in the test losses, due to the randomness in both the training and test sets. To obtain an accurate estimate of the expected generalization risk (2.6), take the average of the test losses. We see that for $k \leqslant 8$ the estimate in Figure 2.7 is close to the true expected generalization risk.

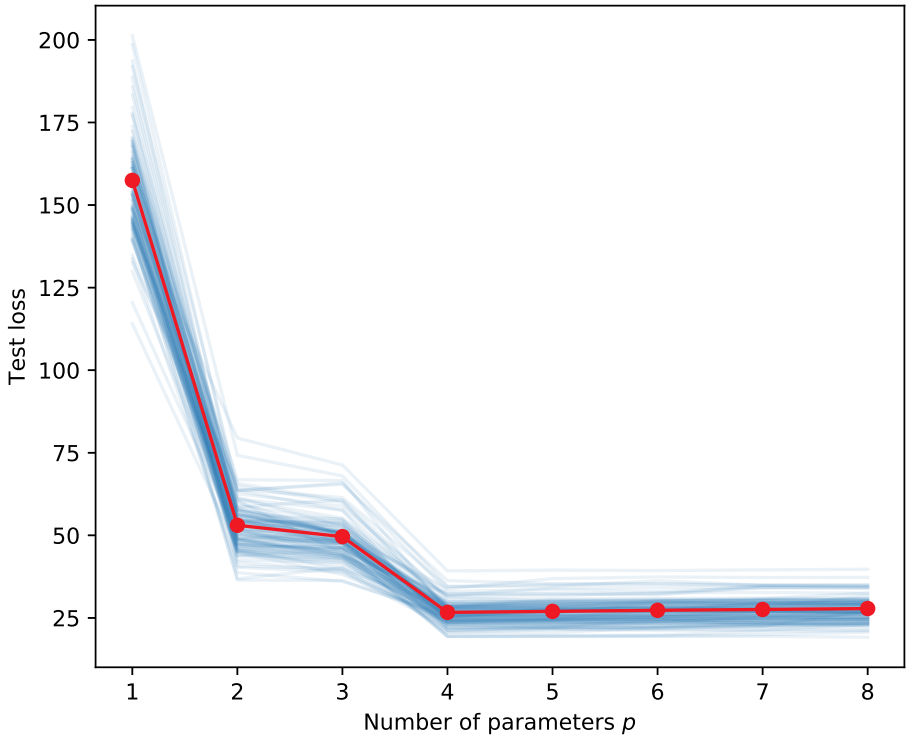


Figure 3.7: Independent estimates of the test loss show some variability.

CMCtestloss.py

```
import numpy as np, matplotlib.pyplot as plt
from numpy.random import rand, randn
from numpy.linalg import solve

def generate_data(beta, sig, n):
    u = rand(n, 1)
    y = (u ** np.arange(0, 4)) @ beta + sig * randn(n, 1)
    return u, y

beta = np.array([[10, -140, 400, -250]]).T
n = 100
sig = 5
betahat = {}
plt.figure(figsize=[6,5])
totMSE = np.zeros(8)
max_p = 8
p_range = np.arange(1, max_p + 1, 1)

for N in range(0,100):
```

```
    u, y = generate_data(beta, sig, n)  #training data
    X = np.ones((n, 1))
    for p in p_range:
       if p > 1:
         X = np.hstack((X, u**(p-1)))
       betahat[p] = solve(X.T @ X, X.T @ y)

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

    totMSE = totMSE + np.array(MSE)
    plt.plot(p_range, MSE,'C0',alpha=0.1)

plt.plot(p_range,totMSE/N,'r-o')
plt.xticks(ticks=p_range)
plt.xlabel('Number of parameters $p$')
plt.ylabel('Test loss')
plt.tight_layout()
plt.savefig('MSErepeatpy.pdf',format='pdf')
plt.show()
```

3.3.2 Bootstrap Method

The bootstrap method [37] combines CMC estimation with the resampling procedure of
☞ 76 Section 3.2.4. The idea is as follows: Suppose we wish to estimate a number μ via some estimator $Y = H(\mathcal{T})$, where $\mathcal{T} := \{X_1, \ldots, X_n\}$ is an iid sample from some unknown cdf F. It is assumed that Y does not depend on the order of the $\{X_i\}$. To assess the quality (for example, accuracy) of the estimator Y, one could draw independent replications $\mathcal{T}_1, \ldots, \mathcal{T}_N$ of $\mathcal{T}$ and find sample estimates for quantities such as the variance $\mathbb{V}\mathrm{ar}Y$, the bias $\mathbb{E}Y - \mu$, and the mean squared error $\mathbb{E}(Y - \mu)^2$. However, it may be too time-consuming or simply not feasible to obtain such replications. An alternative is to *resample* the original data. To reiterate, given an outcome $\tau = \{x_1, \ldots, x_n\}$ of $\mathcal{T}$, we simulate an iid sample $\mathcal{T}^* := \{X_1^*, \ldots, X_n^*\}$ from the empirical cdf F_n, via Algorithm 3.2.6 (hence the resampling size is $N = n$ here).
☞ 76

The rationale is that the empirical cdf F_n is close to the actual cdf F and gets closer as n gets larger. Hence, any quantities depending on F, such as $\mathbb{E}_F g(Y)$, where g is a function, can be approximated by $\mathbb{E}_{F_n} g(Y)$. The latter is usually still difficult to evaluate, but it can be simply estimated via CMC as

$$\frac{1}{K} \sum_{i=1}^{K} g(Y_i^*),$$

where $Y_1^*, \ldots, Y_K^*$ are independent random variables, each distributed as $Y^* = H(\mathcal{T}^*)$. This seemingly self-referent procedure is called *bootstrapping* — alluding to Baron von Mün-

BOOTSTRAPPING

chausen, who pulled himself out of a swamp by his own bootstraps. As an example, the bootstrap estimate of the expectation of Y is

$$\widehat{\mathbb{E}Y} = \overline{Y}^* = \frac{1}{K}\sum_{i=1}^{K} Y_i^*,$$

which is simply the sample mean of $\{Y_i^*\}$. Similarly, the bootstrap estimate for $\mathbb{V}\mathrm{ar}Y$ is the sample variance

$$\widehat{\mathbb{V}\mathrm{ar}Y} = \frac{1}{K-1}\sum_{i=1}^{K}(Y_i^* - \overline{Y}^*)^2. \tag{3.14}$$

Bootstrap estimators for the bias and MSE are $\overline{Y}^* - Y$ and $\frac{1}{K}\sum_{i=1}^{K}(Y_i^* - Y)^2$, respectively. Note that for these estimators the unknown quantity μ is replaced with its original estimator Y. Confidence intervals can be constructed in the same fashion. We mention two variants: the *normal method* and the *percentile method*. In the normal method, a $1-\alpha$ confidence interval for μ is given by

NORMAL METHOD

PERCENTILE METHOD

$$(Y \pm z_{1-\alpha/2} S^*),$$

where S^* is the bootstrap estimate of the standard deviation of Y; that is, the square root of (3.14). In the percentile method, the upper and lower bounds of the $1-\alpha$ confidence interval for μ are given by the $1-\alpha/2$ and $\alpha/2$ quantiles of Y, which in turn are estimated via the corresponding sample quantiles of the bootstrap sample $\{Y_i^*\}$.

The following example illustrates the usefulness of the bootstrap method for *ratio estimation* and also introduces the *renewal reward process* model for data.

■ **Example 3.12 (Bootstrapping the Ratio Estimator)** A common scenario in stochastic simulation is that the output of the simulation consists of independent pairs of data $(C_1, R_1), (C_2, R_2), \ldots$, where each C is interpreted as the length of a period of time — a so-called *cycle* — and R is the *reward* obtained during that cycle. Such a collection of random variables $\{(C_i, R_i)\}$ is called a *renewal reward process*. Typically, the reward R_i depends on the cycle length C_i. Let A_t be the *average reward* earned by time t; that is, $A_t = \sum_{i=1}^{N_t} R_i/t$, where $N_t = \max\{n : C_1 + \cdots + C_n \leqslant t\}$ counts the number of complete cycles at time t. It can be shown, see Exercise 20, that if the expectations of the cycle length and reward are finite, then A_t converges to the constant $\mathbb{E}R/\mathbb{E}C$. This ratio can thus be interpreted as the *long-run average reward*.

RENEWAL REWARD PROCESS

☞ 118

LONG-RUN AVERAGE REWARD

Estimation of the ratio $\mathbb{E}R/\mathbb{E}C$ from data $(C_1, R_1), \ldots, (C_n, R_n)$ is easy: take the *ratio estimator*

RATIO ESTIMATOR

$$A = \frac{\overline{R}}{\overline{C}}.$$

However, this estimator A is not unbiased and it is not obvious how to derive confidence intervals. Fortunately, the bootstrap method can come to the rescue: simply resample the pairs $\{(C_i, R_i)\}$, obtain ratio estimators $A_1^*, \ldots, A_K^*$, and from these compute quantities of interest such as confidence intervals.

As a concrete example, let us return to the Markov chain in Example 3.6. Recall that the chain starts at state 1 at time 0. After a certain amount of time T_1, the process returns to state 1. The time steps $0, \ldots, T_1 - 1$ form a natural "cycle" for this process, as from time T_1 onwards the process behaves probabilistically *exactly the same* as when it started,

☞ 75

independently of $X_0, \ldots, X_{T_1-1}$. Thus, if we define $T_0 = 0$, and let T_i be the i-th time that the chain returns to state 1, then we can break up the time interval into independent cycles of lengths $C_i = T_i - T_{i-1}$, $i = 1, 2, \ldots$. Now suppose that during the i-th cycle a reward

$$R_i = \sum_{t=T_{i-1}}^{T_i-1} \varrho^{t-T_{i-1}} \, r(X_t)$$

is received, where $r(i)$ is some fixed reward for visiting state $i \in \{1, 2, 3, 4\}$ and $\varrho \in (0, 1)$ is a discounting factor. Clearly, $\{(C_i, R_i)\}$ is a renewal reward process. Figure 3.8 shows the outcomes of 1000 pairs (C, R), using $r(1) = 4, r(2) = 3, r(3) = 10, r(4) = 1$, and $\varrho = 0.9$.

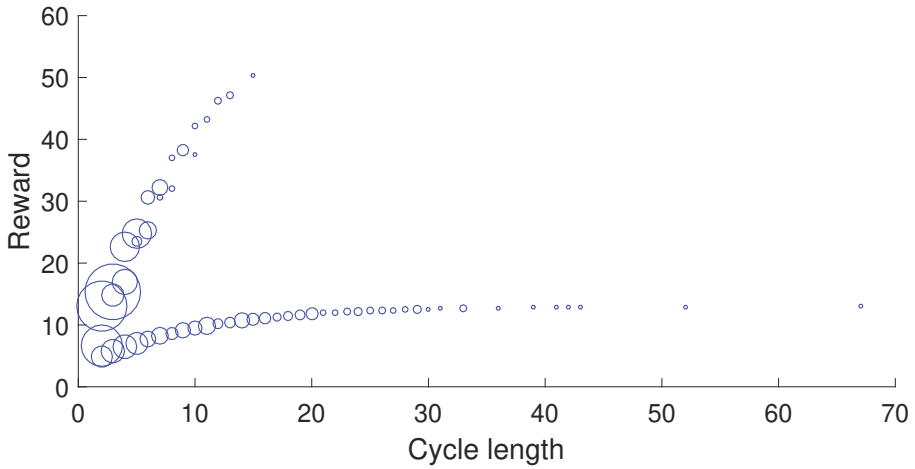


Figure 3.8: Each circle represents a (cycle length, reward) pair. The varying circle sizes indicate the number of occurrences for a given pair. For example, (2,15.43) is the most likely pair here, occurring 186 out of a 1000 times. It corresponds to the cycle path $1 \to 3 \to 2 \to 1$.

The long-run average reward is estimated as 2.50 for our data. But how accurate is this estimate? Figure 3.9 shows a density plot of the bootstrapped ratio estimates, where we independently resampled the data pairs 1000 times.

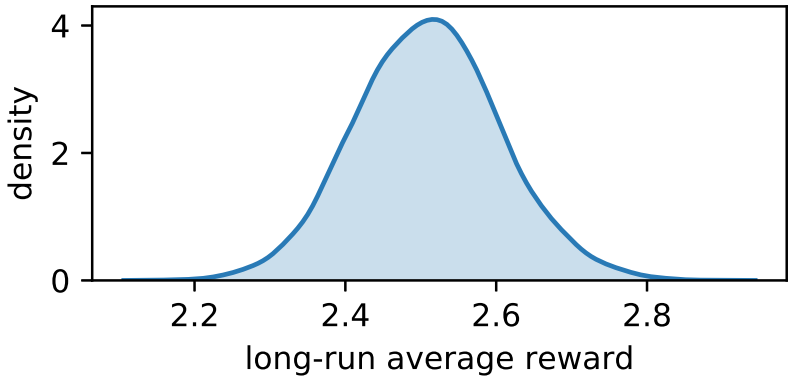


Figure 3.9: Density plot of the bootstrapped ratio estimates for the Markov chain renewal reward process.

Figure 3.9 indicates that the true long-run average reward lies between 2.2 and 2.8 with high confidence. More precisely, the 99% bootstrap confidence interval (percentile method) is here (2.27, 2.77). The following Python script spells out the procedure.

ratioest.py

```
import numpy as np, matplotlib.pyplot as plt, seaborn as sns
from numba import jit

np.random.seed(123)
n = 1000
P = np.array([[0, 0.2, 0.5, 0.3],
              [0.5 ,0, 0.5, 0],
              [0.3, 0.7, 0, 0],
              [0.1, 0, 0, 0.9]])
r = np.array([4,3,10,1])
Corg = np.array(np.zeros((n,1)))
Rorg = np.array(np.zeros((n,1)))
rho=0.9

@jit()   #for speed-up; see Appendix
def generate_cyclereward(n):
   for i in range(n):
       t=1
       xreg = 1   #regenerative state  (out of 1,2,3,4)
       reward = r[0]
       x= np.amin(np.argwhere(np.cumsum(P[xreg-1,:]) > np.random.
          rand())) + 1
       while x != xreg:
           t += 1
           reward += rho**(t-1)*r[x-1]
           x = np.amin(np.where(np.cumsum(P[x-1,:]) > np.random.rand
              ())) + 1
       Corg[i] = t
       Rorg[i] = reward
   return Corg, Rorg

Corg, Rorg = generate_cyclereward(n)

Aorg = np.mean(Rorg)/np.mean(Corg)
K = 5000
A = np.array(np.zeros((K,1)))
C = np.array(np.zeros((n,1)))
R = np.array(np.zeros((n,1)))
for i in range(K):
    ind = np.ceil(n*np.random.rand(1,n)).astype(int)[0]-1
    C = Corg[ind]
    R = Rorg[ind]
    A[i] = np.mean(R)/np.mean(C)

plt.xlabel('long-run average reward')
plt.ylabel('density')
sns.kdeplot(A.flatten(),shade=True)
plt.show()
```

■

3.3.3 Variance Reduction

The estimation of performance measures in Monte Carlo simulation can be made more efficient by utilizing known information about the simulation model. Variance reduction techniques include antithetic variables, control variables, importance sampling, conditional Monte Carlo, and stratified sampling; see, for example, [71, Chapter 9]. We shall only deal with control variables and importance sampling here.

CONTROL VARIABLE

Suppose Y is the output of a simulation experiment. A random variable $\widetilde{Y}$, obtained from the same simulation run, is called a *control variable* for Y if Y and $\widetilde{Y}$ are correlated (negatively or positively) and the expectation of $\widetilde{Y}$ is known. The use of control variables for variance reduction is based on the following theorem. We leave its proof to Exercise 21.

☞ 118

Theorem 3.3: Control Variable Estimation

Let $Y_1, \ldots, Y_N$ be the output of N independent simulation runs and let $\widetilde{Y}_1, \ldots, \widetilde{Y}_N$ be the corresponding control variables, with $\mathbb{E}\widetilde{Y}_k = \widetilde{\mu}$ known. Let $\varrho_{Y,\widetilde{Y}}$ be the correlation coefficient between each Y_k and $\widetilde{Y}_k$. For each $\alpha \in \mathbb{R}$ the estimator

$$\widehat{\mu}^{(c)} = \frac{1}{N} \sum_{k=1}^{N} \left[Y_k - \alpha \left(\widetilde{Y}_k - \widetilde{\mu} \right) \right] \tag{3.15}$$

is an unbiased estimator for $\mu = \mathbb{E}Y$. The minimal variance of $\widehat{\mu}^{(c)}$ is

$$\mathbb{V}\mathrm{ar}\, \widehat{\mu}^{(c)} = \frac{1}{N} (1 - \varrho^2_{Y,\widetilde{Y}}) \, \mathbb{V}\mathrm{ar}\, Y, \tag{3.16}$$

which is obtained for $\alpha = \varrho_{Y,\widetilde{Y}} \sqrt{\mathbb{V}\mathrm{ar} Y / \mathbb{V}\mathrm{ar} \widetilde{Y}}$.

From (3.16) we see that, by using the optimal α in (3.15), the variance of the control variate estimator is a factor $1 - \varrho^2_{Y,\widetilde{Y}}$ smaller than the variance of the crude Monte Carlo estimator. Thus, if $\widetilde{Y}$ is highly correlated with Y, a significant variance reduction can be achieved. The optimal α is usually unknown, but it can be easily estimated from the sample covariance matrix of $\{(Y_k, \widetilde{Y}_k)\}$.

☞ 456

In the next example, we estimate the multiple integral in Example 3.10 using control variables.

☞ 86

■ **Example 3.13 (Monte Carlo Integration (cont.))** The random variable $Y = |X_1 + X_2 + X_3|^{1/2} (2\pi)^{3/2}$ is positively correlated with the random variable $\widetilde{Y} = X_1^2 + X_2^2 + X_3^2$, for the same choice of $X_1, X_2, X_3 \overset{\text{iid}}{\sim} \mathcal{N}(0, 1)$. As $\mathbb{E}\widetilde{Y} = \mathbb{V}\mathrm{ar}(X_1 + X_2 + X_3) = 3$, we can use it as a control variable to estimate the expectation of Y. The following Python program is based on Theorem 3.3. It imports the crude Monte Carlo sampling code from Example 3.10.

mcintCV.py

```
from mcint import *

Yc = np.sum(x**2, axis=1) # control variable data
yc = 3  # true expectation of control variable
C = np.cov(y,Yc) # sample covariance matrix
cor = C[0][1]/np.sqrt(C[0][0]*C[1][1])
alpha = C[0][1]/C[1][1]

est = np.mean(y-alpha*(Yc-yc))
RECV = np.sqrt((1-cor**2)*C[0][0]/N)/est  #relative error

print('Estimate = {:3.3f}, CI = ({:3.3f},{:3.3f}), Corr = {:3.3f}'.
      format(est, est*(1-z*RECV), est*(1+z*RECV),cor))
```

```
Estimate = 17.045, CI = (17.032,17.057), Corr = 0.480
```

A typical estimate of the correlation coefficient $\varrho_{Y,\widetilde{Y}}$ is 0.48, which gives a reduction of the variance with a factor $1-0.48^2 \approx 0.77$ — a simulation speed-up of 23% compared with crude Monte Carlo. Although the gain is small in this case, due to the modest correlation between Y and $\widetilde{Y}$, little extra work was required to achieve this variance reduction. ■

IMPORTANCE SAMPLING

One of the most important variance reduction techniques is *importance sampling*. This technique is especially useful for the estimation of very small probabilities. The standard setting is the estimation of a quantity

$$\mu = \mathbb{E}_f H(\boldsymbol{X}) = \int H(\boldsymbol{x})\, f(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}, \tag{3.17}$$

where H is a real-valued function and f the probability density of a random vector $\boldsymbol{X}$, called the *nominal pdf*. The subscript f is added to the expectation operator to indicate that it is taken with respect to the density f.

NOMINAL PDF

Let g be another probability density such that $g(\boldsymbol{x}) = 0$ implies that $H(\boldsymbol{x})\, f(\boldsymbol{x}) = 0$. Using the density g we can represent μ as

$$\mu = \int H(\boldsymbol{x}) \frac{f(\boldsymbol{x})}{g(\boldsymbol{x})}\, g(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \mathbb{E}_g \left[H(\boldsymbol{X}) \frac{f(\boldsymbol{X})}{g(\boldsymbol{X})} \right]. \tag{3.18}$$

Consequently, if $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N \sim_{\text{iid}} g$, then

$$\widehat{\mu} = \frac{1}{N} \sum_{k=1}^{N} H(\boldsymbol{X}_k) \frac{f(\boldsymbol{X}_k)}{g(\boldsymbol{X}_k)} \tag{3.19}$$

is an unbiased estimator of μ. This estimator is called the *importance sampling estimator* and g is called the importance sampling density. The ratio of densities, $f(\boldsymbol{x})/g(\boldsymbol{x})$, is called the *likelihood ratio*. The importance sampling pseudo-code is given in Algorithm 3.3.2.

IMPORTANCE SAMPLING ESTIMATOR

LIKELIHOOD RATIO

Algorithm 3.3.2: Importance Sampling Estimation

input: Function H, importance sampling density g such that $g(\boldsymbol{x}) = 0$ for all $\boldsymbol{x}$ for which $H(\boldsymbol{x})f(\boldsymbol{x}) = 0$, sample size N, confidence level $1-\alpha$.

output: Point estimate and approximate $(1-\alpha)$ confidence interval for $\mu = \mathbb{E}H(\boldsymbol{X})$, where $\boldsymbol{X} \sim f$.

1 Simulate $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N \overset{\text{iid}}{\sim} g$ and let $Y_i = H(\boldsymbol{X}_i)f(\boldsymbol{X}_i)/g(\boldsymbol{X}_i)$, $i = 1, \ldots, N$.

2 Estimate μ via $\widehat{\mu} = \overline{Y}$ and determine an approximate $(1-\alpha)$ confidence interval as

$$\mathcal{I} := \left(\widehat{\mu} - z_{1-\alpha/2} \frac{S}{\sqrt{N}}, \widehat{\mu} + z_{1-\alpha/2} \frac{S}{\sqrt{N}} \right),$$

where z_γ denotes the γ-quantile of the $\mathcal{N}(0,1)$ distribution and S is the sample standard deviation of $Y_1, \ldots, Y_N$.

3 **return** $\widehat{\mu}$ and the interval $\mathcal{I}$.

■ **Example 3.14 (Importance Sampling)** Let us examine the workings of importance sampling by estimating the area, μ say, under the graph of the function

$$M(x_1, x_2) = \mathrm{e}^{-\frac{1}{4}\sqrt{x_1^2+x_2^2}} \left(\sin\left(2\sqrt{x_1^2 + x_2^2} \right) + 1 \right), \quad (x_1, x_2) \in \mathbb{R}^2. \tag{3.20}$$

☞ 80

We saw a similar function in Example 3.8 (but note the different domain). A natural approach to estimate the area is to truncate the domain to the square $[-b, b]^2$, for large enough b, and to estimate the integral

$$\mu_b = \int_{-b}^{b} \int_{-b}^{b} \underbrace{(2b)^2 M(\boldsymbol{x})}_{H(\boldsymbol{x})} f(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} = \mathbb{E}_f H(\boldsymbol{X})$$

via crude Monte Carlo, where $f(\boldsymbol{x}) = 1/(2b)^2, \boldsymbol{x} \in [-b, b]^2$, is the pdf of the uniform distribution on $[-b, b]^2$. Here is the Python code which does just that.

impsamp1.py

```
import numpy as np
from numpy import exp, sqrt, sin, pi, log, cos
from numpy.random import rand

b = 1000
H = lambda x1, x2: (2*b)**2 * exp(-sqrt(x1**2+x2**2)/4)*(sin(2*sqrt(
        x1**2+x2**2))+1)*(x1**2 + x2**2 < b**2)
f = 1/((2*b)**2)
N = 10**6
X1 = -b + 2*b*rand(N,1)
X2 = -b + 2*b*rand(N,1)
Z = H(X1,X2)
estCMC = np.mean(Z).item()  # to obtain scalar
RECMC = np.std(Z)/estCMC/sqrt(N).item()
print('CI = ({:3.3f},{:3.3f}), RE = {: 3.3f}'.format(estCMC*(1-1.96*
    RECMC), estCMC*(1+1.96*RECMC),RECMC))
```

```
CI = (82.663,135.036), RE =  0.123
```

For a truncation level of $b = 1000$ and a sample size of $N = 10^6$, a typical estimate is 108.8, with an estimated relative error of 0.123. We have two sources of error here. The first is the error in approximating μ by μ_b. However, as the function H decays exponentially fast, $b = 1000$ is more than enough to ensure this error is negligible. The second type of error is the statistical error, due to the estimation process itself. This can be quantified by the estimated relative error, and can be reduced by increasing the sample size.

Let us now consider an importance sampling approach in which the importance sampling pdf g is radially symmetric and decays exponentially in the radius, similar to the function H. In particular, we simulate (X_1, X_2) in a way akin to Example 3.1, by first generating a radius $R \sim \mathsf{Exp}(\lambda)$ and an angle $\Theta \sim \mathcal{U}(0, 2\pi)$, and then returning $X_1 = R\cos(\Theta)$ and $X_2 = R\sin(\Theta)$. By the Transformation Rule (Theorem C.4) we then have ☞ 69 ☞ 433

$$g(\boldsymbol{x}) = f_{R,\Theta}(r,\theta)\frac{1}{r} = \lambda\,\mathrm{e}^{-\lambda r}\frac{1}{2\pi}\frac{1}{r} = \frac{\lambda \mathrm{e}^{-\lambda\sqrt{x_1^2+x_2^2}}}{2\pi\sqrt{x_1^2+x_2^2}}, \quad \boldsymbol{x} \in \mathbb{R}^2 \setminus \{\mathbf{0}\}.$$

The following code, which imports the one given above, implements the importance sampling steps, using the parameter $\lambda = 0.1$.

impsamp2.py

```
from impsamp1 import *

lam = 0.1;
g = lambda x1, x2: lam*exp(-sqrt(x1**2 + x2**2)*lam)/sqrt(x1**2 + x2
    **2)/(2*pi);
U = rand(N,1); V = rand(N,1)
R = -log(U)/lam
X1 = R*cos(2*pi*V)
X2 = R*sin(2*pi*V)
Z = H(X1,X2)*f/g(X1,X2)
estIS = np.mean(Z).item()  # obtain scalar
REIS = np.std(Z)/estIS/sqrt(N).item()
print('CI = ({:3.3f},{:3.3f}), RE = {: 3.3f}'.format(estIS*(1-1.96*
    REIS), estIS*(1+1.96*REIS),REIS))
```

```
CI = (100.723,101.077), RE =  0.001
```

A typical estimate is 100.90 with an estimated relative error of $1 \cdot 10^{-4}$, which gives a substantial variance reduction. In terms of approximate 95% confidence intervals, we have (82.7,135.0) in the CMC case versus (100.7,101.1) in the importance sampling case. Of course, we could have reduced the truncation level b to improve the performance of CMC, but then the approximation error might become more significant. For the importance sampling case, the relative error is hardly affected by the threshold level, but does depend on the choice of λ. We chose λ such that the decay rate is slower than the decay rate of the function H, which is 0.25. ■

As illustrated in the above example, a main difficulty in importance sampling is how to choose the importance sampling distribution. A poor choice of g may seriously affect the accuracy of both the estimate and the confidence interval. The theoretically optimal choice

g^* for the importance sampling density minimizes the variance of $\widehat{\mu}$ and is therefore the solution to the functional minimization program

$$\min_g \mathbb{V}\mathrm{ar}_g \left(H(\boldsymbol{X}) \frac{f(\boldsymbol{X})}{g(\boldsymbol{X})} \right). \tag{3.21}$$

☞ 118

OPTIMAL IMPORTANCE SAMPLING PDF

It is not difficult to show, see also Exercise 22, that if either $H(\boldsymbol{x}) \geqslant 0$ or $H(\boldsymbol{x}) \leqslant 0$ for all $\boldsymbol{x}$, then the *optimal importance sampling pdf* is

$$g^*(\boldsymbol{x}) = \frac{H(\boldsymbol{x})\, f(\boldsymbol{x})}{\mu}. \tag{3.22}$$

Namely, in this case $\mathbb{V}\mathrm{ar}_{g^*} \widehat{\mu} = \mathbb{V}\mathrm{ar}_{g^*}(H(\boldsymbol{X}) f(\boldsymbol{X})/g(\boldsymbol{X})) = \mathbb{V}\mathrm{ar}_{g^*} \mu = 0$, so that the estimator $\widehat{\mu}$ is *constant* under g^*. An obvious difficulty is that the evaluation of the optimal importance sampling density g^* is usually not possible, since $g^*(\boldsymbol{x})$ in (3.22) depends on the unknown quantity μ. Nevertheless, one can typically choose a good importance sampling density g "close" to the minimum variance density g^*.

One of the main considerations for choosing a good importance sampling pdf is that the estimator (3.19) should have finite variance. This is equivalent to the requirement that

$$\mathbb{E}_g \left[H^2(\boldsymbol{X}) \frac{f^2(\boldsymbol{X})}{g^2(\boldsymbol{X})} \right] = \mathbb{E}_f \left[H^2(\boldsymbol{X}) \frac{f(\boldsymbol{X})}{g(\boldsymbol{X})} \right] < \infty. \tag{3.23}$$

This suggests that g should not have lighter tails than f and that, preferably, the likelihood ratio, f/g, should be bounded.

3.4 Monte Carlo for Optimization

In this section we describe several Monte Carlo methods for optimization. Such randomized algorithms can be useful for solving optimization problems with many local optima and complicated constraints, possibly involving a mix of continuous and discrete variables. Randomized algorithms are also used to solve *noisy* optimization problems, in which the objective function is unknown and has to be obtained via Monte Carlo simulation.

3.4.1 Simulated Annealing

SIMULATED ANNEALING

Simulated annealing is a Monte Carlo technique for minimization that emulates the physical state of atoms in a metal when the metal is heated up and then slowly cooled down. When the cooling is performed very slowly, the atoms settle down to a minimum-energy state. Denoting the state as $\boldsymbol{x}$ and the energy of a state as $S(\boldsymbol{x})$, the probability distribution of the (random) states is described by the *Boltzmann pdf*

$$f(\boldsymbol{x}) \propto \mathrm{e}^{-\frac{S(\boldsymbol{x})}{kT}}, \quad \boldsymbol{x} \in \mathcal{X},$$

where k is Boltzmann's constant and T is the temperature.

Going beyond the physical interpretation, suppose that $S(\boldsymbol{x})$ is an arbitrary function to be minimized, with $\boldsymbol{x}$ taking values in some discrete or continuous set $\mathcal{X}$. The *Gibbs pdf* corresponding to $S(\boldsymbol{x})$ is defined as

GIBBS PDF

$$f_T(\boldsymbol{x}) = \frac{\mathrm{e}^{-\frac{S(\boldsymbol{x})}{T}}}{z_T}, \quad \boldsymbol{x} \in \mathcal{X},$$

provided that the normalization constant $z_T := \sum_{\boldsymbol{x}} \exp(-S(\boldsymbol{x})/T)$ is finite. Note that this is simply the Boltzmann pdf with the Boltzmann constant k removed. As $T \to 0$, the pdf becomes more and more peaked around the set of global minimizers of S.

The idea of simulated annealing is to create a sequence of points $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots$ that are approximately distributed according to pdfs $f_{T_1}(\boldsymbol{x}), f_{T_2}(\boldsymbol{x}), \ldots$, where $T_1, T_2, \ldots$ is a sequence of "temperatures" that decreases (is "cooled") to 0 — known as the *annealing schedule*. If each $\boldsymbol{X}_t$ were sampled *exactly* from f_{T_t}, then $\boldsymbol{X}_t$ would converge to a global minimum of $S(\boldsymbol{x})$ as $T_t \to 0$. However, in practice sampling is *approximate* and convergence to a global minimum is not assured. A generic simulated annealing algorithm is as follows.

ANNEALING SCHEDULE

Algorithm 3.4.1: Simulated Annealing

input: Annealing schedule $T_0, T_1, \ldots,$, function S, initial value $\boldsymbol{x}_0$.
output: Approximations to the global minimizer $\boldsymbol{x}^*$ and minimum value $S(\boldsymbol{x}^*)$.

```
Set X_0 ← x_0 and t ← 1.
while not stopping do
    Approximately simulate X_t from f_{T_t}(x).
    t ← t + 1
return X_t, S(X_t)
```

A popular annealing schedule is *geometric cooling*, where $T_t = \beta T_{t-1}$, $t = 1, 2, \ldots$, for a given initial temperature T_0 and a *cooling factor* $\beta \in (0, 1)$. Appropriate values for T_0 and β are problem-dependent and this has traditionally required tuning on the part of the user. A possible stopping criterion is to stop after a fixed number of iterations, or when the temperature is "small enough".

GEOMETRIC COOLING
COOLING FACTOR

Approximate sampling from a Gibbs distribution is most often carried out via Markov chain Monte Carlo. For each iteration t, the Markov chain should theoretically run for a large number of steps to accurately sample from the Gibbs pdf f_{T_t}. However, in practice, one often only runs a *single* step of the Markov chain, before updating the temperature, as in Algorithm 3.4.2 below.

To sample from a Gibbs distribution f_T, this algorithm uses a random walk Metropolis–Hastings sampler. From (3.7), the acceptance probability of a proposal $\boldsymbol{y}$ is thus

☞ 80

$$\alpha(\boldsymbol{x}, \boldsymbol{y}) = \min\left\{\frac{\mathrm{e}^{-\frac{1}{T}S(\boldsymbol{y})}}{\mathrm{e}^{-\frac{1}{T}S(\boldsymbol{x})}},\ 1\right\} = \min\left\{\mathrm{e}^{-\frac{1}{T}(S(\boldsymbol{y})-S(\boldsymbol{x}))},\ 1\right\}.$$

Hence, if $S(\boldsymbol{y}) < S(\boldsymbol{x})$, then the proposal is aways accepted. Otherwise, the proposal is accepted with probability $\exp(-\frac{1}{T}(S(\boldsymbol{y}) - S(\boldsymbol{x})))$.

Algorithm 3.4.2: Simulated Annealing with a Random Walk Sampler

input: Objective function S, starting state $\boldsymbol{X}_0$, initial temperature T_0, number of iterations N, symmetric proposal density $q(\boldsymbol{y} \mid \boldsymbol{x})$, constant β.
output: Approximate minimizer and minimum value of S.

for $t = 0$ **to** $N - 1$ **do**
  Simulate a new state $\boldsymbol{Y}$ from the symmetric proposal $q(\boldsymbol{y} \mid \boldsymbol{X}_t)$.
  if $S(\boldsymbol{Y}) < S(\boldsymbol{X}_t)$ **then**
    $\boldsymbol{X}_{t+1} \leftarrow \boldsymbol{Y}$
  else
    Draw $U \sim \mathcal{U}(0, 1)$.
    if $U \leqslant \mathrm{e}^{-(S(\boldsymbol{Y})-S(\boldsymbol{X}_t))/T_t}$ **then**
      $\boldsymbol{X}_{t+1} \leftarrow \boldsymbol{Y}$
    else
      $\boldsymbol{X}_{t+1} \leftarrow \boldsymbol{X}_t$
  $T_{t+1} \leftarrow \beta T_t$
return $\boldsymbol{X}_N$ and $S(\boldsymbol{X}_N)$

■ **Example 3.15 (Simulated Annealing for Minimization)** Let us minimize the "wiggly" function depicted in the bottom panel of Figure 3.10 and given by:

$$S(x) = \begin{cases} -\mathrm{e}^{-x^2/100} \sin(13x - x^4)^5 \sin(1 - 3x^2)^2, & \text{if } -2 \leqslant x \leqslant 2, \\ \infty, & \text{otherwise.} \end{cases}$$

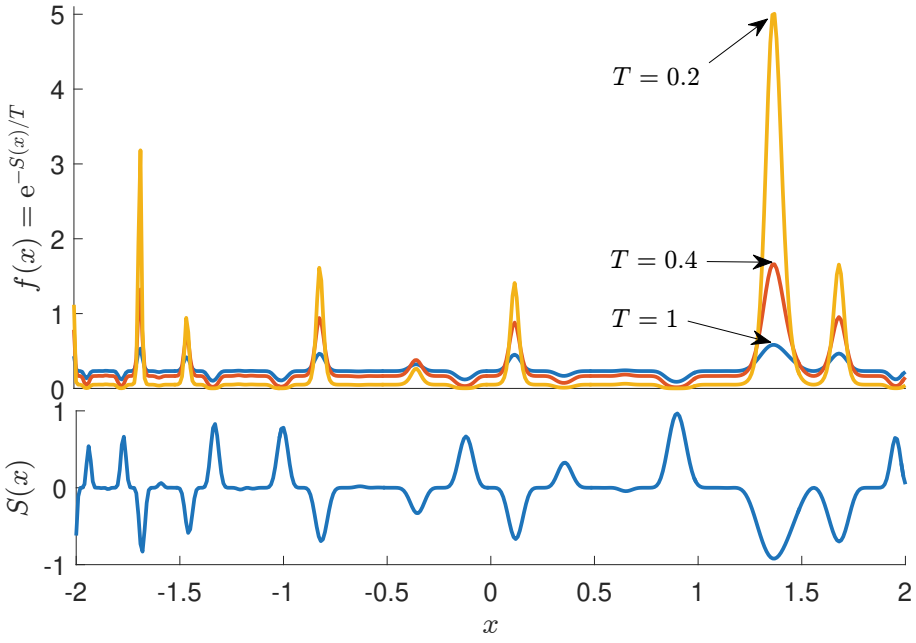


Figure 3.10: Lower panel: the "wiggly" function $S(x)$. Upper panel: three (unnormalized) Gibbs pdfs for temperatures $T = 1, 0.4.0.2$. As the temperature decreases, the Gibbs pdf converges to the pdf that has all its mass concentrated at the minimizer of S.

The function has many local minima and maxima, with a global minimum around 1.4. The figure also illustrates the relationship between S and the (unnormalized) Gibbs pdf f_T.

The following Python code implements a slight variant of Algorithm 3.4.2 where, instead of stopping after a fixed number of iterations, the algorithm stops when the temperature is lower than some threshold (here 10^{-3}).

Instead of stopping after a fixed number N of iterations or when the temperature is low enough, it is useful to stop when consecutive function values are closer than some distance ε to each other, or when the best found function value has not changed over a fixed number d of iterations.

For a "current" state x, the proposal state Y is here drawn from the $\mathcal{N}(x, 0.5^2)$ distribution. We use geometric cooling with decay parameter $\beta = 0.999$ and initial temperature $T_0 = 1$. We set the initial state to $x_0 = 0$. Figure 3.11 depicts a realization of the sequence of states x_t for $t = 0, 1, \ldots$. After initially fluctuating wildly, the sequence settles down to a value around 1.37, with $S(1.37) = -0.92$, corresponding to the global optimizer and minimum, respectively.

simann.py

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

def wiggly(x):
    y = -np.exp(x**2/100)*np.sin(13*x-x**4)**5*np.sin(1-3*x**2)**2
    ind = np.vstack((np.argwhere(x<-2),np.argwhere(x>2)))
    y[ind]=float('inf')
    return y

S = wiggly
beta = 0.999
sig = 0.5
T=1
x= np.array([0])
xx=[]
Sx=S(x)
while T>10**(-3):
    T=beta*T
    y = x+sig*np.random.randn()
    Sy = S(y)
    alpha = np.amin((np.exp(-(Sy-Sx)/T),1))
    if np.random.uniform()<alpha:
        x=y
        Sx=Sy
    xx=np.hstack((xx,x))

print('minimizer = {:3.3f}, minimum ={:3.3f}'.format(x[0],Sx[0]))
plt.plot(xx)
plt.show()
```

```
minimizer = 1.365, minimum = -0.958
```

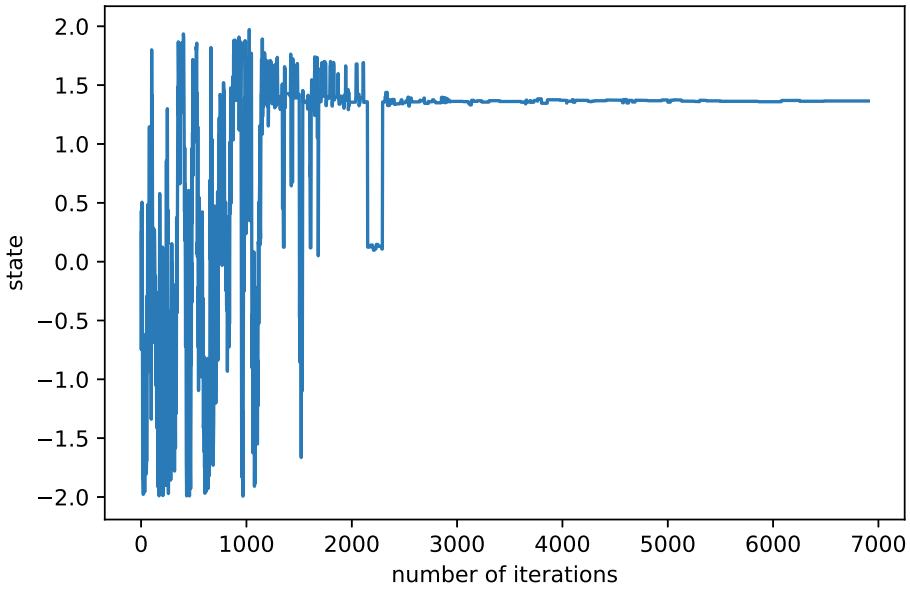


Figure 3.11: Typical states generated by the simulated annealing algorithm.

3.4.2 Cross-Entropy Method

CROSS-ENTROPY

The *cross-entropy* (CE) method [103] is a simple Monte Carlo algorithm that can be used for both optimization and estimation.

The basic idea of the CE method for minimizing a function S on a set $\mathcal{X}$ is to define a parametric family of probability densities $\{f(\cdot\,|\,\boldsymbol{v}), \boldsymbol{v} \in \mathcal{V}\}$ on $\mathcal{X}$ and to iteratively update the parameter $\boldsymbol{v}$ so that $f(\cdot\,|\,\boldsymbol{v})$ places more mass on states $\boldsymbol{x}$ that have smaller S values than on the previous iteration. In particular, the CE algorithm has two basic phases:

- *Sampling*: Samples $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N$ are drawn independently according to $f(\cdot\,|\,\boldsymbol{v})$. The objective function S is evaluated at these points.
- *Updating*: A new parameter $\boldsymbol{v}'$ is selected on the basis of those $\boldsymbol{X}_i$ for which $S(\boldsymbol{X}_i) \leqslant \gamma$ for some level γ. These $\{\boldsymbol{X}_i\}$ form the *elite sample* set, $\mathcal{E}$.

ELITE SAMPLE

At each iteration the level parameter γ is chosen as the worst of the $N^{\text{elite}} := \lceil \varrho N \rceil$ best performing samples, where $\varrho \in (0,1)$ is the *rarity parameter* — typically, $\varrho = 0.1$ or $\varrho = 0.01$. The parameter $\boldsymbol{v}$ is updated as a smoothed average $\alpha \boldsymbol{v}' + (1-\alpha)\boldsymbol{v}$, where $\alpha \in (0,1)$ is the *smoothing parameter* and

RARITY PARAMETER

SMOOTHING PARAMETER

$$\boldsymbol{v}' := \operatorname*{argmax}_{\boldsymbol{v} \in \mathcal{V}} \sum_{\boldsymbol{X} \in \mathcal{E}} \ln f(\boldsymbol{X}\,|\,\boldsymbol{v}). \tag{3.24}$$

The updating rule (3.24) is the result of minimizing the Kullback–Leibler divergence between the conditional density of $\boldsymbol{X} \sim f(\boldsymbol{x}\,|\,\boldsymbol{v})$ given $S(\boldsymbol{X}) \leqslant \gamma$, and $f(\boldsymbol{x}; \boldsymbol{v})$; see [103]. Note that (3.24) yields the *maximum likelihood estimator* (MLE) of $\boldsymbol{v}$ based on the elite samples. Hence, for many specific families of distributions, explicit solutions can be found. An important example is where $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \operatorname{diag}(\boldsymbol{\sigma}^2))$; that is, $\boldsymbol{X}$ has independent Gaussian

☞ 456

components. In this case, the mean vector $\boldsymbol{\mu}$ and the vector of variances $\boldsymbol{\sigma}^2$ are simply updated via the sample mean and sample variance of the elite samples. This is known as *normal updating*. A generic CE procedure for minimization is given in Algorithm 3.4.3.

NORMAL UPDATING

Algorithm 3.4.3: Cross-Entropy Method for Minimization

input: Function S, initial sampling parameter $\boldsymbol{v}_0$, sample size N, rarity parameter ϱ, smoothing parameter α.
output: Approximate minimum of S and optimal sampling parameter $\boldsymbol{v}$.

1 Initialize $\boldsymbol{v}_0$, set $N^{\text{elite}} \leftarrow \lceil \varrho N \rceil$ and $t \leftarrow 0$.
2 **while** a stopping criterion is not met **do**
3 $t \leftarrow t + 1$
4 Simulate an iid sample $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N$ from the density $f(\cdot \,|\, \boldsymbol{v}_{t-1})$.
5 Evaluate the performances $S(\boldsymbol{X}_1), \ldots, S(\boldsymbol{X}_N)$ and sort them from smallest to largest: $S_{(1)}, \ldots, S_{(N)}$.
6 Let γ_t be the sample ϱ-quantile of the performances:

$$\gamma_t \leftarrow S_{(N^{\text{elite}})}. \tag{3.25}$$

7 Determine the set of elite samples $\mathcal{E}_t = \{\boldsymbol{X}_i : S(\boldsymbol{X}_i) \leqslant \gamma_t\}$.
8 Let $\boldsymbol{v}_t'$ be the MLE of the elite samples:

$$\boldsymbol{v}_t' \leftarrow \operatorname*{argmax}_{\boldsymbol{v}} \sum_{\boldsymbol{X} \in \mathcal{E}_t} \ln f(\boldsymbol{X} \,|\, \boldsymbol{v}). \tag{3.26}$$

9 Update the sampling parameter as

$$\boldsymbol{v}_t \leftarrow \alpha \boldsymbol{v}_t' + (1 - \alpha) \boldsymbol{v}_{t-1}. \tag{3.27}$$

10 **return** γ_t, $\boldsymbol{v}_t$

The CE algorithm produces a sequence of pairs $(\gamma_1, \boldsymbol{v}_1), (\gamma_2, \boldsymbol{v}_2), \ldots$, such that γ_t converges (approximately) to the minimal function value, and $f(\cdot \,|\, \boldsymbol{v}_t)$ to a degenerate pdf that (approximately) concentrates all its mass at a minimizer of S, as $t \to \infty$. A possible stopping condition is to stop when the sampling distribution $f(\cdot \,|\, \boldsymbol{v}_t)$ is sufficiently close to a degenerate distribution. For normal updating this means that the standard deviation is sufficiently small.

The output of the CE algorithm could also include the overall best function value and corresponding solution.

In the following example, we minimize the same function as in Example 3.15, but instead use the CE algorithm. ☞ 97

■ **Example 3.16 (Cross-Entropy Method for Minimization)** In this case we take the family of normal distributions $\{\mathcal{N}(\mu, \sigma^2)\}$ for the sampling step (Step 4 of Algorithm 3.4.3), starting with $\mu = 0$ and $\sigma = 3$. The choice of the initial parameter is quite arbitrary, as long as σ is large enough to sample a wide range of points. We take $N = 100$ samples at each iteration, set $\varrho = 0.1$, and keep the $N^{\text{elite}} = 10 = \lceil N\varrho \rceil$ smallest ones as the elite samples. The parameters μ and σ are then updated via the sample mean and sample standard deviation

of the elite samples. In this case we do not use any smoothing ($\alpha = 1$). In the following Python code the 100×2 matrix `Sx` stores the x-values in the first column and the function values in the second column. The rows of this matrix are sorted in ascending order according to the function values, giving the matrix `sortSx`. The first $N^{\text{elite}} = 10$ rows of this sorted matrix correspond to the elite samples and their function values. The updating of μ and σ is done in Lines 14 and 15. Figure 3.12 shows how the pdfs of the $\mathcal{N}(\mu_t, \sigma_t^2)$ sampling distributions degenerate to the point mass at the global minimizer 1.366.

CEmethod.py

```
from simann import wiggly
import numpy as np
np.set_printoptions(precision=3)
mu, sigma = 0, 3
N, Nel = 100, 10
eps = 10**-5
S = wiggly
while sigma > eps:
    X = np.random.randn(N,1)*sigma + np.array(np.ones((N,1)))*mu
    Sx = np.hstack((X, S(X)))
    sortSx = Sx[Sx[:,1].argsort(),]
    Elite = sortSx[0:Nel,:-1]
    mu = np.mean(Elite, axis=0)
    sigma = np.std(Elite, axis=0)
    print('S(mu)= {}, mu: {}, sigma: {}\n'.format(S(mu), mu, sigma))
```

```
S(mu)= [0.071], mu: [0.414], sigma: [0.922]
S(mu)= [0.063], mu: [0.81], sigma: [0.831]
S(mu)= [-0.033], mu: [1.212], sigma: [0.69]
S(mu)= [-0.588], mu: [1.447], sigma: [0.117]
S(mu)= [-0.958], mu: [1.366], sigma: [0.007]
S(mu)= [-0.958], mu: [1.366], sigma: [0.]
S(mu)= [-0.958], mu: [1.366], sigma: [3.535e-05]
S(mu)= [-0.958], mu: [1.366], sigma: [2.023e-06]
```

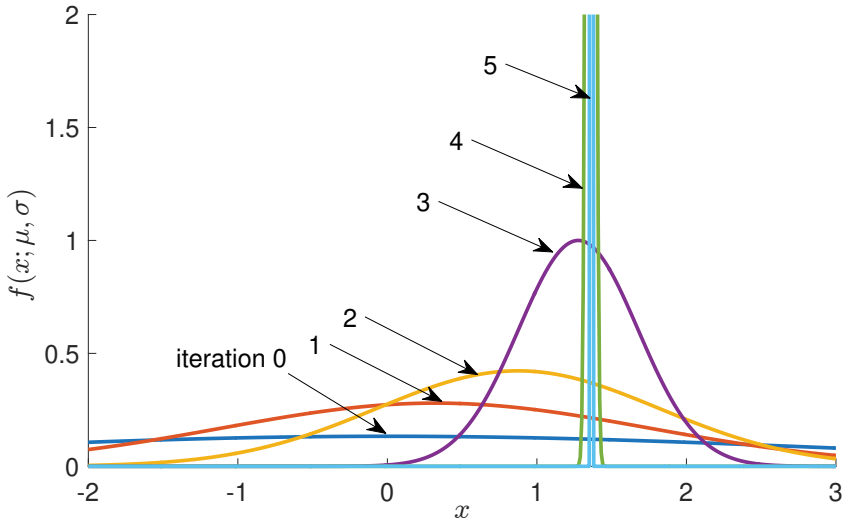


Figure 3.12: The normal pdfs of the first five sampling distributions, truncated to the interval $[-2, 3]$. The initial sampling distribution is $\mathcal{N}(0, 3^2)$.

3.4.3 Splitting for Optimization

Minimizing a function $S(\boldsymbol{x})$, $\boldsymbol{x} \in \mathcal{X}$ is closely related to drawing a random sample from a *level set* of the form $\{\boldsymbol{x} \in \mathcal{X} : S(\boldsymbol{x}) \leqslant \gamma\}$. Suppose S has minimum value γ^* attained at $\boldsymbol{x}^*$. As long as $\gamma \geqslant \gamma^*$, this level set contains the minimizer. Moreover, if γ is close to γ^*, the volume of this level set will be small. So, a randomly selected point from this set is expected to be close to $\boldsymbol{x}^*$. Thus, by gradually decreasing the level parameter γ, the level sets will gradually shrink towards the set $\{\boldsymbol{x}^*\}$. Indeed, the CE method was developed with exactly this connection in mind; see, e.g., [102]. Note that the CE method employs a *parametric* sampling distribution to obtain samples from the level sets (the elite samples). In [34] a *non-parametric* sampling mechanism is introduced that uses an evolving collection of particles. The resulting optimization algorithm, called *splitting for continuous optimization* (SCO), provides a fast and accurate way to optimize complicated continuous functions. The details of SCO are given in Algorithm 3.4.4.

LEVEL SET

SPLITTING FOR CONTINUOUS OPTIMIZATION

Algorithm 3.4.4: Splitting for Continuous Optimization (SCO)

input: Objective function S, sample size N, rarity parameter ϱ, scale factor w, bounded region $\mathcal{B} \subset \mathcal{X}$ that is known to contain a global minimizer, and maximum number of attempts `MaxTry`.

output: Final iteration number t and sequence $(\boldsymbol{X}_{\text{best},1}, b_1), \ldots, (\boldsymbol{X}_{\text{best},t}, b_t)$ of best solutions and function values at each iteration.

Simulate $\mathcal{Y}_0 = \{\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_N\}$ uniformly on $\mathcal{B}$. Set $t \leftarrow 0$ and $N^{\text{elite}} \leftarrow \lceil N\varrho \rceil$.
while stopping condition is not satisfied **do**
  Determine the N^{elite} smallest values, $S_{(1)} \leqslant \cdots \leqslant S_{(N^{\text{elite}})}$, of $\{S(\boldsymbol{X}), \boldsymbol{X} \in \mathcal{Y}_t\}$, and store the corresponding vectors, $\boldsymbol{X}_{(1)}, \ldots, \boldsymbol{X}_{(N^{\text{elite}})}$, in $\mathcal{X}_{t+1}$. Set $b_{t+1} \leftarrow S_{(1)}$ and $\boldsymbol{X}_{\text{best},t+1} \leftarrow X_{(1)}$.
  Draw $B_i \sim \text{Bernoulli}(\frac{1}{2})$, $i = 1, \ldots, N^{\text{elite}}$, with $\sum_{i=1}^{N^{\text{elite}}} B_i = N \bmod N^{\text{elite}}$.
  for $i = 1$ **to** N^{elite} **do**
    $R_i \leftarrow \left\lfloor \frac{N}{N^{\text{elite}}} \right\rfloor + B_i$ `// random splitting factor`
    $\boldsymbol{Y} \leftarrow \boldsymbol{X}_{(i)}$; $\boldsymbol{Y}' \leftarrow \boldsymbol{Y}$
    for $j = 1$ **to** R_i **do**
      Draw $I \in \{1, \ldots, N^{\text{elite}}\} \setminus \{i\}$ uniformly and let $\boldsymbol{\sigma}_i \leftarrow w|\boldsymbol{X}^{(i)} - \boldsymbol{X}^{(I)}|$.
      Simulate a uniform permutation $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_n)$ of $(1, \ldots, n)$.
      for $k = 1$ **to** n **do**
        for `Try` $= 1$ **to** `MaxTry` **do**
          $\boldsymbol{Y}'(\pi_k) \leftarrow \boldsymbol{Y}(\pi_k) + \boldsymbol{\sigma}_i(\pi_k) Z, \quad Z \sim \mathcal{N}(0,1)$
          if $S(\boldsymbol{Y}') < S(\boldsymbol{Y})$ **then** $\boldsymbol{Y} \leftarrow \boldsymbol{Y}'$ and **break**.
      Add $\boldsymbol{Y}$ to $\mathcal{Y}_{t+1}$
  $t \leftarrow t + 1$
return $\{(\boldsymbol{X}_{\text{best},k}, b_k), k = 1, \ldots, t\}$

At iteration $t = 0$, the algorithm starts with a population of particles $\mathcal{Y}_0 = \{\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_N\}$ that are uniformly generated on some bounded region $\mathcal{B}$, which is large enough to contain a global minimizer. The function values of all particles in $\mathcal{Y}_0$ are sorted, and the best

$N^{\text{elite}} = \lceil N\varrho \rceil$ form the elite particle set $\mathcal{X}_1$, exactly as in the CE method. Next, the elite particles are "split" into $\lceil N/N^{\text{elite}} \rceil$ children particles, adding one extra child to some of the elite particles to ensure that the total number of children is again N. The purpose of Line 4 is to randomize which elite particles receive an extra child. Lines 8–15 describe how the children of the i-th elite particle are generated. First, in Line 9, we select one of the *other* elite particles uniformly at random. The same line defines an n-dimensional vector $\boldsymbol{\sigma}_i$ whose components are the absolute differences between the vectors $\boldsymbol{X}_{(i)}$ and $\boldsymbol{X}_{(I)}$, multiplied by a constant w. That is,

$$\boldsymbol{\sigma}_i = w\,|\boldsymbol{X}_{(i)} - \boldsymbol{X}_{(I)}| := w \begin{bmatrix} |X_{(i),1} - X_{(I),1}| \\ |X_{(i),2} - X_{(I),2}| \\ \vdots \\ |X_{(i),n} - X_{(I),n}| \end{bmatrix}.$$

☞ 115

Next, a uniform random permutation $\boldsymbol{\pi}$ of $(1,\ldots,n)$ is simulated (see Exercise 9). Lines 11–14 describe how, starting from a candidate child point $\boldsymbol{Y}$, each coordinate of $\boldsymbol{Y}$ is resampled, in the order determined by $\boldsymbol{\pi}$, by adding a standard normal random variable to that component, multiplied by the corresponding component of $\boldsymbol{\sigma}_i$ (Line 13). If the resulting $\boldsymbol{Y}'$ has a function value that is less than that of $\boldsymbol{Y}$, then the new candidate is accepted. Otherwise, the *same* coordinate is tried again. If no improvement is found in `MaxTry` attempts, the original component is retained. This process is performed for all elite samples, to produce the first-generation population $\mathcal{Y}_1$. The procedure is then repeated for iterations $t = 1, 2, \ldots$, until some stopping criterion is met, e.g., when the best found function value does not change for a number of consecutive iterations, or when the total number of function evaluations exceeds some threshold. The best found function value and corresponding argument (particle) are returned at the conclusion of the algorithm.

The input variable `MaxTry` governs how much computational time is dedicated to updating a component. In most cases we have encountered, the choices $w = 0.5$ and `MaxTry` $= 5$ work well. Empirically, relatively high value for ϱ work well, such as $\varrho = 0.4, 0.8$, or even $\varrho = 1$. The latter case means that at each stage t *all* samples from $\mathcal{Y}_{t-1}$ carry over to the elite set $\mathcal{X}_t$.

■ **Example 3.17 (Test Problem 112)** Hock and Schittkowski [58] provide a rich source of test problems for multiextremal optimization. A challenging one is Problem 112, where the goal is to find $\boldsymbol{x}$ so as to minimize the function

$$S(\boldsymbol{x}) = \sum_{j=1}^{10} x_j \left(c_j + \ln \frac{x_j}{x_1 + \cdots + x_{10}} \right),$$

subject to the following set of constraints:

$$\begin{aligned} x_1 + 2x_2 + 2x_3 + x_6 + x_{10} - 2 &= 0, \\ x_4 + 2x_5 + x_6 + x_7 - 1 &= 0, \\ x_3 + x_7 + x_8 + 2x_9 + x_{10} - 1 &= 0, \\ x_j &\geqslant 0.000001, \quad j = 1,\ldots,10, \end{aligned}$$

where the constants $\{c_i\}$ are given in Table 3.1.

Table 3.1: Constants for Test Problem 112.

$c_1 = -6.089$	$c_2 = -17.164$	$c_3 = -34.054$	$c_4 = -5.914$	$c_5 = -24.721$
$c_6 = -14.986$	$c_7 = -24.100$	$c_8 = -10.708$	$c_9 = -26.662$	$c_{10} = -22.179$

The best known minimal value in [58] was −47.707579. In [89] a better solution was found, −47.760765, using a genetic algorithm. The corresponding solution vector was completely different from the one in [58]. A further improvement,−47.76109081, was found in [70], using the CE method, giving a similar solution vector to that in [89]:

0.04067247	0.14765159	0.78323637	0.00141368	0.48526222
0.00069291	0.02736897	0.01794290	0.03729653	0.09685870

To obtain a solution with SCO, we first converted this 10-dimensional problem into a 7-dimensional one by defining the objective function

$$S_7(\boldsymbol{y}) = S(\boldsymbol{x}),$$

where $x_2 = y_1, x_3 = y_2, x_5 = y_3, x_6 = y_4, x_7 = y_5, x_9 = y_6, x_{10} = y_7$, and

$$\begin{aligned} x_1 &= 2 - (2y_1 + 2y_2 + y_4 + x_7), \\ x_4 &= 1 - (2y_3 + y_4 + y_5), \\ x_8 &= 1 - (y_2 + y_5 + 2y_6 + y_7), \end{aligned}$$

subject to $x_1, \ldots, x_{10} \geqslant 0.000001$, where the $\{x_i\}$ are taken as functions of the $\{y_i\}$. We then adopted a penalty approach (see Section B.4) by adding a penalty function to the original objective function:

☞ 415

$$\widetilde{S}_7(\boldsymbol{y}) = S(\boldsymbol{x}) + 1000 \sum_{i=1}^{10} \max\{-(x_i - 0.000001), 0\},$$

where, again, the $\{x_i\}$ are defined in terms of the $\{y_i\}$ as above.

Optimizing this last function with SCO, we found, in less time than the other algorithms, a slightly smaller function value: −47.761090859365858, with solution vector

0.040668102417464	0.147730393049955	0.783153291185250	0.001414221643059
0.485246633088859	0.000693172682617	0.027399339496606	0.017947274343948
0.037314369272343	0.096871356429511		

in line with the earlier solutions. ■

3.4.4 Noisy Optimization

In *noisy optimization*, the objective function is unknown, but estimates of function values are available, e.g., via simulation. For example, to find an optimal prediction function g in supervised learning, the exact risk $\ell(g) = \mathbb{E}\,\mathrm{Loss}(Y, g(\boldsymbol{x}))$ is usually unknown and only estimates of the risk are available. Optimizing the risk is thus typically a noisy optimization problem. Noisy optimization features prominently in simulation studies where

NOISY OPTIMIZATION

☞ 20

the behavior of some system (e.g., vehicles on a road network) is simulated under certain parameters (e.g., the lengths of the traffic light intervals) and the aim is to choose those parameters optimally (e.g., to maximize the traffic throughput). For each parameter setting the exact value for the objective function is unknown but estimates can be obtained via the simulation.

In general, suppose the goal is to minimize a function S, where S is unknown, but an estimate of $S(\boldsymbol{x})$ can be obtained for any choice of $\boldsymbol{x} \in \mathcal{X}$. Because the gradient ∇S is unknown, one cannot directly apply classical optimization methods. The *stochastic approximation* method mimics the classical gradient descent method by replacing a deterministic gradient with an estimate $\widehat{\nabla S}(\boldsymbol{x})$.

STOCHASTIC APPROXIMATION

A simple estimator for the i-th component of $\nabla S(\boldsymbol{x})$ (that is, $\partial S(\boldsymbol{u})/\partial x_i$), is the *central difference estimator*

CENTRAL DIFFERENCE ESTIMATOR

$$\frac{\widehat{S}(\boldsymbol{x} + \boldsymbol{e}_i\,\delta/2) - \widehat{S}(\boldsymbol{x} - \boldsymbol{e}_i\,\delta/2)}{\delta}, \tag{3.28}$$

where $\boldsymbol{e}_i$ denotes the i-th unit vector, and $\widehat{S}(\boldsymbol{x}+\boldsymbol{e}_i\,\delta/2)$ and $\widehat{S}(\boldsymbol{x}-\boldsymbol{e}_i\,\delta/2)$ can be any estimators of $S(\boldsymbol{x} + \boldsymbol{e}_i\,\delta/2)$ and $S(\boldsymbol{x} - \boldsymbol{e}_i\,\delta/2)$, respectively. The difference parameter $\delta > 0$ should be small enough to reduce the bias of the estimator, but large enough to keep the variance of the estimator small.

To reduce the variance in the estimator (3.28) it is important to have $\widehat{S}(\boldsymbol{x} + \boldsymbol{e}_i\,\delta/2)$ and $\widehat{S}(\boldsymbol{x} - \boldsymbol{e}_i\,\delta/2)$ positively correlated. This can for example be achieved by using *common random numbers* in the simulation.

COMMON RANDOM NUMBERS

☞ 412

In direct analogy to gradient descent methods, the stochastic approximation method produces a sequence of iterates, starting with some $\boldsymbol{x}_1 \in \mathcal{X}$, via

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \beta_t\,\widehat{\nabla S}(\boldsymbol{x}_t), \tag{3.29}$$

where $\beta_1, \beta_2, \ldots$ is a sequence of strictly positive step sizes. A generic stochastic approximation algorithm for minimizing a function S is thus as follows.

Algorithm 3.4.5: Stochastic Approximation

input: A mechanism to estimate any gradient $\nabla S(\boldsymbol{x})$ and step sizes $\beta_1, \beta_2, \ldots$.
output: Approximate optimizer of S.

1 Initialize $\boldsymbol{x}_1 \in \mathcal{X}$. Set $t \leftarrow 1$.
2 **while** a stopping criterion is not met **do**
3 Obtain an estimated gradient $\widehat{\nabla S}(\boldsymbol{x}_t)$ of S at $\boldsymbol{x}_t$.
4 Determine a step size β_t.
5 Set $\boldsymbol{x}_{t+1} \leftarrow \boldsymbol{x}_t - \beta_t\,\widehat{\nabla S}(\boldsymbol{x}_t)$.
6 $t \leftarrow t + 1$
7 **return** $\boldsymbol{x}_t$

When $\widehat{\nabla S}(\boldsymbol{x}_t)$ is an *unbiased* estimator of $\nabla S(\boldsymbol{x}_t)$ in (3.29) the stochastic approximation Algorithm 3.4.5 is referred to as the *Robbins–Monro* algorithm. When finite differences are used to estimate $\widehat{\nabla S}(\boldsymbol{x}_t)$, as in (3.28), the resulting algorithm is known as the

ROBBINS–MONRO

Kiefer–Wolfowitz algorithm. In Section 9.4.1 we will see how stochastic gradient descent is employed in deep learning to minimize the training loss, based on a "minibatch" of training data.

KIEFER–WOLFOWITZ
☞ 334

It can be shown [72] that, under certain regularity conditions on S, the sequence $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots$ converges to the true minimizer $\boldsymbol{x}^*$ when the step sizes decrease slowly enough to 0; in particular, when

$$\sum_{t=1}^{\infty} \beta_t = \infty \quad \text{and} \quad \sum_{t=1}^{\infty} \beta_t^2 < \infty. \tag{3.30}$$

In practice, one rarely uses step sizes that satisfy (3.30), as the convergence of the sequence will be too slow to be of practical use.

An alternative approach to stochastic approximation is the *stochastic counterpart* method, also called *sample average approximation*. It can be applied in situations where the noisy objective function is of the form

STOCHASTIC COUNTERPART

$$S(\boldsymbol{x}) = \mathbb{E}\widetilde{S}(\boldsymbol{x}, \boldsymbol{\xi}), \quad \boldsymbol{x} \in \mathcal{X}, \tag{3.31}$$

where $\boldsymbol{\xi}$ is a random vector that can be simulated and $\widetilde{S}(\boldsymbol{x}, \boldsymbol{\xi})$ can be evaluated exactly. The idea is to replace the optimization of (3.31) with that of the sample average

$$\widehat{S}(\boldsymbol{x}) = \frac{1}{N} \sum_{i=1}^{N} \widetilde{S}(\boldsymbol{x}, \boldsymbol{\xi}_i), \quad \boldsymbol{x} \in \mathcal{X}, \tag{3.32}$$

where $\boldsymbol{\xi}_1, \ldots, \boldsymbol{\xi}_N$ are iid copies of $\boldsymbol{\xi}$. Note that $\widehat{S}$ is a deterministic function of $\boldsymbol{x}$ and so can be optimized using any optimization algorithm. A solution to this sample average version is taken to be an estimator of a solution $\boldsymbol{x}^*$ to the original problem (3.31).

■ **Example 3.18 (Determining Good Importance Sampling Parameters)** The selection of good importance sampling parameters can be viewed as a stochastic optimization problem. Consider, for instance, the importance sampling estimator in Example 3.14. Recall that the nominal distribution is the uniform distribution on the square $[-b, b]^2$, with pdf

☞ 94

$$f_b(\boldsymbol{x}) = \frac{1}{(2b)^2}, \quad \boldsymbol{x} \in [-b, b]^2,$$

where b is large enough to ensure that μ_b is close to μ; in that example, we chose $b = 1000$. The importance sampling pdf is

$$g_\lambda(\boldsymbol{x}) = f_{R,\Theta}(r, \theta)\frac{1}{r} = \lambda \mathrm{e}^{-\lambda r}\frac{1}{2\pi}\frac{1}{r} = \frac{\lambda \mathrm{e}^{-\lambda\sqrt{x_1^2+x_2^2}}}{2\pi\sqrt{x_1^2+x_2^2}}, \quad \boldsymbol{x} = (x_1, x_2) \in \mathbb{R}^2 \setminus \{\mathbf{0}\},$$

which depends on a free parameter λ. In the example we chose $\lambda = 0.1$. Is this the best choice? Maybe $\lambda = 0.05$ or 0.2 would have resulted in a more accurate estimate. The important thing to realize is that the "effectiveness" of λ can be measured in terms of the variance of the estimator $\widehat{\mu}$ in (3.19), which is given by

☞ 93

$$\frac{1}{N}\mathbb{V}\mathrm{ar}_{g_\lambda}\left(H(\boldsymbol{X})\frac{f(\boldsymbol{X})}{g_\lambda(\boldsymbol{X})}\right) = \frac{1}{N}\mathbb{E}_{g_\lambda}\left[H^2(\boldsymbol{X})\frac{f^2(\boldsymbol{X})}{g_\lambda^2(\boldsymbol{X})}\right] - \frac{\mu^2}{N} = \frac{1}{N}\mathbb{E}_f\left[H^2(\boldsymbol{X})\frac{f(\boldsymbol{X})}{g_\lambda(\boldsymbol{X})}\right] - \frac{\mu^2}{N}.$$

Hence, the optimal parameter λ^* minimizes the function $S(\lambda) = \mathbb{E}_f[H^2(\boldsymbol{X})f(\boldsymbol{X})/g_\lambda(\boldsymbol{X})]$, which is unknown, but can be estimated from simulation. To solve this stochastic minimization problem, we first use stochastic approximation. Thus, at each step of the algorithm, the gradient of $\mathbb{E}S(\lambda)$ is estimated from realizations of $\widehat{S}(\lambda) = H^2(\boldsymbol{X})f(\boldsymbol{X})/g_\lambda(\boldsymbol{X})$, where $\boldsymbol{X} \sim f_b$. As in the original problem (that is, the estimation of μ), the parameter b should be large enough to avoid any bias in the estimator of λ^*, but also small enough to ensure a small variance. The following Python code implements a particular instance of Algorithm 3.4.5. For sampling from f_b here, we used $b = 100$ instead of $b = 1000$, as this will improve the crude Monte Carlo estimation of λ^*, without noticeably affecting the bias. The gradient of $\mathbb{E}S(\lambda)$ is estimated in Lines 11–17, using the central difference estimator (3.28). Notice how for the $S(\lambda-\delta/2)$ and $S(\lambda+\delta/2)$ the *same* random vector $\boldsymbol{X} = [X_1, X_2]^\top$ is used.

☞ 118

This significantly reduces the variance of the gradient estimator; see also Exercise 23. The step size β_t should be such that $\beta_t\widehat{\nabla S}(\boldsymbol{x}_t) \approx \lambda_t$. Given the large gradient here, we choose $\beta_0 = 10^{-7}$ and decrease it each step by a factor of 0.99. Figure 3.13 shows how the sequence $\lambda_0, \lambda_1, \ldots$ decreases towards approximately 0.125, which we take as an estimator for the optimal importance sampling parameter λ^*.

stochapprox.py

```
import numpy as np
from numpy import pi
import matplotlib.pyplot as plt

b=100    # choose b large enough, but not too large
delta = 0.01
H = lambda x1, x2: (2*b)**2*np.exp(-np.sqrt(x1**2 + x2**2)/4)*(np.
    sin(2*np.sqrt(x1**2+x2**2)+1))*(x1**2+x2**2<b**2)
f = 1/(2*b)**2
g = lambda x1, x2, lam: lam*np.exp(-np.sqrt(x1**2+x2**2)*lam)/np.
    sqrt(x1**2+x2**2)/(2*pi)
beta = 10**-7   #step size very small, as the gradient is large
lam=0.25
lams = np.array([lam])
N=10**4
for i in range(200):
    x1 = -b + 2*b*np.random.rand(N,1)
    x2 = -b + 2*b*np.random.rand(N,1)
    lamL = lam - delta/2
    lamR = lam + delta/2
    estL = np.mean(H(x1,x2)**2*f/g(x1, x2, lamL))
    estR = np.mean(H(x1,x2)**2*f/g(x1, x2, lamR))  #use SAME x1,x2
    gr = (estR-estL)/delta  #gradient
    lam = lam - gr*beta  #gradient descend
    lams = np.hstack((lams, lam))
    beta = beta*0.99

lamsize=range(0, (lams.size))
plt.plot(lamsize, lams)
plt.show()
```

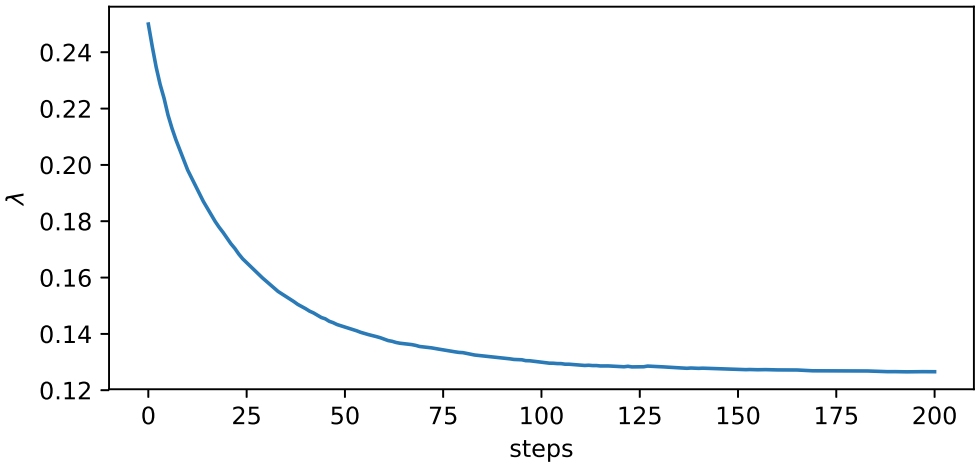


Figure 3.13: The stochastic optimization algorithm produces a sequence $\lambda_t, t = 0, 1, 2, \ldots$ that tends to an approximate estimate of the optimal importance sampling parameter $\lambda^* \approx 0.125$.

Next, we estimate λ^* using a stochastic counterpart approach. As the objective function $S(\lambda)$ is of the form (3.31) (with λ taking the role of $\boldsymbol{x}$ and $\boldsymbol{X}$ the role of $\boldsymbol{\xi}$), we obtain the sample average

$$\widehat{S}(\lambda) = \frac{1}{N} \sum_{i=1}^{N} H^2(\boldsymbol{X}_i) \frac{f(\boldsymbol{X}_i)}{g_\lambda(\boldsymbol{X}_i)}, \tag{3.33}$$

where $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N \sim_{\text{iid}} f_b$. Once the $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N \sim_{\text{iid}} f_b$ have been simulated, $\widehat{S}(\lambda)$ is a deterministic function of λ, which can be optimized by any means. We take the most basic approach and simply evaluate the function for $\lambda = 0.01, 0.02, \ldots, 0.3$ and select the minimizing λ on this grid. The code is given below and Figure 3.14 shows $\widehat{S}(\lambda)$ as a function of λ. The minimum value found was $1.60 \cdot 10^4$ for minimizer $\widehat{\lambda}^* = 0.12$, which is in accordance with the value obtained via stochastic approximation. The sensitivity of this estimate can be assessed from the graph: for a wide range of values (say from 0.04 to 0.15) $\widehat{S}$ stays rather flat. So any of these values could be used in an importance sampling procedure to estimate μ. However, very small values (less than 0.02) and large values (greater than 0.25) should be avoided. Our original choice of $\lambda = 0.1$ was therefore justified and we could not have done much better.

stochcounterpart.py

```
from stochapprox import *

lams = np.linspace(0.01, 0.31, 1000)
res=[]
res = np.array(res)
for i in range(lams.size):
    lam = lams[i]
    np.random.seed(1)
    g = lambda x1, x2: lam*np.exp(-np.sqrt(x1**2+x2**2)*lam)/np.sqrt
        (x1**2+x2**2)/(2*pi)
```

```
    X=-b+2*b*np.random.rand(N,1)
    Y=-b+2*b*np.random.rand(N,1)
    Z=H(X,Y)**2*f/g(X,Y)
    estCMC = np.mean(Z)
    res = np.hstack((res, estCMC))

plt.plot(lams, res)
plt.xlabel(r'$\lambda$')
plt.ylabel(r'$\hat{S}(\lambda)$')
plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
plt.show()
```

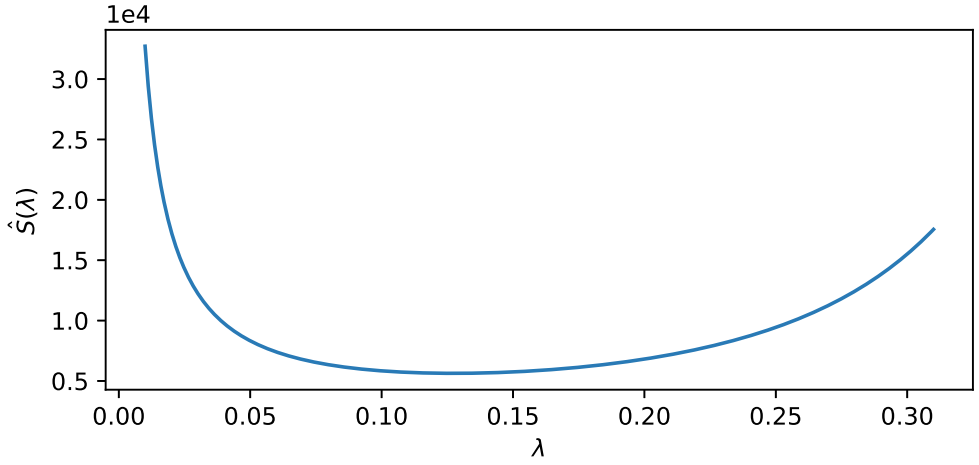


Figure 3.14: The stochastic counterpart method replaces the unknown $S(\lambda)$ (that is, the scaled variance of the importance sampling estimator) with its sample average, $\widehat{S}(\lambda)$. The minimum value of $\widehat{S}$ is attained around $\lambda = 0.12$.

■

A third method for stochastic optimization is the cross-entropy method. In particular, Algorithm 3.4.3 can easily be modified to minimize *noisy* functions $S(\boldsymbol{x}) = \mathbb{E}\widetilde{S}(\boldsymbol{x}, \boldsymbol{\xi})$, as defined in (3.31). The only change required in the algorithm is that every function value $S(\boldsymbol{x})$ be replaced by its estimate $\widehat{S}(\boldsymbol{x})$. Depending on the level of noise in the function, the sample size N might have to be increased considerably.

☞ 101

■ **Example 3.19 (Cross-Entropy Method for Noisy Optimization)** To explore the use of the CE method for noisy optimization, take the following noisy discrete optimization problem. Suppose there is a "black box" that contains an unknown binary sequence of n bits. If one feeds the black box any input vector, it will first scramble the input by independently flipping the bits (changing 0 to 1 and 1 to 0) with a probability θ and then return the number of bits that match the true (unknown) binary sequence. This is illustrated in Figure 3.15 for $n = 10$.

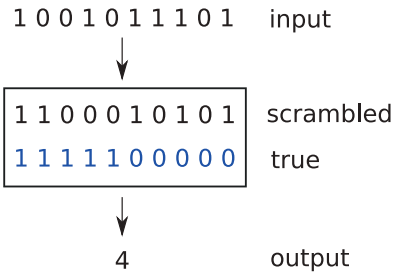


Figure 3.15: A noisy optimization function as a black box. The input to the black box is a binary vector. Inside the black box the digits of the input vector are scrambled by flipping bits with probability θ. The output is the number of bits of the scrambled vector that match the true (unknown) binary vector.

Denoting by $S(\boldsymbol{x})$ the true number of matching digits for a binary input vector $\boldsymbol{x}$, the black box thus returns a noisy estimate $\widehat{S}(\boldsymbol{x})$. The objective is to estimate the binary sequence inside the black box, by feeding it with many input vectors and observing their output. Or, to put it in a different way, to maximize $S(\boldsymbol{x})$ using $\widehat{S}(\boldsymbol{x})$ as a proxy. Since there are 2^n possible input vectors, it is infeasible to try all possible vectors $\boldsymbol{x}$ even for moderate n.

The following Python program implements the noisy function $\widehat{S}(\boldsymbol{x})$ for $n = 100$. Each input bit is flipped with a rather high probability $\theta = 0.4$, so that the output is a poor indicator of how many bits actually match the true vector. This true vector has 1s at positions $1, \ldots, 50$ and 0s at $51, \ldots, 100$.

Snoisy.py

```
import numpy as np

def Snoisy(X):   #takes a matrix
    n = X.shape[1]
    N = X.shape[0]
    # true binary vector
    xorg = np.hstack((np.ones((1,n//2)), np.zeros((1,n//2))))
    theta = 0.4 # probability to flip the input
    # storing the number of bits unequal to the true vector
    s = np.zeros(N)
    for i in range(0,N):
       # determine which bits to flip
       flip = (np.random.uniform(size=(n)) < theta).astype(int)
       ind = flip>0
       X[i][ind] = 1-X[i][ind]
       s[i] = (X[i] != xorg).sum()
    return s
```

The CE code below to optimize $S(\boldsymbol{x})$ is quite similar to the continuous optimization code in Example 3.16. However, instead of sampling iid random variables $X_1, \ldots, X_N$ from a normal distribution, we now sample iid binary vectors $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N$ from a $\mathsf{Ber}(\boldsymbol{p})$ distribution. More precisely, given a row vector of probabilities $\boldsymbol{p} = [p_1, \ldots, p_n]$, we independently simulate the components $X_1, \ldots, X_n$ of each binary vector $\boldsymbol{X}$ according to $X_i \sim \mathsf{Ber}(p_i)$, $i = 1, \ldots, n$. After each iteration, the vector $\boldsymbol{p}$ is updated as the (vector) mean of the elite

☞ 101

samples. Note that, in contrast to the minimization problem in Example 3.16, the elite samples now correspond to the *largest* function values. The sample size is $N = 1000$ and the number of elite samples is 200. The components of the initial sampling vector $\boldsymbol{p}$ are all equal to $1/2$; that is, the $\boldsymbol{X}$ are initially uniformly sampled from the set of all binary vectors of length $n = 100$. At each subsequent iteration the parameter vector is updated via the mean of the elite samples and evolves towards a degenerate vector $\boldsymbol{p}^*$ with only 1s and 0s. Sampling from such a $\mathsf{Ber}(\boldsymbol{p}^*)$ distribution gives an outcome $\boldsymbol{x}^* = \boldsymbol{p}^*$, which can be taken as an estimate for the maximizer of S; that is, the true binary vector hidden in the black box. The algorithm stops when $\boldsymbol{p}$ has degenerated sufficiently.

Figure 3.16 shows the evolution of the vector of probabilities $\boldsymbol{p}$. This figure may be seen as the discrete analogue of Figure 3.12. We see that, despite the high noise, the CE method is able to find the true state of the black box, and hence the maximum value of S.

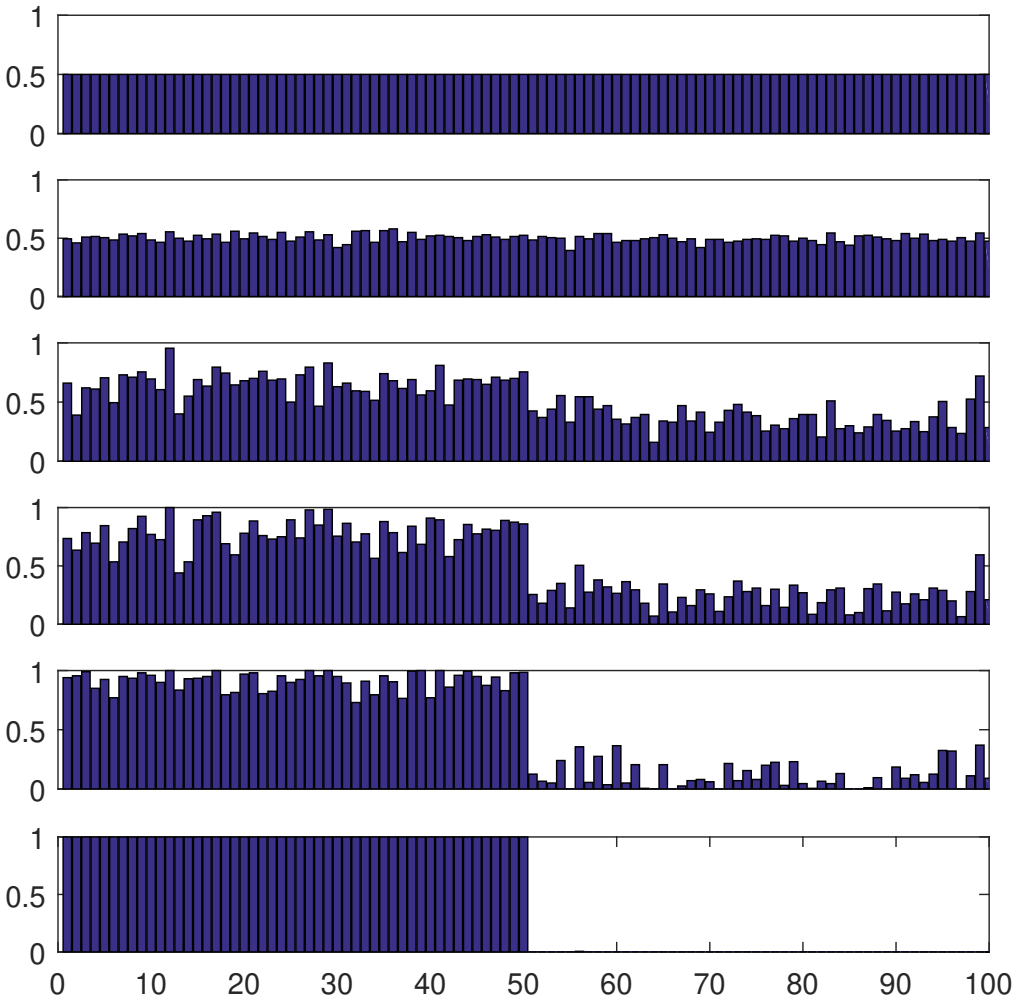


Figure 3.16: Evolution of the vector of probabilities $\boldsymbol{p} = [p_1, \ldots, p_n]$ towards the degenerate solution.

CEnoisy.py

```
from Snoisy import Snoisy
import numpy as np
n = 100
rho = 0.1
N = 1000; Nel = int(N*rho); eps = 0.01
p = 0.5*np.ones(n)
i = 0
pstart = p
ps = np.zeros((1000,n))
ps[0] = pstart
pdist = np.zeros((1,1000))
while np.max(np.minimum(p,1-p)) > eps:
    i += 1
    X = (np.random.uniform(size=(N,n)) < p).astype(int)
    X_tmp = np.array(X, copy=True)
    SX = Snoisy(X_tmp)
    ids = np.argsort(SX,axis=0)
    Elite = X[ids[0:Nel],:]
    p = np.mean(Elite,axis=0)
    ps[i] = p
print(p)
```

Further Reading

The article [68] explores why the Monte Carlo method is so important in today's quantitative investigations. The *Handbook of Monte Carlo Methods* [71] provides a comprehensive overview of Monte Carlo simulation that explores the latest topics, techniques, and real-world applications. Popular books on simulation and the Monte Carlo method include [42], [75], and [104]. A classic reference on random variable generation is [32]. Easy introductions to stochastic simulation are given in [49], [98], and [100]. More advanced theory can be found in [5]. Markov chain Monte Carlo is detailed in [50] and [99]. The research monograph on the cross-entropy method is [103] and a tutorial is provided in [30]. A range of optimization applications of the CE method is given in [16]. Theoretical results on adaptive tuning schemes for simulated annealing may be found, for example, in [111]. There are several established ways for gradient estimation. These include the finite difference method, infinitesimal perturbation analysis, the score function method, and the method of weak derivatives; see, for example, [51, Chapter 7].

Exercises

1. We can modify the Box–Muller method in Example 3.1 to draw X and Y uniformly on the unit disc, $\{(x, y) \in \mathbb{R}^2 : x^2 + y^2 \leqslant 1\}$, in the following way: Independently draw a radius R and an angle $\Theta \sim \mathcal{U}(0, 2\pi)$, and return $X = R\cos(\Theta)$, $Y = R\sin(\Theta)$. The question is how to draw R. ☞ 69

   (a) Show that the cdf of R is given by $F_R(r) = r^2$ for $0 \leqslant r \leqslant 1$ (with $F_R(r) = 0$ and

$F_R(r) = 1$ for $r < 0$ and $r > 1$, respectively).

(b) Explain how to simulate R using the inverse-transform method.

(c) Simulate 100 independent draws of $[X, Y]^\top$ according to the method described above.

2. A simple acceptance–rejection method to simulate a vector $\boldsymbol{X}$ in the unit d-ball $\{\boldsymbol{x} \in \mathbb{R}^d : \|\boldsymbol{x}\| \leqslant 1\}$ is to first generate $\boldsymbol{X}$ uniformly in the hyper cube $[-1, 1]^d$ and then to accept the point only if $\|\boldsymbol{X}\| \leqslant 1$. Determine an analytic expression for the probability of acceptance as a function of d and plot this for $d = 1, \ldots, 50$.

3. Let the random variable X have pdf

$$f(x) = \begin{cases} \frac{1}{2}\, x\,, & 0 \leqslant x < 1, \\ \frac{1}{2}\,, & 1 \leqslant x \leqslant \frac{5}{2}\,. \end{cases}$$

Simulate a random variable from $f(x)$, using

(a) the inverse-transform method;

(b) the acceptance–rejection method, using the proposal density

$$g(x) = \frac{8}{25}\, x\,, \quad 0 \leqslant x \leqslant \frac{5}{2}\,.$$

4. Construct simulation algorithms for the following distributions:

(a) The $\mathsf{Weib}(\alpha, \lambda)$ distribution, with cdf $F(x) = 1 - \mathrm{e}^{-(\lambda x)^\alpha}$, $x \geqslant 0$, where $\lambda > 0$ and $\alpha > 0$.

(b) The $\mathsf{Pareto}(\alpha, \lambda)$ distribution, with pdf $f(x) = \alpha\lambda(1 + \lambda x)^{-(\alpha+1)}$, $x \geqslant 0$, where $\lambda > 0$ and $\alpha > 0$.

5. We wish to sample from the pdf

$$f(x) = x\,\mathrm{e}^{-x}, \quad x \geqslant 0,$$

using acceptance–rejection with the proposal pdf $g(x) = \mathrm{e}^{-x/2}/2$, $x \geqslant 0$.

(a) Find the smallest C for which $Cg(x) \geqslant f(x)$ for all x.

(b) What is the efficiency of this acceptance–rejection method?

6. Let $[X, Y]^\top$ be uniformly distributed on the triangle with corners $(0, 0), (1, 2)$, and $(-1, 1)$. Give the distribution of $[U, V]^\top$ defined by the linear transformation

$$\begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}.$$

7. Explain how to generate a random variable from the *extreme value distribution*, which has cdf

$$F(x) = 1 - \mathrm{e}^{-\exp\left(\frac{x-\mu}{\sigma}\right)}\,, \quad -\infty < x < \infty, \quad (\sigma > 0),$$

via the inverse-transform method.

8. Write a program that generates and displays 100 random vectors that are uniformly distributed within the ellipse

$$5\,x^2 + 21\,x\,y + 25\,y^2 = 9.$$

[Hint: Consider generating uniformly distributed samples within the circle of radius 3 and use the fact that linear transformations preserve uniformity to transform the circle to the given ellipse.]

9. Suppose that $X_i \sim \mathsf{Exp}(\lambda_i)$, independently, for all $i = 1, \ldots, n$. Let $\mathbf{\Pi} = [\Pi_1, \ldots, \Pi_n]^\top$ be the random permutation induced by the ordering $X_{\Pi_1} < X_{\Pi_2} < \cdots < X_{\Pi_n}$, and define $Z_1 := X_{\Pi_1}$ and $Z_j := X_{\Pi_j} - X_{\Pi_{j-1}}$ for $j = 2, \ldots, n$.

   (a) Determine an $n \times n$ matrix $\mathbf{A}$ such that $\boldsymbol{Z} = \mathbf{A}\boldsymbol{X}$ and show that $\det(\mathbf{A}) = 1$.

   (b) Denote the joint pdf of $\boldsymbol{X}$ and $\mathbf{\Pi}$ as

$$f_{\boldsymbol{X},\mathbf{\Pi}}(\boldsymbol{x}, \boldsymbol{\pi}) = \prod_{i=1}^{n} \lambda_{\pi_i} \exp\left(-\lambda_{\pi_i} x_{\pi_i}\right) \times \mathbb{1}\{x_{\pi_1} < \cdots < x_{\pi_n}\}, \quad \boldsymbol{x} \geqslant \mathbf{0},\ \boldsymbol{\pi} \in \mathcal{P}_n,$$

   where $\mathcal{P}_n$ is the set of all $n!$ permutations of $\{1, \ldots, n\}$. Use the multivariate transformation formula (C.22) to show that ☞ 432

$$f_{\boldsymbol{Z},\mathbf{\Pi}}(\boldsymbol{z}, \boldsymbol{\pi}) = \exp\left(-\sum_{i=1}^{n} z_i \sum_{k \geqslant i} \lambda_{\pi_k}\right) \prod_{i=1}^{n} \lambda_i, \qquad \boldsymbol{z} \geqslant \mathbf{0},\ \boldsymbol{\pi} \in \mathcal{P}_n.$$

   Hence, conclude that the probability mass function of the random permutation $\mathbf{\Pi}$ is:

$$\mathbb{P}[\mathbf{\Pi} = \boldsymbol{\pi}] = \prod_{i=1}^{n} \frac{\lambda_{\pi_i}}{\sum_{k \geqslant i} \lambda_{\pi_k}}, \quad \boldsymbol{\pi} \in \mathcal{P}_n.$$

   (c) Write pseudo-code to simulate a *uniform* random permutation $\mathbf{\Pi} \in \mathcal{P}_n$; that is, such that $\mathbb{P}[\mathbf{\Pi} = \boldsymbol{\pi}] = \frac{1}{n!}$, and explain how this uniform random permutation can be used to reshuffle a training set τ_n.

10. Consider the Markov chain with transition graph given in Figure 3.17, starting in state 1.

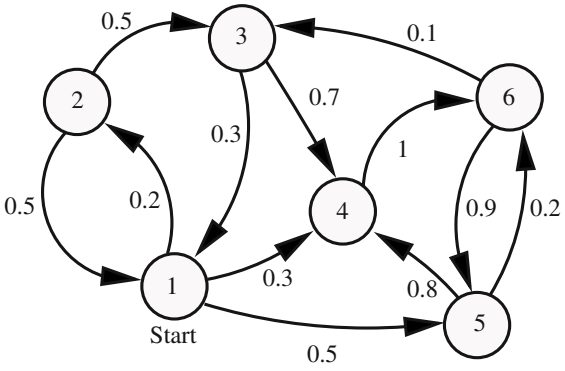


Figure 3.17: The transition graph for the Markov chain $\{X_t, t = 0, 1, 2, \ldots\}$.

(a) Construct a computer program to simulate the Markov chain, and show a realization for $N = 100$ steps.

(b) Compute the limiting probabilities that the Markov chain is in state 1,2,...,6, by solving the global balance equations (C.42).

☞ 452

(c) Verify that the exact limiting probabilities correspond to the average fraction of times that the Markov process visits states 1,2,...,6, for a large number of steps N.

☞ 453

11. As a generalization of Example C.9, consider a random walk on an arbitrary undirected connected graph with a finite vertex set $\mathcal{V}$. For any vertex $v \in \mathcal{V}$, let $d(v)$ be the number of neighbors of v — called the *degree* of v. The random walk can jump to each one of the neighbors with probability $1/d(v)$ and can be described by a Markov chain. Show that, if the chain is *aperiodic*, the limiting probability that the chain is in state v is equal to $d(v)/\sum_{v' \in \mathcal{V}} d(v')$.

☞ 76

12. Let $U, V \sim_{\text{iid}} \mathcal{U}(0,1)$. The reason why in Example 3.7 the sample mean and sample median behave very differently is that $\mathbb{E}[U/V] = \infty$, while the median of U/V is finite. Show this, and compute the median. [Hint: start by determining the cdf of $Z = U/V$ by writing it as an expectation of an indicator function.]

13. Consider the problem of generating samples from $Y \sim \mathsf{Gamma}(2, 10)$.

(a) Direct simulation: Let $U_1, U_2 \sim_{\text{iid}} \mathcal{U}(0,1)$. Show that $-\ln(U_1)/10 - \ln(U_2)/10 \sim \mathsf{Gamma}(2, 10)$. [Hint: derive the distribution of $-\ln(U_1)/10$ and use Example C.1.]

☞ 427

(b) Simulation via MCMC: Implement an independence sampler to simulate from the $\mathsf{Gamma}(2, 10)$ target pdf

$$f(x) = 100\, x\, \mathrm{e}^{-10x}, \quad x \geqslant 0,$$

using proposal transition density $q(y \,|\, x) = g(y)$, where $g(y)$ is the pdf of an $\mathsf{Exp}(5)$ random variable. Generate $N = 500$ samples, and compare the true cdf with the empirical cdf of the data.

14. Let $\boldsymbol{X} = [X, Y]^\top$ be a random column vector with a bivariate normal distribution with expectation vector $\boldsymbol{\mu} = [1, 2]^\top$ and covariance matrix

$$\boldsymbol{\Sigma} = \begin{bmatrix} 1 & a \\ a & 4 \end{bmatrix}.$$

(a) What are the conditional distributions of $(Y \,|\, X = x)$ and $(X \,|\, Y = y)$? [Hint: use Theorem C.8.]

☞ 436

(b) Implement a Gibbs sampler to draw 10^3 samples from the bivariate distribution $\mathcal{N}(\boldsymbol{\mu}, \Sigma)$ for $a = 0$, 1, and 1.75, and plot the resulting samples.

15. Here the objective is to sample from the 2-dimensional pdf

$$f(x, y) = c\, \mathrm{e}^{-(xy+x+y)}, \quad x \geqslant 0, \quad y \geqslant 0,$$

for some normalization constant c, using a Gibbs sampler. Let $(X, Y) \sim f$.

(a) Find the conditional pdf of X given $Y = y$, and the conditional pdf of Y given $X = x$.

(b) Write working Python code that implements the Gibbs sampler and outputs 1000 points that are approximately distributed according to f.

(c) Describe how the normalization constant c could be estimated via Monte Carlo simulation, using random variables $X_1, \dots, X_N, Y_1, \dots, Y_N \overset{\text{iid}}{\sim} \mathsf{Exp}(1)$.

16. We wish to estimate $\mu = \int_{-2}^{2} \mathrm{e}^{-x^2/2}\,\mathrm{d}x = \int H(x) f(x)\,\mathrm{d}x$ via Monte Carlo simulation using two different approaches: (1) defining $H(x) = 4\,\mathrm{e}^{-x^2/2}$ and f the pdf of the $\mathcal{U}[-2, 2]$ distribution and (2) defining $H(x) = \sqrt{2\pi}\,\mathbb{1}\{-2 \leqslant x \leqslant 2\}$ and f the pdf of the $\mathcal{N}(0, 1)$ distribution.

(a) For both cases estimate μ via the estimator $\widehat{\mu}$

$$\widehat{\mu} = N^{-1} \sum_{i=1}^{N} H(\boldsymbol{X}_i). \tag{3.34}$$

Use a sample size of $N = 1000$.

(b) For both cases estimate the relative error κ of $\widehat{\mu}$ using $N = 100$.

(c) Give a 95% confidence interval for μ for both cases using $N = 100$.

(d) From part (b), assess how large N should be such that the relative width of the confidence interval is less than 0.01, and carry out the simulation with this N. Compare the result with the true value of μ.

17. Consider estimation of the tail probability $\mu = \mathbb{P}[X \geqslant \gamma]$ of some random variable X, where γ is large. The crude Monte Carlo estimator of μ is

$$\widehat{\mu} = \frac{1}{N} \sum_{i=1}^{N} Z_i, \tag{3.35}$$

where $X_1, \dots, X_N$ are iid copies of X and $Z_i = \mathbb{1}\{X_i \geqslant \gamma\}$, $i = 1, \dots, N$.

(a) Show that $\widehat{\mu}$ is unbiased; that is, $\mathbb{E}\,\widehat{\mu} = \mu$.

(b) Express the relative error of $\widehat{\mu}$, i.e.,

$$\mathrm{RE} = \frac{\sqrt{\mathbb{V}\mathrm{ar}\,\widehat{\mu}}}{\mathbb{E}\,\widehat{\mu}},$$

in terms of N and μ.

(c) Explain how to estimate the relative error of $\widehat{\mu}$ from outcomes $x_1, \dots, x_N$ of $X_1, \dots, X_N$, and how to construct a 95% confidence interval for μ.

(d) An unbiased estimator Z of μ is said to be *logarithmically efficient* if

$$\lim_{\gamma \to \infty} \frac{\ln \mathbb{E} Z^2}{\ln \mu^2} = 1. \tag{3.36}$$

Show that the CMC estimator (3.35) with $N = 1$ is not logarithmically efficient.

18. One of the test cases in [70] involves the minimization of the *Hougen* function. Implement a cross-entropy and a simulated annealing algorithm to carry out this optimization task.

19. In the *binary knapsack problem*, the goal is to solve the optimization problem:

$$\max_{\boldsymbol{x}\in\{0,1\}^n} \boldsymbol{p}^\top \boldsymbol{x},$$

subject to the constraints

$$\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{c},$$

where $\boldsymbol{p}$ and $\boldsymbol{w}$ are $n \times 1$ vectors of non-negative numbers, $\mathbf{A} = (a_{ij})$ is an $m \times n$ matrix, and $\boldsymbol{c}$ is an $m \times 1$ vector. The interpretation is that $x_j = 1$ or 0 depending on whether item j with value p_j is packed into the knapsack or not , $j = 1, \ldots, n$; The variable a_{ij} represents the i-th attribute (e.g., volume, weight) of the j-th item. Associated with each attribute is a maximal capacity, e.g., c_1 could be the maximum volume of the knapsack, c_2 the maximum weight, etc.

Write a CE program to solve the `Sento1.dat` knapsack problem at http://people.brunel.ac.uk/~mastjjb/jeb/orlib/files/mknap2.txt, as described in [16].

20. Let $(C_1, R_1), (C_2, R_2), \ldots$ be a renewal reward process, with $\mathbb{E}R_1 < \infty$ and $\mathbb{E}C_1 < \infty$. Let $A_t = \sum_{i=1}^{N_t} R_i/t$ be the average reward at time $t = 1, 2, \ldots$, where $N_t = \max\{n : T_n \leqslant t\}$ and we have defined $T_n = \sum_{i=1}^{n} C_i$ as the time of the n-th renewal.

    (a) Show that $T_n/n \xrightarrow{\text{a.s.}} \mathbb{E}C_1$ as $n \to \infty$.

    (b) Show that $N_t \xrightarrow{\text{a.s.}} \infty$ as $t \to \infty$.

    (c) Show that $N_t/t \xrightarrow{\text{a.s.}} 1/\mathbb{E}C_1$ as $t \to \infty$. [Hint: Use the fact that $T_{N_t} \leqslant t \leqslant T_{N_t+1}$ for all $t = 1, 2, \ldots$.]

    (d) Show that

$$A_t \xrightarrow{\text{a.s.}} \frac{\mathbb{E}R_1}{\mathbb{E}C_1} \quad \text{as} \quad t \to \infty.$$

☞ 92

21. Prove Theorem 3.3.

☞ 96

22. Prove that if $H(\boldsymbol{x}) \geqslant 0$ the importance sampling pdf g^* in (3.22) gives the zero-variance importance sampling estimator $\widehat{\mu} = \mu$.

23. Let X and Y be random variables (not necessarily independent) and suppose we wish to estimate the expected difference $\mu = \mathbb{E}[X - Y] = \mathbb{E}X - \mathbb{E}Y$.

    (a) Show that if X and Y are *positively correlated*, the variance of $X - Y$ is smaller than if X and Y are *independent*.

    (b) Suppose now that X and Y have cdfs F and G, respectively, and are simulated via the inverse-transform method: $X = F^{-1}(U)$, $Y = G^{-1}(V)$, with $U, V \sim \mathcal{U}(0, 1)$, not necessarily independent. Intuitively, one might expect that

if U and V are positively correlated, the variance of $X-Y$ would be smaller than if U and V are independent. Show that this is not always the case by providing a counter-example.

(c) Continuing (b), assume now that F and G are continuous. Show that the variance of $X - Y$ by taking *common random numbers* $U = V$ is no larger than when U and V are independent. [Hint: Use the following lemma of Hoeffding [41]: If (X, Y) have joint cdf H with marginal cdfs of X and Y being F and G, respectively, then

$$\mathbb{C}\mathrm{ov}(X, Y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} (H(x, y) - F(x)\, G(y))\, \mathrm{d}x\, \mathrm{d}y,$$

provided $\mathbb{C}\mathrm{ov}(X, Y)$ exists.]


Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 4

UNSUPERVISED LEARNING

When there is no distinction between response and explanatory variables, unsupervised methods are required to learn the structure of the data. In this chapter we look at various unsupervised learning techniques, such as density estimation, clustering, and principal component analysis. Important tools in unsupervised learning include the cross-entropy training loss, mixture models, the Expectation–Maximization algorithm, and the Singular Value Decomposition.

4.1 Introduction

In contrast to supervised learning, where an "output" (response) variable y is explained by an "input" (explanatory) vector $\boldsymbol{x}$, in unsupervised learning there is no response variable and the overall goal is to extract useful information and patterns from the data, e.g., in the form $\tau = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ or as a matrix $\mathbf{X}^\top = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]$. In essence, the objective of unsupervised learning is to learn about the underlying probability distribution of the data.

We start in Section 4.2 by setting up a framework for unsupervised learning that is similar to the framework used for supervised learning in Section 2.3. That is, we formulate ☞ 23 unsupervised learning in terms of risk and loss minimization; but now involving the cross-entropy risk, rather than the squared-error risk. In a natural way this leads to fundamental learning concepts such as likelihood, Fisher information, and the Akaike information criterion. Section 4.3 introduces the Expectation–Maximization (EM) algorithm as a useful method for maximizing likelihood functions when their solution cannot be found easily in closed form.

If the data forms an iid sample from some unknown distribution, the "empirical distribution" of the data provides valuable information about the unknown distribution. In Section 4.4 we formalize the concept of the empirical distribution (a generalization of the empirical cdf) and explain how we can produce an estimate of the underlying probability ☞ 11 density function of the data using kernel density estimators.

Most unsupervised learning techniques focus on identifying certain traits of the underlying distribution, such as its local maximizers. A related idea is to partition the data into clusters of points that are in some sense "similar" to each other. In Section 4.5 we formulate the clustering problem in terms of a mixture model. In particular, the data are assumed ☞ 135

to come from a mixture of (usually Gaussian) distributions, and the objective is to recover the parameters of the mixture distributions from the data. The principal tool for parameter estimation in mixture models is the EM algorithm.

Section 4.6 discusses a more heuristic approach to clustering, where the data are grouped according to certain "cluster centers", whose positions are found by solving an optimization problem. Section 4.7 describes how clusters can be constructed in a hierarchical manner.

Finally, in Section 4.8 we discuss the unsupervised learning technique called Principal Component Analysis (PCA), which is an important tool for reducing the dimensionality of the data.

We will revisit various unsupervised learning techniques in subsequent chapters on *supervised* learning. For example, cross-entropy training loss minimization will be important in logistic regression (Section 5.7) and classification (Chapter 7), and PCA can be used for variable selection and dimensionality reduction, to make models easier to train and increase their predictive power; see e.g., Sections 6.8 and 7.4.

☞ 204
☞ 251

4.2 Risk and Loss in Unsupervised Learning

In unsupervised learning, the training data $\mathcal{T} := \{\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\}$ only consists of (what are usually assumed to be) independent copies of a feature vector $\boldsymbol{X}$; there is no response data. Suppose our objective is to learn the unknown pdf f of $\boldsymbol{X}$ based on an outcome $\tau = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ of the training data $\mathcal{T}$. Conveniently, we can follow the same line of reasoning as for *supervised* learning, discussed in Sections 2.3–2.5. Table 4.1 gives a summary of definitions for the case of unsupervised learning. Compare this with Table 2.1 for the supervised case.

☞ 23
☞ 25

Similar to supervised learning, we wish to find a function g, which is now a probability density (continuous or discrete), that best approximates the pdf f in terms of minimizing a risk

$$\ell(g) := \mathbb{E}\,\mathrm{Loss}(f(\boldsymbol{X}), g(\boldsymbol{X})), \tag{4.1}$$

where Loss is a loss function. In (2.27), we already encountered the Kullback–Leibler risk

$$\ell(g) := \mathbb{E}\ln\frac{f(\boldsymbol{X})}{g(\boldsymbol{X})} = \mathbb{E}\ln f(\boldsymbol{X}) - \mathbb{E}\ln g(\boldsymbol{X}). \tag{4.2}$$

If $\mathcal{G}$ is a class of functions that contains f, then minimizing the Kullback–Leibler risk over $\mathcal{G}$ will yield the (correct) minimizer f. Of course, the problem is that minimization of (4.2) depends on f, which is generally not known. However, since the term $\mathbb{E}\ln f(\boldsymbol{X})$ does not depend on g, it plays no role in the minimization of the Kullback–Leibler risk. By removing this term, we obtain the *cross-entropy risk* (for discrete $\boldsymbol{X}$ replace the integral with a sum):

CROSS-ENTROPY RISK

$$\ell(g) := -\mathbb{E}\ln g(\boldsymbol{X}) = -\int f(\boldsymbol{x})\,\ln g(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}. \tag{4.3}$$

Thus, minimizing the cross-entropy risk (4.3) over all $g \in \mathcal{G}$, again gives the minimizer f, provided that $f \in \mathcal{G}$. Unfortunately, solving (4.3) is also infeasible in general, as it still

Table 4.1: Summary of definitions for unsupervised learning.

$\boldsymbol{x}$	Fixed feature vector.
$\boldsymbol{X}$	Random feature vector.
$f(\boldsymbol{x})$	Pdf of $\boldsymbol{X}$ evaluated at the point $\boldsymbol{x}$.
τ or τ_n	Fixed training data $\{\boldsymbol{x}_i, i = 1, \ldots, n\}$.
$\mathcal{T}$ or $\mathcal{T}_n$	Random training data $\{\boldsymbol{X}_i, i = 1, \ldots, n\}$.
g	Approximation of the pdf f.
$\text{Loss}(f(\boldsymbol{x}), g(\boldsymbol{x}))$	Loss incurred when approximating $f(\boldsymbol{x})$ with $g(\boldsymbol{x})$.
$\ell(g)$	Risk for approximation function g; that is, $\mathbb{E}\,\text{Loss}(f(\boldsymbol{X}), g(\boldsymbol{X}))$.
$g^{\mathcal{G}}$	Optimal approximation function in function class $\mathcal{G}$; that is, $\text{argmin}_{g\in\mathcal{G}}\,\ell(g)$.
$\ell_\tau(g)$	Training loss for approximation function (guess) 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 *learner*: $\text{argmin}_{g\in\mathcal{G}}\,\ell_\tau(g)$. That is, the optimal approximation 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 for a random training set $\mathcal{T}$.

depends on f. Instead, we seek to minimize the *cross-entropy training loss*:

CROSS-ENTROPY TRAINING LOSS

$$\ell_\tau(g) := \frac{1}{n}\sum_{i=1}^{n} \text{Loss}(f(\boldsymbol{x}_i), g(\boldsymbol{x}_i)) = -\frac{1}{n}\sum_{i=1}^{n} \ln g(\boldsymbol{x}_i) \tag{4.4}$$

over the class of functions $\mathcal{G}$, where $\tau = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ is an iid sample from f. This optimization is doable without knowing f and is equivalent to solving the maximization problem

$$\max_{g\in\mathcal{G}} \sum_{i=1}^{n} \ln g(\boldsymbol{x}_i). \tag{4.5}$$

A key step in setting up the learning procedure is to select a suitable function class $\mathcal{G}$ over which to optimize. The standard approach is to parameterize g with a parameter $\boldsymbol{\theta}$ and let $\mathcal{G}$ be the class of functions $\{g(\cdot\,|\,\boldsymbol{\theta}), \boldsymbol{\theta} \in \Theta\}$ for some p-dimensional parameter set Θ. For the remainder of Section 4.2, we will be using this function class, as well as the cross-entropy risk.

The function $\boldsymbol{\theta} \mapsto g(\boldsymbol{x}\,|\,\boldsymbol{\theta})$ is called the *likelihood function*. It gives the likelihood of the observed feature vector $\boldsymbol{x}$ under $g(\cdot\,|\,\boldsymbol{\theta})$, as a function of the parameter $\boldsymbol{\theta}$. The natural logarithm of the likelihood function is called the *log-likelihood* function and its gradient with respect to $\boldsymbol{\theta}$ is called the *score function*, denoted $\boldsymbol{S}(\boldsymbol{x}\,|\,\boldsymbol{\theta})$; that is,

LIKELIHOOD FUNCTION

SCORE FUNCTION

$$\boldsymbol{S}(\boldsymbol{x}\,|\,\boldsymbol{\theta}) := \frac{\partial \ln g(\boldsymbol{x}\,|\,\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \frac{\frac{\partial g(\boldsymbol{x}\,|\,\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}}{g(\boldsymbol{x}\,|\,\boldsymbol{\theta})}. \tag{4.6}$$

The random score $\boldsymbol{S}(\boldsymbol{X} \mid \boldsymbol{\theta})$, with $\boldsymbol{X} \sim g(\cdot \mid \boldsymbol{\theta})$, is of particular interest. In many cases, its expectation is *equal to the zero vector*; namely,

$$\begin{aligned} \mathbb{E}_{\boldsymbol{\theta}} \boldsymbol{S}(\boldsymbol{X} \mid \boldsymbol{\theta}) &= \int \frac{\frac{\partial g(\boldsymbol{x} \mid \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}}{g(\boldsymbol{x} \mid \boldsymbol{\theta})} g(\boldsymbol{x} \mid \boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{x} \\ &= \int \frac{\partial g(\boldsymbol{x} \mid \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \, \mathrm{d}\boldsymbol{x} = \frac{\partial \int g(\boldsymbol{x} \mid \boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{x}}{\partial \boldsymbol{\theta}} = \frac{\partial 1}{\partial \boldsymbol{\theta}} = \mathbf{0}, \end{aligned} \tag{4.7}$$

provided that the interchange of differentiation and integration is justified. This is true for a large number of distributions, including the normal, exponential, and binomial distributions. Notable exceptions are distributions whose support depends on the distributional parameter; for example the $\mathcal{U}(0, \theta)$ distribution.

It is important to see whether expectations are taken with respect to $\boldsymbol{X} \sim g(\cdot \mid \boldsymbol{\theta})$ or $\boldsymbol{X} \sim f$. We use the expectation symbols $\mathbb{E}_{\boldsymbol{\theta}}$ and $\mathbb{E}$ to distinguish the two cases.

From now on we simply assume that the interchange of differentiation and integration is permitted; see, e.g., [76] for sufficient conditions. The covariance matrix of the random score $\boldsymbol{S}(\boldsymbol{X} \mid \boldsymbol{\theta})$ is called the *Fisher information matrix*, which we denote by $\mathbf{F}$ or $\mathbf{F}(\boldsymbol{\theta})$ to show its dependence on $\boldsymbol{\theta}$. Since the expected score is $\mathbf{0}$, we have

FISHER INFORMATION MATRIX

$$\mathbf{F}(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\theta}}[\boldsymbol{S}(\boldsymbol{X} \mid \boldsymbol{\theta}) \, \boldsymbol{S}(\boldsymbol{X} \mid \boldsymbol{\theta})^{\top}]. \tag{4.8}$$

☞ 398

A related matrix is the expected Hessian matrix of $-\ln g(\boldsymbol{X} \mid \boldsymbol{\theta})$:

$$\mathbf{H}(\boldsymbol{\theta}) := \mathbb{E}\left[-\frac{\partial \boldsymbol{S}(\boldsymbol{X}; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right] = -\mathbb{E}\begin{bmatrix} \frac{\partial^2 \ln g(\boldsymbol{X} \mid \boldsymbol{\theta})}{\partial^2 \theta_1} & \frac{\partial^2 \ln g(\boldsymbol{X} \mid \boldsymbol{\theta})}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 \ln g(\boldsymbol{X} \mid \boldsymbol{\theta})}{\partial \theta_1 \partial \theta_p} \\ \frac{\partial^2 \ln g(\boldsymbol{X} \mid \boldsymbol{\theta})}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 \ln g(\boldsymbol{X} \mid \boldsymbol{\theta})}{\partial^2 \theta_2} & \cdots & \frac{\partial^2 \ln g(\boldsymbol{X} \mid \boldsymbol{\theta})}{\partial \theta_2 \partial \theta_p} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \ln g(\boldsymbol{X} \mid \boldsymbol{\theta})}{\partial \theta_p \partial \theta_1} & \frac{\partial^2 \ln g(\boldsymbol{X} \mid \boldsymbol{\theta})}{\partial \theta_p \partial \theta_2} & \cdots & \frac{\partial^2 \ln g(\boldsymbol{X} \mid \boldsymbol{\theta})}{\partial^2 \theta_p} \end{bmatrix}. \tag{4.9}$$

Note that the expectation here is with respect to $\boldsymbol{X} \sim f$. It turns out that if $f = g(\cdot \mid \boldsymbol{\theta})$, the two matrices are the *same*; that is,

$$\mathbf{F}(\boldsymbol{\theta}) = \mathbf{H}(\boldsymbol{\theta}), \tag{4.10}$$

provided that we may swap the order of differentiation and integration (expectation). This result is called the *information matrix equality*. We leave the proof as Exercise 1.

INFORMATION MATRIX EQUALITY

The matrices $\mathbf{F}(\boldsymbol{\theta})$ and $\mathbf{H}(\boldsymbol{\theta})$ play important roles in approximating the cross-entropy risk for large n. To set the scene, let $g^{\mathcal{G}} = g(\cdot \mid \boldsymbol{\theta}^*)$ be the minimizer of the cross-entropy risk

$$r(\boldsymbol{\theta}) := -\mathbb{E} \ln g(\boldsymbol{X} \mid \boldsymbol{\theta}).$$

We assume that r, as a function of $\boldsymbol{\theta}$, is well-behaved; in particular, that in the neighborhood of $\boldsymbol{\theta}^*$ it is strictly convex and twice continuously differentiable (this holds true, for example, if g is a Gaussian density). It follows that $\boldsymbol{\theta}^*$ is a root of $\mathbb{E}\, \boldsymbol{S}(\boldsymbol{X} \mid \boldsymbol{\theta})$, because

$$\mathbf{0} = \frac{\partial r(\boldsymbol{\theta}^*)}{\partial \boldsymbol{\theta}} = -\frac{\partial \, \mathbb{E} \ln g(\boldsymbol{X} \mid \boldsymbol{\theta}^*)}{\partial \boldsymbol{\theta}} = -\mathbb{E} \frac{\partial \ln g(\boldsymbol{X} \mid \boldsymbol{\theta}^*)}{\partial \boldsymbol{\theta}} = -\mathbb{E}\, \boldsymbol{S}(\boldsymbol{X} \mid \boldsymbol{\theta}^*),$$

again provided that the order of differentiation and integration (expectation) can be swapped. In the same way, $\mathbf{H}(\boldsymbol{\theta})$ is then the Hessian matrix of r. Let $g(\cdot \,|\, \widehat{\boldsymbol{\theta}}_n)$ be the minimizer of the training loss

$$r_{\mathcal{T}_n}(\boldsymbol{\theta}) := -\frac{1}{n}\sum_{i=1}^{n} \ln g(\boldsymbol{X}_i \,|\, \boldsymbol{\theta}),$$

where $\mathcal{T}_n = \{\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\}$ is a random training set. Let r^* be the smallest possible cross-entropy risk, taken over all functions; clearly, $r^* = -\mathbb{E} \ln f(\boldsymbol{X})$, where $\boldsymbol{X} \sim f$. Similar to the supervised learning case, we can decompose the generalization risk, $\ell(g(\cdot \,|\, \widehat{\boldsymbol{\theta}}_n)) = r(\widehat{\boldsymbol{\theta}}_n)$, into

$$r(\widehat{\boldsymbol{\theta}}_n) = r^* + \underbrace{r(\boldsymbol{\theta}^*) - r^*}_{\text{approx. error}} + \underbrace{r(\widehat{\boldsymbol{\theta}}_n) - r(\boldsymbol{\theta}^*)}_{\text{statistical error}} = r(\boldsymbol{\theta}^*) - \mathbb{E} \ln \frac{g(\boldsymbol{X} \,|\, \boldsymbol{\theta}^*)}{g(\boldsymbol{X} \,|\, \widehat{\boldsymbol{\theta}}_n)}.$$

The following theorem specifies the asymptotic behavior of the components of the generalization risk. In the proof we assume that $\widehat{\boldsymbol{\theta}}_n \xrightarrow{\mathbb{P}} \boldsymbol{\theta}^*$ as $n \to \infty$. ☞ 439

Theorem 4.1: Approximating the Cross-Entropy Risk

It holds asymptotically ($n \to \infty$) that

$$\mathbb{E} r(\widehat{\boldsymbol{\theta}}_n) - r(\boldsymbol{\theta}^*) \simeq \operatorname{tr}\left(\mathbf{F}(\boldsymbol{\theta}^*)\, \mathbf{H}^{-1}(\boldsymbol{\theta}^*)\right)/(2n), \tag{4.11}$$

where

$$r(\boldsymbol{\theta}^*) \simeq \mathbb{E} r_{\mathcal{T}_n}(\widehat{\boldsymbol{\theta}}_n) + \operatorname{tr}\left(\mathbf{F}(\boldsymbol{\theta}^*)\, \mathbf{H}^{-1}(\boldsymbol{\theta}^*)\right)/(2n). \tag{4.12}$$

Proof: A Taylor expansion of $r(\widehat{\boldsymbol{\theta}}_n)$ around $\boldsymbol{\theta}^*$ gives the statistical error ☞ 400

$$r(\widehat{\boldsymbol{\theta}}_n) - r(\boldsymbol{\theta}^*) = (\widehat{\boldsymbol{\theta}}_n - \boldsymbol{\theta}^*)^\top \underbrace{\frac{\partial r(\boldsymbol{\theta}^*)}{\partial \boldsymbol{\theta}}}_{=\mathbf{0}} + \frac{1}{2}(\widehat{\boldsymbol{\theta}}_n - \boldsymbol{\theta}^*)^\top \mathbf{H}(\bar{\boldsymbol{\theta}}_n)(\widehat{\boldsymbol{\theta}}_n - \boldsymbol{\theta}^*), \tag{4.13}$$

where $\bar{\boldsymbol{\theta}}_n$ lies on the line segment between $\boldsymbol{\theta}^*$ and $\widehat{\boldsymbol{\theta}}_n$. For large n we may replace $\mathbf{H}(\bar{\boldsymbol{\theta}}_n)$ with $\mathbf{H}(\boldsymbol{\theta}^*)$ as, by assumption, $\widehat{\boldsymbol{\theta}}_n$ converges to $\boldsymbol{\theta}^*$. The matrix $\mathbf{H}(\boldsymbol{\theta}^*)$ is positive definite because $r(\boldsymbol{\theta})$ is strictly convex at $\boldsymbol{\theta}^*$ by assumption, and therefore invertible. It is important to realize that $\widehat{\boldsymbol{\theta}}_n$ is in fact an M-estimator of $\boldsymbol{\theta}^*$. In particular, in the notation of Theorem C.19, we have $\boldsymbol{\psi} = \boldsymbol{S}$, $\mathbf{A} = \mathbf{H}(\boldsymbol{\theta}^*)$, and $\mathbf{B} = \mathbf{F}(\boldsymbol{\theta}^*)$. Consequently, by that same theorem, ☞ 449

$$\sqrt{n}\,(\widehat{\boldsymbol{\theta}}_n - \boldsymbol{\theta}^*) \xrightarrow{\mathrm{d}} \mathcal{N}\left(\mathbf{0}, \mathbf{H}^{-1}(\boldsymbol{\theta}^*)\, \mathbf{F}(\boldsymbol{\theta}^*)\, \mathbf{H}^{-\top}(\boldsymbol{\theta}^*)\right). \tag{4.14}$$

Combining (4.13) with (4.14), it follows from Theorem C.2 that asymptotically the expected estimation error is given by (4.11). ☞ 430

Next, we consider a Taylor expansion of $r_{\mathcal{T}_n}(\boldsymbol{\theta}^*)$ around $\widehat{\boldsymbol{\theta}}_n$:

$$r_{\mathcal{T}_n}(\boldsymbol{\theta}^*) = r_{\mathcal{T}_n}(\widehat{\boldsymbol{\theta}}_n) + (\boldsymbol{\theta}^* - \widehat{\boldsymbol{\theta}}_n)^\top \underbrace{\frac{\partial r_{\mathcal{T}_n}(\widehat{\boldsymbol{\theta}}_n)}{\partial \boldsymbol{\theta}}}_{=\mathbf{0}} + \frac{1}{2}(\boldsymbol{\theta}^* - \widehat{\boldsymbol{\theta}}_n)^\top \mathbf{H}_{\mathcal{T}_n}(\bar{\boldsymbol{\theta}}_n)(\boldsymbol{\theta}^* - \widehat{\boldsymbol{\theta}}_n), \tag{4.15}$$

where $\mathbf{H}_{\mathcal{T}_n}(\overline{\boldsymbol{\theta}}_n) := -\frac{1}{n}\sum_{i=1}^{n}\frac{\partial S(\boldsymbol{X}_i\,|\,\overline{\boldsymbol{\theta}}_n)}{\partial \boldsymbol{\theta}}$ is the Hessian of $r_{\mathcal{T}_n}(\boldsymbol{\theta})$ at some $\overline{\boldsymbol{\theta}}_n$ between $\widehat{\boldsymbol{\theta}}_n$ and $\boldsymbol{\theta}^*$. Taking expectations on both sides of (4.15), we obtain

$$r(\boldsymbol{\theta}^*) = \mathbb{E} r_{\mathcal{T}_n}(\widehat{\boldsymbol{\theta}}_n) + \frac{1}{2}\mathbb{E}\,(\boldsymbol{\theta}^* - \widehat{\boldsymbol{\theta}}_n)^\top \mathbf{H}_{\mathcal{T}_n}(\overline{\boldsymbol{\theta}}_n)(\boldsymbol{\theta}^* - \widehat{\boldsymbol{\theta}}_n).$$

Replacing $\mathbf{H}_{\mathcal{T}_n}(\overline{\boldsymbol{\theta}}_n)$ with $\mathbf{H}(\boldsymbol{\theta}^*)$ for large n and using (4.14), we have

$$n\,\mathbb{E}\,(\boldsymbol{\theta}^* - \widehat{\boldsymbol{\theta}}_n)^\top \mathbf{H}_{\mathcal{T}_n}(\overline{\boldsymbol{\theta}}_n)(\boldsymbol{\theta}^* - \widehat{\boldsymbol{\theta}}_n) \longrightarrow \operatorname{tr}\left(\mathbf{F}(\boldsymbol{\theta}^*)\,\mathbf{H}^{-1}(\boldsymbol{\theta}^*)\right), \quad n \to \infty.$$

Therefore, asymptotically as $n \to \infty$, we have (4.12). □

Theorem 4.1 has a number of interesting consequences:

☞ 35

1. Similar to Section 2.5.1, the training loss $\ell_{\mathcal{T}_n}(g_{\mathcal{T}_n}) = r_{\mathcal{T}_n}(\widehat{\boldsymbol{\theta}}_n)$ tends to underestimate the risk $\ell(g^{\mathcal{G}}) = r(\boldsymbol{\theta}^*)$, because the training set $\mathcal{T}_n$ is used to both train $g \in \mathcal{G}$ (that is, estimate $\boldsymbol{\theta}^*$) and to estimate the risk. The relation (4.12) tells us that on average the training loss underestimates the true risk by $\operatorname{tr}(\mathbf{F}(\boldsymbol{\theta}^*)\,\mathbf{H}^{-1}(\boldsymbol{\theta}^*))/(2n)$.

2. Adding equations (4.11) and (4.12), yields the following asymptotic approximation to the expected generalization risk:

$$\mathbb{E}\, r(\widehat{\boldsymbol{\theta}}_n) \simeq \mathbb{E}\, r_{\mathcal{T}_n}(\widehat{\boldsymbol{\theta}}_n) + \frac{1}{n}\operatorname{tr}\left(\mathbf{F}(\boldsymbol{\theta}^*)\,\mathbf{H}^{-1}(\boldsymbol{\theta}^*)\right) \tag{4.16}$$

The first term on the right-hand side of (4.16) can be estimated (without bias) via the training loss $r_{\mathcal{T}_n}(\widehat{\boldsymbol{\theta}}_n)$. As for the second term, we have already mentioned that when the true model $f \in \mathcal{G}$, then $\mathbf{F}(\boldsymbol{\theta}^*) = \mathbf{H}(\boldsymbol{\theta}^*)$. Therefore, when $\mathcal{G}$ is deemed to be a sufficiently rich class of models parameterized by a p-dimensional vector $\boldsymbol{\theta}$, we may approximate the second term as $\operatorname{tr}(\mathbf{F}(\boldsymbol{\theta}^*)\mathbf{H}^{-1}(\boldsymbol{\theta}^*))/n \approx \operatorname{tr}(\mathbf{I}_p)/n = p/n$. This suggests the following heuristic approximation to the (expected) generalization risk:

$$\mathbb{E}\, r(\widehat{\boldsymbol{\theta}}_n) \approx r_{\mathcal{T}_n}(\widehat{\boldsymbol{\theta}}_n) + \frac{p}{n}. \tag{4.17}$$

3. Multiplying both sides of (4.16) by $2n$ and substituting $\operatorname{tr}\left(\mathbf{F}(\boldsymbol{\theta}^*)\mathbf{H}^{-1}(\boldsymbol{\theta}^*)\right) \approx p$, we obtain the approximation:

$$2n\, r(\widehat{\boldsymbol{\theta}}_n) \approx -2\sum_{i=1}^{n} \ln g(\boldsymbol{X}_i\,|\,\widehat{\boldsymbol{\theta}}_n) + 2p. \tag{4.18}$$

AKAIKE INFORMATION CRITERION

The right-hand side of (4.18) is called the *Akaike information criterion* (AIC). Just like (4.17), the AIC approximation can be used to compare the difference in generalization risk of two or more learners. We prefer the learner with the smallest (estimated) generalization risk.

Suppose that, for a training set $\mathcal{T}$, the training loss $r_{\mathcal{T}}(\boldsymbol{\theta})$ has a unique minimum point $\widehat{\boldsymbol{\theta}}$ which lies in the interior of Θ. If $r_{\mathcal{T}}(\boldsymbol{\theta})$ is a differentiable function with respect to $\boldsymbol{\theta}$, then we can find the optimal parameter $\widehat{\boldsymbol{\theta}}$ by solving

$$\frac{\partial r_{\mathcal{T}}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \underbrace{\frac{1}{n}\sum_{i=1}^{n} \boldsymbol{S}(\boldsymbol{X}_i\,|\,\boldsymbol{\theta})}_{\boldsymbol{S}_{\mathcal{T}}(\boldsymbol{\theta})} = \mathbf{0}.$$

In other words, the maximum likelihood estimate $\widehat{\boldsymbol{\theta}}$ for $\boldsymbol{\theta}$ is obtained by solving the root of the average score function, that is, by solving

$$\boldsymbol{S}_{\mathcal{T}}(\boldsymbol{\theta}) = \mathbf{0}. \tag{4.19}$$

It is often not possible to find $\widehat{\boldsymbol{\theta}}$ in an explicit form. In that case one needs to solve the equation (4.19) numerically. There exist many standard techniques for root-finding, e.g., via *Newton's method* (see Section B.3.1), whereby, starting from an initial guess $\boldsymbol{\theta}_0$, subsequent iterates are obtained via the iterative scheme

NEWTON'S METHOD

☞ 409

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{H}_{\mathcal{T}}^{-1}(\boldsymbol{\theta}_t)\, \boldsymbol{S}_{\mathcal{T}}(\boldsymbol{\theta}_t),$$

where

$$\mathbf{H}_{\mathcal{T}}(\boldsymbol{\theta}) := \frac{-\partial \boldsymbol{S}_{\mathcal{T}}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \frac{1}{n}\sum_{i=1}^{n} -\frac{\partial \boldsymbol{S}(\boldsymbol{X}_i \,|\, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$$

is the average Hessian matrix of $\{-\ln g(\boldsymbol{X}_i \,|\, \boldsymbol{\theta})\}_{i=1}^n$. Under $f = g(\cdot \,|\, \boldsymbol{\theta})$, the expectation of $\mathbf{H}_{\mathcal{T}}(\boldsymbol{\theta})$ is equal to the information matrix $\mathbf{F}(\boldsymbol{\theta})$, which does not depend on the data. This suggests an alternative iterative scheme, called *Fisher's scoring method*:

FISHER'S SCORING METHOD

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{F}^{-1}(\boldsymbol{\theta}_t)\, \boldsymbol{S}_{\mathcal{T}}(\boldsymbol{\theta}_t), \tag{4.20}$$

which is not only easier to implement (if the information matrix can be readily evaluated), but also is more numerically stable.

■ **Example 4.1 (Maximum Likelihood for the Gamma Distribution)** We wish to approximate the density of the Gamma(α^*, λ^*) distribution for some true but unknown parameters α^* and λ^*, on the basis of a training set $\tau = \{x_1, \ldots, x_n\}$ of iid samples from this distribution. Choosing our approximating function $g(\cdot \,|\, \alpha, \lambda)$ in the same class of gamma densities,

$$g(x \,|\, \alpha, \lambda) = \frac{\lambda^\alpha x^{\alpha-1} \mathrm{e}^{-\lambda x}}{\Gamma(\alpha)}, \quad x \geqslant 0, \tag{4.21}$$

with $\alpha > 0$ and $\lambda > 0$, we seek to solve (4.19). Taking the logarithm in (4.21), the log-likelihood function is given by

$$l(x \,|\, \alpha, \lambda) := \alpha \ln \lambda - \ln \Gamma(\alpha) + (\alpha - 1) \ln x - \lambda x.$$

It follows that

$$\boldsymbol{S}(\alpha, \lambda) = \begin{bmatrix} \frac{\partial}{\partial \alpha} l(x \,|\, \alpha, \lambda) \\ \frac{\partial}{\partial \lambda} l(x \,|\, \alpha, \lambda) \end{bmatrix} = \begin{bmatrix} \ln \lambda - \psi(\alpha) + \ln x \\ \frac{\alpha}{\lambda} - x \end{bmatrix},$$

where ψ is the derivative of $\ln \Gamma$: the so-called *digamma function*. Hence,

DIGAMMA FUNCTION

$$\mathbf{H}(\alpha, \lambda) = -\mathbb{E}\begin{bmatrix} \frac{\partial^2}{\partial \alpha^2} l(X \,|\, \alpha, \lambda) & \frac{\partial^2}{\partial \alpha \partial \lambda} l(X \,|\, \alpha, \lambda) \\ \frac{\partial^2}{\partial \alpha \partial \lambda} l(X \,|\, \alpha, \lambda) & \frac{\partial^2}{\partial \lambda^2} l(X \,|\, \alpha, \lambda) \end{bmatrix} = -\mathbb{E}\begin{bmatrix} -\psi'(\alpha) & \frac{1}{\lambda} \\ \frac{1}{\lambda} & -\frac{\alpha}{\lambda^2} \end{bmatrix} = \begin{bmatrix} \psi'(\alpha) & -\frac{1}{\lambda} \\ -\frac{1}{\lambda} & \frac{\alpha}{\lambda^2} \end{bmatrix}.$$

Fisher's scoring method (4.20) can now be used to solve (4.19), with

$$\boldsymbol{S}_\tau(\alpha, \lambda) = \begin{bmatrix} \ln \lambda - \psi(\alpha) + n^{-1} \sum_{i=1}^n \ln x_i \\ \frac{\alpha}{\lambda} - n^{-1} \sum_{i=1}^n x_i \end{bmatrix}$$

and $\mathbf{F}(\alpha, \lambda) = \mathbf{H}(\alpha, \lambda)$. ■

4.3 Expectation–Maximization (EM) Algorithm

The *Expectation–Maximization* algorithm (EM) is a general algorithm for maximization of complicated (log-)likelihood functions, through the introduction of auxiliary variables.

To simplify the notation in this section, we use a Bayesian notation system, where the same symbol is used for different (conditional) probability densities.

As in the previous section, given independent observations $\tau = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ from some unknown pdf f, the objective is to find the best approximation to f in a function class $\mathcal{G} = \{g(\cdot \,|\, \boldsymbol{\theta}), \boldsymbol{\theta} \in \Theta\}$ by solving the maximum likelihood problem:

$$\boldsymbol{\theta}^* = \operatorname*{argmax}_{\boldsymbol{\theta} \in \Theta} g(\tau \,|\, \boldsymbol{\theta}), \tag{4.22}$$

where $g(\tau \,|\, \boldsymbol{\theta}) := g(\boldsymbol{x}_1 \,|\, \boldsymbol{\theta}) \cdots g(\boldsymbol{x}_n \,|\, \boldsymbol{\theta})$. The key element of the EM algorithm is the augmentation of the data τ with a suitable vector of *latent variables*, $\boldsymbol{z}$, such that

LATENT VARIABLES

$$g(\tau \,|\, \boldsymbol{\theta}) = \int g(\tau, \boldsymbol{z} \,|\, \boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{z}.$$

COMPLETE-DATA LIKELIHOOD

The function $\boldsymbol{\theta} \mapsto g(\tau, \boldsymbol{z} \,|\, \boldsymbol{\theta})$ is usually referred to as the *complete-data likelihood* function. The choice of the latent variables is guided by the desire to make the maximization of $g(\tau, \boldsymbol{z} \,|\, \boldsymbol{\theta})$ much easier than that of $g(\tau \,|\, \boldsymbol{\theta})$.

Suppose p denotes an arbitrary density of the latent variables $\boldsymbol{z}$. Then, we can write:

$$\begin{aligned}
\ln g(\tau \,|\, \boldsymbol{\theta}) &= \int p(\boldsymbol{z}) \ln g(\tau \,|\, \boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{z} \\
&= \int p(\boldsymbol{z}) \ln \left(\frac{g(\tau, \boldsymbol{z} \,|\, \boldsymbol{\theta}) / p(\boldsymbol{z})}{g(\boldsymbol{z} \,|\, \tau, \boldsymbol{\theta}) / p(\boldsymbol{z})} \right) \mathrm{d}\boldsymbol{z} \\
&= \int p(\boldsymbol{z}) \ln \left(\frac{g(\tau, \boldsymbol{z} \,|\, \boldsymbol{\theta})}{p(\boldsymbol{z})} \right) \mathrm{d}\boldsymbol{z} - \int p(\boldsymbol{z}) \ln \left(\frac{g(\boldsymbol{z} \,|\, \tau, \boldsymbol{\theta})}{p(\boldsymbol{z})} \right) \mathrm{d}\boldsymbol{z} \\
&= \int p(\boldsymbol{z}) \ln \left(\frac{g(\tau, \boldsymbol{z} \,|\, \boldsymbol{\theta})}{p(\boldsymbol{z})} \right) \mathrm{d}\boldsymbol{z} + \mathcal{D}(p, g(\cdot \,|\, \tau, \boldsymbol{\theta})),
\end{aligned} \tag{4.23}$$

☞ 42

where $\mathcal{D}(p, g(\cdot \,|\, \tau, \boldsymbol{\theta}))$ is the Kullback–Leibler divergence from the density p to $g(\cdot \,|\, \tau, \boldsymbol{\theta})$. Since $\mathcal{D} \geqslant 0$, it follows that

$$\ln g(\tau \,|\, \boldsymbol{\theta}) \geqslant \int p(\boldsymbol{z}) \ln \left(\frac{g(\tau, \boldsymbol{z} \,|\, \boldsymbol{\theta})}{p(\boldsymbol{z})} \right) \mathrm{d}\boldsymbol{z} =: \mathcal{L}(p, \boldsymbol{\theta})$$

for all $\boldsymbol{\theta}$ and any density p of the latent variables. In other words, $\mathcal{L}(p, \boldsymbol{\theta})$ is a lower bound on the log-likelihood that involves the complete-data likelihood. The EM algorithm then aims to increase this lower bound as much as possible by starting with an initial guess $\boldsymbol{\theta}^{(0)}$ and then, for $t = 1, 2, \ldots$, solving the following two steps:

1. $p^{(t)} = \operatorname{argmax}_p \mathcal{L}(p, \boldsymbol{\theta}^{(t-1)})$,
2. $\boldsymbol{\theta}^{(t)} = \operatorname{argmax}_{\boldsymbol{\theta} \in \Theta} \mathcal{L}(p^{(t)}, \boldsymbol{\theta})$.

The first optimization problem can be solved explicitly. Namely, by (4.23), we have that

$$p^{(t)} = \operatorname*{argmin}_{p} \mathcal{D}(p, g(\cdot \mid \tau, \boldsymbol{\theta}^{(t-1)})) = g(\cdot \mid \tau, \boldsymbol{\theta}^{(t-1)}).$$

That is, the optimal density is the conditional density of the latent variables given the data τ and the parameter $\boldsymbol{\theta}^{(t-1)}$. The second optimization problem can be simplified by writing $\mathcal{L}(p^{(t)}, \boldsymbol{\theta}) = Q^{(t)}(\boldsymbol{\theta}) - \mathbb{E}_{p^{(t)}} \ln p^{(t)}(\boldsymbol{Z})$, where

$$Q^{(t)}(\boldsymbol{\theta}) := \mathbb{E}_{p^{(t)}} \ln g(\tau, \boldsymbol{Z} \mid \boldsymbol{\theta})$$

is the expected complete-data log-likelihood under $\boldsymbol{Z} \sim p^{(t)}$. Consequently, the maximization of $\mathcal{L}(p^{(t)}, \boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$ is equivalent to finding

$$\boldsymbol{\theta}^{(t)} = \operatorname*{argmax}_{\boldsymbol{\theta} \in \Theta} Q^{(t)}(\boldsymbol{\theta}).$$

This leads to the following generic EM algorithm.

Algorithm 4.3.1: Generic EM Algorithm

input: Data τ, initial guess $\boldsymbol{\theta}^{(0)}$.
output: Approximation of the maximum likelihood estimate.

1 $t \leftarrow 1$
2 **while** a stopping criterion is not met **do**
3 **Expectation Step**: Find $p^{(t)}(z) := g(z \mid \tau, \boldsymbol{\theta}^{(t-1)})$ and compute the expectation

$$Q^{(t)}(\boldsymbol{\theta}) := \mathbb{E}_{p^{(t)}} \ln g(\tau, \boldsymbol{Z} \mid \boldsymbol{\theta}). \tag{4.24}$$

4 **Maximization Step**: Let $\boldsymbol{\theta}^{(t)} \leftarrow \operatorname{argmax}_{\boldsymbol{\theta} \in \Theta} Q^{(t)}(\boldsymbol{\theta})$.
5 $t \leftarrow t + 1$
6 **return** $\boldsymbol{\theta}^{(t)}$

A possible stopping criterion is to stop when

$$\left| \frac{\ln g(\tau \mid \boldsymbol{\theta}^{(t)}) - \ln g(\tau \mid \boldsymbol{\theta}^{(t-1)})}{\ln g(\tau \mid \boldsymbol{\theta}^{(t)})} \right| \leqslant \varepsilon$$

for some small tolerance $\varepsilon > 0$.

■ **Remark 4.1 (Properties of the EM Algorithm)** The identity (4.23) can be used to show that the likelihood $g(\tau \mid \boldsymbol{\theta}^{(t)})$ does not decrease with every iteration of the algorithm. This property is one of the strengths of the algorithm. For example, it can be used to debug computer implementations of the EM algorithm: if the likelihood is observed to decrease at any iteration, then one has detected a bug in the program.

The convergence of the sequence $\{\boldsymbol{\theta}^{(t)}\}$ to a global maximum (if it exists) is highly dependent on the initial value $\boldsymbol{\theta}^{(0)}$ and, in many cases, an appropriate choice of $\boldsymbol{\theta}^{(0)}$ may not be clear. Typically, practitioners run the algorithm from different random starting points over Θ, to ascertain empirically that a suitable optimum is achieved. ■

■ **Example 4.2 (Censored Data)** Suppose the lifetime (in years) of a certain type of machine is modeled via a $\mathcal{N}(\mu, \sigma^2)$ distribution. To estimate μ and σ^2, the lifetimes of n (independent) machines are recorded up to c years. Denote these *censored* lifetimes by $x_1, \ldots, x_n$. The $\{x_i\}$ are thus realizations of iid random variables $\{X_i\}$, distributed as $\min\{Y, c\}$, where $Y \sim \mathcal{N}(\mu, \sigma^2)$.

☞ 428

By the law of total probability (see (C.9)), the marginal pdf of each X can be written as:

$$g(x \,|\, \mu, \sigma^2) = \underbrace{\Phi((c-\mu)/\sigma)}_{\mathbb{P}[Y<c]} \frac{\varphi_{\sigma^2}(x-\mu)}{\Phi((c-\mu)/\sigma)} \mathbb{1}\{x < c\} + \underbrace{\overline{\Phi}((c-\mu)/\sigma)}_{\mathbb{P}[Y \geqslant c]} \mathbb{1}\{x = c\},$$

where $\varphi_{\sigma^2}(\cdot)$ is the pdf of the $\mathcal{N}(0, \sigma^2)$ distribution, Φ is the cdf of the standard normal distribution, and $\overline{\Phi} := 1 - \Phi$. It follows that the likelihood of the data $\tau = \{x_1, \ldots, x_n\}$ as a function of the parameter $\boldsymbol{\theta} := [\mu, \sigma^2]^\top$ is:

$$g(\tau \,|\, \boldsymbol{\theta}) = \prod_{i:x_i<c} \frac{\exp\left(-\frac{(x_i-\mu)^2}{2\sigma^2}\right)}{\sqrt{2\pi\sigma^2}} \times \prod_{i:x_i=c} \overline{\Phi}((c-\mu)/\sigma).$$

Let n_c be the total number of x_i such that $x_i = c$. Using n_c latent variables $\boldsymbol{z} = [z_1, \ldots, z_{n_c}]^\top$, we can write the joint pdf:

$$g(\tau, \boldsymbol{z} \,|\, \boldsymbol{\theta}) = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left(-\frac{\sum_{i:x_i<c}(x_i-\mu)^2}{2\sigma^2} - \frac{\sum_{i=1}^{n_c}(z_i-\mu)^2}{2\sigma^2}\right) \mathbb{1}\left\{\min_i z_i \geqslant c\right\},$$

so that $\int g(\tau, \boldsymbol{z} \,|\, \boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{z} = g(\tau \,|\, \boldsymbol{\theta})$. We can thus apply the EM algorithm to maximize the likelihood, as follows.

For the E(xpectation)-step, we have for a fixed $\boldsymbol{\theta}$:

$$g(\boldsymbol{z} \,|\, \tau, \boldsymbol{\theta}) = \prod_{i=1}^{n_c} g(z_i \,|\, \tau, \boldsymbol{\theta}),$$

where $g(z \,|\, \tau, \boldsymbol{\theta}) = \mathbb{1}\{z \geqslant c\} \varphi_{\sigma^2}(z-\mu)/\overline{\Phi}((c-\mu)/\sigma)$ is simply the pdf of the $\mathcal{N}(\mu, \sigma^2)$ distribution, truncated to $[c, \infty)$.

For the M(aximization)-step, we compute the expectation of the complete log-likelihood with respect to a fixed $g(\boldsymbol{z} \,|\, \tau, \boldsymbol{\theta})$ and use the fact that $Z_1, \ldots, Z_{n_c}$ are iid:

$$\mathbb{E} \ln g(\tau, \boldsymbol{Z} \,|\, \boldsymbol{\theta}) = -\frac{\sum_{i:x_i<c}(x_i-\mu)^2}{2\sigma^2} - \frac{n_c \mathbb{E}(Z-\mu)^2}{2\sigma^2} - \frac{n}{2} \ln \sigma^2 - \frac{n}{2} \ln(2\pi),$$

where Z has a $\mathcal{N}(\mu, \sigma^2)$ distribution, truncated to $[c, \infty)$. To maximize the last expression with respect to μ we set the derivative with respect to μ to zero, and obtain:

$$\mu = \frac{n_c \mathbb{E}Z + \sum_{i:x_i<c} x_i}{n}.$$

Similarly, setting the derivative with respect to σ^2 to zero gives:

$$\sigma^2 = \frac{n_c \mathbb{E}(Z-\mu)^2 + \sum_{i:x_i<c}(x_i-\mu)^2}{n}.$$

In summary, the EM iterates for $t = 1, 2, \ldots$ are as follows.

E-step. Given the current estimate $\boldsymbol{\theta}_t := [\mu_t, \sigma_t^2]^\top$, compute the expectations $\nu_t := \mathbb{E}Z$ and $\zeta_t^2 := \mathbb{E}(Z - \mu_t)^2$, where $Z \sim \mathcal{N}(\mu_t, \sigma_t^2)$, conditional on $Z \geqslant c$; that is,

$$\nu_t := \mu_t + \sigma_t^2 \frac{\varphi_{\sigma_t^2}(c - \mu_t)}{\overline{\Phi}((c - \mu_t)/\sigma_t)}$$
$$\zeta_t^2 := \sigma_t^2 \left(1 + (c - \mu_t) \frac{\varphi_{\sigma_t^2}(c - \mu_t)}{\overline{\Phi}((c - \mu_t)/\sigma_t)}\right).$$

M-step. Update the estimate to $\boldsymbol{\theta}_{t+1} := [\mu_{t+1}, \sigma_{t+1}^2]^\top$ via the formulas:

$$\mu_{t+1} = \frac{n_c \nu_t + \sum_{i:x_i<c} x_i}{n}$$
$$\sigma_{t+1}^2 = \frac{n_c \zeta_t^2 + \sum_{i:x_i<c} (x_i - \mu_{t+1})^2}{n}.$$

■

4.4 Empirical Distribution and Density Estimation

In Section 1.5.2.3 we saw how the empirical cdf $\widehat{F}_n$, obtained from an iid training set $\tau = \{x_1, \ldots, x_n\}$ from an unknown distribution on $\mathbb{R}$, gives an estimate of the unknown cdf F of this sampling distribution. The function $\widehat{F}_n$ is a genuine cdf, as it is right-continuous, increasing, and lies between 0 and 1. The corresponding discrete probability distribution is called the *empirical distribution* of the data. A random variable X distributed according to this empirical distribution takes the values $x_1, \ldots, x_n$ with equal probability $1/n$. The concept of empirical distribution naturally generalizes to higher dimensions: a random vector $\boldsymbol{X}$ that is distributed according to the empirical distribution of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ has discrete pdf $\mathbb{P}[\boldsymbol{X} = \boldsymbol{x}_i] = 1/n, i = 1, \ldots, n$. Sampling from such a distribution — in other words *resampling* the original data — was discussed in Section 3.2.4. The preeminent usage of such sampling is the bootstrap method, discussed in Section 3.3.2.

☞ 11

EMPIRICAL DISTRIBUTION

☞ 76
☞ 88

In a way, the empirical distribution is the natural answer to the unsupervised learning question: what is the underlying probability distribution of the data? However, the empirical distribution is, by definition, a discrete distribution, whereas the true sampling distribution might be continuous. For continuous data it makes sense to also consider estimation of the pdf of the data. A common approach is to estimate the density via a *kernel density estimate* (KDE), the most prevalent learner to carry this out is given next.

Definition 4.1: Gaussian KDE

Let $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$ be the outcomes of an iid sample from a continuous pdf f. A *Gaussian kernel density estimate* of f is a mixture of normal pdfs, of the form

$$g_{\tau_n}(\boldsymbol{x} \,|\, \sigma) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{(2\pi)^{d/2} \sigma^d} \, \mathrm{e}^{-\frac{\|\boldsymbol{x} - \boldsymbol{x}_i\|^2}{2\sigma^2}}, \quad \boldsymbol{x} \in \mathbb{R}^d, \tag{4.25}$$

where $\sigma > 0$ is called the *bandwidth*.

GAUSSIAN KERNEL DENSITY ESTIMATE

We see that g_{τ_n} in (4.25) is the average of a collection of n normal pdfs, where each normal distribution is centered at the data point $\boldsymbol{x}_i$ and has covariance matrix $\sigma^2\mathbf{I}_d$. A major question is how to choose the bandwidth σ so as to best approximate the unknown pdf f. Choosing very small σ will result in a "spiky" estimate, whereas a large σ will produce an over-smoothed estimate that may not identify important peaks that are present in the unknown pdf. Figure 4.1 illustrates this phenomenon. In this case the data are comprised of 20 points uniformly drawn from the unit square. The true pdf is thus 1 on $[0,1]^2$ and 0 elsewhere.

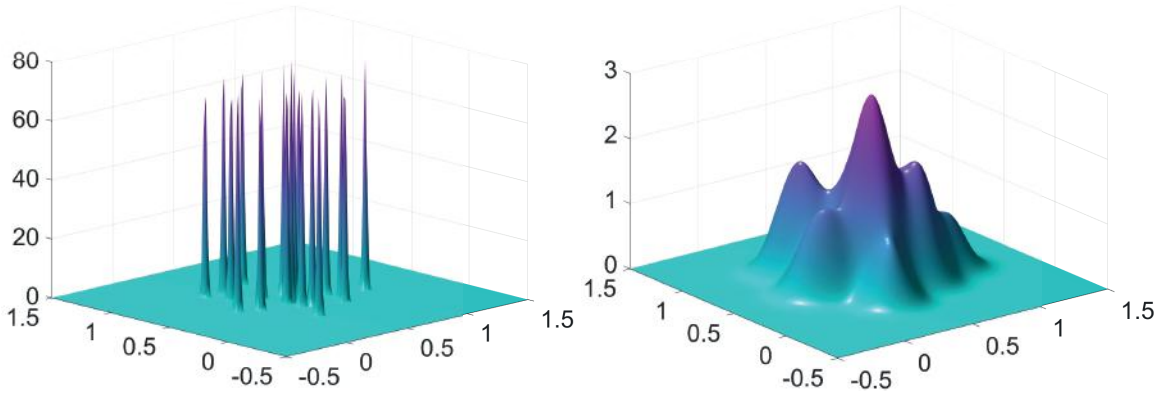


Figure 4.1: Two two-dimensional Gaussian KDEs, with $\sigma = 0.01$ (left) and $\sigma = 0.1$ (right).

Let us write the Gaussian KDE in (4.25) as

$$g_{\tau_n}(\boldsymbol{x} \,|\, \sigma) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{\sigma^d}\phi\left(\frac{\boldsymbol{x}-\boldsymbol{x}_i}{\sigma}\right), \tag{4.26}$$

where

$$\phi(\boldsymbol{z}) = \frac{1}{(2\pi)^{d/2}}\, \mathrm{e}^{-\frac{\|\boldsymbol{z}\|^2}{2}}, \quad \boldsymbol{z} \in \mathbb{R}^d \tag{4.27}$$

is the pdf of the d-dimensional standard normal distribution. By choosing a different probability density ϕ in (4.26), satisfying $\phi(\boldsymbol{x}) = \phi(-\boldsymbol{x})$ for all $\boldsymbol{x}$, we can obtain a wide variety of kernel density estimates. A simple pdf ϕ is, for example, the uniform pdf on $[-1,1]^d$:

$$\phi(\boldsymbol{z}) = \begin{cases} 2^{-d}, & \text{if } \boldsymbol{z} \in [-1,1]^d, \\ 0, & \text{otherwise.} \end{cases}$$

Figure 4.2 shows the graph of the corresponding KDE, using the same data as in Figure 4.1 and with bandwidth $\sigma = 0.1$. We observe qualitatively similar behavior for the Gaussian and uniform KDEs. As a rule, the choice of the function ϕ is less important than the choice of the bandwidth in determining the quality of the estimate.

The important issue of bandwidth selection has been extensively studied for one-dimensional data. To explain the ideas, we use our usual setup and let $\tau = \{x_1, \ldots, x_n\}$ be the observed (one-dimensional) data from the unknown pdf f. First, we define the loss function as

$$\mathrm{Loss}(f(x), g(x)) = \frac{(f(x) - g(x))^2}{f(x)}. \tag{4.28}$$

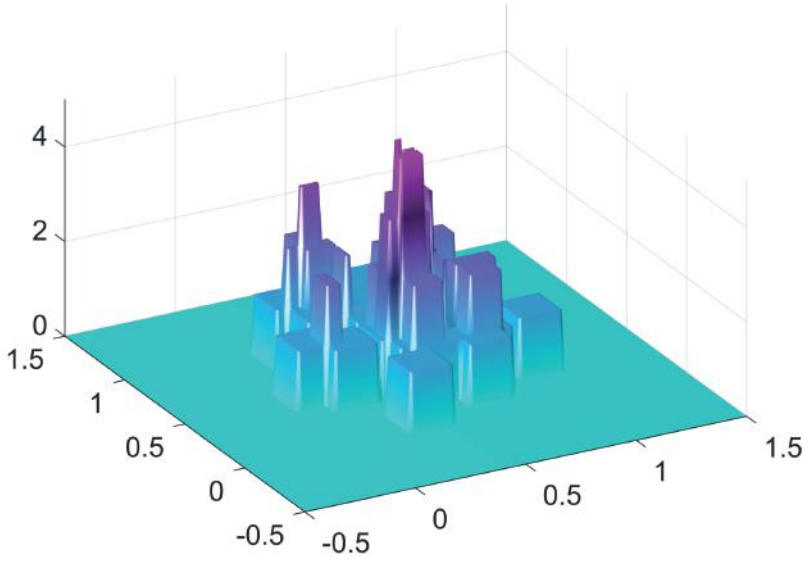


Figure 4.2: A two-dimensional uniform KDE, with bandwidth $\sigma = 0.1$.

The risk to minimize is thus $\ell(g) := \mathbb{E}_f \text{Loss}(f(X), g(X)) = \int (f(x) - g(x))^2 \, \mathrm{d}x$. We bypass the selection of a class of approximation functions by choosing the learner to be specified by (4.25) for a fixed σ. The objective is now to find a σ that minimizes the generalization risk $\ell(g_\tau(\cdot \,|\, \sigma))$ or the expected generalization risk $\mathbb{E}\ell(g_{\mathcal{T}}(\cdot \,|\, \sigma))$. The generalization risk is in this case

$$\int (f(x) - g_\tau(x \,|\, \sigma))^2 \, \mathrm{d}x = \int f^2(x) \, \mathrm{d}x - 2 \int f(x) g_\tau(x \,|\, \sigma) \, \mathrm{d}x + \int g_\tau^2(x \,|\, \sigma) \, \mathrm{d}x.$$

Minimizing this expression with respect to σ is equivalent to minimizing the last two terms, which can be written as

$$-2 \, \mathbb{E}_f g_\tau(X \,|\, \sigma) + \int \left(\frac{1}{n} \sum_{i=1}^{n} \frac{1}{\sigma} \phi\left(\frac{x - x_i}{\sigma} \right) \right)^2 \mathrm{d}x.$$

This expression in turn can be estimated by using a test sample $\{x'_1 \ldots, x'_{n'}\}$ from f, yielding the following minimization problem:

$$\min_\sigma -\frac{2}{n'} \sum_{i=1}^{n'} g_\tau(x'_i \,|\, \sigma) + \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \int \frac{1}{\sigma^2} \phi\left(\frac{x - x_i}{\sigma} \right) \phi\left(\frac{x - x_j}{\sigma} \right) \mathrm{d}x,$$

where $\int \frac{1}{\sigma^2} \phi\left(\frac{x-x_i}{\sigma}\right) \phi\left(\frac{x-x_j}{\sigma}\right) \mathrm{d}x = \frac{1}{\sqrt{2}\sigma} \phi\left(\frac{x_i - x_j}{\sqrt{2}\sigma}\right)$ in the case of the Gaussian kernel (4.27) with $d = 1$. To estimate σ in this way clearly requires a test sample, or at least an application of *cross-validation*. Another approach is to minimize the *expected* generalization risk, (that is, averaged over all training sets):

☞ 37

$$\mathbb{E} \int (f(x) - g_{\mathcal{T}}(x \,|\, \sigma))^2 \, \mathrm{d}x.$$

This is called the *mean integrated squared error* (MISE). It can be decomposed into an integrated squared bias and integrated variance component:

MEAN INTEGRATED SQUARED ERROR

$$\int (f(x) - \mathbb{E} g_{\mathcal{T}}(x \,|\, \sigma))^2 \, \mathrm{d}x + \int \mathbb{V}\text{ar}(g_{\mathcal{T}}(x \,|\, \sigma)) \, \mathrm{d}x.$$

A typical analysis now proceeds by investigating how the MISE behaves for large n, under various assumptions on f. For example, it is shown in [114] that, for $\sigma \to 0$ and $n\sigma \to \infty$, the asymptotic approximation to the MISE of the Gaussian kernel density estimator (4.25) (for $d = 1$) is given by

$$\frac{1}{4}\sigma^4 \|f''\|^2 + \frac{1}{2n\sqrt{\pi\sigma^2}}, \tag{4.29}$$

where $\|f''\|^2 := \int (f''(x))^2 \, \mathrm{d}x$. The asymptotically optimal value of σ is the minimizer

$$\sigma^* := \left(\frac{1}{2n\sqrt{\pi}\,\|f''\|^2} \right)^{1/5}. \tag{4.30}$$

GAUSSIAN RULE OF THUMB

To compute the optimal σ^* in (4.30), one needs to estimate the functional $\|f''\|^2$. The *Gaussian rule of thumb* is to assume that f is the density of the $\mathcal{N}(\bar{x}, s^2)$ distribution, where $\bar{x}$ and s^2 are the sample mean and variance of the data, respectively [113]. In this case $\|f''\|^2 = s^{-5}\pi^{-1/2}3/8$ and the Gaussian rule of thumb becomes:

$$\sigma_{\text{rot}} = \left(\frac{4\, s^5}{3\, n} \right)^{1/5} \approx 1.06\, s\, n^{-1/5}.$$

THETA KDE

We recommend, however, the fast and reliable *theta KDE* of [14], which chooses the bandwidth in an optimal way via a fixed-point procedure. Figures 4.1 and 4.2 illustrate a common problem with traditional KDEs: for distributions on a bounded domain, such as the uniform distribution on $[0, 1]^2$, the KDE assigns positive probability mass *outside* this domain. An additional advantage of the theta KDE is that it largely avoids this boundary effect. We illustrate the theta KDE with the following example.

■ **Example 4.3 (Comparison of Gaussian and theta KDEs)** The following Python program draws an iid sample from the Exp(1) distribution and constructs a Gaussian kernel density estimate. We see in Figure 4.3 that with an appropriate choice of the bandwidth a good fit to the true pdf can be achieved, except at the boundary $x = 0$. The theta KDE does not exhibit this boundary effect. Moreover, it chooses the bandwidth automatically, to achieve a superior fit. The theta KDE source code is available as `kde.py` on the book's GitHub site.

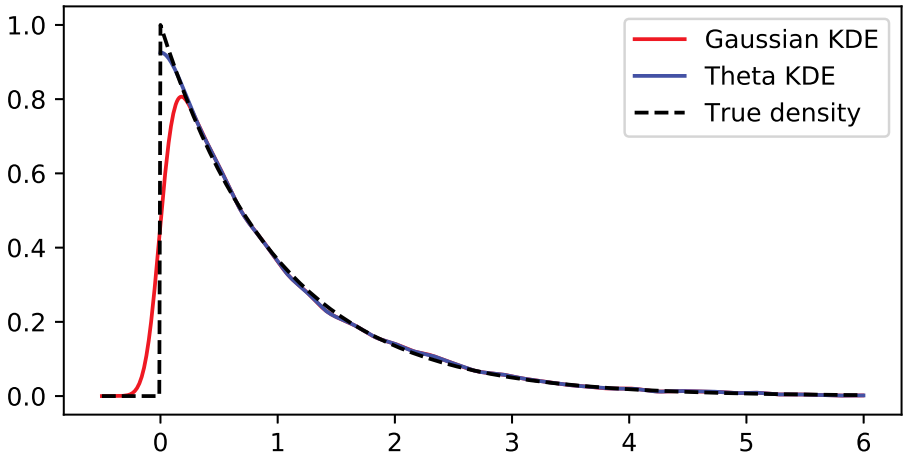


Figure 4.3: Kernel density estimates for Exp(1)-distributed data.

gausthetakde.py

```
import matplotlib.pyplot as plt
import numpy as np
from kde import *

sig = 0.1; sig2 = sig**2; c = 1/np.sqrt(2*np.pi)/sig #Constants
phi = lambda x,x0: np.exp(-(x-x0)**2/(2*sig2)) #Unscaled Kernel
f = lambda x: np.exp(-x)*(x >= 0) # True PDF
n = 10**4 # Sample Size
x = -np.log(np.random.uniform(size=n))# Generate Data via IT method
xx = np.arange(-0.5,6,0.01, dtype = "d")# Plot Range
phis = np.zeros(len(xx))
for i in range(0,n):
    phis = phis + phi(xx,x[i])
phis = c*phis/n
plt.plot(xx,phis,'r')# Plot Gaussian KDE
[bandwidth,density,xmesh,cdf] = kde(x,2**12,0,max(x))
idx = (xmesh <= 6)
plt.plot(xmesh[idx],density[idx])# Plot Theta KDE
plt.plot(xx,f(xx))# Plot True PDF
```

4.5 Clustering via Mixture Models

Clustering is concerned with the grouping of unlabeled feature vectors into clusters, such that samples within a cluster are more similar to each other than samples belonging to different clusters. Usually, it is assumed that the number of clusters is known in advance, but otherwise no prior information is given about the data. Applications of clustering can be found in the areas of communication, data compression and storage, database searching, pattern matching, and object recognition.

A common approach to clustering analysis is to assume that the data comes from a mixture of (usually Gaussian) distributions, and thus the objective is to estimate the parameters of the mixture model by maximizing the likelihood function for the data. Direct optimization of the likelihood function in this case is not a simple task, due to necessary constraints on the parameters (more about this later) and the complicated nature of the likelihood function, which in general has a great number of local maxima and saddle-points. A popular method to estimate the parameters of the mixture model is the EM algorithm, which was discussed in a more general setting in Section 4.3. In this section we explain the basics of mixture modeling and explain the workings of the EM method in this context. In addition, we show how direct optimization methods can be used to maximize the likelihood.

☞ 128

4.5.1 Mixture Models

Let $\mathcal{T} := \{\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\}$ be iid random vectors taking values in some set $\mathcal{X} \subseteq \mathbb{R}^d$, each $\boldsymbol{X}_i$ being distributed according to the *mixture density*

MIXTURE DENSITY

$$g(\boldsymbol{x} \,|\, \boldsymbol{\theta}) = w_1 \phi_1(\boldsymbol{x}) + \cdots + w_K \phi_K(\boldsymbol{x}), \quad \boldsymbol{x} \in \mathcal{X}, \tag{4.31}$$

WEIGHTS

where $\phi_1, \ldots, \phi_K$ are probability densities (discrete or continuous) on $\mathcal{X}$, and the positive *weights* $w_1, \ldots, w_K$ sum up to 1. This mixture pdf can be interpreted in the following way. Let Z be a discrete random variable taking values $1, 2, \ldots, K$ with probabilities $w_1, \ldots, w_K$, and let $\boldsymbol{X}$ be a random vector whose conditional pdf, given $Z = z$, is ϕ_z. By the product rule

☞ 431

(C.17), the joint pdf of Z and $\boldsymbol{X}$ is given by

$$\phi_{Z,\boldsymbol{X}}(z, \boldsymbol{x}) = \phi_Z(z)\, \phi_{\boldsymbol{X}|Z}(\boldsymbol{x} \,|\, z) = w_z\, \phi_z(\boldsymbol{x})$$

and the marginal pdf of $\boldsymbol{X}$ is found by summing the joint pdf over the values of z, which gives (4.31). A random vector $\boldsymbol{X} \sim g$ can thus be simulated in two steps:

1. First, draw Z according to the probabilities $\mathbb{P}[Z = z] = w_z$, $z = 1, \ldots, K$.
2. Then draw $\boldsymbol{X}$ according to the pdf ϕ_Z.

As $\mathcal{T}$ only contain the $\{\boldsymbol{X}_i\}$ variables, the $\{Z_i\}$ are viewed as *latent* variables. We can interpret Z_i as the hidden label of the cluster to which $\boldsymbol{X}_i$ belongs.

Typically, each ϕ_k in (4.31) is assumed to be known up to some parameter vector $\boldsymbol{\eta}_k$. It is customary[1] in clustering analysis to work with *Gaussian* mixtures; that is, each density ϕ_k is Gaussian with some unknown expectation vector $\boldsymbol{\mu}_k$ and covariance matrix $\boldsymbol{\Sigma}_k$. We gather all unknown parameters, including the weights $\{w_k\}$, into a parameter vector $\boldsymbol{\theta}$. As usual, $\tau = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ denotes the outcome of $\mathcal{T}$. As the components of $\mathcal{T}$ are iid, their (joint) pdf is given by

$$g(\tau \,|\, \boldsymbol{\theta}) := \prod_{i=1}^{n} g(\boldsymbol{x}_i \,|\, \boldsymbol{\theta}) = \prod_{i=1}^{n} \sum_{k=1}^{K} w_k\, \phi_k(\boldsymbol{x}_i \,|\, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k). \tag{4.32}$$

Following the same reasoning as for (4.5), we can estimate $\boldsymbol{\theta}$ from an outcome τ by maximizing the log-likelihood function

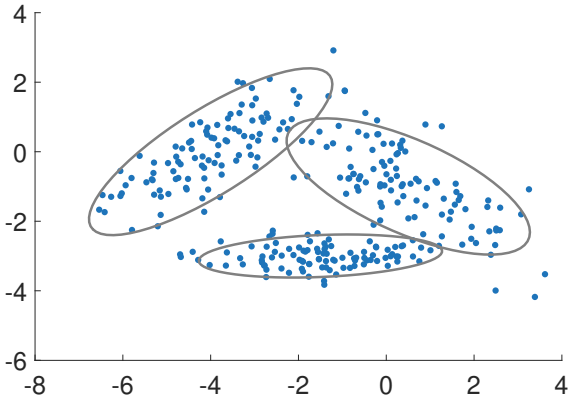
$$l(\boldsymbol{\theta} \,|\, \tau) := \sum_{i=1}^{n} \ln g(\boldsymbol{x}_i \,|\, \boldsymbol{\theta}) = \sum_{i=1}^{n} \ln \left(\sum_{k=1}^{K} w_k\, \phi_k(\boldsymbol{x}_i \,|\, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right). \tag{4.33}$$

However, finding the maximizer of $l(\boldsymbol{\theta} \,|\, \tau)$ is not easy in general, since the function is typically multiextremal.

■ **Example 4.4 (Clustering via Mixture Models)** The data depicted in Figure 4.4 consists of 300 data points that were independently generated from three bivariate normal distributions, whose parameters are given in that same figure. For each of these three distributions, exactly 100 points were generated. Ideally, we would like to cluster the data into three clusters that correspond to the three cases.

To cluster the data into three groups, a possible model for the data is to assume that the points are iid draws from an (unknown) mixture of three 2-dimensional Gaussian distributions. This is a sensible approach, although in reality the data were not simulated in this way. It is instructive to understand the difference between the two models. In the mixture model, each cluster label Z takes the value $\{1, 2, 3\}$ with equal probability, and hence, drawing the labels independently, the total number of points in each cluster would

[1] Other common mixture distributions include Student t and Beta distributions.



cluster	mean vector	covariance matrix
1	$\begin{bmatrix} -4 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 2 & 1.4 \\ 1.4 & 1.5 \end{bmatrix}$
2	$\begin{bmatrix} 0.5 \\ -1 \end{bmatrix}$	$\begin{bmatrix} 2 & -0.95 \\ -0.95 & 1 \end{bmatrix}$
3	$\begin{bmatrix} -1.5 \\ -3 \end{bmatrix}$	$\begin{bmatrix} 2 & 0.1 \\ 0.1 & 0.1 \end{bmatrix}$

Figure 4.4: Cluster the 300 data points (left) into three clusters, without making any assumptions about the probability distribution of the data. In fact, the data were generated from three bivariate normal distributions, whose parameters are listed on the right.

be $\mathsf{Bin}(300, 1/3)$ distributed. However, in the actual simulation, the number of points in each cluster is exactly 100. Nevertheless, the mixture model would be an accurate (although not exact) model for these data. Figure 4.5 displays the "target" Gaussian mixture density for the data in Figure 4.4; that is, the mixture with equal weights and with the exact parameters as specified in Figure 4.4.

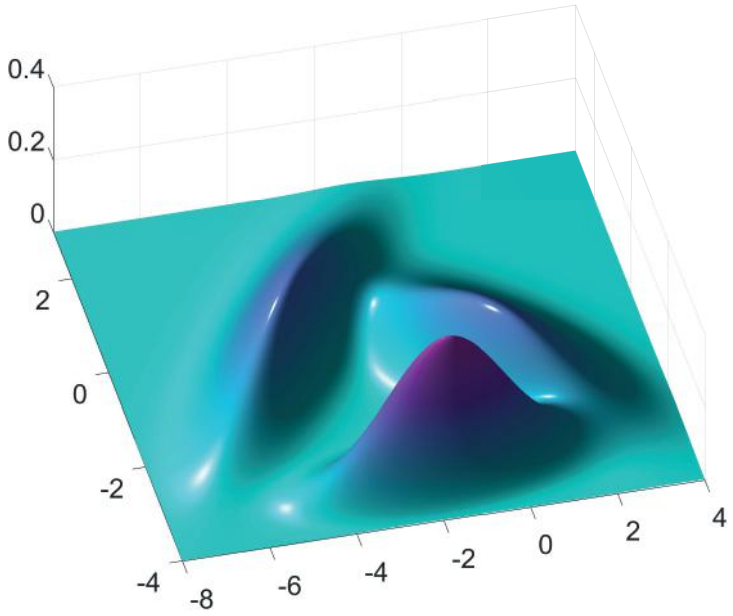


Figure 4.5: The target mixture density for the data in Figure 4.4.

In the next section we will carry out the clustering by using the EM algorithm. ■

4.5.2 EM Algorithm for Mixture Models

As we saw in Section 4.3, instead of maximizing the log-likelihood function (4.33) directly from the data $\tau = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$, the EM algorithm first *augments* the data with the vector of latent variables — in this case the hidden cluster labels $\boldsymbol{z} = \{z_1, \ldots, z_n\}$. The idea is that τ is

DATA AUGMENTATION

only the *observed* part of the complete random data $(\mathcal{T}, \boldsymbol{Z})$, which were generated via the two-step procedure described above. That is, for each data point $\boldsymbol{X}$, first draw the cluster label $Z \in \{1, \ldots, K\}$ according to probabilities $\{w_1, \ldots, w_K\}$ and then, given $Z = z$, draw $\boldsymbol{X}$ from ϕ_z. The joint pdf of $\mathcal{T}$ and $\boldsymbol{Z}$ is

$$g(\tau, z \,|\, \boldsymbol{\theta}) = \prod_{i=1}^{n} w_{z_i} \, \phi_{z_i}(\boldsymbol{x}_i),$$

which is of a much simpler form than (4.32). It follows that the *complete-data log-likelihood* function

COMPLETE-DATA LOG-LIKELIHOOD

$$\widetilde{l}(\boldsymbol{\theta} \,|\, \tau, z) = \sum_{i=1}^{n} \ln[w_{z_i} \, \phi_{z_i}(\boldsymbol{x}_i)] \tag{4.34}$$

is often easier to maximize than the original log-likelihood (4.33), for any given (τ, z). But, of course the latent variables z are not observed and so $\widetilde{l}(\boldsymbol{\theta} \,|\, \tau, z)$ cannot be evaluated. In the E-step of the EM algorithm, the complete-data log-likelihood is replaced with the expectation $\mathbb{E}_p \, \widetilde{l}(\boldsymbol{\theta} \,|\, \tau, \boldsymbol{Z})$, where the subscript p in the expectation indicates that $\boldsymbol{Z}$ is distributed according to the conditional pdf of $\boldsymbol{Z}$ given $\mathcal{T} = \tau$; that is, with pdf

$$p(z) = g(z \,|\, \tau, \boldsymbol{\theta}) \propto g(\tau, z \,|\, \boldsymbol{\theta}). \tag{4.35}$$

Note that $p(z)$ is of the form $p_1(z_1) \cdots p_n(z_n)$ so that, given $\mathcal{T} = \tau$, the components of $\boldsymbol{Z}$ are independent of each other. The EM algorithm for mixture models can now be formulated as follows.

Algorithm 4.5.1: EM Algorithm for Mixture Models

input: Data τ, initial guess $\boldsymbol{\theta}^{(0)}$.
output: Approximation of the maximum likelihood estimate.

$t \leftarrow 1$
while a stopping criterion is not met **do**
  Expectation Step: Find $p^{(t)}(z) := g(z \,|\, \tau, \boldsymbol{\theta}^{(t-1)})$ and $Q^{(t)}(\boldsymbol{\theta}) := \mathbb{E}_{p^{(t)}} \widetilde{l}(\boldsymbol{\theta} \,|\, \tau, \boldsymbol{Z})$.
  Maximization Step: Let $\boldsymbol{\theta}^{(t)} \leftarrow \operatorname{argmax}_{\boldsymbol{\theta}} Q^{(t)}(\boldsymbol{\theta})$.
  $t \leftarrow t + 1$
return $\boldsymbol{\theta}^{(t)}$

A possible termination condition is to stop when $\left| l(\boldsymbol{\theta}^{(t)} \,|\, \tau) - l(\boldsymbol{\theta}^{(t-1)} \,|\, \tau) \right| / \left| l(\boldsymbol{\theta}^{(t)} \,|\, \tau) \right| < \varepsilon$ for some small tolerance $\varepsilon > 0$. As was mentioned in Section 4.3, the sequence of log-likelihood values *does not decrease* with each iteration. Under certain continuity conditions, the sequence $\{\boldsymbol{\theta}^{(t)}\}$ is guaranteed to converge to a local maximizer of the log-likelihood l. Convergence to a global maximizer (if it exists) depends on the appropriate choice for the starting value. Typically, the algorithm is run from different random starting points.

For the case of Gaussian mixtures, each $\phi_k = \phi(\cdot \,|\, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k), k = 1, \ldots, K$ is the density of a d-dimensional Gaussian distribution. Let $\boldsymbol{\theta}^{(t-1)}$ be the current guess for the optimal parameter vector, consisting of the weights $\{w_k^{(t-1)}\}$, mean vectors $\{\boldsymbol{\mu}_k^{(t-1)}\}$, and covariance matrices $\{\boldsymbol{\Sigma}_k^{(t-1)}\}$. We first determine $p^{(t)}$ — the pdf of $\boldsymbol{Z}$ conditional on $\mathcal{T} = \tau$ — for the given guess $\boldsymbol{\theta}^{(t-1)}$. As mentioned before, the components of $\boldsymbol{Z}$ given $\mathcal{T} = \tau$ are independent,

so it suffices to specify the discrete pdf, $p_i^{(t)}$ say, of each Z_i given the observed point $\boldsymbol{X}_i = \boldsymbol{x}_i$. The latter can be found from Bayes' formula:

$$p_i^{(t)}(k) \propto w_k^{(t-1)}\, \phi_k(\boldsymbol{x}_i \,|\, \boldsymbol{\mu}_k^{(t-1)}, \boldsymbol{\Sigma}_k^{(t-1)}), \quad k = 1, \ldots, K. \tag{4.36}$$

Next, in view of (4.34), the function $Q^{(t)}(\boldsymbol{\theta})$ can be written as

$$Q^{(t)}(\boldsymbol{\theta}) = \mathbb{E}_{p^{(t)}} \sum_{i=1}^{n} \left(\ln w_{Z_i} + \ln\, \phi_{Z_i}(\boldsymbol{x}_i \,|\, \boldsymbol{\mu}_{Z_i}, \boldsymbol{\Sigma}_{Z_i}) \right) = \sum_{i=1}^{n} \mathbb{E}_{p_i^{(t)}} \left[\ln w_{Z_i} + \ln\, \phi_{Z_i}(\boldsymbol{x}_i \,|\, \boldsymbol{\mu}_{Z_i}, \boldsymbol{\Sigma}_{Z_i}) \right],$$

where the $\{Z_i\}$ are independent and Z_i is distributed according to $p_i^{(t)}$ in (4.36). This completes the *E-step*. In the *M-step* we maximize $Q^{(t)}$ with respect to the parameter $\boldsymbol{\theta}$; that is, with respect to the $\{w_k\}$, $\{\boldsymbol{\mu}_k\}$, and $\{\boldsymbol{\Sigma}_k\}$. In particular, we maximize

$$\sum_{i=1}^{n} \sum_{k=1}^{K} p_i^{(t)}(k) \left[\ln w_k + \ln \phi_k(\boldsymbol{x}_i \,|\, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right],$$

under the condition $\sum_k w_k = 1$. Using Lagrange multipliers and the fact that $\sum_{k=1}^{K} p_i^{(t)}(k) = 1$ gives the solution for the $\{w_k\}$:

$$w_k = \frac{1}{n} \sum_{i=1}^{n} p_i^{(t)}(k), \quad k = 1, \ldots, K. \tag{4.37}$$

The solutions for $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$ now follow from maximizing $\sum_{i=1}^{n} p_i^{(t)}(k)\, \ln \phi_k(\boldsymbol{x}_i \,|\, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$, leading to

$$\boldsymbol{\mu}_k = \frac{\sum_{i=1}^{n} p_i^{(t)}(k)\, \boldsymbol{x}_i}{\sum_{i=1}^{n} p_i^{(t)}(k)}, \quad k = 1, \ldots, K \tag{4.38}$$

and

$$\boldsymbol{\Sigma}_k = \frac{\sum_{i=1}^{n} p_i^{(t)}(k)\, (\boldsymbol{x}_i - \boldsymbol{\mu}_k)(\boldsymbol{x}_i - \boldsymbol{\mu}_k)^\top}{\sum_{i=1}^{n} p_i^{(t)}(k)}, \quad k = 1, \ldots, K, \tag{4.39}$$

which are very similar to the well-known formulas for the MLEs of the parameters of a Gaussian distribution. After assigning the solution parameters to $\boldsymbol{\theta}^{(t)}$ and increasing the iteration counter t by 1, the steps (4.36), (4.37), (4.38), and (4.39) are repeated until convergence is reached. Convergence of the EM algorithm is very sensitive to the choice of initial parameters. It is therefore recommended to try various different starting conditions. For a further discussion of the theoretical and practical aspects of the EM algorithm we refer to [85].

■ **Example 4.5 (Clustering via EM)** We return to the data in Example 4.4, depicted in Figure 4.4, and adopt the model that the data is coming from a mixture of three bivariate Gaussian distributions.

The Python code below implements the EM procedure described in Algorithm 4.5.1. The initial mean vectors $\{\boldsymbol{\mu}_k\}$ of the bivariate Gaussian distributions are chosen (from visual inspection) to lie roughly in the middle of each cluster, in this case $[-2, -3]^\top$, $[-4, 1]^\top$, and $[0, -1]^\top$. The corresponding covariance matrices are initially chosen as identity matrices, which is appropriate given the observed spread of the data in Figure 4.4. Finally, the initial weights are $1/3, 1/3, 1/3$. For simplicity, the algorithm stops after 100 iterations, which in this case is more than enough to guarantee convergence. The code and data are available from the book's website in the GitHub folder Chapter4.

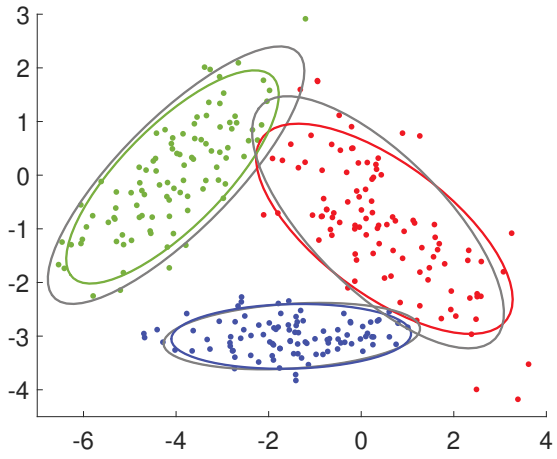
EMclust.py

```
import numpy as np
from scipy.stats import multivariate_normal

Xmat = np.genfromtxt('clusterdata.csv', delimiter=',')
K = 3
n, D = Xmat.shape

W = np.array([[1/3,1/3,1/3]])
M  = np.array([[-2.0,-4,0],[-3,1,-1]], dtype=np.float32)
# Note that if above *all* entries were written as integers, M would
# be defined to be of integer type, which will give the wrong answer

C = np.zeros((3,2,2))

C[:,0,0] = 1
C[:,1,1] = 1

p = np.zeros((3,300))

for i in range(0,100):

#E-step
    for k in range(0,K):
        mvn = multivariate_normal( M[:,k].T, C[k,:,:] )
        p[k,:] = W[0,k]*mvn.pdf(Xmat)

# M-Step
    p = (p/sum(p,0))   #normalize
    W = np.mean(p,1).reshape(1,3)

    for k in range(0,K):
        M[:,k] = (Xmat.T @ p[k,:].T)/sum(p[k,:])
        xm = Xmat.T - M[:,k].reshape(2,1)
        C[k,:,:] = xm @ (xm*p[k,:]).T/sum(p[k,:])
```

The estimated parameters of the mixture distribution are given on the right-hand side of Figure 4.6. After relabeling of the clusters, we can observe a close match with the parameters in Figure 4.4.

The ellipses on the left-hand side of Figure 4.6 show a close match between the 95% probability ellipses[2] of the original Gaussian distributions (in gray) and the estimated ones. A natural way to cluster each point $\boldsymbol{x}_i$ is to assign it to the cluster k for which the conditional probability $p_i(k)$ is maximal (with ties resolved arbitrarily). This gives the clustering of the points into red, green, and blue clusters in the figure.

[2]For each mixture component, the contour of the corresponding bivariate normal pdf is shown that encloses 95% of the probability mass.



weight	mean vector	covariance matrix
0.33	$\begin{bmatrix} -1.51 \\ -3.01 \end{bmatrix}$	$\begin{bmatrix} 1.75 & 0.03 \\ 0.03 & 0.095 \end{bmatrix}$
0.32	$\begin{bmatrix} -4.08 \\ -0.033 \end{bmatrix}$	$\begin{bmatrix} 1.37 & 0.92 \\ 0.92 & 1.03 \end{bmatrix}$
0.35	$\begin{bmatrix} 0.36 \\ -0.88 \end{bmatrix}$	$\begin{bmatrix} 1.93 & -1.20 \\ -1.20 & 1.44 \end{bmatrix}$

Figure 4.6: The results of the EM clustering algorithm applied to the data depicted in Figure 4.4.

As an alternative to the EM algorithm, one can of course use continuous multiextremal optimization algorithms to directly optimize the log-likelihood function $l(\boldsymbol{\theta}\,|\,\tau) = \ln g(\tau\,|\,\boldsymbol{\theta})$ in (4.33) over the set Θ of all possible $\boldsymbol{\theta}$. This is done for example in [15], demonstrating superior results to EM when there are few data points. Closer investigation of the likelihood function reveals that there is a hidden problem with any maximum likelihood approach for clustering if Θ is chosen as large as possible — i.e., any mixture distribution is possible. To demonstrate this problem, consider Figure 4.7, depicting the probability density function, $g(\cdot\,|\,\boldsymbol{\theta})$ of a mixture of two Gaussian distributions, where $\boldsymbol{\theta} = [w, \mu_1, \sigma_1^2, \mu_2, \sigma_2^2]^\top$ is the vector of parameters for the mixture distribution. The log-likelihood function is given by $l(\boldsymbol{\theta}\,|\,\tau) = \sum_{i=1}^{4} \ln g(x_i\,|\,\boldsymbol{\theta})$, where $x_1, \ldots, x_4$ are the data (indicated by dots in the figure).

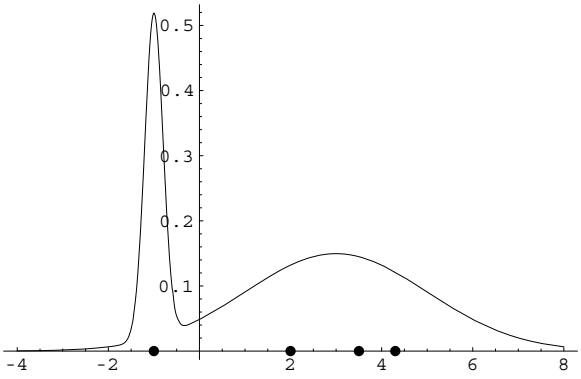


Figure 4.7: Mixture of two Gaussian distributions.

It is clear that by fixing the mixing constant w at 0.25 (say) and centering the first cluster at x_1, one can obtain an arbitrarily large likelihood value by taking the variance of the first cluster to be arbitrarily small. Similarly, for higher dimensional data, by choosing "point" or "line" clusters, or in general "degenerate" clusters, one can make the value of the likelihood infinite. This is a manifestation of the familiar overfitting problem for the

training loss that we already encountered in Chapter 2. Thus, the unconstrained maximization of the log-likelihood function is an ill-posed problem, irrespective of the choice of the optimization algorithm!

Two possible solutions to this "overfitting" problem are:

1. Restrict the parameter set Θ in such a way that degenerate clusters (sometimes called spurious clusters) are not allowed.
2. Run the given algorithm and if the solution is degenerate, discard it and run the algorithm afresh. Keep restarting the algorithm until a non-degenerate solution is obtained.

The first approach is usually applied to multiextremal optimization algorithms and the second is used for the EM algorithm.

4.6 Clustering via Vector Quantization

In the previous section we introduced clustering via mixture models, as a form of parametric density estimation (as opposed to the nonparametric density estimation in Section 4.4). The clusters were modeled in a natural way via the latent variables and the EM algorithm provided a convenient way to assign the cluster members. In this section we consider a more heuristic approach to clustering by ignoring the distributional properties of the data. The resulting algorithms tend to scale better with the number of samples n and the dimensionality d.

In mathematical terms, we consider the following clustering (also called data segmentation) problem. Given a collection $\tau = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ of data points in some d-dimensional space $\mathcal{X}$, divide this data set into K clusters (groups) such that some loss function is minimized. A convenient way to determine these clusters is to first divide up the entire space $\mathcal{X}$, using some distance function $\mathrm{dist}(\cdot,\cdot)$ on this space. A standard choice is the Euclidean (or L_2) distance:

$$\mathrm{dist}(\boldsymbol{x}, \boldsymbol{x}') = \|\boldsymbol{x} - \boldsymbol{x}'\| = \sqrt{\sum_{i=1}^{d} (x_i - x_i')^2}.$$

MANHATTAN DISTANCE

Other commonly used distance measures on $\mathbb{R}^d$ include the *Manhattan distance*:

$$\sum_{i=1}^{d} |x_i - x_i'|$$

MAXIMUM DISTANCE

and the *maximum distance*:

$$\max_{i=1,\ldots,d} |x_i - x_i'|.$$

HAMMING DISTANCE

On the set of strings of length d, an often-used distance measure is the *Hamming distance*:

$$\sum_{i=1}^{d} \mathbb{1}\{x_i \neq x_i'\},$$

that is, the number of mismatched characters. For example, the Hamming distance between 010101 and 011010 is 4.

We can partition the space $\mathcal{X}$ into regions as follows: First, we choose K points $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_K$ called *cluster centers* or *source vectors*. For each $k = 1, \ldots, K$, let

SOURCE VECTORS

$$\mathcal{R}_k = \{\boldsymbol{x} \in \mathcal{X} : \text{dist}(\boldsymbol{x}, \boldsymbol{c}_k) \leqslant \text{dist}(\boldsymbol{x}, \boldsymbol{c}_i) \text{ for all } i \neq k\}$$

be the set of points in $\mathcal{X}$ that lie closer to $\boldsymbol{c}_k$ than any other center. The regions or *cells* $\{\mathcal{R}_k\}$ divide the space $\mathcal{X}$ into what is called a *Voronoi diagram* or a *Voronoi tessellation*. Figure 4.8 shows a Voronoi tessellation of the plane into ten regions, using the Euclidean distance. Note that here the boundaries between the Voronoi cells are straight line segments. In particular, if cell $\mathcal{R}_i$ and $\mathcal{R}_j$ share a border, then a point on this border must satisfy $\|\boldsymbol{x} - \boldsymbol{c}_i\| = \|\boldsymbol{x} - \boldsymbol{c}_j\|$; that is, it must lie on the line that passes through the point $(\boldsymbol{c}_j + \boldsymbol{c}_i)/2$ (that is, the midway point of the line segment between $\boldsymbol{c}_i$ and $\boldsymbol{c}_j$) and be perpendicular to $\boldsymbol{c}_j - \boldsymbol{c}_i$.

VORONOI TESSELLATION

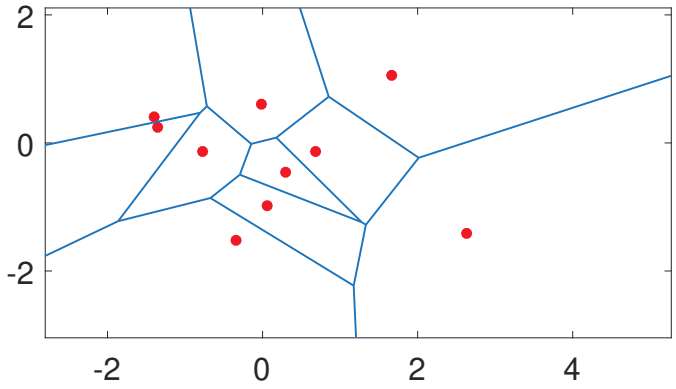


Figure 4.8: A Voronoi tessellation of the plane into ten cells, determined by the (red) centers.

Once the centers (and thus the cells $\{\mathcal{R}_k\}$) are chosen, the points in τ can be clustered according to their nearest center. Points on the boundary have to be treated separately. This is a moot point for continuous data, as generally no data points will lie exactly on the boundary.

The main remaining issue is how to choose the centers so as to cluster the data in some optimal way. In terms of our (unsupervised) learning framework, we wish to approximate a vector $\boldsymbol{x}$ via one of $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_K$, using a piecewise constant vector-valued function

$$\boldsymbol{g}(\boldsymbol{x} \,|\, \mathbf{C}) := \sum_{k=1}^{K} \boldsymbol{c}_k \, \mathbb{1}\{\boldsymbol{x} \in \mathcal{R}_k\},$$

where $\mathbf{C}$ is the $d \times K$ matrix $[\boldsymbol{c}_1, \ldots, \boldsymbol{c}_K]$. Thus, $\boldsymbol{g}(\boldsymbol{x} \,|\, \mathbf{C}) = \boldsymbol{c}_k$ when $\boldsymbol{x}$ falls in region $\mathcal{R}_k$ (we ignore ties). Within this class $\mathcal{G}$ of functions, parameterized by $\mathbf{C}$, our aim is to minimize the training loss. In particular, for the squared-error loss, $\text{Loss}(\boldsymbol{x}, \boldsymbol{x}') = \|\boldsymbol{x} - \boldsymbol{x}'\|^2$, the training loss is

$$\ell_{\tau_n}(\boldsymbol{g}(\cdot \,|\, \mathbf{C})) = \frac{1}{n} \sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{g}(\boldsymbol{x}_i \,|\, \mathbf{C})\|^2 = \frac{1}{n} \sum_{k=1}^{K} \sum_{\boldsymbol{x} \in \mathcal{R}_k \cap \tau_n} \|\boldsymbol{x} - \boldsymbol{c}_k\|^2. \tag{4.40}$$

Thus, the training loss minimizes the average squared distance between the centers. This framework also combines both the encoding and decoding steps in *vector quantization*

VECTOR QUANTIZATION

[125]. Namely, we wish to "quantize" or "encode" the vectors in τ in such a way that each vector is represented by one of K source vectors $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_K$, such that the loss (4.40) of this representation is minimized.

Most well-known clustering and vector quantization methods update the vector of centers, starting from some initial choice and using iterative (typically gradient-based) procedures. It is important to realize that in this case (4.40) is seen as a function of the centers, where each point $\boldsymbol{x}$ is assigned to the nearest center, thus determining the clusters. It is well known that this type of problem — optimization with respect to the centers — is highly multiextremal and, depending on the initial clusters, gradient-based procedures tend to converge to a *local minimum* rather than a global minimum.

4.6.1 K-Means

One of the simplest methods for clustering is the K-means method. It is an iterative method where, starting from an initial guess for the centers, new centers are formed by taking sample means of the current points in each cluster. The new centers are thus the *centroids* of the points in each cell. Although there exist many different varieties of the K-means algorithm, they are all essentially of the following form:

CENTROIDS

Algorithm 4.6.1: K-Means

input: Collection of points $\tau = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$, number of clusters K, initial centers $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_K$.

output: Cluster centers and cells (regions).

1 **while** a stopping criterion is not met **do**
2 $\quad \mathcal{R}_1, \ldots, \mathcal{R}_K \leftarrow \emptyset$ (empty sets).
3 $\quad$ **for** $i = 1$ **to** n **do**
4 $\quad\quad \boldsymbol{d} \leftarrow [\mathrm{dist}(\boldsymbol{x}_i, \boldsymbol{c}_1), \ldots, \mathrm{dist}(\boldsymbol{x}_i, \boldsymbol{c}_K)]$ `// distances to centers`
5 $\quad\quad k \leftarrow \mathrm{argmin}_j \, d_j$
6 $\quad\quad \mathcal{R}_k \leftarrow \mathcal{R}_k \cup \{\boldsymbol{x}_i\}$ `// assign` $\boldsymbol{x}_i$ `to cluster` k
7 $\quad$ **for** $k = 1$ **to** K **do**
8 $\quad\quad \boldsymbol{c}_k \leftarrow \dfrac{\sum_{\boldsymbol{x} \in \mathcal{R}_k} \boldsymbol{x}}{|\mathcal{R}_k|}$ `// compute the new center as a centroid of points`
9 **return** $\{\boldsymbol{c}_k\}$, $\{\mathcal{R}_k\}$

Thus, at each iteration, for a given choice of centers, each point in τ is assigned to its nearest center. After all points have been assigned, the centers are recomputed as the centroids of all the points in the current cluster (Line 8). A typical stopping criterion is to stop when the centers no longer change very much. As the algorithm is quite sensitive to the choice of the initial centers, it is prudent to try multiple starting values, e.g., chosen randomly from the bounding box of the data points.

We can see the K-means method as a deterministic (or "hard") version of the probabilistic (or "soft") EM algorithm as follows. Suppose in the EM algorithm we have Gaussian mixtures with a fixed covariance matrix $\boldsymbol{\Sigma}_k = \sigma^2 \mathbf{I}_d$, $k = 1, \ldots, K$, where σ^2 should be thought of as being infinitesimally small. Consider iteration t of the EM algorithm. Having obtained the expectation vectors $\boldsymbol{\mu}_k^{(t-1)}$ and weights $w_k^{(t-1)}, k = 1, \ldots, K$, each point $\boldsymbol{x}_i$ is assigned a cluster label Z_i according to the probabilities $p_i^{(t)}(k), k = 1, \ldots, K$ given in (4.36).

But for $\sigma^2 \to 0$ the probability distribution $\{p_i^{(t)}(k)\}$ becomes degenerate, putting all its probability mass on $\operatorname{argmin}_k \|\boldsymbol{x}_i - \boldsymbol{\mu}_k\|^2$. This corresponds to the K-means rule of assigning $\boldsymbol{x}_i$ to its nearest cluster center. Moreover, in the M-step (4.38) each cluster center $\boldsymbol{\mu}_k^{(t)}$ is now updated according to the average of the $\{\boldsymbol{x}_i\}$ that have been assigned to cluster k. We thus obtain the same deterministic updating rule as in K-means.

■ **Example 4.6 (K-means Clustering)** We cluster the data from Figure 4.8 via K-means, using the Python implementation below. Note that the data points are stored as a 300×2 matrix `Xmat`. We take the same starting centers as in the EM example: $\boldsymbol{c}_1 = [-2, -3]^\top$, $\boldsymbol{c}_2 = [-4, 1]^\top$, and $\boldsymbol{c}_3 = [0, -1]^\top$. Note also that *squared* Euclidean distances are used in the computations, as these are slightly faster to compute than Euclidean distances (as no square root computations are required) while yielding exactly the same cluster center evaluations.

Kmeans.py

```
import numpy as np
Xmat = np.genfromtxt('clusterdata.csv', delimiter=',')
K = 3
n, D = Xmat.shape
c  = np.array([[-2.0,-4,0],[-3,1,-1]])  #initialize centers
cold = np.zeros(c.shape)
dist2 = np.zeros((K,n))
while np.abs(c - cold).sum() > 0.001:
   cold = c.copy()
   for i in range(0,K): #compute the squared distances
        dist2[i,:] = np.sum((Xmat - c[:,i].T)**2, 1)

   label = np.argmin(dist2,0) #assign the points to nearest centroid
   minvals = np.amin(dist2,0)
   for i in range(0,K): # recompute the centroids
      c[:,i] = np.mean(Xmat[np.where(label == i),:],1).reshape(1,2)

print('Loss = {:3.3f}'.format(minvals.mean()))
```

```
Loss = 2.288
```

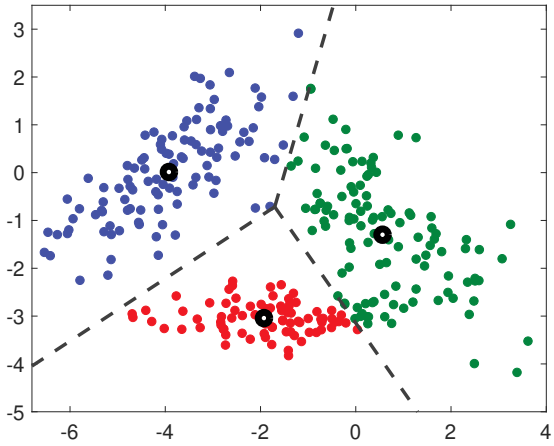


Figure 4.9: Results of the K-means algorithm applied to the data in Figure 4.4. The thick black circles are the centroids and the dotted lines define the cell boundaries.

We found the cluster centers $\boldsymbol{c}_1 = [-1.9286, -3.0416]^\top$, $\boldsymbol{c}_2 = [-3.9237, 0.0131]^\top$, and $\boldsymbol{c}_3 = [0.5611, -1.2980]^\top$, giving the clustering depicted in Figure 4.9. The corresponding loss (4.40) was found to be 2.288. ■

4.6.2 Clustering via Continuous Multiextremal Optimization

As already mentioned, the exact minimization of the loss function (4.40) is difficult to accomplish via standard local search methods such as gradient descent, as the function is highly multimodal. However, nothing is preventing us from using global optimization methods such as the CE or SCO methods discussed in Sections 3.4.2 and 3.4.3.

☞ 100

■ **Example 4.7 (Clustering via CE)** We take the same data set as in Example 4.6 and cluster the points via minimization of the loss (4.40) using the CE method. The Python code below is very similar to the code in Example 3.16, except that now we are dealing with a six-dimensional optimization problem. The loss function is implemented in the function `Scluster`, which essentially reuses the squared distance computation of the K-means code in Example 4.6. The CE program typically converges to a loss of 2.287, corresponding to the (global) minimizers $\boldsymbol{c}_1 = [-1.9286, -3.0416]^\top$, $\boldsymbol{c}_2 = [-3.8681, 0.0456]^\top$, and $\boldsymbol{c}_3 = [0.5880, -1.3526]^\top$, which slightly differs from the local minimizers for the K-means algorithm.

☞ 101

clustCE.py

```
import numpy as np
np.set_printoptions(precision=4)

Xmat = np.genfromtxt('clusterdata.csv', delimiter=',')
K = 3
n, D = Xmat.shape

def Scluster(c):
    n, D = Xmat.shape
    dist2 = np.zeros((K,n))
    cc = c.reshape(D,K)
    for i in range(0,K):
        dist2[i,:] = np.sum((Xmat - cc[:,i].T)**2, 1)
    minvals = np.amin(dist2,0)
    return minvals.mean()

numvar = K*D
mu = np.zeros(numvar)  #initialize centers
sigma = np.ones(numvar)*2
rho = 0.1
N = 500; Nel = int(N*rho); eps = 0.001

func = Scluster
best_trj = np.array(numvar)
best_perf = np.Inf
trj = np.zeros(shape=(N,numvar))

while(np.max(sigma)>eps):
   for i in range(0,numvar):
```

```
        trj[:,i] = (np.random.randn(N,1)*sigma[i]+ mu[i]).reshape(N,)
    S = np.zeros(N)
    for i in range(0,N):
        S[i] = func(trj[i])

    sortedids = np.argsort(S) # from smallest to largest
    S_sorted = S[sortedids]
    best_trj = np.array(n)
    best_perf = np.Inf
    eliteids = sortedids[range(0,Nel)]
    eliteTrj = trj[eliteids,:]
    mu = np.mean(eliteTrj,axis=0)
    sigma = np.std(eliteTrj,axis=0)

    if(best_perf>S_sorted[0]):
        best_perf = S_sorted[0]
        best_trj = trj[sortedids[0]]

print(best_perf)
print(best_trj.reshape(2,3))
```

```
2.2874901831572947
[[-3.9238 -1.8477  0.5895]
 [ 0.0134 -3.0292 -1.2442]]
```

4.7 Hierarchical Clustering

It is sometimes useful to determine data clusters in a hierarchical manner; an example is the construction of evolutionary relationships between animal species. Establishing a hierarchy of clusters can be done in a bottom-up or a top-down manner. In the bottom-up approach, also called *agglomerative clustering*, the data points are merged in larger and larger clusters until all the points have been merged into a single cluster. In the top-down or *divisive clustering* approach, the data set is divided up into smaller and smaller clusters. The left panel of Figure 4.10 depicts a hierarchy of clusters.

AGGLOMERATIVE CLUSTERING

DIVISIVE CLUSTERING

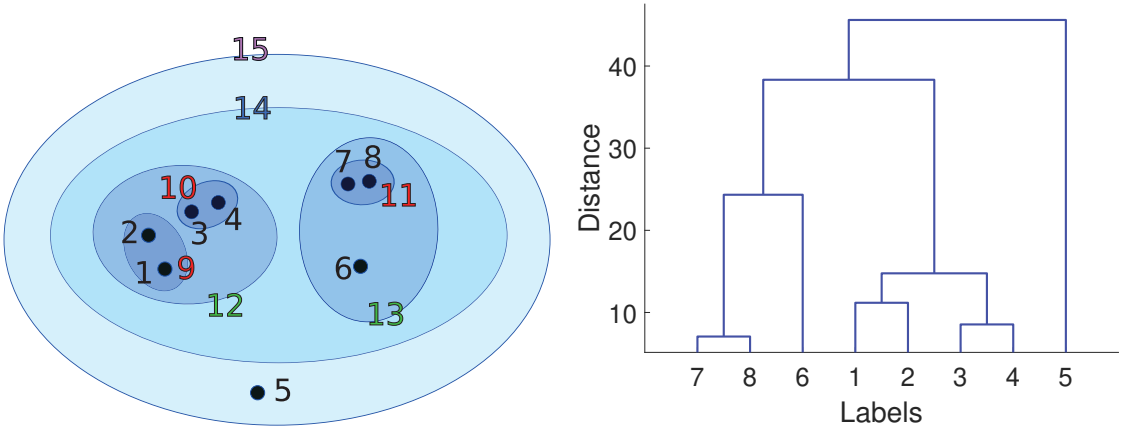


Figure 4.10: Left: a cluster hierarchy of 15 clusters. Right: the corresponding dendrogram.

In Figure 4.10, each cluster is given a cluster identifier. At the lowest level are clusters comprised of the original data points (identifiers $1, \ldots, 8$). The union of clusters 1 and 2 form a cluster with identifier 9, and the union of 3 and 4 form a cluster with identifier 10. In turn the union of clusters 9 and 10 constitutes cluster 12, and so on.

DENDROGRAM

The right panel of Figure 4.10 shows a convenient way to visualize cluster hierarchies using a *dendrogram* (from the Greek *dendro* for tree). A dendrogram not only summarizes how clusters are merged or split, but also shows the distance between clusters, here on the vertical axis. The horizontal axis shows which cluster each data point (label) belongs to.

Many different types of hierarchical clustering can be performed, depending on how the distance is defined between two data points and between two clusters. Denote the data set by $\mathcal{X} = \{\boldsymbol{x}_i, i = 1, \ldots, n\}$. As in Section 4.6, let $\text{dist}(\boldsymbol{x}_i, \boldsymbol{x}_j)$ be the distance between data points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. The default choice is the Euclidean distance $\text{dist}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \|\boldsymbol{x}_i - \boldsymbol{x}_j\|$.

Let $\mathcal{I}$ and $\mathcal{J}$ be two disjoint subsets of $\{1, \ldots, n\}$. These sets correspond to two disjoint subsets (that is, clusters) of $\mathcal{X}$: $\{\boldsymbol{x}_i, i = \mathcal{I}\}$ and $\{\boldsymbol{x}_i, i = \mathcal{J}\}$. We denote the distance between these two clusters by $d(\mathcal{I}, \mathcal{J})$. By specifying the function d, we indicate how the clusters are linked. For this reason it is also referred to as the *linkage* criterion. We give a number of examples:

LINKAGE

- **Single linkage.** The closest distance between the clusters.

$$d_{\min}(\mathcal{I}, \mathcal{J}) := \min_{i \in \mathcal{I}, j \in \mathcal{J}} \text{dist}(\boldsymbol{x}_i, \boldsymbol{x}_j).$$

- **Complete linkage.** The furthest distance between the clusters.

$$d_{\max}(\mathcal{I}, \mathcal{J}) := \max_{i \in \mathcal{I}, j \in \mathcal{J}} \text{dist}(\boldsymbol{x}_i, \boldsymbol{x}_j).$$

- **Group average.** The mean distance between the clusters. Note that this depends on the cluster sizes.

$$d_{\text{avg}}(\mathcal{I}, \mathcal{J}) := \frac{1}{|\mathcal{I}|\,|\mathcal{J}|} \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}} \text{dist}(\boldsymbol{x}_i, \boldsymbol{x}_j).$$

For these linkage criteria, $\mathcal{X}$ is usually assumed to be $\mathbb{R}^d$ with the Euclidean distance.

WARD'S LINKAGE

Another notable measure for the distance between clusters is *Ward's minimum variance linkage* criterion. Here, the distance between clusters is expressed as the additional amount of "variance" (expressed in terms of the sum of squares) that would be introduced if the two clusters were merged. More precisely, for any set $\mathcal{K}$ of indices (labels) let $\overline{\boldsymbol{x}}_{\mathcal{K}} = \sum_{k \in \mathcal{K}} \boldsymbol{x}_k / |\mathcal{K}|$ denote its corresponding cluster mean. Then

$$d_{\text{Ward}}(\mathcal{I}, \mathcal{J}) := \sum_{k \in I \cup \mathcal{J}} \|\boldsymbol{x}_k - \overline{\boldsymbol{x}}_{\mathcal{I} \cup \mathcal{J}}\|^2 - \left(\sum_{i \in \mathcal{I}} \|\boldsymbol{x}_i - \overline{\boldsymbol{x}}_{\mathcal{I}}\|^2 + \sum_{j \in \mathcal{J}} \|\boldsymbol{x}_j - \overline{\boldsymbol{x}}_{\mathcal{J}}\|^2 \right). \tag{4.41}$$

It can be shown (see Exercise 8) that the Ward linkage depends only on the cluster means and the cluster sizes for $\mathcal{I}$ and $\mathcal{J}$:

$$d_{\text{Ward}}(\mathcal{I}, \mathcal{J}) = \frac{|\mathcal{I}|\,|\mathcal{J}|}{|\mathcal{I}| + |\mathcal{J}|} \|\overline{\boldsymbol{x}}_{\mathcal{I}} - \overline{\boldsymbol{x}}_{\mathcal{J}}\|^2.$$

In software implementations, the Ward linkage function is often rescaled by multiplying it by a factor of 2. In this way, the distance between one-point clusters $\{\boldsymbol{x}_i\}$ and $\{\boldsymbol{x}_j\}$ is the *squared* Euclidean distance $\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2$.

Having chosen a distance on $\mathcal{X}$ and a linkage criterion, a general agglomerative clustering algorithm proceeds in the following "greedy" manner.

Algorithm 4.7.1: Greedy Agglomerative Clustering

input: Distance function dist, linkage function d, number of clusters K.
output: The label sets for the tree.

Initialize the set of cluster identifiers: $\mathcal{I} = \{1, \dots, n\}$.
Initialize the corresponding label sets: $\mathcal{L}_i = \{i\}$, $i \in \mathcal{I}$.
Initialize a distance matrix $\mathbf{D} = [d_{ij}]$ with $d_{ij} = d(\{i\}, \{j\})$.
for $k = n + 1$ **to** $2n - K$ **do**
    Find i and $j > i$ in $\mathcal{I}$ such that d_{ij} is minimal.
    Create a new label set $\mathcal{L}_k := \mathcal{L}_i \cup \mathcal{L}_j$.
    Add the new identifier k to $\mathcal{I}$ and remove the old identifiers i and j from $\mathcal{I}$.
    Update the distance matrix $\mathbf{D}$ with respect to the identifiers i, j, and k.
return $\mathcal{L}_i, i = 1, \dots, 2n - K$

Initially, the distance matrix $\mathbf{D}$ contains the (linkage) distances between the one-point clusters containing one of the data points $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n$, and hence with identifiers $1, \dots, n$. Finding the shortest distance amounts to a table lookup in $\mathbf{D}$. When the closest clusters are found, they are merged into a new cluster, and a new identifier k (the smallest positive integer that has not yet been used as an identifier) is assigned to this cluster. The old identifiers i and j are removed from the cluster identifier set $\mathcal{I}$. The matrix $\mathbf{D}$ is then updated by adding a k-th column and row that contain the distances between k and any $m \in \mathcal{I}$. This updating step could be computationally quite costly if the cluster sizes are large and the linkage distance between the clusters depends on all points within the clusters. Fortunately, for many linkage functions, the matrix $\mathbf{D}$ can be updated in an efficient manner.

Suppose that at some stage in the algorithm, clusters $\mathcal{I}$ and $\mathcal{J}$, with identifiers i and j, respectively, are merged into a cluster $\mathcal{K} = \mathcal{I} \cup \mathcal{J}$ with identifier k. Let $\mathcal{M}$, with identifier m, be a previously assigned cluster. An update rule of the linkage distance d_{km} between $\mathcal{K}$ and $\mathcal{M}$ is called a *Lance–Williams* update if it can be written in the form

Lance–Williams

$$d_{km} = \alpha\, d_{im} + \beta\, d_{jm} + \gamma\, d_{ij} + \delta\, |d_{im} - d_{jm}|,$$

where $\alpha, \dots, \delta$ depend only on simple characteristics of the clusters involved, such as the number of elements within the clusters. Table 4.2 shows the update constants for a number of common linkage functions. For example, for single linkage, d_{im} is the minimal distance between $\mathcal{I}$ and $\mathcal{M}$, and d_{jm} is the minimal distance between $\mathcal{J}$ and $\mathcal{M}$. The smallest of these is the minimal distance between $\mathcal{K}$ and $\mathcal{M}$. That is, $d_{km} = \min\{d_{im}, d_{jm}\} = d_{im}/2 + d_{jm}/2 - |d_{im} - d_{jm}|$.

Table 4.2: Constants for the Lance–Williams update rule for various linkage functions, with n_i, n_j, n_m denoting the number of elements in the corresponding clusters.

Linkage	α	β	γ	δ
Single	$1/2$	$1/2$	0	$-1/2$
Complete	$1/2$	$1/2$	0	$1/2$
Group avg.	$\frac{n_i}{n_i+n_j}$	$\frac{n_j}{n_i+n_j}$	0	0
Ward	$\frac{n_i+n_m}{n_i+n_j+n_m}$	$\frac{n_j+n_m}{n_i+n_j+n_m}$	$\frac{-n_m}{n_i+n_j+n_m}$	0

LINKAGE MATRIX

In practice, Algorithm 4.7.1 is run until a single cluster is obtained. Instead of returning the label sets of all $2n-1$ clusters, a *linkage matrix* is returned that contains the same information. At the end of each iteration (Line 8) the linkage matrix stores the merged labels i and j, as well as the (minimal) distance d_{ij}. Other information such as the number of elements in the merged cluster can also be stored. Dendrograms and cluster labels can be directly constructed from the linkage matrix. In the following example, the linkage matrix is returned by the method `agg_cluster`.

■ **Example 4.8 (Agglomerative Hierarchical Clustering)** The Python code below gives a basic implementation of Algorithm 4.7.1 using the Ward linkage function. The methods `fcluster` and `dendrogram` from the `scipy` module can be used to identify the labels in a cluster and to draw the corresponding dendrogram.

AggCluster.py

```
import numpy as np
from scipy.spatial.distance import cdist

def update_distances(D,i,j, sizes): # distances for merged cluster
    n = D.shape[0]
    d = np.inf * np.ones(n+1)
    for k in range(n): # Update distances
       d[k] = ((sizes[i]+sizes[k])*D[i,k] +
       (sizes[j]+sizes[k])*D[j,k] -
       sizes[k]*D[i,j])/(sizes[i] + sizes[j] + sizes[k])

    infs =  np.inf * np.ones(n) # array of infinity
    D[i,:],D[:,i],D[j,:],D[:,j] =  infs,infs,infs,infs # deactivate
    new_D = np.inf * np.ones((n+1,n+1))
    new_D[0:n,0:n] = D # copy old matrix into new_D
    new_D[-1,:], new_D[:,-1] = d,d # add new row and column
    return new_D

def agg_cluster(X):
    n = X.shape[0]
    sizes = np.ones(n)
    D = cdist(X, X,metric = 'sqeuclidean') # initialize dist. matrix
        .
    np.fill_diagonal(D, np.inf * np.ones(D.shape[0]))
    Z = np.zeros((n-1,4))  #linkage matrix encodes hierarchy tree
    for t in range(n-1):
```

```
        i,j = np.unravel_index(D.argmin(), D.shape) # minimizer pair
        sizes = np.append(sizes, sizes[i] + sizes[j])
        Z[t,:]=np.array([i, j, np.sqrt(D[i,j]), sizes[-1]])
        D = update_distances(D, i,j, sizes)  # update distance matr.
    return Z

import scipy.cluster.hierarchy as h

X = np.genfromtxt('clusterdata.csv',delimiter=',') # read the data
Z = agg_cluster(X)  # form the linkage matrix

h.dendrogram(Z) # SciPy can produce a dendrogram from Z
# fcluster function assigns cluster ids to all points based on Z
cl = h.fcluster(Z, criterion = 'maxclust', t=3)

import matplotlib.pyplot as plt
plt.figure(2), plt.clf()
cols = ['red','green','blue']
colors = [cols[i-1] for i in cl]
plt.scatter(X[:,0], X[:,1],c=colors)
plt.show()
```

Note that the distance matrix is initialized with the squared Euclidean distance, so that the Ward linkage is rescaled by a factor of 2. Also, note that the linkage matrix stores the square root of the minimal cluster distances rather than the distances themselves. We leave it as an exercise to check that by using these modifications the results agree with the `linkage` method from `scipy`; see Exercise 9. ■

In contrast to the bottom-up (agglomerative) approach to hierarchical clustering, the divisive approach starts with one cluster, which is divided into two clusters that are as "dissimilar" as possible, which can then be further divided, and so on. We can use the same linkage criteria as for agglomerative clustering to divide a parent cluster into two child clusters by *maximizing* the distance between the child clusters. Although it is a natural to try to group together data by separating dissimilar ones as far as possible, the implementation of this idea tends to scale poorly with n. The problem is related to the well-known *max-cut problem*: given an $n \times n$ matrix of positive costs $c_{ij}, i, j \in \{1, \ldots, n\}$, partition the index set $\mathcal{I} = \{1, \ldots, n\}$ into two subsets $\mathcal{J}$ and $\mathcal{K}$ such that the total cost across the sets, that is,

MAX-CUT PROBLEM

$$\sum_{j \in \mathcal{J}} \sum_{k \in \mathcal{K}} d_{jk},$$

is maximal. If instead we maximize according to the *average* distance, we obtain the group average linkage criterion.

■ **Example 4.9 (Divisive Clustering via CE)** The following Python code is used to divide a small data set (of size 300) into two parts according to maximal group average linkage. It uses a short cross-entropy algorithm similar to the one presented in Example 3.19. Given a vector of probabilities $\{p_i, i = 1, \ldots, n\}$, the algorithm generates an $n \times n$ matrix of Bernoulli random variables with success probability p_i for column i. For each row, the 0s and 1s divide the index set into two clusters, and the corresponding average linkage

☞ 110

distance is computed. The matrix is then sorted row-wise according to these distances. Finally, the probabilities $\{p_i\}$ are updated according to the mean values of the best 10% rows. The process is repeated until the $\{p_i\}$ degenerate to a binary vector. This then presents the (approximate) solution.

clustCE2.py

```
import numpy as np
from numpy import genfromtxt
from scipy.spatial.distance import squareform
from scipy.spatial.distance import pdist
import matplotlib.pyplot as plt


def S(x,D):
    V1 = np.where(x==0)[0] # {V1,V2} is the partition
    V2 = np.where(x==1)[0]
    tmp = D[V1]
    tmp = tmp[:,V2]
    return np.mean(tmp) # the size of the cut


def maxcut(D,N,eps,rho,alpha):
    n = D.shape[1]
    Ne = int(rho*N)
    p = 1/2*np.ones(n)
    p[0] = 1.0
    while (np.max(np.minimum(p,np.subtract(1,p))) > eps):
        x = np.array(np.random.uniform(0,1,(N,n))<=p, dtype=np.int64)
        sx = np.zeros(N)
        for i in range(N):
            sx[i] = S(x[i],D)

        sortSX = np.flip(np.argsort(sx))
        #print("gamma = ",sx[sortSX[Ne-1]], " best=",sx[sortSX[0]])
        elIds = sortSX[0:Ne]
        elites = x[elIds]
        pnew = np.mean(elites, axis=0)
        p = alpha*pnew + (1.0-alpha)*p

    return np.round(p)

Xmat = genfromtxt('clusterdata.csv', delimiter=',')
n = Xmat.shape[0]
D = squareform(pdist(Xmat))
N = 1000
eps = 10**-2
rho = 0.1
alpha = 0.9

# CE
pout = maxcut(D,N,eps,rho, alpha);

cutval = S(pout,D)
```

```
print("cutvalue ",cutval)

#plot
V1 = np.where(pout==0)[0]
xblue = Xmat[V1]
V2 = np.where(pout==1)[0]
xred = Xmat[V2]
plt.scatter(xblue[:,0],xblue[:,1], c="blue")
plt.scatter(xred[:,0],xred[:,1], c="red")
```

```
cutvalue  4.625207676517948
```

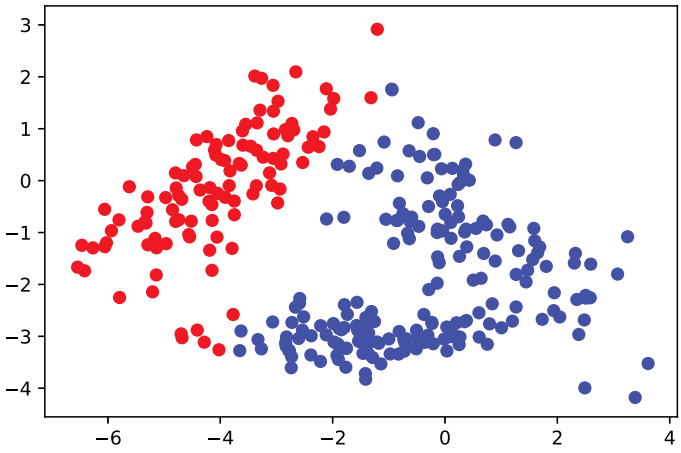


Figure 4.11: Division of the data in Figure 4.4 into two clusters, via the cross-entropy method.

4.8 Principal Component Analysis (PCA)

PRINCIPAL COMPONENT ANALYSIS

The main idea of *principal component analysis* (PCA) is to reduce the dimensionality of a data set consisting of many variables. PCA is a *feature reduction* (or *feature extraction*) mechanism, that helps us to handle high-dimensional data with more features than is convenient to interpret.

4.8.1 Motivation: Principal Axes of an Ellipsoid

Consider a d-dimensional normal distribution with mean vector $\mathbf{0}$ and covariance matrix $\boldsymbol{\Sigma}$. The corresponding pdf (see (2.33)) is ☞ 45

$$f(\boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^n \, |\boldsymbol{\Sigma}|}} \, \mathrm{e}^{-\frac{1}{2} \, \boldsymbol{x}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{x}}, \quad \boldsymbol{x} \in \mathbb{R}^d.$$

If we were to draw many iid samples from this pdf, the points would roughly have an *ellipsoid* pattern, as illustrated in Figure 3.1, and correspond to the contours of f: sets of ☞ 71

points $\boldsymbol{x}$ such that $\boldsymbol{x}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{x} = c$, for some $c \geqslant 0$. In particular, consider the ellipsoid

$$\boldsymbol{x}^\top \boldsymbol{\Sigma}^{-1} \boldsymbol{x} = 1, \quad \boldsymbol{x} \in \mathbb{R}^d. \tag{4.42}$$

☞ 373
☞ 366

PRINCIPAL AXES

SINGULAR VALUE DECOMPOSITION

☞ 378

Let $\boldsymbol{\Sigma} = \mathbf{B}\mathbf{B}^\top$, where $\mathbf{B}$ is for example the (lower) Cholesky matrix. Then, as explained in Example A.5, the ellipsoid (4.42) can also be viewed as the linear transformation of d-dimensional unit sphere via matrix $\mathbf{B}$. Moreover, the *principal axes* of the ellipsoid can be found via a *singular value decomposition* (SVD) of $\mathbf{B}$ (or $\boldsymbol{\Sigma}$); see Section A.6.5 and Example A.8. In particular, suppose that an SVD of $\mathbf{B}$ is

$$\mathbf{B} = \mathbf{U}\mathbf{D}\mathbf{V}^\top \qquad \text{(note that an SVD of } \boldsymbol{\Sigma} \text{ is then } \mathbf{U}\mathbf{D}^2\mathbf{U}^\top\text{)}.$$

The columns of the matrix $\mathbf{UD}$ correspond to the principal axes of the ellipsoid, and the relative magnitudes of the axes are given by the elements of the diagonal matrix $\mathbf{D}$. If some of these magnitudes are small compared to the others, a reduction in the dimension of the space may be achieved by *projecting* each point $\boldsymbol{x} \in \mathbb{R}^d$ onto the subspace spanned by the main (say $k \ll d$) columns of $\mathbf{U}$ — the so-called *principal components*. Suppose without loss of generality that the first k principal components are given by the first k columns of $\mathbf{U}$, and let $\mathbf{U}_k$ be the corresponding $d \times k$ matrix.

PRINCIPAL COMPONENTS

With respect to the standard basis $\{\boldsymbol{e}_i\}$, the vector $\boldsymbol{x} = x_1\boldsymbol{e}_1 + \cdots + x_d\boldsymbol{e}_d$ is represented by the d-dimensional vector $[x_1, \ldots, x_d]^\top$. With respect to the orthonormal basis $\{\boldsymbol{u}_i\}$ formed by the columns of matrix $\mathbf{U}$, the representation of $\boldsymbol{x}$ is $\mathbf{U}^\top\boldsymbol{x}$. Similarly, the projection of any point $\boldsymbol{x}$ onto the subspace spanned by the first k principal vectors is represented by the k-dimensional vector $\mathbf{U}_k^\top\boldsymbol{x}$, with respect to the orthonormal basis formed by the columns of $\mathbf{U}_k$. So, the idea is that if a point $\boldsymbol{x}$ lies close to its projection $\mathbf{U}_k\mathbf{U}_k^\top\boldsymbol{x}$, we may represent it via k numbers instead of d, using the combined features given by the k principal components. See Section A.4 for a review of projections and orthonormal bases.

☞ 362

■ **Example 4.10 (Principal Components)** Consider the matrix

$$\boldsymbol{\Sigma} = \begin{bmatrix} 14 & 8 & 3 \\ 8 & 5 & 2 \\ 3 & 2 & 1 \end{bmatrix},$$

which can be written as $\boldsymbol{\Sigma} = \mathbf{B}\mathbf{B}^\top$, with

$$\mathbf{B} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}.$$

Figure 4.12 depicts the ellipsoid $\boldsymbol{x}^\top\boldsymbol{\Sigma}\boldsymbol{x} = 1$, which can be obtained by linearly transforming the points on the unit sphere by means of the matrix $\mathbf{B}$. The principal axes and sizes of the ellipsoid are found through a singular value decomposition $\mathbf{B} = \mathbf{U}\mathbf{D}\mathbf{V}^\top$, where $\mathbf{U}$ and $\mathbf{D}$ are

$$\mathbf{U} = \begin{bmatrix} 0.8460 & 0.4828 & 0.2261 \\ 0.4973 & -0.5618 & -0.6611 \\ 0.1922 & -0.6718 & 0.7154 \end{bmatrix} \quad \text{and} \quad \mathbf{D} = \begin{bmatrix} 4.4027 & 0 & 0 \\ 0 & 0.7187 & 0 \\ 0 & 0 & 0.3160 \end{bmatrix}.$$

The columns of $\mathbf{U}$ show the directions of the principal axes of the ellipsoid, and the diagonal elements of $\mathbf{D}$ indicate the relative magnitudes of the principal axes. We see that

the first principal component is given by the first column of $\mathbf{U}$, and the second principal component by the second column of $\mathbf{U}$.

The projection of the point $\boldsymbol{x} = [1.052, 0.6648, 0.2271]^\top$ onto the 1-dimensional space spanned by the first principal component $\boldsymbol{u}_1 = [0.8460, 0.4972, 0.1922]^\top$ is $z = \boldsymbol{u}_1\boldsymbol{u}_1^\top\boldsymbol{x} = [1.0696, 0.6287, 0.2429]^\top$. With respect to the basis vector $\boldsymbol{u}_1$, z is represented by the number $\boldsymbol{u}_1^\top z = 1.2643$. That is, $z = 1.2643\boldsymbol{u}_1$.

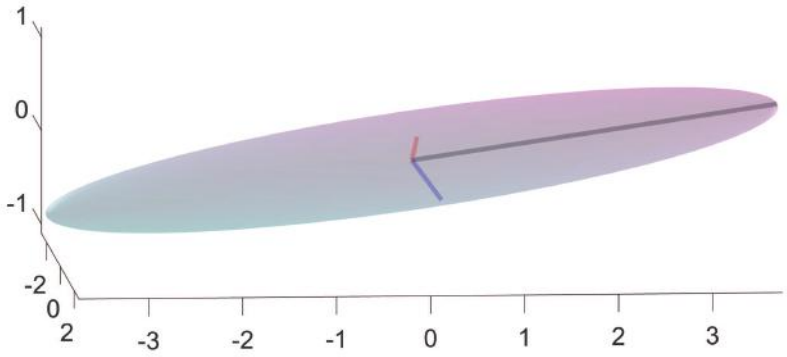


Figure 4.12: A "surfboard" ellipsoid where one principal axis is significantly larger than the other two.

■

4.8.2 PCA and Singular Value Decomposition (SVD)

In the setting above, we did not consider any data set drawn from a multivariate pdf f. The whole analysis rested on linear algebra. In *principal component analysis* (PCA) we start with data $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, where each $\boldsymbol{x}$ is d-dimensional. PCA does not require assumptions how the data were obtained, but to make the link with the previous section, we can think of the data as iid draws from a multivariate normal pdf.

PRINCIPAL COMPONENT ANALYSIS

Let us collect the data in a matrix $\mathbf{X}$ in the usual way; that is,

☞ 43

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1d} \\ x_{21} & x_{22} & \ldots & x_{2d} \\ \vdots & \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & \ldots & x_{nd} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_1^\top \\ \boldsymbol{x}_2^\top \\ \vdots \\ \boldsymbol{x}_n^\top \end{bmatrix}.$$

The matrix $\mathbf{X}$ will be the PCA's input. Under this setting, the data consists of points in d-dimensional space, and our goal is to present the data using n feature vectors of dimension $k < d$.

In accordance with the previous section, we assume that underlying distribution of the data has expectation vector $\mathbf{0}$. In practice, this means that before PCA is applied, the data needs to be *centered* by subtracting the *column* mean in every column:

$$x'_{ij} = x_{ij} - \overline{x}_j,$$

where $\overline{x}_j = \frac{1}{n}\sum_{i=1}^n x_{ij}$.

We assume from now on that the data comes from a general d-dimensional distribution with mean vector $\mathbf{0}$ and some covariance matrix $\boldsymbol{\Sigma}$. The covariance matrix $\boldsymbol{\Sigma}$ is by definition equal to the expectation of the random matrix $\boldsymbol{X}\boldsymbol{X}^\top$, and can be estimated from the data $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ via the sample average

$$\widehat{\boldsymbol{\Sigma}} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i \boldsymbol{x}_i^\top = \frac{1}{n}\mathbf{X}^\top\mathbf{X}.$$

As $\widehat{\boldsymbol{\Sigma}}$ is a covariance matrix, we may conduct the same analysis for $\widehat{\boldsymbol{\Sigma}}$ as we did for $\boldsymbol{\Sigma}$ in the previous section. Specifically, suppose $\widehat{\boldsymbol{\Sigma}} = \mathbf{U}\mathbf{D}^2\mathbf{U}^\top$ is an SVD of $\widehat{\boldsymbol{\Sigma}}$ and let $\mathbf{U}_k$ be the matrix whose columns are the k principal components; that is, the k columns of $\mathbf{U}$ corresponding to the largest diagonal elements in $\mathbf{D}^2$. Note that we have used $\mathbf{D}^2$ instead of $\mathbf{D}$ to be compatible with the previous section. The transformation $\boldsymbol{z} = \mathbf{U}_k\mathbf{U}_k^\top\boldsymbol{x}_i$ maps each vector $\boldsymbol{x}_i \in \mathbb{R}^d$ (thus, with d features) to a vector $\boldsymbol{z}_i \in \mathbb{R}^d$ lying in the subspace spanned by the columns of $\mathbf{U}_k$. With respect to this basis, the point $\boldsymbol{z}_i$ has representation $\boldsymbol{z}_i = \mathbf{U}_k^\top(\mathbf{U}_k\mathbf{U}_k^\top\boldsymbol{x}_i) = \mathbf{U}_k^\top\boldsymbol{x}_i \in \mathbb{R}^k$ (thus with k features). The corresponding covariance matrix of the $\boldsymbol{z}_i, i = 1, \ldots, n$ is diagonal. The diagonal elements $\{d_{\ell\ell}\}$ of $\mathbf{D}$ can be interpreted as standard deviations of the data in the directions of the principal components. The quantity $v = \sum_{\ell=1} d_{\ell\ell}^2$ (that is, the trace of $\mathbf{D}^2$) is thus a measure for the amount of variance in the data. The proportion $d_{\ell\ell}^2/v$ indicates how much of the variance in the data is explained by the ℓ-th principal component.

Another way to look at PCA is by considering the question: How can we best project the data onto a k-dimensional subspace in such a way that the total squared distance between
☞ 362 the projected points and the original points is minimal? From Section A.4, we know that any orthogonal projection to a k-dimensional subspace $\mathcal{V}_k$ can be represented by a matrix $\mathbf{U}_k\mathbf{U}_k^\top$, where $\mathbf{U}_k = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k]$ and the $\{\boldsymbol{u}_\ell, \ell = 1, \ldots, k\}$ are orthogonal vectors of length 1 that span $\mathcal{V}_k$. The above question can thus be formulated as the minimization program:

$$\min_{\boldsymbol{u}_1,\ldots,\boldsymbol{u}_k} \sum_{i=1}^{n} \|\boldsymbol{x}_i - \mathbf{U}_k\mathbf{U}_k^\top\boldsymbol{x}_i\|^2. \tag{4.43}$$

Now observe that

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n} \|\boldsymbol{x}_i - \mathbf{U}_k\mathbf{U}_k^\top\boldsymbol{x}_i\|^2 &= \frac{1}{n}\sum_{i=1}^{n} (\boldsymbol{x}_i^\top - \boldsymbol{x}_i^\top\mathbf{U}_k\mathbf{U}_k^\top)(\boldsymbol{x}_i - \mathbf{U}_k\mathbf{U}_k^\top\boldsymbol{x}_i) \\
&= \underbrace{\frac{1}{n}\sum_{i=1}^{n} \|\boldsymbol{x}_i\|^2}_{c} - \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i^\top\mathbf{U}_k\mathbf{U}_k^\top\boldsymbol{x}_i = c - \frac{1}{n}\sum_{i=1}^{n}\sum_{\ell=1}^{k} \operatorname{tr}(\boldsymbol{x}_i^\top\boldsymbol{u}_\ell\boldsymbol{u}_\ell^\top\boldsymbol{x}_i) \\
&= c - \frac{1}{n}\sum_{\ell=1}^{k}\sum_{i=1}^{n} \boldsymbol{u}_\ell^\top\boldsymbol{x}_i\boldsymbol{x}_i^\top\boldsymbol{u}_\ell = c - \sum_{\ell=1}^{k} \boldsymbol{u}_\ell^\top\widehat{\boldsymbol{\Sigma}}\boldsymbol{u}_\ell,
\end{aligned}$$

☞ 357 where we have used the cyclic property of a trace (Theorem A.1) and the fact that $\mathbf{U}_k\mathbf{U}_k^\top$ can be written as $\sum_{\ell=1}^{k} \boldsymbol{u}_\ell\boldsymbol{u}_\ell^\top$. It follows that the minimization problem(4.43) is equivalent to the maximization problem

$$\max_{\boldsymbol{u}_1,\ldots,\boldsymbol{u}_k} \sum_{\ell=1}^{k} \boldsymbol{u}_\ell^\top\widehat{\boldsymbol{\Sigma}}\boldsymbol{u}_\ell. \tag{4.44}$$

This maximum can be at most $\sum_{\ell=1}^{k} d_{\ell\ell}^2$ and is attained precisely when $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_k$ are the first k principal components of $\widehat{\boldsymbol{\Sigma}}$.

■ **Example 4.11 (Singular Value Decomposition)** The following data set consists of independent samples from the three-dimensional Gaussian distribution with mean vector $\mathbf{0}$ and covariance matrix $\boldsymbol{\Sigma}$ given in Example 4.10:

$$\mathbf{X} = \begin{bmatrix} 3.1209 & 1.7438 & 0.5479 \\ -2.6628 & -1.5310 & -0.2763 \\ 3.7284 & 3.0648 & 1.8451 \\ 0.4203 & 0.3553 & 0.4268 \\ -0.7155 & -0.6871 & -0.1414 \\ 5.8728 & 4.0180 & 1.4541 \\ 4.8163 & 2.4799 & 0.5637 \\ 2.6948 & 1.2384 & 0.1533 \\ -1.1376 & -0.4677 & -0.2219 \\ -1.2452 & -0.9942 & -0.4449 \end{bmatrix}.$$

After replacing $\mathbf{X}$ with its centered version, an SVD $\mathbf{U}\mathbf{D}^2\mathbf{U}^\top$ of $\widehat{\boldsymbol{\Sigma}} = \mathbf{X}^\top\mathbf{X}/n$ yields the principal component matrix $\mathbf{U}$ and diagonal matrix $\mathbf{D}$:

$$\mathbf{U} = \begin{bmatrix} -0.8277 & 0.4613 & 0.3195 \\ -0.5300 & -0.4556 & -0.7152 \\ -0.1843 & -0.7613 & 0.6216 \end{bmatrix} \quad \text{and} \quad \mathbf{D} = \begin{bmatrix} 3.3424 & 0 & 0 \\ 0 & 0.4778 & 0 \\ 0 & 0 & 0.1038 \end{bmatrix}.$$

We also observe that, apart from the sign of the first column, the principal component matrix $\mathbf{U}$ is similar to that in Example 4.10. Likewise for the matrix $\mathbf{D}$. We see that 97.90% of the total variance is explained by the first principal component. Figure 4.13 shows the projection of the centered data onto the subspace spanned by this principal component.

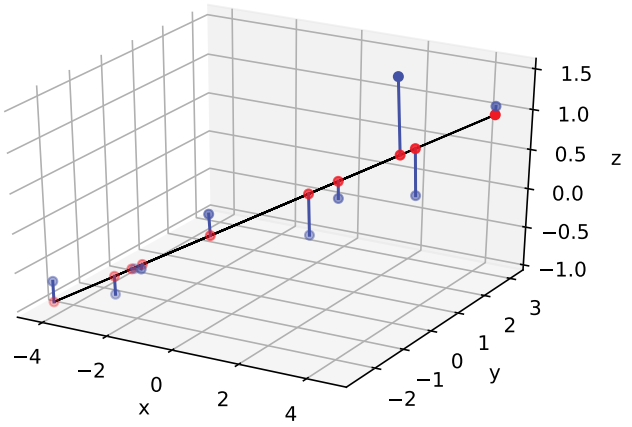


Figure 4.13: Data from the "surfboard" pdf is projected onto the subspace spanned by the largest principal component.

The following Python code was used.