

Data Science BOOKCAMP

Five real-world Python projects

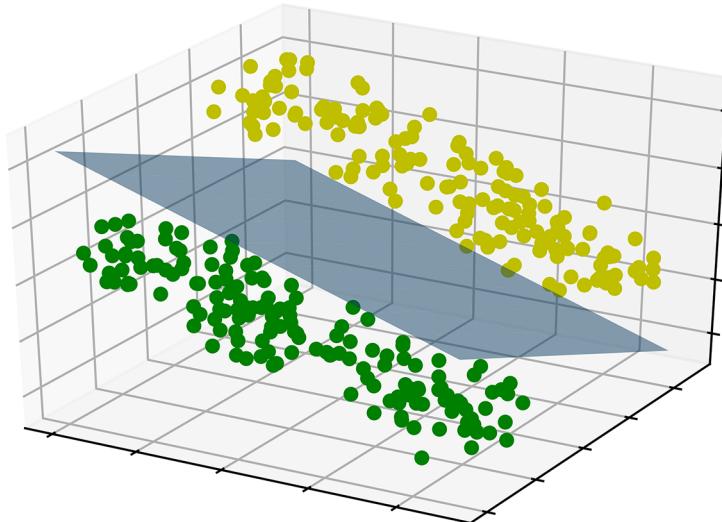
Leonard Apeltsin



MANNING

Core algorithms inside the book

Algorithm	Use case	First introduced
K-means	Clustering	Section 10
DBSCAN	Clustering	Section 10
Jaccard similarity computation	Text comparison	Section 13
Cosine similarity computation	Text comparison	Section 13
Principal component analysis	Dimension reduction	Section 14
Singular value decomposition	Dimension reduction	Section 14
Power iteration	Eigenvector computation	Section 14
TFIDF vectorization	Text comparison	Section 15
Shortest path length computation	Network path optimization	Section 18
PageRank	Network centrality measurement	Section 19
Markov clustering	Social network clustering	Section 19
K-nearest neighbors	Supervised classification	Section 20
Cross-validation	Model performance testing	Section 20
Perceptron	Supervised classification	Section 21
Linear regression	Supervised classification	Section 21
Decision tree	Supervised classification	Section 22
Random forest	Supervised classification	Section 22



A trained logistic regression classifier distinguishes between two classes of points by slicing like a cleaver through 3D space (see section 21).

Data Science Bookcamp

Data Science

Bookcamp

FIVE PYTHON PROJECTS

LEONARD APELTSIN



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2021 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The author and publisher have made every effort to ensure that the information in this book was correct at press time. The author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.

 Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Elesha Hyde
Technical development editors: Arthur Zubarev and Alvin Raj
Review editors: Ivan Martinović and Adriana Sabo
Production editor: Deirdre S. Hiam
Copy editor: Tiffany Taylor
Proofreader: Katie Tennant
Technical proofreader: Raffaella Ventaglio
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781617296253
Printed in the United States of America

*To my teacher, Alexander Vishnevsky,
who taught me how to think*

brief contents

CASE STUDY 1	FINDING THE WINNING STRATEGY IN A CARD GAME	1
1	■ Computing probabilities using Python	3
2	■ Plotting probabilities using Matplotlib	17
3	■ Running random simulations in NumPy	33
4	■ Case study 1 solution	58
CASE STUDY 2	ASSESSING ONLINE AD CLICKS FOR SIGNIFICANCE	69
5	■ Basic probability and statistical analysis using SciPy	71
6	■ Making predictions using the central limit theorem and SciPy	94
7	■ Statistical hypothesis testing	114
8	■ Analyzing tables using Pandas	137
9	■ Case study 2 solution	154

CASE STUDY 3	TRACKING DISEASE OUTBREAKS USING NEWS HEADLINES	165
10	■ Clustering data into groups	167
11	■ Geographic location visualization and analysis	194
12	■ Case study 3 solution	226
CASE STUDY 4	USING ONLINE JOB POSTINGS TO IMPROVE YOUR DATA SCIENCE RESUME	245
13	■ Measuring text similarities	249
14	■ Dimension reduction of matrix data	292
15	■ NLP analysis of large text datasets	340
16	■ Extracting text from web pages	385
17	■ Case study 4 solution	404
CASE STUDY 5	PREDICTING FUTURE FRIENDSHIPS FROM SOCIAL NETWORK DATA	445
18	■ An introduction to graph theory and network analysis	451
19	■ Dynamic graph theory techniques for node ranking and social network analysis	482
20	■ Network-driven supervised machine learning	518
21	■ Training linear classifiers with logistic regression	548
22	■ Training nonlinear classifiers with decision tree techniques	586
23	■ Case study 5 solution	634

contents

<i>preface</i>	xvii
<i>acknowledgments</i>	xix
<i>about this book</i>	xxi
<i>about the author</i>	xxv
<i>about the cover illustration</i>	xxvi

CASE STUDY 1 FINDING THE WINNING STRATEGY IN A CARD GAME1

1	<i>Computing probabilities using Python</i>	3
1.1	Sample space analysis: An equation-free approach for measuring uncertainty in outcomes	4 <i>Analyzing a biased coin</i> 7
1.2	Computing nontrivial probabilities	8 <i>Problem 1: Analyzing a family with four children</i> 8 □ <i>Problem 2:</i> <i>Analyzing multiple die rolls</i> 10 □ <i>Problem 3: Computing die-roll</i> <i>probabilities using weighted sample spaces</i> 11
1.3	Computing probabilities over interval ranges	13 <i>Evaluating extremes using interval analysis</i> 13

2 Plotting probabilities using Matplotlib 17

- 2.1 Basic Matplotlib plots 17
- 2.2 Plotting coin-flip probabilities 22
 - Comparing multiple coin-flip probability distributions* 26

3 Running random simulations in NumPy 33

- 3.1 Simulating random coin flips and die rolls using NumPy 34
 - Analyzing biased coin flips* 36
- 3.2 Computing confidence intervals using histograms and NumPy arrays 38
 - Binning similar points in histogram plots* 41 ▪ *Deriving probabilities from histograms* 43 ▪ *Shrinking the range of a high confidence interval* 46 ▪ *Computing histograms in NumPy* 49
- 3.3 Using confidence intervals to analyze a biased deck of cards 51
- 3.4 Using permutations to shuffle cards 54

4 Case study 1 solution 58

- 4.1 Predicting red cards in a shuffled deck 59
 - Estimating the probability of strategy success* 60
- 4.2 Optimizing strategies using the sample space for a 10-card deck 64

CASE STUDY 2 ASSESSING ONLINE AD CLICKS FOR SIGNIFICANCE.....69

- 4.3 Problem statement 69
- 4.4 Dataset description 70
- 4.5 Overview 70

5 Basic probability and statistical analysis using SciPy 71

- 5.1 Exploring the relationships between data and probability using SciPy 72
- 5.2 Mean as a measure of centrality 76
 - Finding the mean of a probability distribution* 83
- 5.3 Variance as a measure of dispersion 85
 - Finding the variance of a probability distribution* 90

6 Making predictions using the central limit theorem and SciPy 94

- 6.1 Manipulating the normal distribution using SciPy 95
 - Comparing two sampled normal curves* 99
- 6.2 Determining the mean and variance of a population through random sampling 103
- 6.3 Making predictions using the mean and variance 107
 - Computing the area beneath a normal curve* 109 ▪ *Interpreting the computed probability* 112

7 Statistical hypothesis testing 114

- 7.1 Assessing the divergence between sample mean and population mean 115
- 7.2 Data dredging: Coming to false conclusions through oversampling 121
- 7.3 Bootstrapping with replacement: Testing a hypothesis when the population variance is unknown 124
- 7.4 Permutation testing: Comparing means of samples when the population parameters are unknown 132

8 Analyzing tables using Pandas 137

- 8.1 Storing tables using basic Python 138
- 8.2 Exploring tables using Pandas 138
- 8.3 Retrieving table columns 141
- 8.4 Retrieving table rows 143
- 8.5 Modifying table rows and columns 145
- 8.6 Saving and loading table data 148
- 8.7 Visualizing tables using Seaborn 149

9 Case study 2 solution 154

- 9.1 Processing the ad-click table in Pandas 155
- 9.2 Computing p-values from differences in means 157
- 9.3 Determining statistical significance 161
- 9.4 41 shades of blue: A real-life cautionary tale 162

CASE STUDY 3 **TRACKING DISEASE OUTBREAKS USING NEWS HEADLINES** **165**

9.5 Problem statement 165

Dataset description 165

9.6 Overview 166

- 10** *Clustering data into groups* **167**
- 10.1 Using centrality to discover clusters 168
 - 10.2 K-means: A clustering algorithm for grouping data into K central groups 174
 - K-means clustering using scikit-learn* 175 ▪ *Selecting the optimal K using the elbow method* 177
 - 10.3 Using density to discover clusters 181
 - 10.4 DBSCAN: A clustering algorithm for grouping data based on spatial density 185
 - Comparing DBSCAN and K-means* 186 ▪ *Clustering based on non-Euclidean distance* 187
 - 10.5 Analyzing clusters using Pandas 191

- 11** *Geographic location visualization and analysis* **194**
- 11.1 The great-circle distance: A metric for computing the distance between two global points 195
 - 11.2 Plotting maps using Cartopy 198
 - Manually installing GEOS and Cartopy* 199 ▪ *Utilizing the Conda package manager* 199 ▪ *Visualizing maps* 201
 - 11.3 Location tracking using GeoNamesCache 211
 - Accessing country information* 212 ▪ *Accessing city information* 215 ▪ *Limitations of the GeoNamesCache library* 219
 - 11.4 Matching location names in text 221

- 12** *Case study 3 solution* **226**
- 12.1 Extracting locations from headline data 227
 - 12.2 Visualizing and clustering the extracted location data 233
 - 12.3 Extracting insights from location clusters 238

CASE STUDY 4 USING ONLINE JOB POSTINGS TO IMPROVE YOUR DATA SCIENCE RESUME.....245

12.4 Problem statement 245

Dataset description 246

12.5 Overview 247

13 Measuring text similarities 249

13.1 Simple text comparison 250

Exploring the Jaccard similarity 255 ▪ *Replacing words with numeric values* 257

13.2 Vectorizing texts using word counts 262

Using normalization to improve TF vector similarity 264 *Using unit vector dot products to convert between relevance metrics* 272

13.3 Matrix multiplication for efficient similarity calculation 274

Basic matrix operations 277 ▪ *Computing all-by-all matrix similarities* 285

13.4 Computational limits of matrix multiplication 287

14 Dimension reduction of matrix data 292

14.1 Clustering 2D data in one dimension 293

Reducing dimensions using rotation 297

14.2 Dimension reduction using PCA and scikit-learn 309

14.3 Clustering 4D data in two dimensions 315

Limitations of PCA 320

14.4 Computing principal components without rotation 323

Extracting eigenvectors using power iteration 327

14.5 Efficient dimension reduction using SVD and scikit-learn 336

15 NLP analysis of large text datasets 340

15.1 Loading online forum discussions using scikit-learn 341

15.2 Vectorizing documents using scikit-learn 343

15.3 Ranking words by both post frequency and count 350

Computing TFIDF vectors with scikit-learn 356

15.4	Computing similarities across large document datasets	358
15.5	Clustering texts by topic	363
	<i>Exploring a single text cluster</i>	368
15.6	Visualizing text clusters	372
	<i>Using subplots to display multiple word clouds</i>	377

16 *Extracting text from web pages* 385

16.1	The structure of HTML documents	386
16.2	Parsing HTML using BeautifulSoup	394
16.3	Downloading and parsing online data	401

17 *Case study 4 solution* 404

17.1	Extracting skill requirements from job posting data	405
	<i>Exploring the HTML for skill descriptions</i>	406
17.2	Filtering jobs by relevance	412
17.3	Clustering skills in relevant job postings	422
	<i>Grouping the job skills into 15 clusters</i>	425
	<i>Investigating the technical skill clusters</i>	431
	<i>Investigating the soft-skill clusters</i>	434
	<i>Exploring clusters at alternative values of K</i>	436
	<i>Analyzing the 700 most relevant postings</i>	440
17.4	Conclusion	443

CASE STUDY 5 PREDICTING FUTURE FRIENDSHIPS FROM SOCIAL NETWORK DATA.....445

17.5	Problem statement	445
	<i>Introducing the friend-of-a-friend recommendation algorithm</i>	446
	<i>Predicting user behavior</i>	446
17.6	Dataset description	447
	<i>The Profiles table</i>	447
	<i>The Observations table</i>	448
	<i>The Friendships table</i>	449
17.7	Overview	449

18 *An introduction to graph theory and network analysis* 451

18.1	Using basic graph theory to rank websites by popularity	452
	<i>Analyzing web networks using NetworkX</i>	455

- 18.2 Utilizing undirected graphs to optimize the travel time between towns 465
 - Modeling a complex network of towns and counties* 467
 - Computing the fastest travel time between nodes* 473

19 *Dynamic graph theory techniques for node ranking and social network analysis* 482

- 19.1 Uncovering central nodes based on expected traffic in a network 483
 - Measuring centrality using traffic simulations* 486
- 19.2 Computing travel probabilities using matrix multiplication 489
 - Deriving PageRank centrality from probability theory* 492
 - Computing PageRank centrality using NetworkX* 496
- 19.3 Community detection using Markov clustering 498
- 19.4 Uncovering friend groups in social networks 513

20 *Network-driven supervised machine learning* 518

- 20.1 The basics of supervised machine learning 519
- 20.2 Measuring predicted label accuracy 527
 - Scikit-learn's prediction measurement functions* 536
- 20.3 Optimizing KNN performance 537
- 20.4 Running a grid search using scikit-learn 539
- 20.5 Limitations of the KNN algorithm 544

21 *Training linear classifiers with logistic regression* 548

- 21.1 Linearly separating customers by size 549
- 21.2 Training a linear classifier 554
 - Improving perceptron performance through standardization* 562
- 21.3 Improving linear classification with logistic regression 565
 - Running logistic regression on more than two features* 572
- 21.4 Training linear classifiers using scikit-learn 574
 - Training multiclass linear models* 576
- 21.5 Measuring feature importance with coefficients 579
- 21.6 Linear classifier limitations 582

22 *Training nonlinear classifiers with decision tree techniques* 586

- 22.1 Automated learning of logical rules 587
 - Training a nested if/else model using two features* 593 ▪ *Deciding which feature to split on* 599 ▪ *Training if/else models with more than two features* 608
- 22.2 Training decision tree classifiers using scikit-learn 614
 - Studying cancerous cells using feature importance* 621
- 22.3 Decision tree classifier limitations 624
- 22.4 Improving performance using random forest classification 626
- 22.5 Training random forest classifiers using scikit-learn 630

23 *Case study 5 solution* 634

- 23.1 Exploring the data 635
 - Examining the profiles* 635 ▪ *Exploring the experimental observations* 638 ▪ *Exploring the Friendships linkage table* 641
- 23.2 Training a predictive model using network features 645
- 23.3 Adding profile features to the model 652
- 23.4 Optimizing performance across a steady set of features 657
- 23.5 Interpreting the trained model 659
 - Why are generalizable models so important?* 662

index 665

preface

Another promising candidate had failed their data science interview, and I began to wonder why. The year was 2018, and I was struggling to expand the data science team at my startup. I had interviewed dozens of seemingly qualified candidates, only to reject them all. The latest rejected applicant was an economics PhD from a top-notch school. Recently, the applicant had transitioned into data science after completing a 10-week bootcamp. I asked the applicant to discuss an analytics problem that was very relevant to our company. They immediately brought up a trendy algorithm that was not applicable to the situation. When I tried to debate the algorithm's incompatibilities, the candidate was at a loss. They didn't know how the algorithm actually worked or the appropriate circumstances under which to use it. These details hadn't been taught to them at the bootcamp.

After the rejected candidate departed, I began to reflect on my own data science education. How different it had been! Back in 2006, data science was not yet a coveted career choice, and DS bootcamps did not yet exist. In those days, I was a poor grad student struggling to pay the rent in pricey San Francisco. My graduate research required me to analyze millions of genetic links to diseases. I realized that my skills were transferable to other areas of analysis, and thus my data science consultancy was born.

Unbeknownst to my graduate advisor, I began to solicit analytics work from random Bay Area companies. That freelance work helped pay the bills, so I could not be too choosy about the data-driven assignments I tackled. Thus, I would sign up for a variety of data science tasks, ranging from simple statistical analyses to complex predictive modeling. Sometimes I would find myself overwhelmed by a seemingly intractable

data problem, but in the end, I'd persevere. My struggles taught me the nuances of diverse analytics techniques and how to best combine them to reach elegant solutions. More importantly, I learned how common techniques can fail and how to surmount these failure points to deliver impactful results. As my skill set grew, my data science career began to flourish. Eventually, I became a leader in the field.

Would I have achieved the same level of success through rote memorization at a 10-week bootcamp? Probably not. Many bootcamps prioritize the study of standalone algorithms over more cohesive problem-solving skills. Furthermore, the hype over an algorithm's strengths tends to be emphasized over its weaknesses. Consequently, students are sometimes ill prepared to handle data science in real-world settings. That insight inspired me to write this book.

I decided to replicate my own data science education by exposing you, my readers, to a set of increasingly challenging analytics problems. Additionally, I chose to arm you with tools and techniques required to handle these problems effectively. My aim is to holistically help you cultivate your analytic problem-solving skills. This way, when you interview for that junior data science position, you will be much more likely to get the job.

acknowledgments

Writing this book was very hard. I definitely could not have done it alone. Fortunately, my family and friends provided their support during this arduous journey. First and foremost, I thank my mother, Irina Apeltsin. She kept me motivated during those difficult days when the task before me seemed insurmountable. Additionally, I thank my grandmother, Vera Fisher, whose pragmatic advice kept me on track as I plowed through the material for my book.

Furthermore, I'd like to thank my childhood friend Vadim Stolnik. Vadim is a brilliant graphic designer who helped me with the book's myriad illustrations. Also, I want to acknowledge my friend and colleague Emmanuel Yera, who had my back during my initial writing efforts. Moreover, I must mention my dear dance partner Alexandria Law, who kept my spirits up during my struggles and also helped pick out this book's cover.

Next, I thank my editor at Manning, Elesha Hyde. Over the course of the past three years, you've worked tirelessly to ensure that I deliver something truly of value to my readers. I will forever be grateful for your patience, optimism, and ceaseless commitment to quality. You've pushed me to become a better writer, and my readers will ultimately benefit from these efforts. Additionally, I'd like to acknowledge my technical development editor Arthur Zubarov and my technical proofreader Rafaela Ventaglio. Your inputs helped me craft a better, cleaner book. I also thank Deirdre Hiam, my project editor; Tiffany Taylor, my copyeditor; Katie Tennant, my proofreader; and everyone else at Manning who had a hand in this book.

To all the reviewers—Adam Scheller, Adriaan Beiertz, Alan Bogusiewicz, Amaresh Rajasekharan, Ayon Roy, Bill Mitchell, Bob Quintus, David Jacobs, Diego Casella, Duncan McRae, Elias Rangel, Frank L Quintana, Grzegorz Bernas, Jason Hales, Jean-François Morin, Jeff Smith, Jim Amrhein, Joe Justesen, John Kasiewicz, Maxim Kupfer, Michael Johnson, Michał Ambroziewicz, Raffaella Ventaglio, Ravi Sajnani, Robert Diana, Simone Sguazza, Sriram Macharla, and Stuart Woodward—thank you. Your suggestions helped make this a better book.

about this book

Open-ended problem-solving abilities are essential for a data science career. Unfortunately, these abilities cannot be acquired simply by reading. To become a problem solver, you must persistently solve difficult problems. With this in mind, I've structured my book around case studies: open-ended problems modeled on real-world situations. The case studies range from online advertisement analysis to tracking disease outbreaks using news data. Upon completing these case studies, you will be well suited to begin a career in data science.

Who should read this book

This book's intended reader is an educated novice who is interested in transitioning to a data science career. When I imagine a typical reader, I picture a fourth-year college student studying economics who wishes to explore a broader range of analytics opportunities, or a chemistry major already out of school who is searching for a more data-centric career path. Or perhaps the reader is a successful frontend web developer with a very limited mathematics background who would like to give data science a shot. None of my potential readers have ever taken a data science class, leaving them inexperienced when it comes to diverse data analysis. The purpose of this book is to eliminate that skill deficiency.

My readers are required to know the bare-bones basics of Python programming. Self-taught beginning Python should be sufficient to explore the exercises in the book. Your mathematical knowledge is not expected to extend beyond basic high-school trigonometry.

How this book is organized

This book contains five case studies of progressing difficulty. Each case study begins with a detailed problem statement, which you will need to resolve. The problem statement is followed by two to five sections that introduce the data science skills required to solve the problem. These skill sections cover fundamental libraries, as well as mathematical and algorithmic techniques. Each final case study section describes the solution to the problem.

Case study 1 pertains to basic probability theory:

- Section 1 discusses how to compute probabilities using straightforward Python.
- Section 2 introduces the concept of probability distributions. It also introduces the Matplotlib visualization library, which can be used to visualize the distributions.
- Section 3 discusses how to estimate probabilities using randomized simulations. The NumPy numerical computing library is introduced to facilitate efficient simulation execution.
- Section 4 contains the case study solution.

Case study 2 extends beyond probability into statistics:

- Section 5 introduces simple statistical measures of centrality and dispersion. It also introduces the SciPy scientific computing library, which contains a useful statistics module.
- Section 6 dives deep into the central limit theorem, which can be used to make statistical predictions.
- Section 7 discusses various statistical inference techniques, which can be used to distinguish interesting data patterns from random noise. Additionally, this section illustrates the dangers of incorrect inference usage and how these dangers can be best avoided.
- Section 8 introduces the Pandas library, which can be utilized to preprocess tabular data before statistical analysis.
- Section 9 contains the case study solution.

Case study 3 focuses on the unsupervised clustering of geographic data:

- Section 10 illustrates how measures of centrality can be used to cluster data into groups. The scikit-learn library is also introduced to facilitate efficient clustering.
- Section 11 focuses on geographic data extraction and visualization. Extraction from text is carried out with the GeoNamesCache library, while visualization is achieved using the Cartopy map-plotting library.
- Section 12 contains the case study solution.

Case study 4 focuses on natural language processing using large-scale numeric computations:

- Section 13 illustrates how to efficiently compute similarities between texts using matrix multiplication. NumPy's built-in matrix optimizations are used extensively for this purpose.
- Section 14 shows how to utilize dimension reduction for more efficient matrix analysis. Mathematical theory is discussed in conjunction with scikit-learn's dimension-reduction methods.
- Section 15 applies natural language processing techniques to a very large text dataset. The section discusses how to best explore and cluster that text data.
- Section 16 shows how to extract text from online data using the Beautiful Soup HTML-parsing library.
- Section 17 contains the case study solution.

Case study 5 completes the book with a discussion of network theory and supervised machine learning:

- Section 18 introduces basic network theory in conjunction with the NetworkX graph analysis library.
- Section 19 shows how to utilize network flow to find clusters in network data. Probabilistic simulations and matrix multiplications are used to achieve effective clustering.
- Section 20 introduces a simple supervised machine learning algorithm based on network theory. Common machine learning evaluation techniques are also illustrated using scikit-learn.
- Section 21 discusses additional machine learning techniques, which rely on memory-efficient linear classifiers.
- Section 22 dives into the flaws of previously introduced supervised learning methodologies. The flaws are subsequently circumvented using nonlinear decision tree classifiers.
- Section 23 contains the case study solution.

Each section of the book builds on the algorithms and libraries introduced in previous sections. Hence, you are encouraged to go through this book cover to cover to minimize confusion. But if you are already familiar with a subset of the material in the book, feel free to skip that familiar material. Finally, I strongly recommend that you tackle each case study problem on your own before reading the solution. Independently trying to solve each problem will maximize the value of this book.

About the code

This book contains many examples of source code, both in numbered listings and inline with normal text. In both cases, the source code is formatted in a fixed-width font like this to separate it from ordinary text. The source code in the listings is

structured in modular chunks, with written explanations that precede each modular bit of code. That code presentation style is well suited for display in a Jupyter notebook since notebooks bridge functional code samples with written explanations. Consequently, the source code for each case study is available for download in a Jupyter notebook at www.manning.com/books/data-science-bookcamp. These notebooks combine code listings with summarized explanations from the book. Per usual notebook style, interdependencies exist between separate notebook cells. Thus, it's recommended that you run the code samples in the exact order they appear in the notebook: otherwise you risk encountering a dependency-driven error.

about the author

LEONARD APELTSIN is the head of data science at Anomaly. His team applies advanced analytics to uncover healthcare fraud, waste, and abuse. Prior to Anomaly, Leonard led the machine learning development efforts at Primer AI, a startup that specializes in natural language processing. As a founding member, Leonard helped grow the Primer AI team from 4 to nearly 100 employees. Before venturing into startups, Leonard worked in academia, uncovering hidden patterns in genetically linked diseases. His discoveries have been published in the subsidiaries of the journals *Science* and *Nature*. Leonard holds BS degrees in biology and computer science from Carnegie Mellon University and a PhD in bioinformatics from The University of California, San Francisco.

about the cover illustration

The figure on the cover of *Data Science Bookcamp* is captioned “Habitante du Tyrol,” or resident of Tyrol. The illustration is taken from a collection of dress costumes from various countries by Jacques Grasset de Saint-Sauveur (1757–1810), titled *Costumes de Différents Pays*, published in France in 1797. Each illustration is finely drawn and colored by hand. The rich variety of Grasset de Saint-Sauveur’s collection reminds us vividly of how culturally apart the world’s towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. On the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

The way we dress has changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns, regions, or countries. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Grasset de Saint-Sauveur’s pictures.

Case study 1

Finding the winning strategy in a card game

Problem statement

Would you like to win a bit of money? Let's wager on a card game for minor stakes. In front of you is a shuffled deck of cards. All 52 cards lie face down. Half the cards are red, and half are black. I will proceed to flip over the cards one by one. If the last card I flip over is red, you'll win a dollar. Otherwise, you'll lose a dollar.

Here's the twist: you can ask me to halt the game at any time. Once you say "Halt," I will flip over the next card and end the game. That next card will serve as the final card. You will win a dollar if it's red, as shown in figure CS1.1.

We can play the game as many times as you like. The deck will be reshuffled every time. After each round, we'll exchange money. What is your best approach to winning this game?

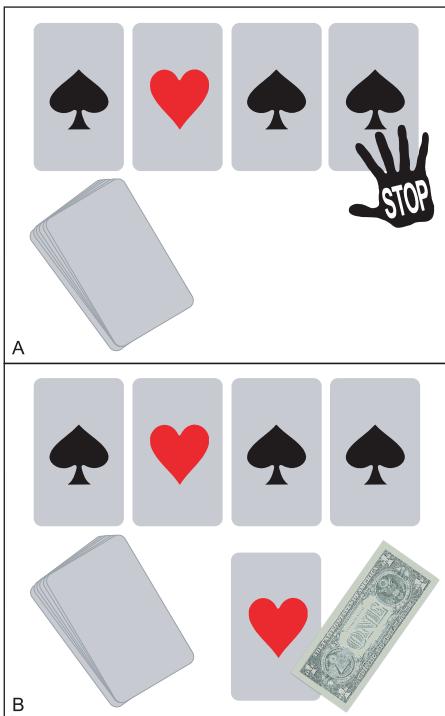


Figure CS1.1 The card-flipping game. We start with a shuffled deck. I repeatedly flip over the top card from the deck. (A) I have just flipped the fourth card. You instruct me to stop. (B) I flip over the fifth and final card. The final card is red. You win a dollar.

Overview

To address the problem at hand, we will need to know how to

- 1 Compute the probabilities of observable events using sample space analysis.
- 2 Plot the probabilities of events across a range of interval values.
- 3 Simulate random processes, such as coin flips and card shuffling, using Python.
- 4 Evaluate our confidence in decisions drawn from simulations using confidence interval analysis.

1

Computing probabilities using Python

This section covers

- What are the basics of probability theory?
- Computing probabilities of a single observation
- Computing probabilities across a range of observations

Few things in life are certain; most things are driven by chance. Whenever we cheer for our favorite sports team, or purchase a lottery ticket, or make an investment in the stock market, we hope for some particular outcome, but that outcome cannot ever be guaranteed. Randomness permeates our day-to-day experiences. Fortunately, that randomness can still be mitigated and controlled. We know that some unpredictable events occur more rarely than others and that certain decisions carry less uncertainty than other much-riskier choices. Driving to work in a car is safer than riding a motorcycle. Investing part of your savings in a retirement account is safer than betting it all on a single hand of blackjack. We can intrinsically sense these trade-offs in certainty because even the most unpredictable systems still show some predictable behaviors. These behaviors have been rigorously studied using *probability theory*. Probability theory is an inherently complex branch of math. However, aspects of the theory can be understood without knowing the mathematical

underpinnings. In fact, difficult probability problems can be solved in Python without needing to know a single math equation. Such an equation-free approach to probability requires a baseline understanding of what mathematicians call a *sample space*.

1.1 Sample space analysis: An equation-free approach for measuring uncertainty in outcomes

Certain actions have measurable outcomes. A *sample space* is the set of all the possible outcomes an action could produce. Let's take the simple action of flipping a coin. The coin will land on either heads or tails. Thus, the coin flip will produce one of two measurable outcomes: *heads* or *tails*. By storing these outcomes in a Python set, we can create a sample space of coin flips.

Listing 1.1 Creating a sample space of coin flips

```
sample_space = {'Heads', 'Tails'} ← Storing elements in curly brackets creates a Python set. A Python set is a collection of unique, unordered elements.
```

Suppose we choose an element of `sample_space` at random. What fraction of the time will the chosen element equal `Heads`? Well, our sample space holds two possible elements. Each element occupies an equal fraction of the space within the set. Therefore, we expect `Heads` to be selected with a frequency of $1/2$. That frequency is formally defined as the *probability* of an outcome. All outcomes within `sample_space` share an identical probability, which is equal to $1 / \text{len}(\text{sample_space})$.

Listing 1.2 Computing the probability of heads

```
probability_heads = 1 / len(sample_space)
print(f'Probability of choosing heads is {probability_heads}')
```

Probability of choosing heads is 0.5

The probability of choosing `Heads` equals 0.5. This relates directly to the action of flipping a coin. We'll assume the coin is unbiased, which means the coin is equally likely to fall on either heads or tails. Thus, a coin flip is conceptually equivalent to choosing a random element from `sample_space`. The probability of the coin landing on heads is therefore 0.5; the probability of it landing on tails is also equal to 0.5.

We've assigned probabilities to our two measurable outcomes. However, there are additional questions we could ask. What is the probability that the coin lands on either heads or tails? Or, more exotically, what is the probability that the coin will spin forever in the air, landing on neither heads nor tails? To find rigorous answers, we need to define the concept of an *event*. An event is the subset of those elements within `sample_space` that satisfy some *event condition* (as shown in figure 1.1). An event condition is a simple Boolean function whose input is a single `sample_space` element. The function returns `True` only if the element satisfies our condition constraints.

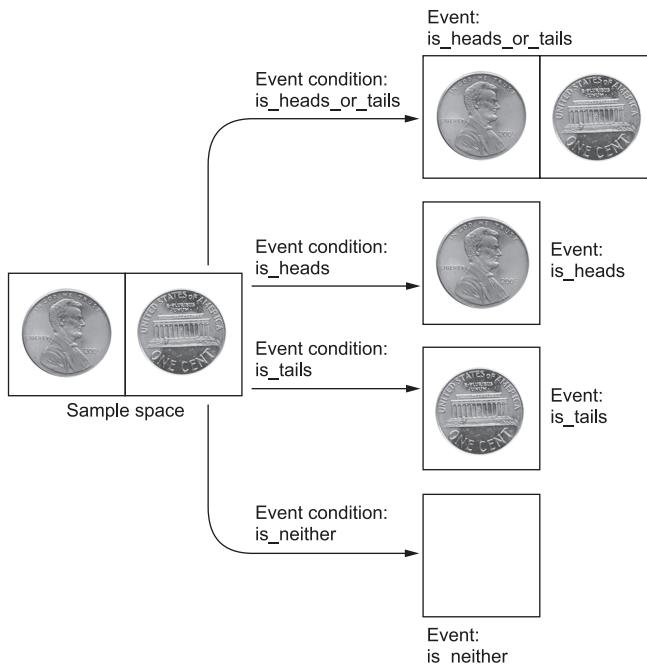


Figure 1.1 Four event conditions applied to a sample space. The sample space contains two outcomes: heads and tails. Arrows represent the event conditions. Every event condition is a yes-or-no function. Each function filters out those outcomes that do not satisfy its terms. The remaining outcomes form an event. Each event contains a subset of the outcomes found in the sample space. Four events are possible: heads, tails, heads or tails, and neither heads nor tails.

Let's define two event conditions: one where the coin lands on either heads or tails, and another where the coin lands on neither heads nor tails.

Listing 1.3 Defining event conditions

```
def is_heads_or_tails(outcome): return outcome in {'Heads', 'Tails'}
def is_neither(outcome): return not is_heads_or_tails(outcome)
```

Also, for the sake of completeness, let's define event conditions for the two basic events in which the coin satisfies exactly one of our two potential outcomes.

Listing 1.4 Defining additional event conditions

```
def is_heads(outcome): return outcome == 'Heads'
def is_tails(outcome): return outcome == 'Tails'
```

We can pass event conditions into a generalized `get_matching_event` function. That function is defined in listing 1.5. Its inputs are an event condition and a generic sample

space. The function iterates through the generic sample space and returns the set of outcomes where `event_condition(outcome)` is True.

Listing 1.5 Defining an event-detection function

```
def get_matching_event(event_condition, sample_space):
    return set([outcome for outcome in sample_space
               if event_condition(outcome)])
```

Let's execute `get_matching_event` on our four event conditions. Then we'll output the four extracted events.

Listing 1.6 Detecting events using event conditions

```
event_conditions = [is_heads_or_tails, is_heads, is_tails, is_neither]

for event_condition in event_conditions:
    print(f"Event Condition: {event_condition.__name__}")      ← Prints the name of an event_condition function
    event = get_matching_event(event_condition, sample_space)
    print(f'Event: {event}\n')

Event Condition: is_heads_or_tails
Event: {'Tails', 'Heads'}

Event Condition: is_heads
Event: {'Heads'}

Event Condition: is_tails
Event: {'Tails'}

Event Condition: is_neither
Event: set()
```

We've successfully extracted four events from `sample_space`. What is the probability of each event occurring? Earlier, we showed that the probability of a single-element outcome for a fair coin is $1 / \text{len}(\text{sample_space})$. This property can be generalized to include multi-element events. The probability of an event is equal to $\text{len}(\text{event}) / \text{len}(\text{sample_space})$, but only if all outcomes are known to occur with equal likelihood. In other words, the probability of a multi-element event for a fair coin is equal to the event size divided by the sample space size. We now use event size to compute the four event probabilities.

Listing 1.7 Computing event probabilities

```
def compute_probability(event_condition, generic_sample_space):
    event = get_matching_event(event_condition, generic_sample_space)      ←
    return len(event) / len(generic_sample_space)                            ←

Probability is equal to event size divided by sample space size.          The compute_probability function extracts the event associated with an inputted event condition to compute its probability.
```

```

for event_condition in event_conditions:
    prob = compute_probability(event_condition, sample_space)
    name = event_condition.__name__
    print(f"Probability of event arising from '{name}' is {prob}")

Probability of event arising from 'is_heads_or_tails' is 1.0
Probability of event arising from 'is_heads' is 0.5
Probability of event arising from 'is_tails' is 0.5
Probability of event arising from 'is_neither' is 0.0

```

The executed code outputs a diverse range of event probabilities, the smallest of which is 0.0 and the largest of which is 1.0. These values represent the lower and upper bounds of probability; no probability can ever fall below 0.0 or rise above 1.0.

1.1.1 Analyzing a biased coin

We computed probabilities for an unbiased coin. What would happen if that coin was biased? Suppose, for instance, that a coin is four times more likely to land on heads relative to tails. How do we compute the likelihoods of outcomes that are not weighted in an equal manner? Well, we can construct a weighted sample space represented by a Python dictionary. Each outcome is treated as a key whose value maps to the associated weight. In our example, Heads is weighted four times as heavily as Tails, so we map Tails to 1 and Heads to 4.

Listing 1.8 Representing a weighted sample space

```
weighted_sample_space = {'Heads': 4, 'Tails': 1}
```

Our new sample space is stored in a dictionary. This allows us to redefine the size of the sample space as the sum of all dictionary weights. Within `weighted_sample_space`, that sum will equal 5.

Listing 1.9 Checking the weighted sample space size

```
sample_space_size = sum(weighted_sample_space.values())
assert sample_space_size == 5
```

We can redefine event size in a similar manner. Each event is a set of outcomes, and those outcomes map to weights. Summing over the weights yields the event size. Thus, the size of the event satisfying the `is_heads_or_tails` event condition is also 5.

Listing 1.10 Checking the weighted event size

```
event = get_matching_event(is_heads_or_tails, weighted_sample_space)
event_size = sum(weighted_sample_space[outcome] for outcome in event)
assert event_size == 5
```

As a reminder, this function iterates over each outcome in the inputted sample space. Thus, it will work as expected on our dictionary input. This is because Python iterates over dictionary keys, not key-value pairs as in many other popular programming languages.

Our generalized definitions of sample space size and event size permit us to create a `compute_event_probability` function. The function takes as input a `generic_sample_space` variable that can be either a weighted dictionary or an unweighted set.

Listing 1.11 Defining a generalized event probability function

```
def compute_event_probability(event_condition, generic_sample_space):
    event = get_matching_event(event_condition, generic_sample_space)
    if type(generic_sample_space) == type(set()):           ← Checks whether
        return len(event) / len(generic_sample_space)          generic_event_space
    event_size = sum(generic_sample_space[outcome]           is a set
                    for outcome in event)
    return event_size / sum(generic_sample_space.values())
```

We can now output all the event probabilities for the biased coin without needing to redefine our four event condition functions.

Listing 1.12 Computing weighted event probabilities

```
for event_condition in event_conditions:
    prob = compute_event_probability(event_condition, weighted_sample_space)
    name = event_condition.__name__
    print(f"Probability of event arising from '{name}' is {prob}")

Probability of event arising from 'is_heads' is 0.8
Probability of event arising from 'is_tails' is 0.2
Probability of event arising from 'is_heads_or_tails' is 1.0
Probability of event arising from 'is_neither' is 0.0
```

With just a few lines of code, we have constructed a tool for solving many problems in probability. Let's apply this tool to problems more complex than a simple coin flip.

1.2 Computing nontrivial probabilities

We'll now solve several example problems using `compute_event_probability`.

1.2.1 Problem 1: Analyzing a family with four children

Suppose a family has four children. What is the probability that exactly two of the children are boys? We'll assume that each child is equally likely to be either a boy or a girl. Thus we can construct an unweighted sample space where each outcome represents one possible sequence of four children, as shown in figure 1.2.

B B B B	B G B G
B B B G	G B G B
B B G B	B G G B
B G B B	B G G G
G B B B	G G G B
G G B B	G G G G
G B B G	G B G G
G B G G	G G B G
B B G G	G G G G

Figure 1.2 The sample space for four sibling children. Each row in the sample space contains 1 of 16 possible outcomes. Every outcome represents a unique combination of four children. The sex of each child is indicated by a letter: B for boy and G for girl. Outcomes with two boys are marked by an arrow. There are six such arrows; thus, the probability of two boys equals 6 / 16.

Listing 1.13 Computing the sample space of children

```
possible_children = ['Boy', 'Girl']
sample_space = set()
for child1 in possible_children:
    for child2 in possible_children:
        for child3 in possible_children:
            for child4 in possible_children:
                outcome = (child1, child2, child3, child4)
                sample_space.add(outcome)
```

Each possible sequence of four children is represented by a four-element tuple.

We ran four nested for loops to explore the sequence of four births. This is not an efficient use of code. We can more easily generate our sample space using Python's built-in `itertools.product` function, which returns all pairwise combinations of all elements across all input lists. Next, we input four instances of the `possible_children` list into `itertools.product`. The product function then iterates over all four instances of the list, computing all the combinations of list elements. The final output equals our sample space.

Listing 1.14 Computing the sample space using product

The * operator unpacks multiple arguments stored within a list. These arguments are then passed into a specified function. Thus, calling `product(*[possible_children])` is equivalent to calling `product(possible_children, possible_children, possible_children, possible_children)`.

```
from itertools import product
all_combinations = product(*[4 * [possible_children]])
```

Note that after running this line, `all_combinations` will be empty. This is because `product` returns a Python iterator, which can be iterated over only once. For us, this isn't an issue. We are about to compute the sample space even more efficiently, and `all_combinations` will not be used in future code.

We can make our code even more efficient by executing `set(product(possible_children, repeat=4))`. In general, running `product(possible_children, repeat=n)` returns an iterable over all possible combinations of n children.

Listing 1.15 Passing repeat into product

```
sample_space_efficient = set(product(possible_children, repeat=4))
assert sample_space == sample_space_efficient
```

Let's calculate the fraction of `sample_space` that is composed of families with two boys. We define a `has_two_boys` event condition and then pass that condition into `compute_event_probability`.

Listing 1.16 Computing the probability of two boys

```
def has_two_boys(outcome): return len([child for child in outcome
                                         if child == 'Boy']) == 2
```

```
prob = compute_event_probability(has_two_boys, sample_space)
print(f"Probability of 2 boys is {prob}")

Probability of 2 boys is 0.375
```

The probability of exactly two boys being born in a family of four children is 0.375. By implication, we expect 37.5% of families with four children to contain an equal number of boys and girls. Of course, the actual observed percentage of families with two boys will vary due to random chance.

1.2.2 Problem 2: Analyzing multiple die rolls

Suppose we're shown a fair six-sided die whose faces are numbered from 1 to 6. The die is rolled six times. What is the probability that these six die rolls add up to 21?

We begin by defining the possible values of any single roll. These are integers that range from 1 to 6.

Listing 1.17 Defining all possible rolls of a six-sided die

```
possible_rolls = list(range(1, 7))
print(possible_rolls)

[1, 2, 3, 4, 5, 6]
```

Next, we create the sample space for six consecutive rolls using the product function.

Listing 1.18 Sample space for six consecutive die rolls

```
sample_space = set(product(possible_rolls, repeat=6))
```

Finally, we define a has_sum_of_21 event condition that we'll subsequently pass into compute_event_probability.

Listing 1.19 Computing the probability of a die-roll sum

```
def has_sum_of_21(outcome): return sum(outcome) == 21
prob = compute_event_probability(has_sum_of_21, sample_space)
print(f"6 rolls sum to 21 with a probability of {prob}")
```

Conceptually, rolling a single die six times is equivalent to rolling six dice simultaneously.

```
6 rolls sum to 21 with a probability of 0.09284979423868313
```

The six die rolls will sum to 21 more than 9% of the time. Note that our analysis can be coded more concisely using a lambda expression. *Lambda expressions* are one-line anonymous functions that do not require a name. In this book, we use lambda expressions to pass short functions into other functions.

Listing 1.20 Computing the probability using a lambda expression

```
prob = compute_event_probability(lambda x: sum(x) == 21, sample_space)    ←
assert prob == compute_event_probability(has_sum_of_21, sample_space)
```

Lambda expressions allow us to define short functions in a single line of code. Coding `lambda x:` is functionally equivalent to coding `func(x):`. Thus, `lambda x: sum(x) == 21` is functionally equivalent to `has_sum_of_21`.

1.2.3 Problem 3: Computing die-roll probabilities using weighted sample spaces

We've just computed the likelihood of six die rolls summing to 21. Now, let's recompute that probability using a weighted sample space. We need to convert our unweighted sample space set into a weighted sample space dictionary; this will require us to identify all possible die-roll sums. Then we must count the number of times each sum appears across all possible die-roll combinations. These combinations are already stored in our computed `sample_space` set. By mapping the die-roll sums to their occurrence counts, we will produce a `weighted_sample_space` result.

Listing 1.21 Mapping die-roll sums to occurrence counts

```
from collections import defaultdict    ←
weighted_sample_space = defaultdict(int)    ←
for outcome in sample_space:    ←
    total = sum(outcome)    ←
    weighted_sample_space[total] += 1    ←
        Updates the occurrence count for a summed dice value
```

This module returns dictionaries whose keys are all assigned a default value. For instance, `defaultdict(int)` returns a dictionary where the default value for each key is set to zero.

The `weighted_sample` dictionary maps each summed six-die-roll combination to its occurrence count.

Each outcome contains a unique combination of six die rolls.

Computes the summed value of six unique die rolls

Before we recompute our probability, let's briefly explore the properties of `weighted_sample_space`. Not all weights in the sample space are equal—some of the weights are much smaller than others. For instance, there is only one way for the rolls to sum to 6: we must roll precisely six 1s to achieve that dice-sum combination. Hence, we expect `weighted_sample_space[6]` to equal 1. We expect `weighted_sample_space[36]` to also equal 1, since we must roll six 6s to achieve a sum of 36.

Listing 1.22 Checking very rare die-roll combinations

```
assert weighted_sample_space[6] == 1
assert weighted_sample_space[36] == 1
```

Meanwhile, the value of `weighted_sample_space[21]` is noticeably higher.

Listing 1.23 Checking a more common die-roll combination

```
num_combinations = weighted_sample_space[21]
print(f"There are {num_combinations} ways for 6 die rolls to sum to 21")
```

There are 4332 ways for 6 die rolls to sum to 21

As the output shows, there are 4,332 ways for six die rolls to sum to 21. For example, we could roll four 4s, followed by a 3 and then a 2. Or we could roll three 4s followed by a 5, a 3, and a 1. Thousands of other combinations are possible. This is why a sum of 21 is much more probable than a sum of 6.

Listing 1.24 Exploring different ways of summing to 21

```
assert sum([4, 4, 4, 4, 3, 2]) == 21
assert sum([4, 4, 4, 5, 3, 1]) == 21
```

Note that the observed count of 4,332 is equal to the length of an unweighted event whose die rolls add up to 21. Also, the sum of values in `weighted_sample` is equal to the length of `sample_space`. Hence, a direct link exists between unweighted and weighted event probability computation.

Listing 1.25 Comparing weighted events and regular events

```
event = get_matching_event(lambda x: sum(x) == 21, sample_space)
assert weighted_sample_space[21] == len(event)
assert sum(weighted_sample_space.values()) == len(sample_space)
```

Let's now recompute the probability using the `weighted_sample_space` dictionary. The final probability of rolling a 21 should remain unchanged.

Listing 1.26 Computing the weighted event probability of die rolls

```
prob = compute_event_probability(lambda x: x == 21,
                                 weighted_sample_space)
assert prob == compute_event_probability(has_sum_of_21, sample_space)
print(f"6 rolls sum to 21 with a probability of {prob}")

6 rolls sum to 21 with a probability of 0.09284979423868313
```

What is the benefit of using a weighted sample space over an unweighted one? Less memory usage! As we see next, the unweighted `sample_space` set has on the order of 150 times more elements than the weighted sample space dictionary.

Listing 1.27 Comparing weighted to unweighted event space size

```
print('Number of Elements in Unweighted Sample Space:')
print(len(sample_space))
print('Number of Elements in Weighted Sample Space:')
print(len(weighted_sample_space))
```

```
Number of Elements in Unweighted Sample Space:  
46656  
Number of Elements in Weighted Sample Space:  
31
```

1.3 Computing probabilities over interval ranges

So far, we've only analyzed event conditions that satisfy some single value. Now we'll analyze event conditions that span intervals of values. An *interval* is the set of all the numbers between and including two boundary cutoffs. Let's define an `is_in_interval` function that checks whether a number falls within a specified interval. We'll control the interval boundaries by passing a `minimum` and a `maximum` parameter.

Listing 1.28 Defining an interval function

Defines a closed interval in which the min/max boundaries are included. However, it's also possible to define open intervals when needed. In open intervals, at least one of the boundaries is excluded.

```
def is_in_interval(number, minimum, maximum):  
    return minimum <= number <= maximum
```

Given the `is_in_interval` function, we can compute the probability that an event's associated value falls within some numeric range. For instance, let's compute the likelihood that our six consecutive die rolls sum to a value between 10 and 21 (inclusive).

Listing 1.29 Computing the probability over an interval

```
prob = compute_event_probability(lambda x: is_in_interval(x, 10, 21), ←  
                                 weighted_sample_space)  
print(f"Probability of interval is {prob}")  
Probability of interval is 0.5446244855967078
```

Lambda function that takes some input x and returns True if x falls in an interval between 10 and 21.
This one-line lambda function serves as our event condition.

The six die rolls will fall into that interval range more than 54% of the time. Thus, if a roll sum of 13 or 20 comes up, we should not be surprised.

1.3.1 Evaluating extremes using interval analysis

Interval analysis is critical to solving a whole class of very important problems in probability and statistics. One such problem involves the evaluation of extremes: the problem boils down to whether observed data is too extreme to be believable.

Data seems extreme when it is too unusual to have occurred by random chance. For instance, suppose we observe 10 flips of an allegedly fair coin, and that coin lands on heads 8 out of 10 times. Is this a sensible result for a fair coin? Or is our coin secretly biased toward landing on heads? To find out, we must answer the following question: what is the probability that 10 fair coin flips lead to an extreme number of heads? We'll define an extreme head count as eight heads or more. Thus, we can

describe the problem as follows: what is the probability that 10 fair coin flips produce from 8 to 10 heads?

We'll find our answer by computing an interval probability. However, first we need the sample space for every possible sequence of 10 flipped coins. Let's generate a weighted sample space. As previously discussed, this is more efficient than using a non-weighted representation.

The following code creates a `weighted_sample_space` dictionary. Its keys equal the total number of observable heads, ranging from 0 through 10. These head counts map to values. Each value holds the number of coin-flip combinations that contain the associated head count. We thus expect `weighted_sample_space[10]` to equal 1, since there is just one possible way to flip a coin 10 times and get 10 heads. Meanwhile, we expect `weighted_sample_space[9]` to equal 10, since a single tail among 9 heads can occur across 10 different positions.

Listing 1.30 Computing the sample space for 10 coin flips

```
For reusability, we define a general function that returns a weighted sample
space for num_flips coin flips. The num_flips parameter is preset to 10 coin flips.

def generate_coin_sample_space(num_flips=10):
    weighted_sample_space = defaultdict(int)
    for coin_flips in product(['Heads', 'Tails'], repeat=num_flips):
        heads_count = len([outcome for outcome in coin_flips
                           if outcome == 'Heads'])
        weighted_sample_space[heads_count] += 1
    return weighted_sample_space

weighted_sample_space = generate_coin_sample_space()
assert weighted_sample_space[10] == 1
assert weighted_sample_space[9] == 10
```

Number of heads in a
unique sequence of
num_flips coin flips

Our weighted sample space is ready. We now compute the probability of observing an interval from 8 to 10 heads.

Listing 1.31 Computing an extreme head-count probability

```
prob = compute_event_probability(lambda x: is_in_interval(x, 8, 10),
                                 weighted_sample_space)
print(f"Probability of observing more than 7 heads is {prob}")
```

Probability of observing more than 7 heads is 0.0546875

Ten fair coin flips produce more than seven heads approximately 5% of the time. Our observed head count does not commonly occur. Does this mean the coin is biased? Not necessarily. We haven't yet considered extreme tail counts. If we had observed eight tails and not eight heads, we would have still been suspicious of the coin. Our computed interval did not take this extreme into account—instead, we treated eight or more tails as just another normal possibility. To evaluate the fairness of our coin, we

must include the likelihood of observing eight tails or more. This is equivalent to observing two heads or fewer.

Let's formulate the problem as follows: what is the probability that 10 fair coin flips produce either 0 to 2 heads or 8 to 10 heads? Or, stated more concisely, what is the probability that the coin flips do *not* produce from 3 to 7 heads? That probability is computed here.

Listing 1.32 Computing an extreme interval probability

```
prob = compute_event_probability(lambda x: not is_in_interval(x, 3, 7),
                                 weighted_sample_space)
print(f"Probability of observing more than 7 heads or 7 tails is {prob}")

Probability of observing more than 7 heads or 7 tails is 0.109375
```

Ten fair coin flips produce at least eight identical results approximately 10% of the time. That probability is low but still within the realm of plausibility. Without additional evidence, it's difficult to decide whether the coin is truly biased. So, let's collect that evidence. Suppose we flip the coin 10 additional times, and 8 more heads come up. This brings us to 16 heads out of 20 coin flips total. Our confidence in the fairness of the coin has been reduced, but by how much? We can find out by measuring the change in probability. Let's find the probability of 20 fair coin flips *not* producing from 5 to 15 heads.

Listing 1.33 Analyzing extreme head counts for 20 fair coin flips

```
weighted_sample_space_20_flips = generate_coin_sample_space(num_flips=20)
prob = compute_event_probability(lambda x: not is_in_interval(x, 5, 15),
                                 weighted_sample_space_20_flips)
print(f"Probability of observing more than 15 heads or 15 tails is {prob}")

Probability of observing more than 15 heads or 15 tails is 0.01181793212890625
```

The updated probability has dropped from approximately 0.1 to approximately 0.01. Thus, the added evidence has caused a tenfold decrease in our confidence in the coin's fairness. Despite this probability drop, the ratio of heads to tails has remained constant at 4 to 1. Both our original and updated experiments produced 80% heads and 20% tails. This leads to an interesting question: why does the probability of observing an extreme result decrease as the coin is flipped more times? We can find out through detailed mathematical analysis. However, a much more intuitive solution is to just visualize the distribution of head counts across our two sample space dictionaries. The visualization would effectively be a plot of keys (head counts) versus values (combination counts) present in each dictionary. We can carry out this plot using Matplotlib, Python's most popular visualization library. In the subsequent section, we discuss Matplotlib usage and its application to probability theory.

Summary

- A *sample space* is the set of all the possible outcomes an action can produce.
- An *event* is a subset of the sample space containing just those outcomes that satisfy some *event condition*. An event condition is a Boolean function that takes as input an outcome and returns either True or False.
- The *probability* of an event equals the fraction of event outcomes over all the possible outcomes in the entire sample space.
- Probabilities can be computed over *numeric intervals*. An interval is defined as the set of all the numbers sandwiched between two boundary values.
- Interval probabilities are useful for determining whether an observation appears extreme.

Plotting probabilities using Matplotlib

This section covers

- Creating simple plots using Matplotlib
- Labeling plotted data
- What is a probability distribution?
- Plotting and comparing multiple probability distributions

Data plots are among the most valuable tools in any data scientist's arsenal. Without good visualizations, we are effectively crippled in our ability to glean insights from our data. Fortunately, we have at our disposal the external Python Matplotlib library, which is fully optimized for outputting high-caliber plots and data visualizations. In this section, we use Matplotlib to better comprehend the coin-flip probabilities that we computed in section 1.

2.1 Basic Matplotlib plots

Let's begin by installing the Matplotlib library.

NOTE Call `pip install matplotlib` from the command line terminal to install the Matplotlib library.

Once installation is complete, import `matplotlib.pyplot`, which is the library's main plot-generation module. According to convention, the module is commonly imported using the shortened alias `plt`.

Listing 2.1 Importing Matplotlib

```
import matplotlib.pyplot as plt
```

We will now plot some data using `plt.plot`. That method takes as input two iterables: `x` and `y`. Calling `plt.plot(x, y)` prepares a 2D plot of `x` versus `y`; displaying the plot requires a subsequent call to `plt.show()`. Let's assign our `x` to equal integers 0 through 10 and our `y` values to equal double the values of `x`. The following code visualizes that linear relationship (figure 2.1).

Listing 2.2 Plotting a linear relationship

```
x = range(0, 10)
y = [2 * value for value in x]
plt.plot(x, y)
plt.show()
```

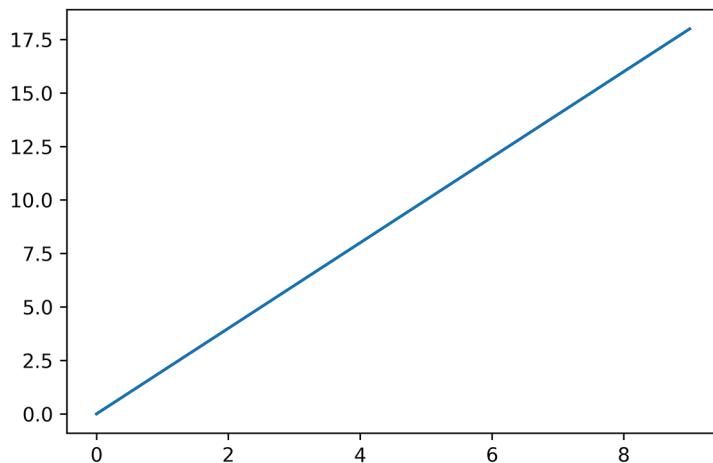


Figure 2.1 A Matplotlib plot of `x` versus `2x`. The `x` variable represents integers 0 through 10.

WARNING The axes in the linear plot are not evenly spaced, so the slope of the plotted line appears less steep than it actually is. We can equalize both axes by calling `plt.axis('equal')`. However, this will lead to an awkward visualization containing too much empty space. Throughout this book, we rely on Matplotlib's automated axes adjustments while also carefully observing the adjusted lengths.

The visualization is complete. Within it, our 10 y-axis points have been connected using smooth line segments. If we prefer to visualize the 10 points individually, we can do so using the `plt.scatter` method (figure 2.2).

Listing 2.3 Plotting individual data points

```
plt.scatter(x, y)
plt.show()
```

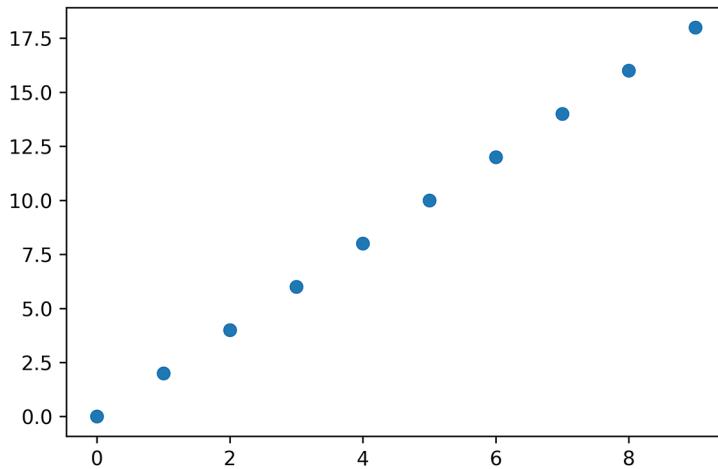


Figure 2.2 A Matplotlib scatter plot of `x` versus `2 * x`. The `x` variable represents integers 0 through 10. The individual integers are visible as scattered points in the plot.

Suppose we want to emphasize the interval where `x` begins at 2 and ends at 6. We do this by shading the area under the plotted curve over the specified interval, using the `plt.fill_between` method. The method takes as input both `x` and `y` and also a `where` parameter, which defines the interval coverage. The input of the `where` parameter is a list of Boolean values in which an element is `True` if the `x` value at the corresponding index falls within the interval we specified. In the following code, we set the `where` parameter to equal `[is_in_interval(value, 2, 6) for value in x]`. We also execute `plt.plot(x,y)` to juxtapose the shaded interval with the smoothly connected line (figure 2.3).

Listing 2.4 Shading an interval beneath a connected plot

```
plt.plot(x, y)
where = [is_in_interval(value, 2, 6) for value in x]
plt.fill_between(x, y, where=where)
plt.show()
```

So far, we have reviewed three visualization methods: `plt.plot`, `plt.scatter`, and `plt.fill_between`. Let's execute all three methods in a single plot (figure 2.4). Doing

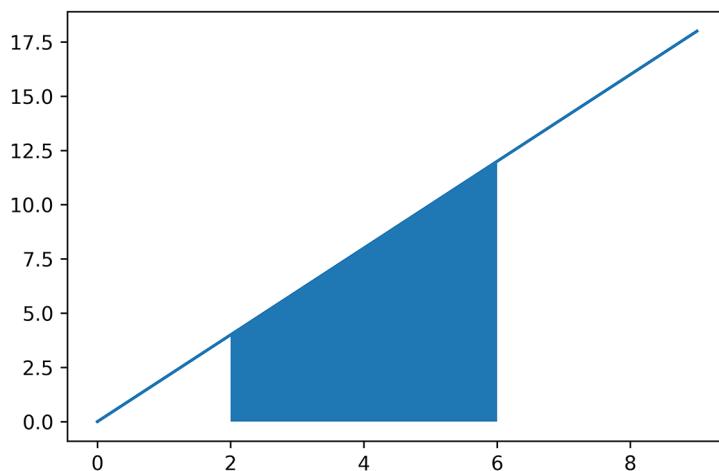


Figure 2.3 A connected plot with a shaded interval. The interval covers all values between 2 and 6.

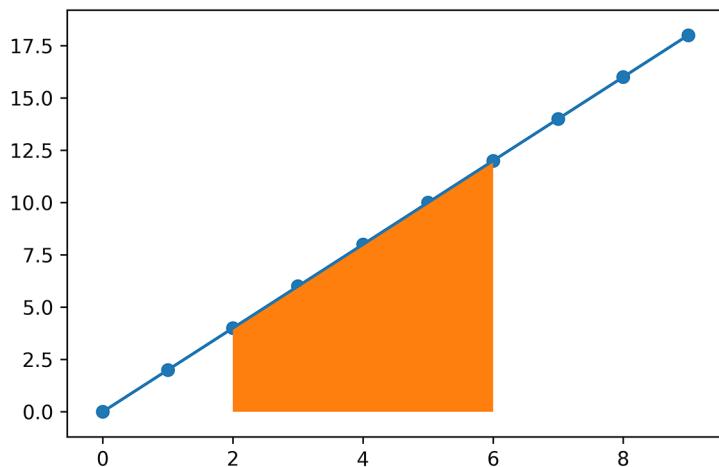


Figure 2.4 A connected plot and a scatter plot combined with a shaded interval. The individual integers in the plot appear as points marking a smooth, indivisible line.

so highlights an interval beneath a continuous line while also exposing individual coordinates.

Listing 2.5 Exposing individual coordinates within a continuous plot

```
plt.scatter(x, y)
plt.plot(x, y)
plt.fill_between(x, y, where=where)
plt.show()
```

No data plot is ever truly complete without descriptive x-axis and y-axis labels. Such labels can be set using the plt.xlabel and plt.ylabel methods (figure 2.5).

Listing 2.6 Adding axis labels

```
plt.plot(x, y)
plt.xlabel('Values between zero and ten')
plt.ylabel('Twice the values of x')
plt.show()
```

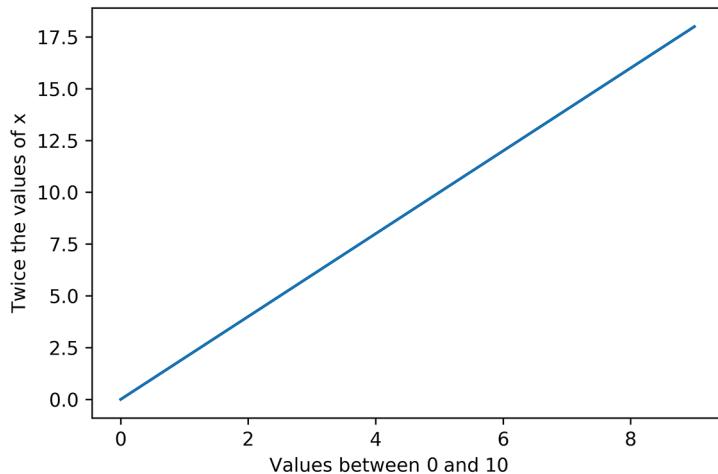


Figure 2.5 A Matplotlib plot with x-axis and y-axis labels

Common Matplotlib methods

- plt.plot(x, y)—Plots the elements of x versus the elements of y. The plotted points are connected using smooth line segments.
- plt.scatter(x, y)—Plots the elements of x versus the elements of y. The plotted points are visualized individually and are not connected by any lines.
- plt.fill_between(x, y, where=booleans)—Highlights a subset of the area beneath a plotted curve. The curve is obtained by plotting x versus y. The where parameter defines all highlighted intervals; it takes a list of Booleans that correspond to elements of x. Each Boolean is True if its corresponding x value is located within a highlighted interval.
- plt.xlabel(label)—Sets the x label of the plotted curve to equal label.
- plt.ylabel(label)—Sets the y label of the plotted curve to equal label.

2.2 Plotting coin-flip probabilities

We now have tools to visualize the relationship between a coin-flip count and the probability of heads. In section 1, we examined the probability of seeing 80% or more heads across a series of coin flips. That probability decreased as the coin-flip count went up, and we wanted to know why. We'll soon find out by plotting head counts versus their associated coin-flip combination counts. These values were already computed in our section 1 analysis. The keys in the `weighted_sample_space` dictionary contain all possible head counts across 10 flipped coins. These head counts map to combination counts. Meanwhile, the `weighted_sample_space_20_flips` dictionary contains the head-count mappings for 20 flipped coins.

Our aim is to compare the plotted data from both these dictionaries. We begin by plotting the elements of `weighted_sample_space`: we plot its keys on the x-axis versus the associated values on the y-axis. The x-axis corresponds to 'Head-count', and the y-axis corresponds to 'Number of coin-flip combinations with x heads'. We use a scatter plot to visualize key-to-value relationships directly without connecting any plotted points (figure 2.6).

Listing 2.7 Plotting the coin-flip weighted sample space

```
x_10_flips = list(weighted_sample_space.keys())
y_10_flips = [weighted_sample_space[key] for key in x_10_flips]
plt.scatter(x_10_flips, y_10_flips)
plt.xlabel('Head-count')
plt.ylabel('Number of coin-flip combinations with x heads')
plt.show()
```

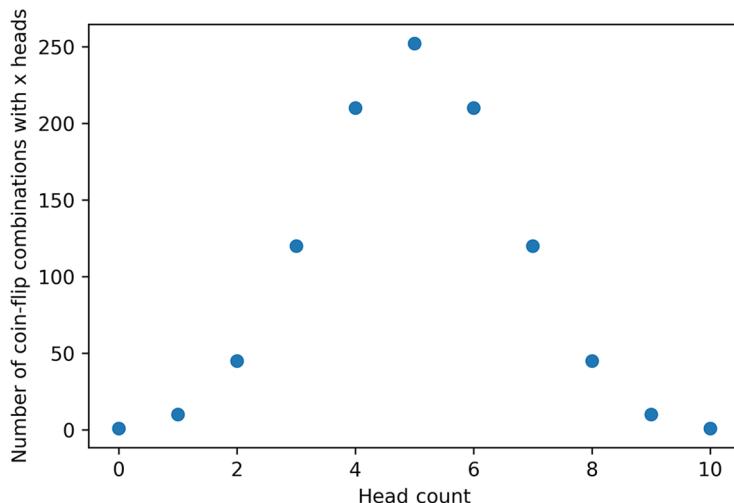


Figure 2.6 A scatter plot representation of the sample space for 10 flipped coins. The symmetric plot is centered around a peak at 5 of 10 counted heads.

The visualized sample space takes on a symmetric shape. The symmetry is set around a peak head count of 5. Therefore, head-count combinations closer to 5 occur more frequently than those that are further from 5. As we learned in the previous section, such frequencies correspond to probabilities. Thus, a head count is more probable if its value is closer to 5. Let's emphasize this by plotting the probabilities directly on the y-axis (figure 2.7). The probability plot will allow us to replace our lengthy y-axis label with a more concisely stated 'Probability'. We can compute the y-axis probabilities by taking our existing combination counts and dividing them by the total sample space size.

Listing 2.8 Plotting the coin-flip probabilities

```
sample_space_size = sum(weighted_sample_space.values())
prob_x_10_flips = [value / sample_space_size for value in y_10_flips]
plt.scatter(x_10_flips, prob_x_10_flips)
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

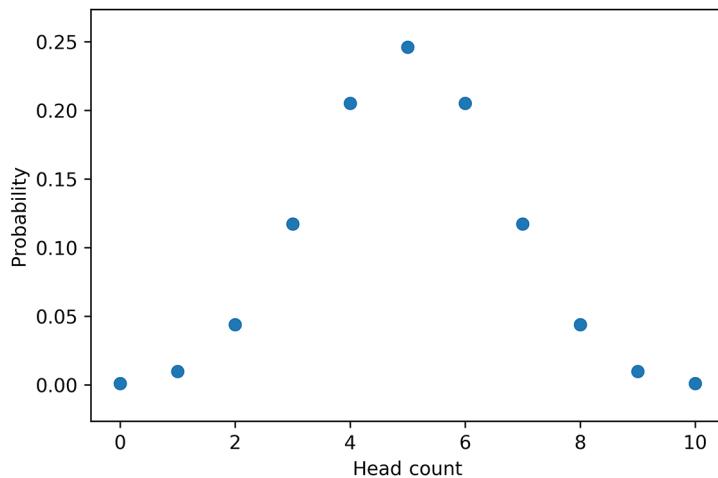


Figure 2.7 A scatter plot mapping head counts to their probability of occurrence. Probabilities can be inferred by looking directly at the plot.

Our plot permits us to visually estimate the probability of any head count. Thus, just by glancing at the plot, we can determine that the probability of observing five heads is approximately 0.25. This mapping between x-values and probabilities is referred to as a *probability distribution*. Probability distributions exhibit certain mathematically consistent properties that make them useful for likelihood analysis. For instance, consider the x-values of any probability distribution: they correspond to all the possible values of a random variable x . The probability that x falls within some interval is equal to the

area beneath the probability curve over the span of that interval. Therefore, the total area beneath a probability distribution always equals 1.0. This holds for any distribution, including our head-count plot. Listing 2.9 confirms this by executing `sum(prob_x_10_flips)`.

NOTE We can compute the area beneath each head-count probability p by using a vertical rectangle. The height of the rectangle is p . The width of the rectangle is 1.0, since all consecutive head counts on the x-axis are spaced one unit apart. Hence, the area of the rectangle is $p * 1.0$, which equals p . Consequently, the total area beneath the distribution equals `sum([p for p in prob_x_10_flips])`. In section 3, we'll do a deeper dive into how rectangles can be used to determine the area.

Listing 2.9 Confirming that all probabilities sum to 1.0

```
assert sum(prob_x_10_flips) == 1.0
```

The area beneath the head-count interval of 8 through 10 is equal to the probability of observing eight heads or more. Let's visualize that area using the `plt.fill_between` method. We also utilize `plt.plot` and `plt.scatter` to display individual head counts encompassing the shaded interval (figure 2.8).

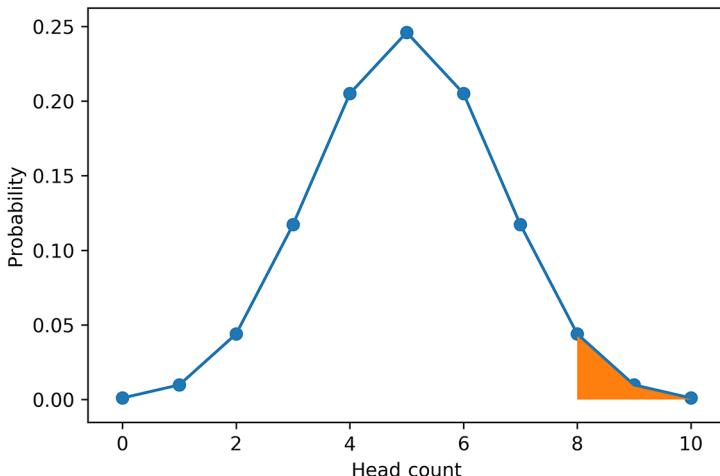


Figure 2.8 Overlaid smooth plot and scatter plot representations of the coin-flip probability distribution. A shaded interval covers head counts 8 through 10. The shaded area equals the probability of observing eight or more heads.

Listing 2.10 Shading the interval under a probability curve

```
plt.plot(x_10_flips, prob_x_10_flips)
plt.scatter(x_10_flips, prob_x_10_flips)
where = [is_in_interval(value, 8, 10) for value in x_10_flips]
```

```
plt.fill_between(x_10_flips, prob_x_10_flips, where=where)
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

NOTE We've purposefully smoothed the shaded interval to make a visually appealing plot. However, the true interval area is not smooth: it is composed of discrete, rectangular blocks, which resemble steps. The steps are discrete because the head counts are indivisible integers. If we wish to visualize the actual step-shaped area, we can pass a `ds="steps-mid"` parameter into `plt.plot` and a `step="mid"` parameter into `plt.fill_between`.

Now, let's also shade the interval demarcating the probability of observing eight tails or more. The following code highlights the extremes along both tail ends of our probability distribution (figure 2.9).

Listing 2.11 Shading the interval under the extremes of a probability curve

```
plt.plot(x_10_flips, prob_x_10_flips)
plt.scatter(x_10_flips, prob_x_10_flips)
where = [not is_in_interval(value, 3, 7) for value in x_10_flips]
plt.fill_between(x_10_flips, prob_x_10_flips, where=where)
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

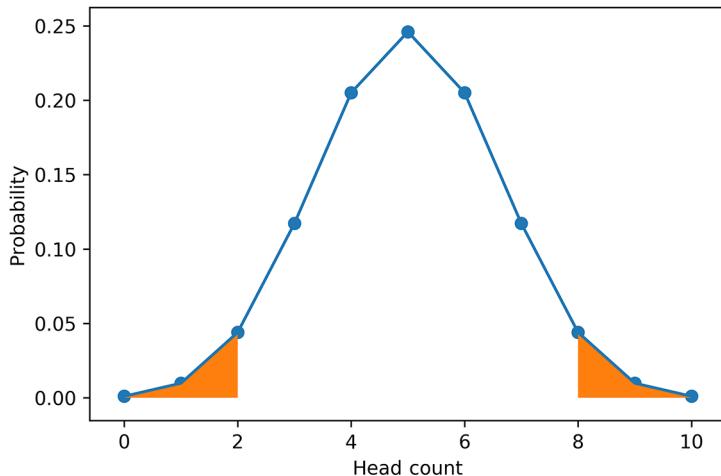


Figure 2.9 Overlaid smooth plot and scatter plot representations of the coin-flip probability distribution. Two shaded intervals span an extreme number of heads and tails. The intervals are symmetric, visually implying that their probabilities are equal.

The two symmetrically shaded intervals cover the right and left tail ends of the coin-flip curve. Based on our previous analysis, we know that the probability of observing more than seven heads or tails is approximately 10%. Therefore, each of the symmetrically shaded tail segments should cover approximately 5% of the total area under the curve.

2.2.1 Comparing multiple coin-flip probability distributions

Plotting the 10-coin-flip distribution makes it easier to visually comprehend the associated interval probabilities. Let's extend our plot to also encompass the distribution for 20 flipped coins. We'll plot both distributions on a single figure, although first we must compute the x-axis head counts and y-axis probabilities for the 20-coin-flip distribution.

Listing 2.12 Computing probabilities for a 20-coin-flip distribution

```
x_20_flips = list(weighted_sample_space_20_flips.keys())
y_20_flips = [weighted_sample_space_20_flips[key] for key in x_20_flips]
sample_space_size = sum(weighted_sample_space_20_flips.values())
prob_x_20_flips = [value / sample_space_size for value in y_20_flips]
```

Now we are ready to visualize the two distributions simultaneously (figure 2.10). We do this by executing `plt.plot` and `plt.scatter` on both probability distributions. We also pass a few style-related parameters into these method calls. One of the parameters is `color`: to distinguish the second distribution, we set its color to `black` by passing `color='black'`. Alternatively, we can avoid typing out the entire color name by passing '`k`', Matplotlib's single-character code for black. We can make the second distribution

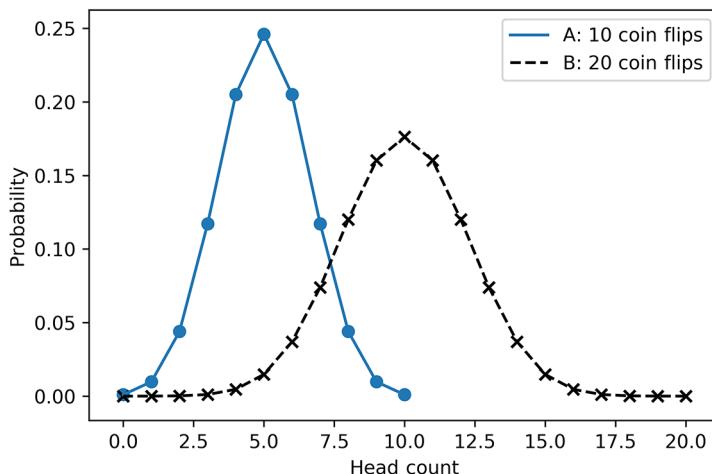


Figure 2.10 The probability distributions for 10 coin flips (A) and 20 coin flips (B). The 20-coin-flip distribution is marked by dashed lines and x-shaped scattered points.

stand out in other ways: passing `linestyle='--'` into `plt.plot` ensures that the distribution points are connected using dashed lines instead of regular lines. We can also distinguish the individual points using x-shaped markers rather than filled circles by passing `marker='x'` into `plt.scatter`. Finally, we add a legend to our figure by passing a `label` parameter into each of our two `plt.plot` calls and executing the `plt.legend()` method to display the legend. Within the legend, the 10-coin-flip distribution and the 20-coin-flip distribution are labeled A and B, respectively.

Listing 2.13 Plotting two simultaneous distributions

```
plt.plot(x_10_flips, prob_x_10_flips, label='A: 10 coin-flips')
plt.scatter(x_10_flips, prob_x_10_flips)
plt.plot(x_20_flips, prob_x_20_flips, color='black', linestyle='--',
         label='B: 20 coin-flips')
plt.scatter(x_20_flips, prob_x_20_flips, color='k', marker='x')
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.legend()
plt.show()
```

Common Matplotlib style parameters

- `color`—Determines the color of the plotted output. This setting can be a color name or a single-character code. Both `color='black'` and `color='k'` generate a black plot, and both `color='red'` and `color='r'` generate a red plot.
- `linestyle`—Determines the style of the plotted line that connects the data points. Its default value equals `'-'`. Inputting `linestyle='-'` generates a connected line, `linestyle='--'` generates a dashed line, `linestyle=':'` generates a dotted line, and `linestyle='.'` generates a line composed of alternating dots and dashes.
- `marker`—Determines the style of markers assigned to individually plotted points. Its default value equals `'o'`. Inputting `marker='o'` generates a circular marker, `marker='x'` generates an x-shaped marker, `marker='s'` generates a square-shaped marker, and `marker='p'` generates a pentagon-shaped marker.
- `label`—Maps a label to the specified color and style. This mapping appears in the legend of the plot. A subsequent call to `plt.legend()` is required to make the legend visible.

We've visualized our two distributions. Next, we highlight our interval of interest (80% heads or tails) across each of the two curves (figure 2.11). Note that the area beneath the tail ends of distribution B is very small; we remove the scatter points to highlight the tail-end intervals more clearly. We also replace the line style of distribution B with the more transparent `linestyle=':'`.

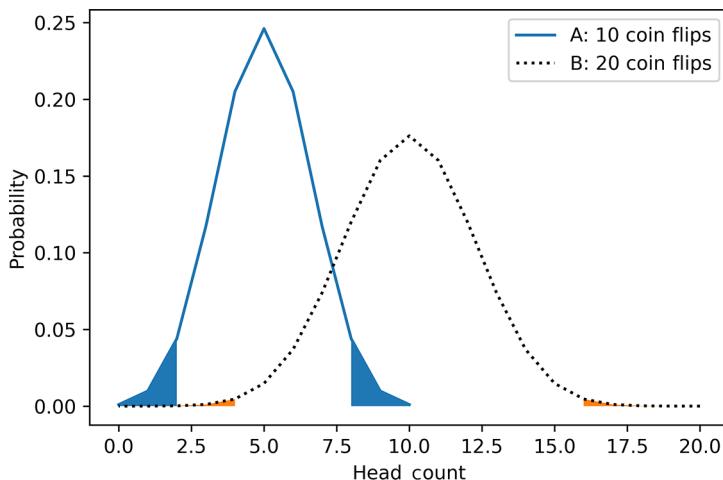


Figure 2.11 The probability distributions for 10 coin flips (A) and 20 coin flips (B). Shaded intervals beneath both distributions represent an extreme number of heads and tails. The shaded interval beneath B occupies one-tenth the area of the shaded interval beneath A.

Listing 2.14 Highlighting intervals beneath two plotted distributions

```

plt.plot(x_10_flips, prob_x_10_flips, label='A: 10 coin-flips')
plt.plot(x_20_flips, prob_x_20_flips, color='k', linestyle=':',
         label='B: 20 coin-flips')

where_10 = [not is_in_interval(value, 3, 7) for value in x_10_flips]
plt.fill_between(x_10_flips, prob_x_10_flips, where=where_10)
where_20 = [not is_in_interval(value, 5, 15) for value in x_20_flips]
plt.fill_between(x_20_flips, prob_x_20_flips, where=where_20)

plt.xlabel('Head-Count')
plt.ylabel('Probability')
plt.legend()
plt.show()

```

The shaded area beneath the tail ends of distribution B is much lower than the shaded interval beneath distribution A. This is because distribution A has fatter, more elevated tail ends that cover a thicker area quantity. Thickness in the tails accounts for differences in interval probabilities.

The visualization is informative, but only if we highlight the interval areas beneath both curves. Without the calls to `plt.fill_between`, we cannot answer the question we posed earlier: why does the probability of observing 80% or more heads decrease as the fair coin is flipped more times? The answer is hard to extrapolate because the two distributions show little overlap, making it difficult to do a direct visual comparison. Perhaps we can improve the plot by aligning the distribution peaks. Distribution

A is centered at 5 head counts (out of 10 coin flips), and distribution B is centered at 10 head counts (out of 20 coin flips). If we convert the head counts into frequencies (by dividing by the total coin flips), then both distribution peaks should align at a frequency of 0.5. The conversion should also align our head-count intervals of 8-to-10 and 16-to-20 so that they both lie on the interval 0.8-to-1.0. Let's execute this conversion and regenerate the plot (figure 2.12).

Listing 2.15 Converting head counts into frequencies

```
x_10_frequencies = [head_count /10 for head_count in x_10_flips]
x_20_frequencies = [head_count /20 for head_count in x_20_flips]

plt.plot(x_10_frequencies, prob_x_10_flips, label='A: 10 coin-flips')
plt.plot(x_20_frequencies, prob_x_20_flips, color='k', linestyle=':', 
         label='B: 20 coin-flips')
plt.legend()

plt.xlabel('Head-Frequency')
plt.ylabel('Probability')
plt.show()
```

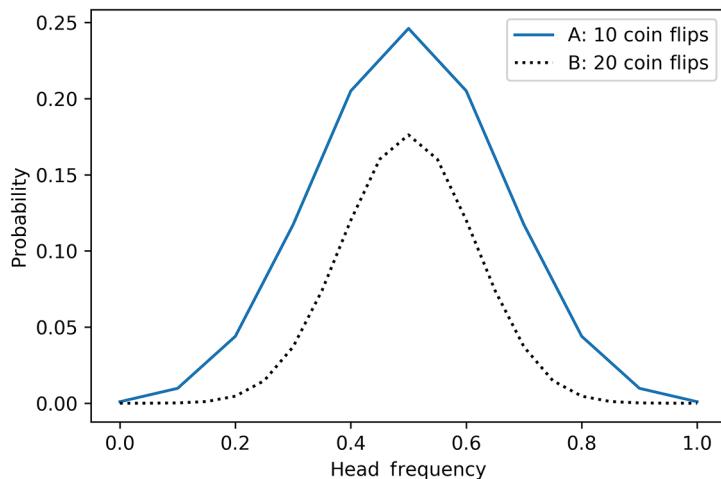


Figure 2.12 The head-count frequencies for 10 coin flips (A) and 20 coin flips (B) plotted against their probabilities. Both y-axis peaks align at a frequency of 0.5. The area of A fully covers the area of B because the total area of each plot no longer sums to 1.0.

As expected, the two peaks now both align at the head frequency of 0.5. However, our division by the head counts has reduced the areas beneath the two curves by tenfold and twentyfold, respectively. The total area beneath each curve no longer equals 1.0. This is a problem: as we've discussed, the total area under a curve must sum to 1.0 if we wish to infer an interval probability. However, we can fix the area sums if we multiply

the y-axis values of curves A and B by 10 and 20. The adjusted y-values will no longer refer to probabilities, so we'll have to name them something else. The appropriate term to use is *relative likelihood*, which mathematically refers to a y-axis value within a curve whose total area is 1.0. We therefore name our new y-axis variables `relative_likelihood_10` and `relative_likelihood_20`.

Listing 2.16 Computing relative likelihoods of frequencies

```
relative_likelihood_10 = [10 * prob for prob in prob_x_10_flips]
relative_likelihood_20 = [20 * prob for prob in prob_x_20_flips]
```

The conversion is complete. It's time to plot our two new curves while also highlighting the intervals associated with our `where_10` and `where_20` Boolean arrays (figure 2.13).

Listing 2.17 Plotting aligned relative likelihood curves

```
plt.plot(x_10_frequencies, relative_likelihood_10, label='A: 10 coin-flips')
plt.plot(x_20_frequencies, relative_likelihood_20, color='k',
          linestyle=':', label='B: 20 coin-flips')

plt.fill_between(x_10_frequencies, relative_likelihood_10, where=where_10)
plt.fill_between(x_20_frequencies, relative_likelihood_20, where=where_20)

plt.legend()
plt.xlabel('Head-Frequency')
plt.ylabel('Relative Likelihood')
plt.show()
```

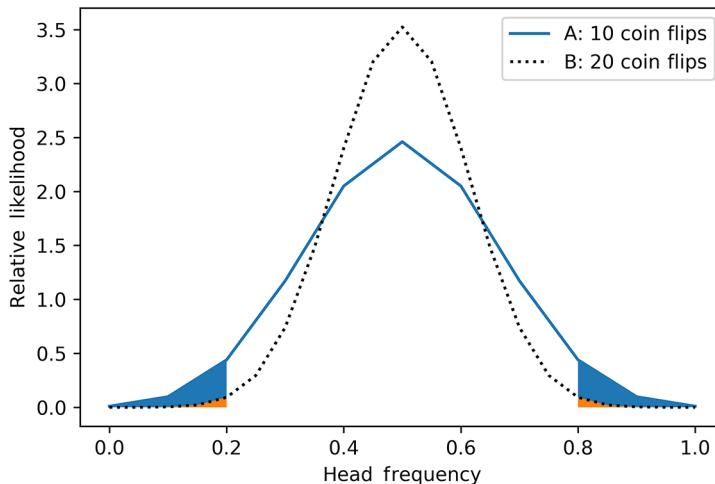


Figure 2.13 The head-count frequencies for 10 coin flips (A) and 20 coin flips (B) plotted against their relative likelihoods. Shaded intervals beneath both plots represent an extreme number of heads and tails. The areas of these intervals correspond to probabilities because the total area of each plot sums to 1.0.

Within the plot, curve A resembles a short yet wide-shouldered bodybuilder, while curve B could be compared to a taller and thinner individual. Since curve A is wider, its area over more extreme head-frequency intervals is larger. Hence, observed recordings of such frequencies are more likely to occur when the coin-flip count is 10 and not 20. Meanwhile, the thinner, more vertical curve B covers more area around the central frequency of 0.5.

If we flip more than 20 coins, how will this influence our frequency distribution? According to probability theory, each additional coin flip will cause the frequency curve to grow even taller and thinner (figure 2.14). The curve will transform like a stretched rubber band that's being pulled vertically upward: it will lose thickness in exchange for vertical length. As the total number of coin flips extends into the millions and billions, the curve will completely lose its girth, becoming a single very long vertical peak whose center lies at a frequency of 0.5. Beyond that frequency, the nonexistent area beneath the vertical line will approach zero. It follows that the area beneath the peak will approach 1.0 because our total area must always equal 1.0. The area of 1.0 corresponds to a probability of 1.0. Thus, as the number of coin flips approaches infinity, the frequency of heads will come to equal the actual probability of heads with absolute certainty.

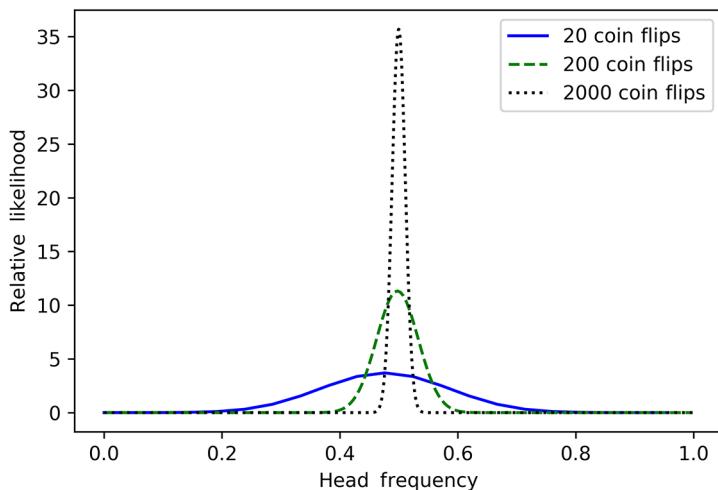


Figure 2.14 Hypothetical head-count frequencies plotted over an increasing number of coin flips. All y-axis peaks align at a frequency of 0.5. The peaks grow higher and more narrow as the coin-flip count goes up. At 2,000 coin flips, the constricted area of the peak is centered almost entirely at 0.5. With infinite coin flips, the resulting peak should stretch into a single, vertical line that's perfectly positioned at 0.5.

The relationship between infinite coin flips and absolute certainty is guaranteed by a fundamental theorem in probability theory: *the law of large numbers*. According to that

law, the frequency of an observation becomes virtually indistinguishable from the probability of that observation when the number of observations grows high. Therefore, with enough coin flips, our frequency of heads will equal the actual probability of heads, which is 0.5. Beyond mere coin flips, we can apply the law to more complex phenomena, such as card games. If we run enough card game simulations, then our frequency of a win will equal the actual probability of a win.

In the subsequent section, we will show how the law of large numbers can be combined with random simulations to approximate complex probabilities. Eventually, we will execute simulations to find the probabilities of randomly drawn cards. However, as the law of large numbers indicates, these simulations must be run on a large, computationally expensive scale. Therefore, efficient simulation implementation requires us to familiarize ourselves with the NumPy numeric computation library. That library is discussed in section 3.

Summary

- By plotting every possible numeric observation versus its probability, we generate a probability distribution. The total area beneath a probability distribution sums to 1.0. The area beneath a specific interval of the distribution equals the probability of observing some value within that interval.
- The y-axis values of a probability distribution do not necessarily need to equal probabilities, as long as the plotted area sums to 1.0.
- The probability distribution of a fair coin-flip sequence resembles a symmetric curve. Its x-axis head counts can be converted into frequencies. During that conversion, we can maintain an area of 1.0 by converting y-axis probabilities into relative likelihoods. The peak of the converted curve is centered at a frequency of 0.5. If the coin-flip count is increased, then the peak will also rise as the curve becomes more narrow on its sides.
- According to the *law of large numbers*, the frequency of any observation will approach the probability of that observation as the observation count grows large. Thus, a fair-coin distribution becomes dominated by its central frequency of 0.5 as the coin-flip count goes up.



Running random simulations in NumPy

This section covers

- Basic usage of the NumPy library
- Simulating random observations using NumPy
- Visualizing simulated data
- Estimating unknown probabilities from simulated observations

NumPy, which stands for Numerical Python, is the engine that powers Pythonic data science. Python, despite its many virtues, is simply not suited for large-scale numeric analysis. Hence, data scientists must rely on the external NumPy library to efficiently manipulate and store numeric data. NumPy is an incredibly powerful tool for processing large collections of raw numbers. Thus, many of Python's external data processing libraries are NumPy compatible. One such library is Matplotlib, which we introduced in the previous section. Other NumPy-driven libraries are discussed in later portions of the book. This section focuses on randomized numerical simulations. We will use NumPy to analyze billions of random data points; these random observations will allow us to learn hidden probabilities.

3.1 Simulating random coin flips and die rolls using NumPy

NumPy should already be installed in your working environment as one of the Matplotlib requirements. Let's import NumPy as np based on common NumPy usage convention.

NOTE NumPy can also be installed independently of Matplotlib by calling `pip install numpy` from the command line terminal.

Listing 3.1 Importing NumPy

```
import numpy as np
```

Now that NumPy is imported, we can carry out random simulations using the `np.random` module. That module is useful for generating random values and simulating random processes. For instance, calling `np.random.randint(1, 7)` produces a random integer between 1 and 6. The method chooses from the six possible integers with equal likelihood, thus simulating a single roll of a standard die.

Listing 3.2 Simulating a randomly rolled die

```
die_roll = np.random.randint(1, 7)
assert 1 <= die_roll <= 6
```

The generated `die_roll` value is random, and its assigned value will vary among the readers of this book. The inconsistency could make it difficult to perfectly re-create certain random simulations in this section. We need a way of ensuring that all our random outputs can be reproduced at home. Conveniently, consistency can easily be maintained by calling `np.random.seed(0)`; this method call makes sequences of randomly chosen values reproducible. After the call, we can directly guarantee that our first three dice rolls will land on values 5, 6, and 1.

Listing 3.3 Seeding reproducible random die rolls

```
np.random.seed(0)
die_rolls = [np.random.randint(1, 7) for _ in range(3)]
assert die_rolls == [5, 6, 1]
```

Adjusting the inputted `x` into `np.random.randint(0, x)` allows us to simulate any number of discrete outcomes. For instance, setting `x` to 52 will simulate a randomly drawn card. Alternatively, setting `x` to 2 will simulate a single flip of an unbiased coin. Let's generate that coin flip by calling `np.random.randint(0, 2)`; this method call returns a random value equal to either 0 or 1. We assume that 0 stands for tails and 1 stands for heads.

Listing 3.4 Simulating one fair coin flip

```
np.random.seed(0)
coin_flip = np.random.randint(0, 2)
print(f"Coin landed on {'heads' if coin_flip == 1 else 'tails'}")
```

Coin landed on tails

Next, we simulate a sequence of 10 coin flips and then compute the observed frequency of heads.

Listing 3.5 Simulating 10 fair coin flips

```
np.random.seed(0)
def frequency_heads(coin_flip_sequence):
    total_heads = len([head for head in coin_flip_sequence if head == 1]) ←
    return total_heads / len(coin_flip_sequence)

coin_flips = [np.random.randint(0, 2) for _ in range(10)]
freq_heads = frequency_heads(coin_flips)
print(f"Frequency of Heads is {freq_heads}")
```

Note that we can
compute the head count
more efficiently by running
`sum(coin_flip_sequence)`.

Frequency of Heads is 0.8

The observed frequency is 0.8, which is quite disproportionate to the actual probability of heads. However, as we have learned, 10 coin flips will produce such extreme frequencies approximately 10% of the time. More coin flips are required to estimate the actual probability.

Let's see what happens when we flip the coin 1,000 times. After each flip, we record the total frequency of heads observed in the sequence. Once the coin flips are completed, we visualize our output by plotting the coin-flip count versus the frequency count (figure 3.1). Our plot also includes a horizontal line along the actual probability of 0.5. We generate that line by calling `plt.axhline(0.5, color='k')`.

Listing 3.6 Plotting simulated fair coin-flip frequencies

```
np.random.seed(0)
coin_flips = []
frequencies = []
for _ in range(1000):
    coin_flips.append(np.random.randint(0, 2))
    frequencies.append(frequency_heads(coin_flips))

plt.plot(list(range(1000)), frequencies)
plt.axhline(0.5, color='k')
plt.xlabel('Number of Coin Flips')
plt.ylabel('Head-Frequency')
plt.show()
```

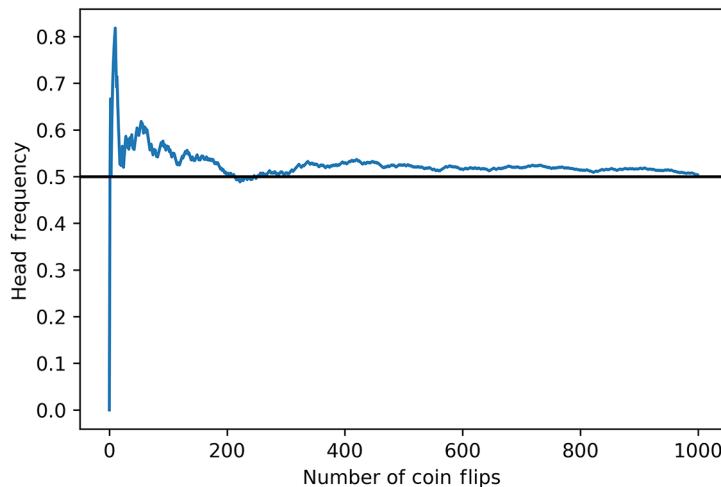


Figure 3.1 The number of fair coin flips plotted against the observed head-count frequency. The frequency fluctuates wildly before stabilizing at around 0.5.

The probability of heads slowly converges to 0.5. Thus, the law of large numbers appears to hold up.

3.1.1 *Analyzing biased coin flips*

We've simulated a sequence of unbiased coin flips, but what if we wish to simulate a coin that falls on heads 70% of the time? Well, we can generate that biased output by calling `np.random.binomial(1, 0.7)`. The binomial method name refers to the generic coin-flip distribution, which mathematicians call the *binomial distribution*. The method takes as input two parameters: the number of coin flips and the probability of the desired coin-flip outcome. The method executes the specified number of biased coin flips and then counts the instances when the desired outcome was observed. When the number of coin flips is set to 1, the method returns a binary value of 0 or 1. In our case, a value of 1 represents our desired observation of heads.

Listing 3.7 Simulating biased coin flips

```
np.random.seed(0)
print("Let's flip the biased coin once.")
coin_flip = np.random.binomial(1, 0.7)
print(f"Biased coin landed on {'heads' if coin_flip == 1 else 'tails'}.")

print("\nLet's flip the biased coin 10 times.")
number_coin_flips = 10
head_count = np.random.binomial(number_coin_flips, .7)
print(f"\n{head_count} heads were observed out of "
      f"\n{number_coin_flips} biased coin flips")
```

```
Let's flip the biased coin once.  
Biased coin landed on heads.
```

```
Let's flip the biased coin 10 times.  
6 heads were observed out of 10 biased coin flips
```

Let's generate a sequence of 1,000 biased coin flips. We then check if the frequency converges to 0.7.

Listing 3.8 Computing coin-flip-frequency convergence

```
np.random.seed(0)  
head_count = np.random.binomial(1000, 0.7)  
frequency = head_count / 1000  
print(f"Frequency of Heads is {frequency}")  
  
Frequency of Heads is 0.697
```

The frequency of heads approximates 0.7 but is not actually equal to 0.7. In fact, the frequency value is 0.003 units smaller than the true probability of heads. Suppose we recompute the frequency of 1,000 coin flips five more times. Will all the frequencies be lower than 0.7? Will certain frequencies hit the exact value of 0.7? We'll find out by executing `np.random.binomial(1000, 0.7)` over five looped iterations.

Listing 3.9 Recomputing coin-flip-frequency convergence

```
np.random.seed(0)  
assert np.random.binomial(1000, 0.7) / 1000 == 0.697      ←  
for i in range(1, 6):  
    head_count = np.random.binomial(1000, 0.7)  
    frequency = head_count / 1000  
    print(f"Frequency at iteration {i} is {frequency}")  
    if frequency == 0.7:  
        print("Frequency equals the probability!\n")  
  
Frequency at iteration 1 is 0.69  
Frequency at iteration 2 is 0.7  
Frequency equals the probability!  
  
Frequency at iteration 3 is 0.707  
Frequency at iteration 4 is 0.702  
Frequency at iteration 5 is 0.699
```

As a reminder, we seeded our random number generator to maintain consistent output. Thus, our first pseudorandom sampling will return the previously observed frequency of 0.697. We'll skip over this result to generate five fresh frequencies.

Just one of the five iterations produced a measurement that equaled the real probability. Twice the measured frequency was slightly too low, and twice it was slightly too high. The observed frequency appears to fluctuate over every sampling of 1,000 coin flips. It seems that even though the law of large numbers allows us to approximate the actual probability, some uncertainty still remains. Data science is somewhat messy, and we cannot always be certain of the conclusions we draw from our data. Nevertheless,

our uncertainty can be measured and contained using what mathematicians call a *confidence interval*.

3.2 Computing confidence intervals using histograms and NumPy arrays

Suppose we're handed a biased coin whose bias we don't know. We flip the coin 1,000 times and observe a frequency of 0.709. We know the frequency approximates the actual probability, but by how much? More precisely, what are the chances of the actual probability falling within an interval close to 0.709 (such as an interval between 0.7 and 0.71)? To find out, we must do additional sampling.

We've previously sampled our coin over five iterations of 1,000 coin flips each. The sampling produced some fluctuations in the frequency. Let's explore these fluctuations by increasing our frequency count from 5 to 500. We can execute this supplementary sampling by running `[np.random.binomial(1000, 0.7) for _ in range(500)]`.

Listing 3.10 Computing frequencies with 500 flips per sample

```
np.random.seed(0)
head_count_list = [np.random.binomial(1000, 0.7) for _ in range(500)]
```

However, we can more efficiently sample over 500 iterations by running `np.random.binomial(coin_flip_count, 0.7, size=500)`. The optional `size` parameter allows us to execute `np.random.binomial(coin_flip_count, 0.7)` 500 times while using NumPy's internal optimizations.

Listing 3.11 Optimizing the coin-flip-frequency computation

```
np.random.seed(0)
head_count_array = np.random.binomial(1000, 0.7, 500)
```

The output is not a Python list but a NumPy array data structure. As previously noted, NumPy arrays can more efficiently store numeric data. The actual numeric quantities stored in both `head_count_array` and `head_count_list` remain the same. We prove this by converting the array into a list using the `head_count_array.tolist()` method.

Listing 3.12 Converting a NumPy array to a Python list

```
assert head_count_array.tolist() == head_count_list
```

Conversely, we can also convert our Python list into a value-equivalent NumPy array by calling `np.array(head_count_list)`. The equality between the converted array and `head_count_array` can be confirmed using the `np.array_equal` method.

Listing 3.13 Converting a Python list to a NumPy array

```
new_array = np.array(head_count_list)
assert np.array_equal(new_array, head_count_array) == True
```

Why should we prefer to use a NumPy array over a standard Python list? Well, besides the aforementioned memory optimizations and analysis speed-ups, NumPy makes it easier to implement clean code. For instance, NumPy offers more straightforward multiplication and division. Dividing a NumPy array directly by `x` creates a new array whose elements are all divided by `x`. Thus, executing `head_count_array / 1000` will automatically transform our head counts into frequencies. By contrast, frequency calculation across `head_count_list` requires that we either iterate over all elements in the list or use Python's convoluted `map` function.

Listing 3.14 Computing frequencies using NumPy

```
frequency_array = head_count_array / 1000
assert frequency_array.tolist() == [head_count / 1000
                                    for head_count in head_count_list]
assert frequency_array.tolist() == list(map(lambda x: x / 1000,
                                            head_count_list))
```

Useful NumPy methods for running random simulations

- `np.random.randint(x, y)`—Returns a random integer between `x` and `y-1`, inclusive.
- `np.random.binomial(1, p)`—Returns a single random value equal to 0 or 1. The probability that the value equals 1 is `p`.
- `np.random.binomial(x, p)`—Runs `x` instances of `np.random.binomial(1, p)` and returns the summed result. The returned value represents the number of nonzero observations across `x` samples.
- `np.random.binomial(x, p, size=y)`—Returns an array of `y` elements. Each array element is equal to a random output of `np.random.binomial(x, p)`.
- `np.random.binomial(x, p, size=y) / x`—Returns an array of `y` elements. Each element represents the frequency of nonzero observations across `x` samples.

We've converted our head-count array into a frequency array using a simple division operation. Let's explore the contents of `frequency_array` in greater detail. We start by outputting the first 20 sampled frequencies using the same `:` index-slicing delimiter utilized by Python lists. Note that unlike a printed list, the NumPy array does not contain commas in its output.

Listing 3.15 Printing a NumPy frequency array

```
print(frequency_array[:20])
[ 0.697  0.69   0.7    0.707  0.702  0.699  0.723  0.67   0.702  0.713
  0.721  0.689  0.711  0.697  0.717  0.691  0.731  0.697  0.722  0.728]
```

The sampled frequencies fluctuate from 0.69 to approximately 0.731. Of course, an additional 480 frequencies remain in `frequency_array`. Let's extract the minimum

and maximum array values by calling the `frequency_array.min()` and `frequency_array.max()` methods.

Listing 3.16 Finding the largest and smallest frequency values

```
min_freq = frequency_array.min()
max_freq = frequency_array.max()
print(f"Minimum frequency observed: {min_freq}")
print(f"Maximum frequency observed: {max_freq}")
print(f"Difference across frequency range: {max_freq - min_freq}")

Minimum frequency observed: 0.656
Maximum frequency observed: 0.733
Difference across frequency range: 0.0769999999999996
```

Somewhere in the frequency range of 0.656 to 0.733 lies the true probability of heads. That interval span is noticeably large, with a more than 7% difference between the largest and smallest sampled values. Perhaps we can narrow the frequency range by plotting all unique frequencies against their occurrence counts (figure 3.2).

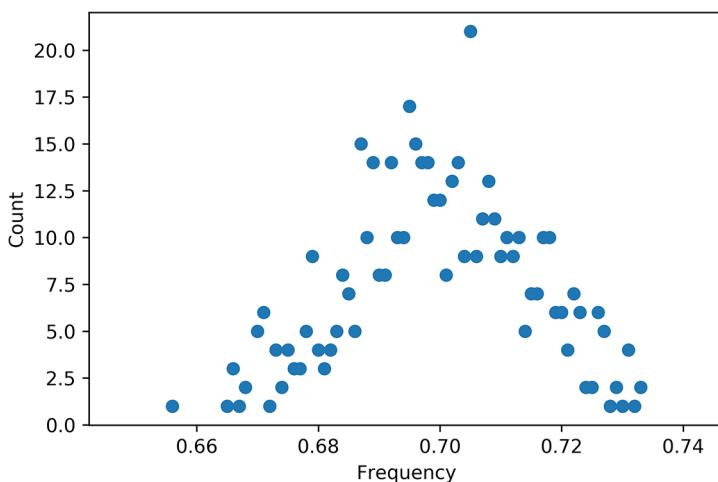


Figure 3.2 A scatter plot of 500 head-count frequencies plotted against the frequency counts. The frequencies are centered around 0.7. Certain proximate frequencies appear as overlapping dots in the plot.

Listing 3.17 Plotting measured frequencies

```
frequency_counts = defaultdict(int)
for frequency in frequency_array:
    frequency_counts[frequency] += 1

frequencies = list(frequency_counts.keys())
counts = [frequency_counts[freq] for freq in frequencies]
```

```
plt.scatter(frequencies, counts)
plt.xlabel('Frequency')
plt.ylabel('Count')
plt.show()
```

The visualization is informative: frequencies close to 0.7 occur more commonly than other, more distant values. However, our plot is also flawed, since nearly identical frequencies appear as overlapping dots in the chart. We should group these proximate frequencies together instead of treating them as individual points.

3.2.1 Binning similar points in histogram plots

Let's try a more nuanced visualization by binning together frequencies that are in close proximity to each other. We subdivide our frequency range into N equally spaced bins and then place all frequency values into one of those bins. By definition, the values in any given bin are at most $1/N$ units apart. Then we count the total values in each bin and visualize the counts using a plot.

The bin-based plot we just described is called a *histogram*. We can generate histograms in Matplotlib by calling `plt.hist`. The method takes as input the sequence of values to be binned and an optional `bins` parameter, which specifies the total number of bins. Thus, calling `plt.hist(frequency_array, bins=77)` will split our data across 77 bins, each covering a width of .01 units. Alternatively, we can pass in `bins='auto'`, and Matplotlib will select an appropriate bin width using a common optimization technique (the details of which are beyond the scope of this book). Let's plot a histogram while optimizing bin width by calling `plt.hist(frequency_array, bins='auto')` (figure 3.3).

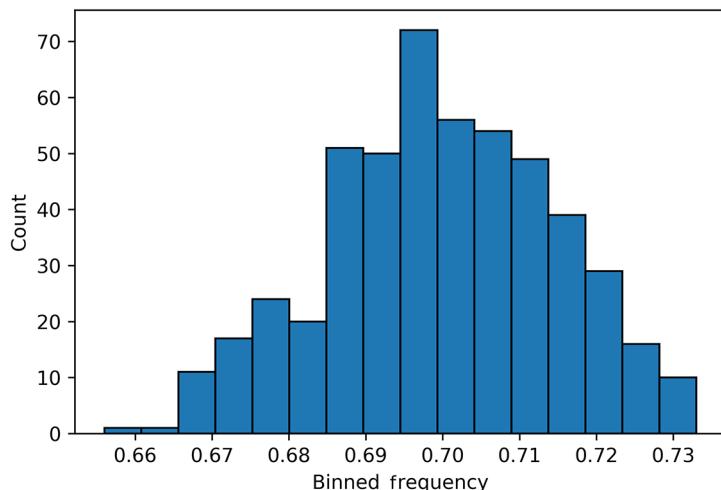


Figure 3.3 A histogram of 500 binned frequencies plotted against the number of elements in each bin. The bin with the most elements is centered around a frequency of 0.7.

NOTE In listing 3.18, we also include an `edgecolor='black'` parameter. This helps us visually distinguish the boundaries between bins by coloring the bin edges black.

Listing 3.18 Plotting a frequency histogram using `plt.hist`

```
plt.hist(frequency_array, bins='auto', edgecolor='black')
plt.xlabel('Binned Frequency')
plt.ylabel('Count')
plt.show()
```

In our plotted histogram, the bin with the highest frequency count falls between 0.69 and 0.70. This bin rises noticeably higher than the dozen or so other bins. We can obtain a more precise bin count using `counts`, which is a NumPy array returned by `plt.hist`. The array holds the y-axis frequency counts for each binned group. Let's call `plt.hist` to return `counts` and subsequently access `counts.size` to find the total number of binned groups.

Listing 3.19 Counting bins in a plotted histogram

```
counts, _, _ = plt.hist(frequency_array, bins='auto',
                       edgecolor='black') ←
print(f"Number of Bins: {counts.size}")
```

counts is one of three variables returned by `plt.hist`. The other variables are discussed later in this section.

Number of Bins: 16

There are 16 bins in the histogram. How wide is each bin? We can find out by dividing the total frequency range by 16. Alternatively, we can use the `bin_edges` array, which is the second variable returned by `plt.hist`. This array holds the x-axis positions of the vertical bin edges in the plot. Thus, the difference between any two consecutive edge positions equals the bin width.

Listing 3.20 Finding the width of bins in a histogram

```
counts, bin_edges, _ = plt.hist(frequency_array, bins='auto',
                               edgecolor='black')

bin_width = bin_edges[1] - bin_edges[0]
assert bin_width == (max_freq - min_freq) / counts.size
print(f"Bin width: {bin_width}")

Bin width: 0.00481249999999997
```

NOTE The size of `bin_edges` is always one greater than the size of the `counts`. Why is that the case? Imagine if we had only one rectangular bin: it would be bounded by two vertical lines. Adding an additional bin would also increase the boundary size by 1. If we extrapolate that logic to N bins, then we'd expect to see $N + 1$ boundary lines.

The `bin_edges` array can be used in tandem with `counts` to output the element count and coverage range for any specified bin. Let's define an `output_bin_coverage` function that prints the count and coverage for any bin at position `i`.

Listing 3.21 Getting a bin's frequency and size

```
def output_bin_coverage(i):
    count = int(counts[i])
    range_start, range_end = bin_edges[i], bin_edges[i+1]
    range_string = f"{range_start} - {range_end}"
    print(f"The bin for frequency range {range_string} contains "
          f"{count} element{'' if count == 1 else 's'}")
```

A bin at position `i` contains `counts[i]` frequencies.

`output_bin_coverage(0)`

A bin at position `i` covers a frequency range of `bin_edges[i]` through `bin_edges[i+1]`.

`output_bin_coverage(5)`

```
The bin for frequency range 0.656 - 0.6608125 contains 1 element
The bin for frequency range 0.6800625 - 0.684875 contains 20 elements
```

Now, let's compute the count and frequency range for the highest peak in our histogram. For this, we need the index of `counts.argmax()`. Conveniently, NumPy arrays have a built-in `argmax` method, which returns the index of the maximum value in an array.

Listing 3.22 Finding the index of an array's maximum value

```
assert counts[counts.argmax()] == counts.max()
```

Thus, calling `output_bin_coverage(counts.argmax())` should provide us with the output we've requested.

Listing 3.23 Using argmax to return a histogram's peak

```
output_bin_coverage(counts.argmax())
```

```
The bin for frequency range 0.6945 - 0.6993125 contains 72 elements
```

3.2.2 Deriving probabilities from histograms

The most-occupied bin in the histogram contains 72 elements and covers a frequency range of approximately 0.694 to 0.699. How can we determine whether the actual probability of heads falls within that range (without knowing the answer in advance)? One option is to calculate the likelihood that a randomly measured frequency falls within 0.694 to 0.699. If that likelihood were 1.0, then 100% of measured frequencies would be covered by the range. These measured frequencies would occasionally include the actual probability of heads, so we would be 100% confident that our true probability lay somewhere between 0.694 and 0.699. Even if the likelihood were lower, at 95%, we would still be fairly confident that the range enclosed our true probability value.

How should we calculate the likelihood? Earlier, we showed that the likelihood of an interval equals its area under a curve, but only when the total plotted area sums to 1.0. The area under our histogram is greater than 1.0 and thus must be modified by passing `density=True` into `plt.hist`. The passed parameter maintains the histogram's shape while forcing its area's sum to equal 1.0.

Listing 3.24 Plotting a histogram's relative likelihoods

```
likelihoods, bin_edges, _ = plt.hist(frequency_array, bins='auto',
                                    edgecolor='black', density=True)
plt.xlabel('Binned Frequency')
plt.ylabel('Relative Likelihood')
plt.show()
```

The binned counts have now been replaced by relative likelihoods, which are stored in the `likelihoods` array (figure 3.4). As we've discussed, *relative likelihood* is a term applied to the y-values of a plot whose area sums to 1.0. Of course, the area beneath our histogram now sums to 1.0. We can prove this by summing the rectangular area of each bin, which equals the bin's vertical likelihood value multiplied by `bin_width`. Hence, the area beneath the histogram is equal to the summed likelihoods multiplied by `bin_width`. Consequently, calling `likelihoods.sum() * bin_width` should return an area of 1.0.

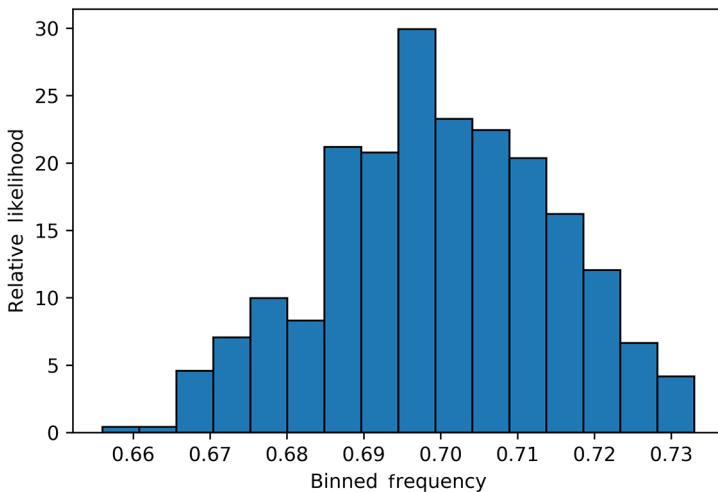


Figure 3.4 A histogram of 500 binned frequencies plotted against their associated relative likelihoods. The area of the histogram sums to 1.0. That area can be computed by summing over the rectangular areas of each bin.

NOTE The total area equals the summed rectangle areas in the histogram. In figure 3.4, the length of the longest rectangles is quite large, so we visually estimate that the total area is greater than 1.0.

Listing 3.25 Computing the total area under a histogram

```
assert likeliabilities.sum() * bin_width == 1.0
```

The histogram's total area sums to 1.0. Thus, the area beneath the histogram's peak now equals the probability of a randomly sampled frequency falling within the 0.694 to 0.699 interval range. Let's compute this value by calculating the area of the bin positioned at `likeliabilities.argmax()`.

Listing 3.26 Computing the probability of the peak frequencies

```
index = likeliabilities.argmax()
area = likeliabilities[index] * bin_width
range_start, range_end = bin_edges[index], bin_edges[index+1]
range_string = f"{range_start} - {range_end}"
print(f"Sampled frequency falls within interval {range_string} with
      probability {area}")

Sampled frequency falls within interval 0.6945 - 0.6993125 with probability
0.144
```

The probability is approximately 14%. That value is low, but we can increase it by expanding our interval range beyond one bin. We stretch the range to cover neighboring bins at indices `likeliabilities.argmax() - 1` and `likeliabilities.argmax() + 1`.

NOTE As a reminder, Python's indexing notation is inclusive of the start index and exclusive of the end index. Hence, we set the end index to equal `likeliabilities.argmax() + 2` to include `likeliabilities.argmax() + 1`.

Listing 3.27 Increasing the probability of a frequency range

```
peak_index = likeliabilities.argmax()
start_index, end_index = (peak_index - 1, peak_index + 2)
area = likeliabilities[start_index: end_index + 1].sum() * bin_width
range_start, range_end = bin_edges[start_index], bin_edges[end_index]
range_string = f"{range_start} - {range_end}"
print(f"Sampled frequency falls within interval {range_string} with
      probability {area}")

Sampled frequency falls within interval 0.6896875 - 0.704125 with probability
0.464
```

The three bins cover a frequency range of approximately 0.689 to 0.704. Their associated probability is 0.464. Thus, the three bins represent what statisticians call a 46.4% *confidence interval*, which means we are 46.4% confident that our true probability falls within the three-bin range. That confidence percentage is too low. Statisticians prefer a confidence interval of 95% or more. We reach that confidence interval by iteratively expanding our leftmost bin and rightmost bin until the interval area stretches past 0.95.

Listing 3.28 Computing a high confidence interval

```
def compute_high_confidence_interval(likelihoods, bin_width):
    peak_index = likelihoods.argmax()
    area = likelihoods[peak_index] * bin_width
    start_index, end_index = peak_index, peak_index + 1
    while area < 0.95:
        if start_index > 0:
            start_index -= 1
        if end_index < likelihoods.size - 1:
            end_index += 1

    area = likelihoods[start_index: end_index + 1].sum() * bin_width

    range_start, range_end = bin_edges[start_index], bin_edges[end_index]
    range_string = f"{range_start:.6f} - {range_end:.6f}"
    print((f"The frequency range {range_string} represents a "
           f"{100 * area:.2f}% confidence interval"))
    return start_index, end_index

compute_high_confidence_interval(likelihoods, bin_width)
```

The frequency range 0.670438 - 0.723375 represents a 95.40% confidence interval

The frequency range of roughly 0.670 to 0.723 represents a 95.4% confidence interval. Thus, a sampled sequence of 1,000 biased coin flips should fall within that range 95.4% of the time. We're fairly confident that the true probability lies somewhere between 0.670 and 0.723. However, we still cannot tell for sure whether the true probability is closer to 0.67 or 0.72. We need to somehow narrow that range to obtain a more informative probability estimation.

3.2.3 Shrinking the range of a high confidence interval

How can we taper down our range while maintaining a 95% confidence interval? Perhaps we should try elevating the frequency count from 500 to something noticeably larger. Previously, we've sampled 500 frequencies, where each frequency represented 1,000 biased coin flips. Instead, let's sample 100,000 frequencies while keeping the coin-flip count constant at 1,000.

Listing 3.29 Sampling 100,000 frequencies

```
np.random.seed(0)
head_count_array = np.random.binomial(1000, 0.7, 100000)
frequency_array = head_count_array / 1000
assert frequency_array.size == 100000
```

We will recompute the histogram on the updated frequency_array, which now holds 200-fold more frequency elements. Then we visualize that histogram while also searching for a high confidence interval. Let's incorporate the confidence interval into our visualization by coloring the histogram bars in its range (figure 3.5). The histogram

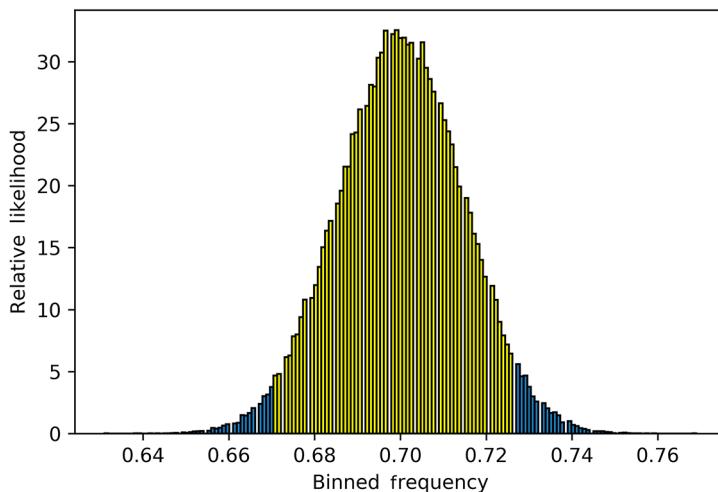


Figure 3.5 A histogram of 100,000 binned frequencies plotted against their associated relative likelihoods. Highlighted bars delineate the 95% confidence interval, which represents 95% of the histogram's area. That interval covers a frequency range of roughly 0.670–0.727.

bars can be visually modified by relying on patches, which is the third variable returned by plt.hist. The graphical details of each bin at index i are accessible through `patches[i]`. If we wish to color the i th bin yellow, we can simply call `patches[i].set_facecolor('yellow')`. In this manner, we can highlight all the specified histogram bars that fall within the updated interval range.

Listing 3.30 Coloring histogram bars over an interval

```
likelihoods, bin_edges, patches = plt.hist(frequency_array, bins='auto',
                                           edgecolor='black', density=True)
bin_width = bin_edges[1] - bin_edges[0]
start_index, end_index = compute_high_confidence_interval(likelihoods,
                                                          bin_width)

for i in range(start_index, end_index):
    patches[i].set_facecolor('yellow')
plt.xlabel('Binned Frequency')
plt.ylabel('Relative Likelihood')

plt.show()
```

The frequency range 0.670429 – 0.727857 represents a 95.42% confidence interval

The recomputed histogram resembles a symmetric bell-shaped curve. Many of its bars have been highlighted using the `set_facecolor` method. The highlighted bars represent a 95% confidence interval. The interval covers a frequency range of roughly

0.670 to 0.727. This new frequency range is nearly identical to the one we saw before: increasing the frequency sample size did not reduce the range. Perhaps we should also increase the number of coin flips per frequency sample from 1,000 to 50,000 (figure 3.6). We keep the frequency sample size steady at 100,000, thus leading to 5 billion flipped coins.

Listing 3.31 Sampling 5 billion flipped coins

```
np.random.seed(0)
head_count_array = np.random.binomial(50000, 0.7, 100000)
frequency_array = head_count_array / 50000

likelihoods, bin_edges, patches = plt.hist(frequency_array, bins='auto',
                                           edgecolor='black', density=True)
bin_width = bin_edges[1] - bin_edges[0]
start_index, end_index = compute_high_confidence_interval(likelihoods,
                                                          bin_width)

for i in range(start_index, end_index):
    patches[i].set_facecolor('yellow')
plt.xlabel('Binned Frequency')
plt.ylabel('Relative Likelihood')

plt.show()
```

The frequency range 0.695769 - 0.703708 represents a 95.06% confidence interval

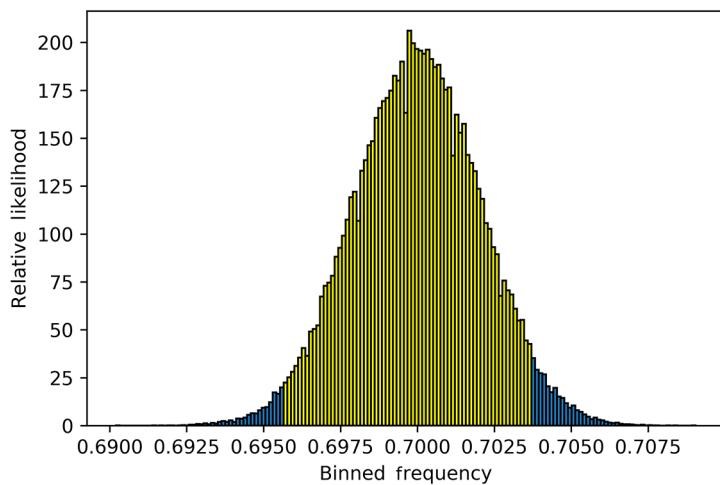


Figure 3.6 A histogram of 100,000 binned frequencies plotted against their associated relative likelihoods. Highlighted bars delineate the 95% confidence interval, which represents 95% of the histogram's area. That interval covers a frequency range of roughly 0.695–0.703.

The new 95.06% confidence interval covers a frequency range of roughly 0.695 to 0.703. If we round the range to two decimal places, it equals 0.70 to 0.70. We are thus exceedingly confident that our true probability is approximately 0.70. By increasing the coin flips per sample, we've successfully narrowed the range of our 95% confidence interval.

On a separate note, our updated histogram once again resembles a bell-shaped curve. That curve is referred to as either the *Gaussian distribution* or the *normal distribution*. The normal distribution is incredibly important to probability theory and statistics due to the *central limit theorem*. According to this theorem, sampled frequency distributions take the shape of a normal distribution when the number of samples is large. Furthermore, the theorem predicts a narrowing of likely frequencies as the size of each frequency sample increases. This is perfectly consistent with our observations, which are summarized here:

- 1 Initially, we sampled 1,000 coin flips 500 times.
- 2 Each sequence of 1,000 coin flips was converted to a frequency.
- 3 We plotted the histogram of 500 frequencies representing 50,000 total coin flips.
- 4 The histogram shape was not symmetric. It peaked at approximately 0.7.
- 5 We increased the frequency count from 500 to 100,000.
- 6 We plotted the histogram of 100,000 frequencies representing 1 million total coin flips.
- 7 The new histogram's shape resembled a normal curve. It continued peaking at 0.7.
- 8 We summed the rectangular area of bins around the peak. We stopped once the added bins covered 95% of the area under the histogram.
- 9 These bins represented a frequency range of approximately 0.670–0.723.
- 10 We increased the coin flips per sample from 1,000 to 50,000.
- 11 We plotted the histogram of 100,000 frequencies representing 5 billion total coin flips.
- 12 The updated histogram's shape continued to resemble a normal curve.
- 13 We recomputed the range covering 95% of the histogram's area.
- 14 The range width shrank to approximately 0.695–0.703.
- 15 Thus, when we increased our per-frequency flip count, the range of likely frequencies began to narrow at around 0.7.

3.2.4 Computing histograms in NumPy

Calling the `plt.hist` method automatically generates a histogram plot. Can we obtain the histogram likelihoods and bin edges without creating a plot? Yes, because `plt.hist` uses NumPy's non-visual `np.histogram` function. This function takes as input all parameters that don't relate to histogram visualization, such as `frequency_arrays`, `bins='auto'`, and `density=True`. It then returns two variables not associated with plot

manipulation: likelihoods and bin_edges. Therefore, we can run `compute_high_confidence_interval` while bypassing visualization simply by calling `np.histogram`.

Listing 3.32 Computing a histogram using `np.histogram`

```
np.random.seed(0)                                We no longer store the start and end index variables
                                                    returned by this function since there is no longer a need
                                                    to highlight an interval range in a histogram plot.

likelihoods, bin_edges = np.histogram(frequency_array, bins='auto',
                                      density=True)
bin_width = bin_edges[1] - bin_edges[0]
compute_high_confidence_interval(likelihoods, bin_width)
```

The frequency range 0.695769 - 0.703708 represents a 95.06% confidence interval

Useful histogram functions

- `plt.hist(data, bins=10)`—Plots a histogram in which the elements of data are distributed across 10 equally spaced bins.
- `plt.hist(data, bins='auto')`—Plots a histogram whose bin count is determined automatically, based on the data distribution. `auto` is the default setting of `_bins`.
- `plt.hist(data, edges='black')`—In the plotted histogram, the edges of each bin are marked by black vertical lines.
- `counts, _, _ = plt.hist(data)`—The `counts` array is the first of the three variables returned by `plt.hist`. It holds the count of elements contained in each bin. These counts appear on the y-axis of the histogram plot.
- `_, bin_edges, _ = plt.hist(data)`—The `bin_edges` array is the second of the three variables returned by `plt.hist`. It holds the x-axis positions of the vertical bin edges in the plot. Subtracting `bin_edges[i]` from `bin_edges[i + 1]` returns the width of every bin. Multiplying the width by `counts[i]` returns the area of the rectangular bin at position `i`.
- `likelihoods, _, _ = plt.hist(data, density=True)`—The binned counts are transformed into likelihoods so that the area beneath the histogram sums to 1.0. Thus, the histogram is transformed into a probability distribution. Multiplying the bin width by `likelihoods[i]` returns the probability of a random outcome falling within a range of `bin_edges[i] - bin_edges[i + 1]`.
- `_, _, patches = plt.hist(data)`—The `patches` list is the third of the three variables returned by `plt.hist`. The graphical settings of each bin at index `i` are stored in `patches[i]`. Calling `patches[i].set_facecolor('yellow')` changes the color of the histogram bin at position `i`.
- `likelihoods, bin_edges = np.histogram(data, density=True)`—Returns the histogram likelihoods and bin edges without actually plotting the results.

3.3 Using confidence intervals to analyze a biased deck of cards

Suppose you're shown a biased 52-card deck. Each card is either red or black, but the color counts aren't equal. How many red cards are present in the deck? You could find out by counting all the red cards one by one, but that would be too easy. Let's add a constraint to make the problem more interesting. You are only allowed to see the first card in the deck! If you wish to see a new card, you must first reshuffle. You're permitted to reshuffle as many times as you like and to view the top card after each shuffle.

Given these constraints, we must solve the problem using random sampling. Let's begin by modeling a 52-card deck with an unknown number of red cards. That red count is an integer between 0 and 52, which we can generate using `np.random.randint`. We'll keep the value of our random `red_card_count` value hidden until we've found a solution using sampling.

Listing 3.33 Generating a random red card count

```
np.random.seed(0)
total_cards = 52
red_card_count = np.random.randint(0, total_cards + 1)
```

Now let's compute `black_card_count` by using the constraint that `red_card_count` and `black_card_count` must sum to 52 cards total. We also maintain bias by ensuring that the two counts are not equal.

Listing 3.34 Generating a black card count

```
black_card_count = total_cards - red_card_count
assert black_card_count != red_card_count
```

During the modeling phase, we'll shuffle the deck and flip over the first card. What is the probability the card will be red? Well, a red card represents one of two possible outcomes: red or black. These outcomes can be characterized by the sample space `{'red_card', 'black_card'}`, but only when the two outcomes are equally likely. However, in our biased deck, the outcomes are weighted by `red_card_count` and `black_card_count`. A weighted sample space dict is therefore required, in which the dictionary values equal the count variables. We label the associated keys `'red_card'` and `'black_card'`. Passing `weighted_sample_space` into `compute_event_probability` will allow us to compute the probability of drawing a red card.

Listing 3.35 Computing card probabilities using a sample space

```
weighted_sample_space = {'red_card': red_card_count,
                        'black_card': black_card_count}
prob_red = compute_event_probability(lambda x: x == 'red_card',
                                      weighted_sample_space)
```

As a reminder, the `compute_event_probability` function divides the `red_card_count` variable by the sum of `red_card_count` and `black_card_count` to compute the probability. Furthermore, the sum of `red_card_count` and `black_card_count` is equal to `total_cards`. Therefore, the probability of drawing a red card is equal to `red_card_count` divided by `total_cards`. Let's verify that.

Listing 3.36 Computing card probabilities using division

```
assert prob_red == red_card_count / total_cards
```

How should we utilize `prob_red` to model a flipped-over first card? Well, the card flip will produce one of two possible outputs: red or black. These two outcomes can be modeled as coin flips in which heads and tails are replaced by colors. Therefore, we can model the flipped card using the binomial distribution. Calling `np.random.binomial(1, prob_red)` returns 1 if the first card is red and 0 otherwise.

Listing 3.37 Simulating a random card

```
np.random.seed(0)
color = 'red' if np.random.binomial(1, prob_red) else 'black'
print(f"The first card in the shuffled deck is {color}")
```

The first card in the shuffled deck is red

We shuffle the deck 10 times and flip over the first card after each shuffle.

Listing 3.38 Simulating 10 random cards

```
np.random.seed(0)
red_count = np.random.binomial(10, prob_red)
print(f"In {red_count} of out 10 shuffles, a red card came up first.")
```

In 8 of out 10 shuffles, a red card came up first.

A red card appeared at the top of the deck in 8 out of 10 random shuffles. Does this mean that 80% of the cards are red? Of course not. We've previously shown how such outcomes are common when the sampling size is low. Instead of shuffling the deck 10 times, let's shuffle it 50,000 times. Then we compute the frequency and then redo the shuffling procedure another 100,000 times. We execute these steps by calling `np.random.binomial(50000, prob_red, 100000)` and dividing by 50,000. The resulting frequency array can be transformed into a histogram that will allow us to compute a 95% confidence interval for flipping over a red card. We compute the confidence interval by expanding the range of bins around the histogram's peak until that range covers 95% of the histogram's area.

Listing 3.39 Computing card probability confidence intervals

```

Converts 100,000 red counts
into 100,000 frequencies
    np.random.seed(0)
    red_card_count_array = np.random.binomial(50000, prob_red, 100000)
    frequency_array = red_card_count_array / 50000

Counts the observed red
cards out of 50,000 shuffles;
repeats 100,000 times

→ likelihoods, bin_edges = np.histogram(frequency_array, bins='auto',
                                         density=True)
    bin_width = bin_edges[1] - bin_edges[0]
    start_index, end_index = compute_high_confidence_interval(likelihoods,
                                                               bin_width)

The frequency range 0.842865 - 0.849139 represents a 95.16% confidence interval

Computes the
frequency histogram
Computes the 95% confidence
interval for the histogram

```

We are very confident that `prob_red` lies between 0.842865 and 0.849139. We also know that `prob_red` equals `red_card_count / total_cards`, and therefore `red_card_count` equals `prob_red * total_cards`. Thus, we are highly confident that `red_card_count` lies between `0.842865 * total_cards` and `0.849139 * total_cards`. Let's compute the likely range of `red_card_count`. We round the end points of the range to the nearest integers because `red_card_count` corresponds to an integer value.

Listing 3.40 Estimating the red card count

```

range_start = round(0.842771 * total_cards)
range_end = round(0.849139 * total_cards)
print(f"The number of red cards in the deck is between {range_start} and
{range_end}")

```

The number of red cards in the deck is between 44 and 44

We are very confident that there are 44 red cards in the deck. Let's check if our solution is correct.

Listing 3.41 Validating the red card count

```

if red_card_count == 44:
    print('We are correct! There are 44 red cards in the deck')
else:
    print('Oops! Our sampling estimation was wrong.')

```

We are correct! There are 44 red cards in the deck

There are indeed 44 red cards in the deck. We were able to determine this without manually counting all the cards. Our use of random card-shuffle sampling and confidence interval calculations proved sufficient to uncover the solution.

3.4 Using permutations to shuffle cards

Card shuffling requires us to randomly reorder the elements of a card deck. That random reordering can be carried out using the `np.random.shuffle` method. The function takes as input an ordered array or list and shuffles its elements in place. The following code randomly shuffles a deck of cards containing two red cards (represented by 1s) and two black cards (represented by 0s).

Listing 3.42 Shuffling a four-card deck

```
np.random.seed(0)
card_deck = [1, 1, 0, 0]
np.random.shuffle(card_deck)
print(card_deck)

[0, 0, 1, 1]
```

The `shuffle` method has rearranged the elements in `card_deck`. If we prefer to carry out the shuffle while retaining a copy of the original unshuffled deck, we can do so using `np.random.permutation`. The method returns a NumPy array containing a random ordering of cards. Meanwhile, the elements of the original inputted deck remain unchanged.

Listing 3.43 Returning a copy of the shuffled deck

```
np.random.seed(0)
unshuffled_deck = [1, 1, 0, 0]
shuffled_deck = np.random.permutation(unshuffled_deck)
assert unshuffled_deck == [1, 1, 0, 0]
print(shuffled_deck)

[0 0 1 1]
```

The random ordering of elements returned by `np.random.permutation` is mathematically called a *permutation*. Random permutations vary from the original ordering most of the time. On rare occasions, they may equal the original, unshuffled permutation. What is the probability that a shuffled permutation will exactly equal `unshuffled_deck`?

We can of course find out through sampling. However, the four-element deck is small enough to be analyzed using sample spaces. Composing the sample space requires us to cycle through all possible permutations of the deck. We can do so using the `itertools.permutations` function. Calling `itertools.permutations(unshuffled_deck)` will return an iterable over every possible permutation of the deck. Let's use the function to output the first three permutations. Note that these permutations are printed as Python tuples, not as arrays or lists. Tuples, unlike arrays or lists, cannot be modified in place: they are represented using parentheses.

Listing 3.44 Iterating over card permutations

```
import itertools
for permutation in list(itertools.permutations(unshuffled_deck))[:3]:
    print(permutation)

(1, 1, 0, 0)
(1, 1, 0, 0)
(1, 0, 1, 0)
```

The first two generated permutations are identical to each other. Why is that the case? Well, the first permutation is just the original unshuffled_deck with no rearranged elements. Meanwhile, the second permutation was generated by swapping the third and fourth elements of the first permutation. However, both those elements were zeros, so the swap did not impact the list. We can confirm that the swap actually took place by examining the first three permutations of [0, 1, 2, 3].

Listing 3.45 Monitoring permutation swaps

```
for permutation in list(itertools.permutations([0, 1, 2, 3]))[:3]:
    print(permutation)

(0, 1, 2, 3)
(0, 1, 3, 2)
(0, 2, 1, 3)
```

Certain permutations of the four-card deck occur more than once. Thus, we can hypothesize that certain permutations might occur more frequently than others. Let's check this hypothesis by storing the permutation counts in a weighted_sample_space dictionary.

Listing 3.46 Computing permutation counts

```
weighted_sample_space = defaultdict(int)
for permutation in itertools.permutations(unshuffled_deck):
    weighted_sample_space[permutation] += 1

for permutation, count in weighted_sample_space.items():
    print(f"Permutation {permutation} occurs {count} times")

Permutation (1, 1, 0, 0) occurs 4 times
Permutation (1, 0, 1, 0) occurs 4 times
Permutation (1, 0, 0, 1) occurs 4 times
Permutation (0, 1, 1, 0) occurs 4 times
Permutation (0, 1, 0, 1) occurs 4 times
Permutation (0, 0, 1, 1) occurs 4 times
```

All the permutations occur with equal frequency. Consequently, all card arrangements are equally likely, and a weighted sample space is not required. An unweighted sample

space equal to `set(itertools.permutations(unshuffled_deck))` should sufficiently resolve the problem.

Listing 3.47 Computing permutation probabilities

Defines a lambda function that takes as input some `x` and returns True if `x` equals our unshuffled deck. This one-line lambda function serves as our event condition.

The unweighted sample space equals the set of all the unique permutations of the deck.

```
sample_space = set(itertools.permutations(unshuffled_deck))
event_condition = lambda x: list(x) == unshuffled_deck
prob = compute_event_probability(event_condition, sample_space)
assert prob == 1 / len(sample_space)
print(f"Probability that a shuffle does not alter the deck is {prob}")
```

Probability that a shuffle does not alter the deck is 0.1666666666666666

Computes the probability of observing an event that satisfies our event condition

Suppose we are handed a generic `unshuffled_deck` of size N containing $N/2$ red cards. Mathematically, it can be shown that all the color permutations of the deck will occur with equal likelihood. Thus, we can compute probabilities directly using the deck's unweighted sample space. Unfortunately, creating this sample space is not feasible for a 52-card deck since the number of possible permutations is astronomically large: 8.06×10^{67} , which is larger than the number of atoms on Earth. If we attempted to compute a 52-card sample space, our program would run for many days before eventually running out of memory. However, such a sample space can easily be computed for a smaller 10-card deck.

Listing 3.48 Computing a 10-card sample space

```
red_cards = 5 * [1]
black_cards = 5 * [0]
unshuffled_deck = red_cards + black_cards
sample_space = set(itertools.permutations(unshuffled_deck))
print(f"Sample space for a 10-card deck contains {len(sample_space)} elements")
```

Sample space for a 10-card deck contains 252 elements

We have been tasked with finding the best strategy for drawing a red card. The 10-card `sample_space` set could prove useful in these efforts: the set allows us to compute the probabilities of various competing strategies directly. We can thus rank our strategies based on their 10-card deck performance and then apply the top-ranking strategies to a 52-card deck.

Summary

- The `np.random.binomial` method can simulate random coin flips. The method gets its name from the *binomial distribution*, which is a generic distribution that captures coin-flip probabilities.
- When a coin is flipped repeatedly, its frequency of heads converges toward the actual probability of heads. However, the final frequency may differ slightly from the actual probability.
- We can visualize the variability of recorded coin-flip frequencies by plotting a *histogram*. A histogram shows binned counts of observed numeric values. The counts can be transformed into relative likelihoods so that the area beneath the histogram sums to 1.0. Effectively, the transformed histogram becomes a probability distribution. The area around the distribution's peak represents a *confidence interval*. A confidence interval is the likelihood that an unknown probability falls within a certain frequency range. Generally, we prefer a confidence interval that is at 95% or higher.
- The shape of a frequency histogram resembles a bell-shaped curve when the number of sampled frequencies is high. That curve is referred to as either the *Gaussian distribution* or the *normal distribution*. According to the *central limit theorem*, the 95% confidence interval associated with the bell curve becomes narrower as the size of each frequency sample increases.
- Simulated card shuffles can be carried out using the `np.random.permutation` method. This method returns a random permutation of the inputted deck of cards. The *permutation* represents a random ordering of card elements. We can iterate over every possible permutation by calling `itertools.permutations`. Iterating over all the permutations for a 52-card deck is computationally impossible. However, we can easily capture all the permutations of a smaller 10-card deck. These permutations can be used to compute the small deck's sample space.

Case study 1 solution

This section covers

- Card game simulations
- Probabilistic strategy optimization
- Confidence intervals

Our aim is to play a card game in which the cards are iteratively flipped until we tell the dealer to stop. Then one additional card is flipped. If that card is red, we win a dollar; otherwise, we lose a dollar. Our goal is to discover a strategy that best predicts a red card in the deck. We will do so by

- 1 Developing multiple strategies for predicting red cards in a randomly shuffled deck.
- 2 Applying each strategy across multiple simulations to compute its probability of success within a high confidence interval. If these computations prove to be intractable, we will instead focus on those strategies that perform best across a 10-card sample space.
- 3 Returning the simplest strategy associated with the highest probability of success.

WARNING Spoiler alert! The solution to case study 1 is about to be revealed. I strongly encourage you to try to solve the problem prior to reading the solution. The original problem statement is available for reference at the beginning of the case study.

4.1 Predicting red cards in a shuffled deck

We start by creating a deck holding 26 red cards and 26 black cards. Black cards are represented by 0s, and red cards are represented by 1s.

Listing 4.1 Modeling a 52-card deck

```
red_cards = 26 * [1]
black_cards = 26 * [0]
unshuffled_deck = red_cards + black_cards
```

We proceed to shuffle the deck.

Listing 4.2 Shuffling a 52-card deck

```
np.random.seed(1)
shuffled_deck = np.random.permutation(unshuffled_deck)
```

Now we iteratively flip over the cards in the deck, stopping when the next card is more likely to be red. Then we flip over the next card. We win if that card is red.

How do we decide when we should stop? One simple strategy is to terminate the game when the number of red cards remaining in the deck is greater than the number of black cards remaining in the deck. Let's execute that strategy on the shuffled deck.

Listing 4.3 Coding a card game strategy

```
remaining_red_cards = 26
for i, card in enumerate(shuffled_deck[:-1]):
    remaining_red_cards -= card
    remaining_total_cards = 52 - i - 1      ←
    if remaining_red_cards / remaining_total_cards > 0.5:
        break

print(f"Stopping the game at index {i}.")
final_card = shuffled_deck[i + 1]
color = 'red' if final_card else 0
print(f"The next card in the deck is {'red' if final_card else 'black'}.")
print(f"We have {'won' if final_card else 'lost'}!")
```

Stopping the game at index 1.
The next card in the deck is red.
We have won!

Subtracts the total cards seen thus far from 52. This total equals $i + 1$, since i is initially set to zero.
Alternatively, we can run `enumerate(shuffled_deck[:-1], 1)` so that i is initially set to 1.

The strategy yielded a win on our very first try. Our strategy halts when the fraction of remaining red cards is greater than half of the remaining total cards. We can generalize that fraction to equal a `min_red_fraction` parameter, thus halting when the red card ratio is greater than the inputted parameter value. This generalized strategy is implemented next with `min_red_fraction` preset to 0.5.

Listing 4.4 Generalizing the card game strategy

```
np.random.seed(0)
total_cards = 52
total_red_cards = 26
def execute_strategy(min_fraction_red=0.5, shuffled_deck=None,
                     return_index=False):
    if shuffled_deck is None:
        shuffled_deck = np.random.permutation(unshuffled_deck) ←
            Shuffles the
            unshuffled
            deck if no
            input deck
            is provided

    remaining_red_cards = total_red_cards

    for i, card in enumerate(shuffled_deck[:-1]):
        remaining_red_cards -= card
        fraction_red_cards = remaining_red_cards / (total_cards - i - 1)
        if fraction_red_cards > min_fraction_red:
            break

    return (i+1, shuffled_deck[i+1]) if return_index else shuffled_deck[i+1] ←
        Optionally returns the card
        index along with the final card
```

4.1.1 Estimating the probability of strategy success

Let's apply our basic strategy to a series of 1,000 random shuffles.

Listing 4.5 Running the strategy over 1,000 shuffles

```
observations = np.array([execute_strategy() for _ in range(1000)])
```

The total fraction of 1s in `observations` corresponds to the observed fraction of red cards and therefore to the fraction of wins. We can compute this fraction by summing the 1s in `observations` and dividing by the array size. As an aside, that computation can also be carried out by calling `observations.mean()`.

Listing 4.6 Computing the frequency of wins

```
frequency_wins = observations.sum() / 1000
assert frequency_wins == observations.mean()
print(f"The frequency of wins is {frequency_wins}")
```

```
The frequency of wins is 0.511
```

We've won 51.1% of the total games! Our strategy appears to be working: 511 wins and 489 losses will net us a total profit of \$22.

Listing 4.7 Computing total profit

```
dollars_won = frequency_wins * 1000
dollars_lost = (1 - frequency_wins) * 1000
total_profit = dollars_won - dollars_lost
print(f"Total profit is ${total_profit:.2f}")
```

Total profit is \$22.00

The strategy worked well for a sample size of 1,000 shuffles. We now plot the strategy's win-frequency convergence over a series of sample sizes ranging from 1 through 10,000 (figure 4.1).

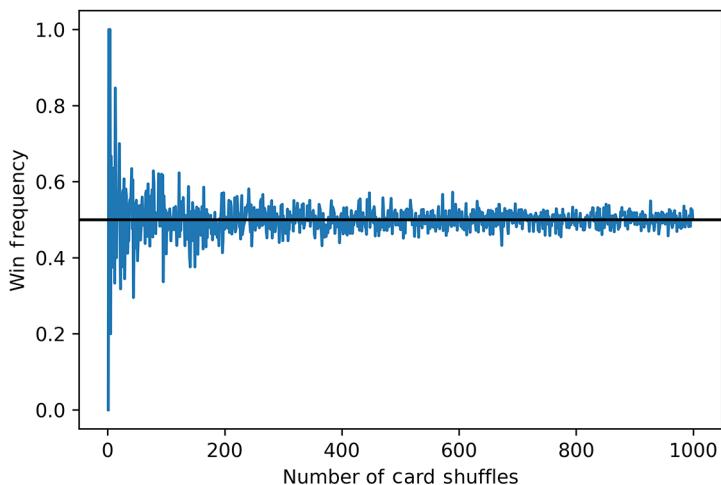


Figure 4.1 The number of played games plotted against the observed win-count frequency. The frequencies fluctuate around a value of 0.5. We cannot tell if the probability of winning is above or below 0.5.

Listing 4.8 Plotting simulated frequencies of wins

```
np.random.seed(0)
def repeat_game(number_repeats):
    observations = np.array([execute_strategy()
                            for _ in range(number_repeats)])
    return observations.mean()

frequencies = []
for i in range(1, 1000):
    frequencies.append(repeat_game(i))
```

↳ Returns the frequency of wins for
a specified number of games

```

plt.plot(list(range(1, 1000)), frequencies)
plt.axhline(0.5, color='k')
plt.xlabel('Number of Card Shuffles')
plt.ylabel('Win-Frequency')
plt.show()
print(f"The win-frequency for 10,000 shuffles is {frequencies[-1]}")

```

The win-frequency for 10,000 shuffles is 0.5035035035035035

The strategy yields a win frequency of over 50% when 10,000 card shuffles are sampled. However, the strategy also fluctuates above and below 50% throughout the entire sampling process. How confident are we that the probability of a win is actually greater than 0.5? We can find out using confidence interval analysis (figure 4.2). We compute the confidence interval by sampling 10,000 card shuffles 300 times, for a total of 3 million shuffles. Shuffling an array is a computationally expensive procedure, so listing 4.9 takes approximately 40 seconds to run.

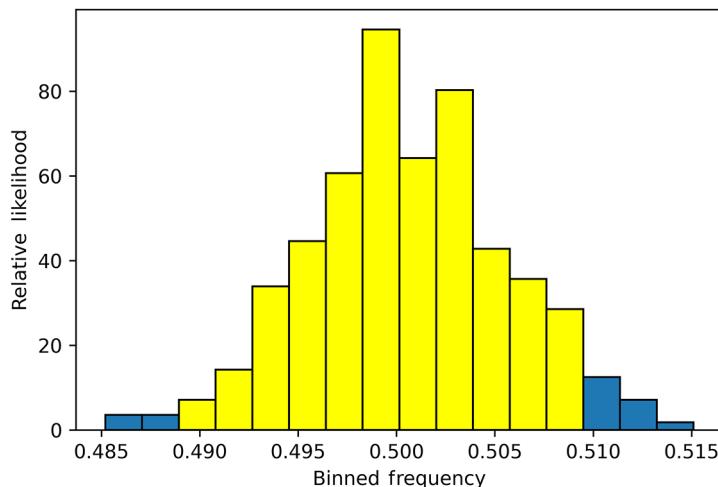


Figure 4.2 A histogram of 300 binned frequencies plotted against their associated relative likelihoods. Highlighted bars delineate the 95% confidence interval. That interval covers a frequency range of roughly 0.488–0.508.

Listing 4.9 Computing the confidence interval for 3 million shuffles

```

np.random.seed(0)
frequency_array = np.array([repeat_game(10000) for _ in range(300)])

likelihoods, bin_edges, patches = plt.hist(frequency_array, bins='auto',
                                           edgecolor='black', density=True)
bin_width = bin_edges[1] - bin_edges[0]

```

```

start_index, end_index = compute_high_confidence_interval(likelihoods,
    bin_width) ←
for i in range(start_index, end_index):
    patches[i].set_facecolor('yellow')
plt.xlabel('Binned Frequency')
plt.ylabel('Relative Likelihood')

plt.show()

The frequency range 0.488938 - 0.509494 represents a 97.00% confidence
interval

```

As a reminder, we defined the `compute_high_confidence_interval` function in section 3.

We are quite confident that the actual probability lies somewhere between 0.488 and 0.509. However, we still don't know whether that probability is above 0.5 or below 0.5. This is a problem: even a minor misinterpretation of the true probability could cause us to lose money.

Imagine that the true probability is 0.5001. If we apply our strategy to 1 billion shuffles, we should expect to win \$200,000. Now suppose we were wrong, and the actual probability is 0.4999. In this scenario, we will lose \$200,000. A tiny error over the fourth decimal space could cost us hundreds of thousands of dollars.

We must be absolutely certain that the true probability lies above 0.5. Thus, we must narrow the 95% confidence interval by increasing the sample size at the expense of running time. The following code samples 50,000 shuffles over 3,000 iterations. It takes approximately an hour to run.

WARNING The following code will take an hour to run.

Listing 4.10 Computing the confidence interval for 150 million shuffles

```

np.random.seed(0)

frequency_array = np.array([repeat_game(50000) for _ in range(3000)])
likelihoods, bin_edges = np.histogram(frequency_array, bins='auto',
                                       density=True)
bin_width = bin_edges[1] - bin_edges[0]
compute_high_confidence_interval(likelihoods, bin_width)

The frequency range 0.495601 - 0.504345 represents a 96.03% confidence
interval

```

We've executed our sampling. Unfortunately, the new confidence interval still does not discern whether the true probability lies above 0.5. So what should we do? Increasing the number of samples is not computationally feasible (unless we're willing to let the simulation run for a couple of days). Perhaps increasing `min_red_fraction` from 0.5 to 0.75 will yield an improvement. Let's update our strategy and go for a long walk as our simulation takes another hour to run.

WARNING The following code will take an hour to run.

Listing 4.11 Computing the confidence interval for an updated strategy

```

np.random.seed(0)
def repeat_game(number_repeats, min_red_fraction):
    observations = np.array([execute_strategy(min_red_fraction)
                             for _ in range(number_repeats)])
    return observations.mean()

frequency_array = np.array([repeat_game(50000, 0.75) for _ in range(3000)])
likeliabilities, bin_edges = np.histogram(frequency_array, bins='auto',
                                         density=True)
bin_width = bin_edges[1] - bin_edges[0]
compute_high_confidence_interval(likeliabilities, bin_width)

The frequency range 0.495535 - 0.504344 represents a 96.43% confidence
interval

```

Nope! The span of our confidence interval remains unresolved since it still covers both profitable and unprofitable probabilities.

Perhaps we can gain more insight by applying our strategies to a 10-card deck. That deck's sample space can be explored in its entirety, thus letting us compute the exact probability of a win.

4.2 Optimizing strategies using the sample space for a 10-card deck

The following code computes the sample space for a 10-card deck. Then it applies our basic strategy to that sample space. The final output is the probability that the strategy will yield a win.

Listing 4.12 Applying a basic strategy to a 10-card deck

```

total_cards = 10
total_red_cards = int(total_cards / 2)
total_black_cards = total_red_cards
unshuffled_deck = [1] * total_red_cards + [0] * total_black_cards
sample_space = set(itertools.permutations(unshuffled_deck))
win_condition = lambda x: execute_strategy(shuffled_deck=np.array(x))
prob_win = compute_event_probability(win_condition, sample_space)
print(f"Probability of a win is {prob_win}")

Probability of a win is 0.5

```

As a reminder, `itertools` was previously imported in section 3.

We defined the `compute_event_probability` function in section 1.

Event condition where our basic strategy yields a win

Surprisingly, our basic strategy yields a win only 50% of the time. This is no better than selecting the first card at random! Maybe our `min_red_fraction` parameter was insufficiently low. We can find out by sampling all the two-decimal `min_red_fraction` values between 0.50 and 1.0. The following code computes the win probabilities over a range of `min_red_fraction` values and returns the minimum and maximum probabilities.

Listing 4.13 Applying multiple strategies to a 10-card deck

```

def scan_strategies():
    fractions = [value / 100 for value in range(50, 100)]
    probabilities = []
    for frac in fractions:
        win_condition = lambda x: execute_strategy(frac,
                                                    shuffled_deck=np.array(x))
        probabilities.append(compute_event_probability(win_condition,
                                                        sample_space))
    return probabilities

probabilities = scan_strategies()
print(f"Lowest probability of win is {min(probabilities)}")
print(f"Highest probability of win is {max(probabilities)}")

Lowest probability of win is 0.5
Highest probability of win is 0.5

```

Both the lowest and highest probabilities are equal to 0.5! None of our strategies have outperformed a random card choice. Perhaps adjusting the deck size will yield some improvement. Let's analyze the sample spaces of decks containing two, four, six, and eight cards. We apply all strategies to each sample space and return their probabilities of winning. Then we search for a probability that isn't equal to 0.5.

Listing 4.14 Applying multiple strategies to multiple decks

```

for total_cards in [2, 4, 6, 8]:
    total_red_cards = int(total_cards / 2)
    total_black_cards = total_red_cards
    unshuffled_deck = [1] * total_red_cards + [0] * total_black_cards

    sample_space = set(itertools.permutations(unshuffled_deck))
    probabilities = scan_strategies()
    if all(prob == 0.5 for prob in probabilities):
        print(f"No winning strategy found for deck of size {total_cards}")
    else:
        print(f"Winning strategy found for deck of size {total_cards}")

No winning strategy found for deck of size 2
No winning strategy found for deck of size 4
No winning strategy found for deck of size 6
No winning strategy found for deck of size 8

```

All of the strategies yield a probability of 0.5 across the small decks. Each time we increase the deck size, we add two additional cards to the deck, but this fails to improve performance. A strategy that fails on a 2-card deck continues to fail on a 4-card deck, and a strategy that fails on an 8-card deck continues to fail on a 10-card deck. We can extrapolate this logic even further. A strategy that fails on a 10-card deck will probably fail on a 12-card deck, and thus on a 14-card deck and a 16-card deck.

Eventually, it will fail on a 52-card deck. Qualitatively, this inductive argument makes sense. Mathematically, it can be proven to be true. Right now, we don't need to concern ourselves with the math. What's important is that our instincts have been proven wrong. Our strategies don't work on a 10-card deck, and we have little reason to believe they will work on a 52-card deck. Why do the strategies fail?

Intuitively, our initial strategy made sense: if there are more red cards than black cards in the deck, then we are more likely to pick a red card from the deck. However, we failed to take into account those scenarios when the red cards never outnumber the black cards. For instance, suppose the first 26 cards are red and the remainder are black. In these circumstances, our strategies will fail to halt, and we will lose. Also, let's consider a shuffled deck where the first 25 cards are red, the next 26 cards are black, and the final card is red. Here, our strategy will fail to halt, but we will still win. It seems each strategy can lead to one of four scenarios:

- Strategy halts and the next card is red. We win.
- Strategy halts and the next card is black. We lose.
- Strategy doesn't halt and the final card is red. We win.
- Strategy doesn't halt and the final card is black. We lose.

Let's sample how frequently the four scenarios occur across 50,000 card shuffles. We record these frequencies over our range of two-digit `min_red_fraction` values. We then plot each `min_red_fraction` value against the occurrence rates observed from the four scenarios (figure 4.3).

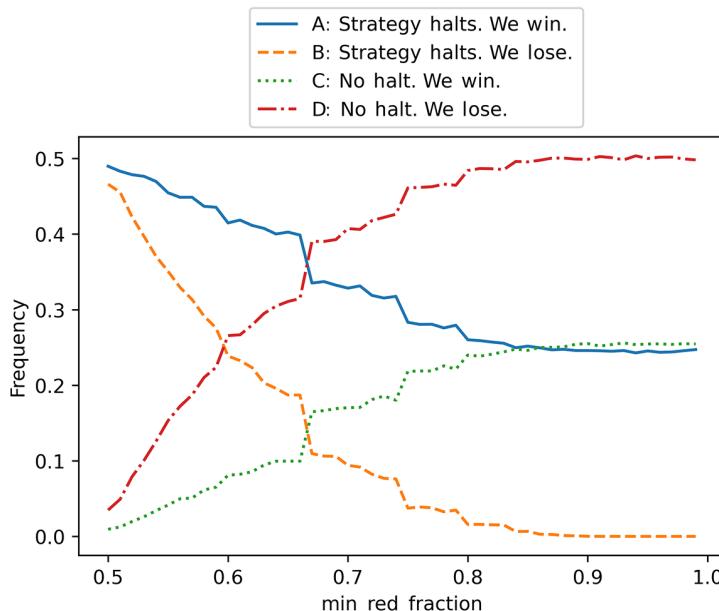


Figure 4.3 The `min_red_fraction` parameter is plotted against the sampled frequencies for all four possible scenarios. Scenario A initially has a frequency of roughly 0.49, but eventually it drops to 0.25. Scenario C has a frequency of roughly 0.01, but eventually it increases to 0.25. The frequency sums for A and C remain at approximately 0.5, thus reflecting a 50% chance of winning the game.

Listing 4.15 Plotting strategy outcomes across a 52-card deck

```

np.random.seed(0)
total_cards = 52
total_red_cards = 26
unshuffled_deck = red_cards + black_cards

def repeat_game_detailed(number_repeats, min_red_fraction):
    observations = [execute_strategy(min_red_fraction, return_index=True)
                    for _ in range(num_repeats)]
    successes = [index for index, card, in observations if card == 1] ←
    halt_success = len([index for index in successes if index != 51]) ←
    no_halt_success = len(successes) - halt_success

    failures = [index for index, card, in observations if card == 0] ←
    halt_failure = len([index for index in failures if index != 51]) ←
    no_halt_failure = len(failures) - halt_failure
    result = [halt_success, halt_failure, no_halt_success, no_halt_failure]
    return [r / number_repeats for r in result]

fractions = [value / 100 for value in range(50, 100)]
num_repeats = 50000
result_types = [[], [], [], []]
for fraction in fractions:   ←
    result = repeat_game_detailed(num_repeats, fraction)
    for i in range(4):
        result_types[i].append(result[i])

plt.plot(fractions, result_types[0],
         label='A) Strategy Halts. We Win.')
plt.plot(fractions, result_types[1], linestyle='--',
         label='B) Strategy Halts. We Lose.')
plt.plot(fractions, result_types[2], linestyle=':',
         label='C) No Halt. We Win.')
plt.plot(fractions, result_types[3], linestyle='-.',
         label='D) No Halt. We Lose.')
plt.xlabel('min_red_fraction')
plt.ylabel('Frequency')
plt.legend(bbox_to_anchor=(1.0, 0.5)) ←
plt.show()

```

We execute a strategy across num_repeats simulations.

Scenario where our strategy doesn't halt and we win

This list contains all instances of losses.

Scenario where our strategy halts and we win

This list contains all instances of wins.

We scan the scenario frequencies across multiple strategies.

We return the observed frequencies for all four scenarios.

Scenario where our strategy doesn't halt and we lose

Scenario where our strategy halts and we lose

The bbox_to_anchor parameter is used to position the legend above the plot to avoid overlap with the four plotted curves.

Let's examine the plot at the `min_red_fraction` value of 0.5. Here, scenario A (*Strategy Halts. We Win.*) is the most common outcome, with a frequency of approximately 0.49. Meanwhile, a halt leads to a loss approximately 46% of the time (strategy B). So why do we maintain a 50% chance of winning the game? Well, in 1% of the cases, our strategy fails to halt, but we still win (scenario C). The strategy's weakness is counterbalanced by random chance.

Within the plot, as the `min_red_fraction` goes up, the frequency of scenario A goes down. The more conservative we are, the less likely we are to stop the game

prematurely and yield a win. Meanwhile, the success rate of scenario C increases. The more conservative we are, the higher the likelihood of reaching the final card and winning by chance.

As `min_red_fraction` increases, both scenario A and scenario C converge to a frequency of 0.25. Thus the probability of a win remains at 50%. Sometimes our strategy halts, and we do win. Other times, the strategy halts, and we still lose. Any advantage that each strategy offers is automatically wiped out by these losses. However, we occasionally get lucky: our strategy fails to halt, yet we win the game. These lucky wins amend our losses, and our probability of winning stays the same. No matter what we do, our likelihood of winning remains fifty-fifty. Therefore, the optimal strategy we can offer is to pick the first card in the shuffled deck.

Listing 4.16 The optimal winning strategy

```
def optimal_strategy(shuffled_deck):  
    return shuffled_deck[0]
```

Summary

- Probabilities can be counterintuitive. Innately, we assumed that our planned card game strategy would perform better than random. However, this proved not to be the case. We must be careful when dealing with random processes. It's best to rigorously test all our intuitive assumptions prior to wagering on any future outcome.
- Sometimes, even large-scale simulations fail to find a probability within the required level of precision. However, by simplifying our problem, we can utilize sample spaces to yield insights. Sample spaces allow us to test our intuition. If our intuitive solution fails on a toy version of the problem, it is also likely to fail on the actual version of the problem.

Case study 2

Assessing online ad clicks for significance

Problem statement

Fred is a loyal friend, and he needs your help. Fred just launched a burger bistro in the city of Brisbane. The bistro is open for business, but business is slow. Fred wants to entice new customers to come and try his tasty burgers. To do this, Fred will run an online advertising campaign directed at Brisbane residents. Every weekday, between 11:00 a.m. and 1:00 p.m., Fred will purchase 3,000 ads aimed at hungry locals. Every ad will be viewed by a single Brisbane resident. The text of every ad will read, "Hungry? Try the Best Burger in Brisbane. Come to Fred's." Clicking the text will take potential customers to Fred's site. Each displayed ad will cost our friend one cent, but Fred believes the investment will be worth it.

Fred is getting ready to execute his ad campaign. However, he runs into a problem. Fred previews his ad, and its text is blue. Fred believes that blue is a boring color. He feels that other colors could yield more clicks. Fortunately, Fred's advertising software allows him to choose from 30 different colors. Is there a text color that will bring more clicks than blue? Fred decides to find out.

Fred instigates an experiment. Every weekday for a month, Fred purchases 3,000 online ads. The text of every ad is assigned to one of 30 possible colors. The advertisements are distributed evenly by color. Thus, 100 ads with the same color are viewed by 100 people every day. For example, 100 people view a blue ad, and another 100 people view a green ad. These numbers add up to 3,000 views that are distributed across the 30 colors. Fred's advertising software automatically

tracks all daily views. It also records the daily clicks associated with each of the 30 colors. The software stores this data in a table. That table holds the clicks per day and views per day for every specified color. Each table row maps a color to the views and clicks for all analyzed days.

Fred has carried out his experiment. He obtained ad-click data for all 20 weekdays of the month. That data is organized by color. Now, Fred wants to know if there is a color that draws significantly more ad clicks than blue. Unfortunately, Fred doesn't know how to properly interpret the results. He's not sure which clicks are meaningful and which clicks have occurred purely randomly. Fred is brilliant at broiling burgers but has no training in data analysis. This is why Fred has turned to you for help. Fred asks you to analyze his table and to compare the counts of daily clicks. He's searching for a color that draws significantly more ad clicks than blue. Are you willing to help Fred? If so, he's promised you free burgers for a year!

Dataset description

Fred's ad-click data is stored in the file colored_ad_click_table.csv. The .csv file extension is an acronym for *comma-separated values*. Our .csv file is a table stored as text. The table columns are separated by commas. The first line in the file contains the comma-separated labels for the columns. The first 99 characters of that line are *Color,Click Count: Day 1,View Count: Day 1,Click Count: Day 2,View Count: Day 2,Click Count: Day 3,*.

Let's briefly clarify the column labels:

- Column 1: *Color*
 - Each row in the column corresponds to one of 30 possible text colors.
- Column 2: *Click Count: Day 1*
 - The column tallies the times each colored ad was clicked on day 1 of Fred's experiment.
- Column 3: *View Count: Day 1*
 - The column tallies the times each ad was viewed on day 1 of Fred's experiment.
 - According to Fred, all daily views are expected to equal 100.
- The remaining 38 columns contain the clicks per day and views per day for the other 19 days of the experiment.

Overview

To address the problem at hand, we need to know how to do the following:

- Measure the centrality and dispersion of sampled data.
- Interpret the significance of two diverging means through p-value calculation.
- Minimize mistakes associated with misleading p-value measurements.
- Load and manipulate data stored in tables using Python.



Basic probability and statistical analysis using SciPy

This section covers

- Analyzing binomials using the SciPy library
- Defining dataset centrality
- Defining dataset dispersion
- Computing the centrality and dispersion of probability distributions

Statistics is a branch of mathematics dealing with the collection and interpretation of numeric data. It is the precursor of all modern data science. The term *statistic* originally signified “the science of the state” because statistical methods were first developed to analyze the data of state governments. Since ancient times, government agencies have gathered data pertaining to their populace. That data would be used to levy taxes and organize large military campaigns. Hence, critical state decisions depended on the quality of data. Poor record keeping could lead to potentially disastrous results. That is why state bureaucrats were very concerned by any random fluctuations in their records. Probability theory eventually tamed these fluctuations, making the randomness interpretable. Ever since then, statistics and probability theory have been closely intertwined.

Statistics and probability theory are closely related, but in some ways, they are very different. Probability theory studies random processes over a potentially infinite number of measurements. It is not bound by real-world limitations. This allows us to model the behavior of a coin by imagining millions of coin flips. In real life, flipping a coin millions of times is a pointlessly time-consuming endeavor. Surely we can sacrifice some data instead of flipping coins all day and night. Statisticians acknowledge these constraints placed on us by the data-gathering process. Real-world data collection is costly and time consuming. Every data point carries a price. We cannot survey a country's population without employing government officials. We cannot test our online ads without paying for every ad that's clicked. Thus, the size of our final dataset usually depends on the size of our initial budget. If the budget is constrained, then the data will also be constrained. This trade-off between data and resourcing lies at the heart of modern statistics. Statistics help us understand exactly how much data is sufficient to draw insights and make impactful decisions. The purpose of statistics is to find meaning in data even when that data is limited in size.

Statistics is highly mathematical and usually taught using math equations. Nevertheless, direct exposure to equations is not a prerequisite for statistical understanding. In fact, many data scientists do not write formulas when running statistical analyses. Instead, they use Python libraries such as SciPy, which handle all the complex math calculations. However, proper library usage still requires an intuitive understanding of statistical procedures. In this section, we cultivate our understanding of statistics by applying probability theory to real-world problems.

5.1 Exploring the relationships between data and probability using SciPy

SciPy, which is short for *Scientific Python*, provides many useful methods for scientific analysis. The SciPy library includes an entire module for addressing problems in probability and statistics: `scipy.stats`. Let's install the library and import the `stats` module.

NOTE Call `pip install scipy` from the command line terminal to install the SciPy library.

Listing 5.1 Importing the stats module from SciPy

```
from scipy import stats
```

The `stats` module is very useful for assessing the randomness of data. For example, in section 1, we computed the probability of a fair coin producing at least 16 heads after 20 flips. Our calculations required us to examine all possible combinations of 20 flipped coins. Then we computed the probability of observing 16 or more heads or 16 or more tails to measure the randomness of our observations. SciPy allows us to measure this probability directly using the `stats.binom_test` method. The method is named after the binomial distribution, which governs how a flipped coin might fall.

The method requires three parameters: the number of heads, the total number of coin flips, and the probability of a coin landing on heads. Let's apply the binomial test to 16 heads observed from 20 coin flips. Our output should equal the previously computed value of approximately 0.011.

NOTE SciPy and standard Python handle low-value decimal points differently. In section 1, when we computed the probability, the final value was rounded to 17 significant digits. SciPy, on the other hand, returns a value containing 18 significant digits. Thus, for consistency's sake, we round our SciPy output to 17 digits.

Listing 5.2 Analyzing extreme head counts using SciPy

```
num_heads = 16
num_flips = 20
prob_head = 0.5
prob = stats.binom_test(num_heads, num_flips, prob_head)
print(f"Probability of observing more than 15 heads or 15 tails is {prob:.17f}")

Probability of observing more than 15 heads or 15 tails is 0.01181793212890625
```

It's worth emphasizing that `stats.binom_test` did not compute the probability of observing 16 heads. Rather, it returned the probability of seeing a coin-flip sequence where 16 or more coins fell on the same face. If we want the probability of seeing exactly 16 heads, then we must utilize the `stats.binom.pmf` method. That method represents the *probability mass function* of the binomial distribution. A probability mass function maps inputted integer values to their probability of occurrence. Thus, calling `stats.binom.pmf(num_heads, num_flips, prob_heads)` returns the likelihood of a coin yielding `num_heads` number of heads. Under current settings, this equals the probability of a fair coin falling on heads 16 out of 20 times.

Listing 5.3 Computing an exact probability using `stats.binom.pmf`

```
prob_16_heads = stats.binom.pmf(num_heads, num_flips, prob_head)
print(f"The probability of seeing {num_heads} of {num_flips} heads is
{prob_16_heads}")

The probability of seeing 16 of 20 heads is 0.004620552062988271
```

We've used `stats.binom.pmf` to find the probability of seeing exactly 16 heads. However, that method is also able to compute multiple probabilities simultaneously. Multiple head-count probabilities can be processed by passing in a list of head-count values. For instance, passing `[4, 16]` returns a two-element NumPy array containing the probabilities of seeing 4 heads and 16 heads, respectively. Conceptually, the probability of seeing 4 heads and 16 tails equals the probability of seeing 4 tails and 16 heads. Thus, executing `stats.binom.pmf([4, 16], num_flips, prob_head)` should return a two-element array whose elements are equal. Let's confirm.

Listing 5.4 Computing an array of probabilities using `stats.binom.pmf`

```
probabilities = stats.binom.pmf([4, 16], num_flips, prob_head)
assert probabilities.tolist() == [prob_16_heads] * 2
```

List-passing allows us to compute probabilities across intervals. For example, if we pass `range(21)` into `stats.binom.pmf`, then the outputted array will contain all probabilities across the interval of every possible head count. As we learned in section 1, the sum of these probabilities should equal 1.0.

NOTE Summing low-value decimals is computationally tricky. Over the course of the summation, tiny errors accumulate. Due to these errors, our final summed probability will marginally diverge from 1.0 unless we round it to 14 significant digits. We do this rounding in the next listing.

Listing 5.5 Computing an interval probability using `stats.binom.pmf`

```
interval_all_counts = range(21)
probabilities = stats.binom.pmf(interval_all_counts, num_flips, prob_head)
total_prob = probabilities.sum()
print(f"Total sum of probabilities equals {total_prob:.14f}")

Total sum of probabilities equals 1.000000000000000
```

Also, as discussed in section 2, plotting `interval_all_counts` versus probabilities reveals the shape of our 20-coin-flip distribution. Thus, we can generate the distribution plot without having to iterate through possible coin-flip combinations (figure 5.1).

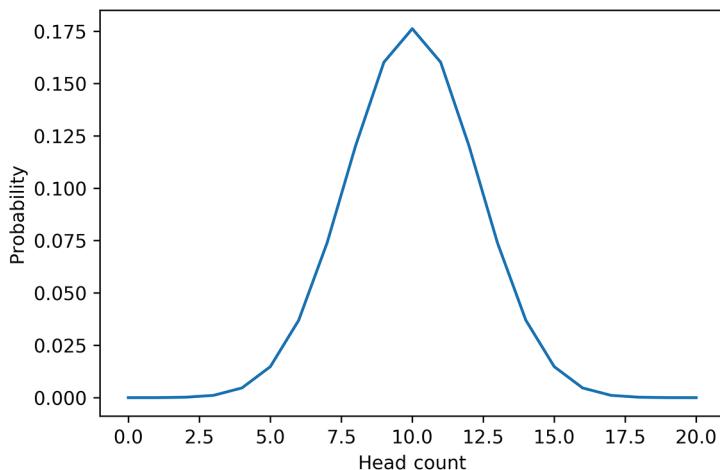


Figure 5.1 The probability distribution for 20 coin flips, generated using SciPy

Listing 5.6 Plotting a 20-coin-flip binomial distribution

```
import matplotlib.pyplot as plt
plt.plot(interval_all_counts, probabilities)
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

In section 2, our ability to visualize the binomial was limited by the total number of coin-flip combinations that we needed to compute. This is no longer the case. The `stats.binom.pmf` method lets us display any distribution associated with an arbitrary coin-flip count. Let's use our newfound freedom to simultaneously plot the distributions for 20, 80, 140, and 200 coin flips (figure 5.2).

Listing 5.7 Plotting five different binomial distributions

```
flip_counts = [20, 80, 140, 200]
linestyles = [':', '--', '-.', ':']
colors = ['b', 'g', 'r', 'k']

for num_flips, linestyle, color in zip(flip_counts, linestyles, colors):
    x_values = range(num_flips + 1)
    y_values = stats.binom.pmf(x_values, num_flips, 0.5)
    plt.plot(x_values, y_values, linestyle=linestyle, color=color,
              label=f'{num_flips} coin-flips')
plt.legend()
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

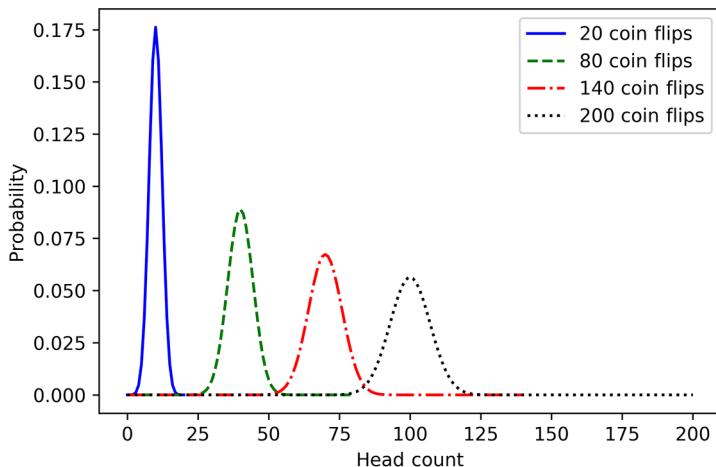


Figure 5.2 Multiple binomial probability distributions across 20, 80, 140, and 200 coin flips. The distribution centers shift right as the coin-flip count goes up. Also, every distribution becomes more dispersed around its center as the coin-flip count increases.

Within the plot, the central peak of each binomial appears to shift rightward as the coin-flip count goes up. Also, the 20-coin-flip distribution is noticeably thinner than the 200-coin-flip distribution. In other words, the plotted distributions grow more dispersed around their central positions as these central positions move to the right.

Such shifts in centrality and dispersion are commonly encountered in data analysis. We previously observed dispersion shifts in section 3, where we used randomly sampled data to visualize several histogram distributions. Subsequently, we observed that the plotted histogram thickness was dependent on our sample size. At the time, our observations were purely qualitative since we lacked a metric for comparing the thickness of two plots. However, simply noting that one plot appears thicker than another is insufficient. Likewise, stating that one plot is more rightward than another is also insufficient. We need to quantify our distribution differences. We must assign specific numbers to centrality and dispersion to discern how these numbers change from plot to plot. Doing so requires that we familiarize ourselves with the concepts of *variance* and *mean*.

5.2 Mean as a measure of centrality

Suppose we wish to study our local temperature over the first week of summer. When summer comes around, we glance at the thermometer outside our window. At noon, the temperature is exactly 80 degrees. We repeat our noon measurements over the next six days. Our measurements are 80, 77, 73, 61, 74, 79, and 81 degrees. Let's store these measurements in a NumPy array.

Listing 5.8 Storing recorded temperatures in a NumPy array

```
import numpy as np
measurements = np.array([80, 77, 73, 61, 74, 79, 81])
```

We'll now attempt to summarize our measurements using a single central value. First, we sort the measurements in place by calling `measurements.sort()`. Then, we plot the sorted temperatures in order to evaluate their centrality (figure 5.3).

Listing 5.9 Plotting the recorded temperatures

```
measurements.sort()
number_of_days = measurements.size
plt.plot(range(number_of_days), measurements)
plt.scatter(range(number_of_days), measurements)
plt.ylabel('Temperature')
plt.show()
```

Based on the plot, a central temperature exists somewhere between 60 degrees and 80 degrees. Therefore, we can naively estimate the center as approximately 70 degrees. Let's quantify our estimate as the midpoint between the lowest value and the highest value in the plot. We compute that midpoint by taking half the difference between the minimum and maximum temperatures and adding it to the minimum temperature.

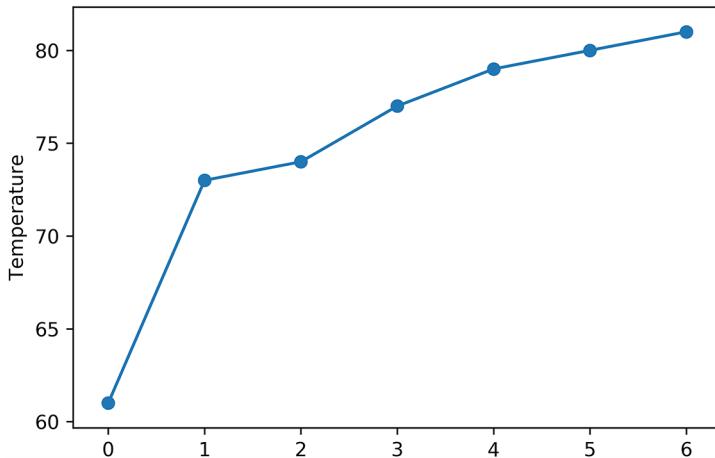


Figure 5.3 A plot containing seven sorted temperatures. A central temperature exists somewhere between 60 and 80 degrees.

(We can also obtain the same value by summing the minimum and maximum directly and dividing that sum by 2.)

Listing 5.10 Finding the midpoint temperature

```
difference = measurements.max() - measurements.min()
midpoint = measurements.min() + difference / 2
assert midpoint == (measurements.max() + measurements.min()) / 2
print(f"The midpoint temperature is {midpoint} degrees")
```

The midpoint temperature is 71.0 degrees

The midpoint temperature is 71 degrees. Let's mark that midpoint on our plot using a horizontal line. We draw the horizontal line by calling `plt.axhline(midpoint)` (figure 5.4).

Listing 5.11 Plotting the midpoint temperature

```
plt.plot(range(number_of_days), measurements)
plt.scatter(range(number_of_days), measurements)
plt.axhline(midpoint, color='k', linestyle='--')
plt.ylabel('Temperature')
plt.show()
```

Our plotted midpoint seems a little low: six of our seven measurements are higher than the midpoint. Intuitively, our central value should split the measurements more evenly—the number of temperatures above and below the center should be approximately equal. We can achieve this equality by choosing the middle element in our sorted seven-element array. The middle element, which statisticians call the *median*,

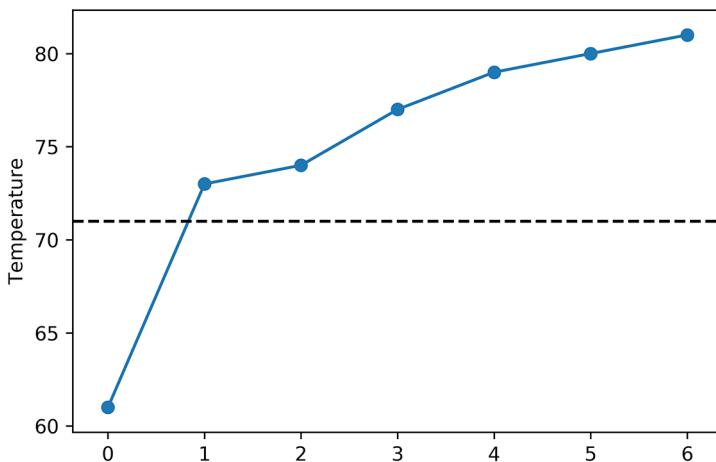


Figure 5.4 A plot containing seven sorted temperatures. A temperature of 71 degrees marks the midpoint between the highest and lowest temperatures. That midpoint seems low: six of seven temperatures are above the midpoint value.

will split our measurements into two equal parts. Three measurements will appear below the median, and three measurements will appear above it. 3 is also the index in the measurements array where the median is present. Let's add the median to our plot (figure 5.5).

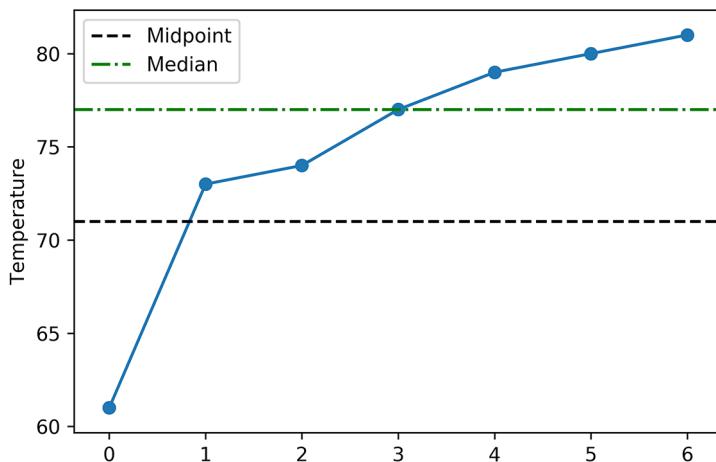


Figure 5.5 A plot containing seven sorted temperatures. A median of 77 degrees splits the temperatures in half. The median appears slightly off balance: it is closer to the three upper temperatures than to the three lower temperatures.

Listing 5.12 Plotting the median temperature

```
median = measurements[3]
print(f"The median temperature is {median} degrees")
plt.plot(range(number_of_days), measurements)
plt.scatter(range(number_of_days), measurements)
plt.axhline(midpoint, color='k', linestyle='--', label='midpoint')
plt.axhline(median, color='g', linestyle='-.', label='median')
plt.legend()
plt.ylabel('Temperature')
plt.show()
```

The median temperature is 77 degrees

Our median of 77 degrees splits the temperatures in half. However, the split is not well balanced since the median is closer to the upper three temperatures in the plot. In particular, the median is noticeably far from our minimum measure of 61 degrees. Perhaps we can balance the split by penalizing the median for being too far from the minimum. We'll implement this penalty using the *squared distance*, which is simply the square of the difference between two values. The squared distance grows quadratically as the two values are pushed further apart. Thus, if we penalize our central value based on its distance to 61, the squared distance penalty will grow noticeably larger as it drifts away from 61 degrees (figure 5.6).

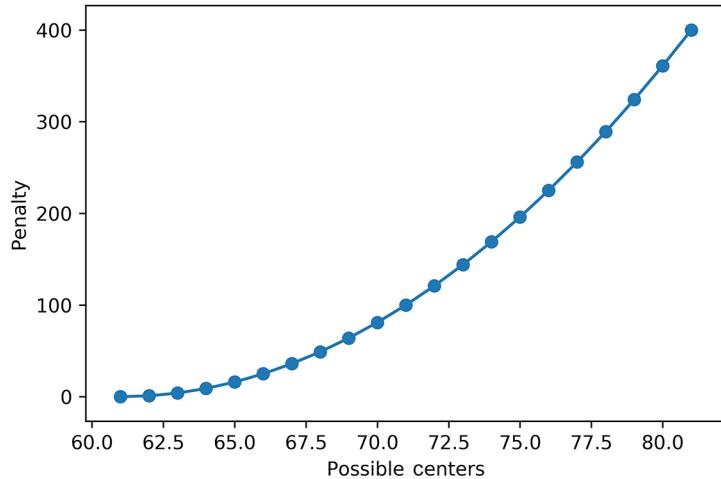


Figure 5.6 A plot of possible centers penalized based on their squared distances relative to the minimum temperature of 61 degrees. Not surprisingly, the minimum penalty occurs at 61 degrees. Unfortunately, the penalty doesn't take into account the distance to the remaining six recorded temperatures.

Listing 5.13 Penalizing centers using the squared distance from the minimum

```

def squared_distance(value1, value2):
    return (value1 - value2) ** 2
    Uses the range of values between  
the minimum and maximum measured  
temperatures as our set of possible centers

possible_centers = range(measurements.min(), measurements.max() + 1)
penalties = [squared_distance(center, 61) for center in possible_centers]
plt.plot(possible_centers, penalties)
plt.scatter(possible_centers, penalties)
plt.xlabel('Possible Centers')
plt.ylabel('Penalty')
plt.show()

```

Our plot displays the penalty across a range of possible centers based on their distance to our minimum. As the centers shift toward 61, the penalty drops, but their distance to the remaining six measurements increases. Thus, we ought to penalize each potential center based on its squared distance to all seven measurements. We'll do so by defining a sum of squared distances function, which will add up the squared distances between some value and the measurement array. That function will serve as our new penalty. Plotting the possible centers against their penalties will allow us to find the center whose penalty is minimized (figure 5.7).

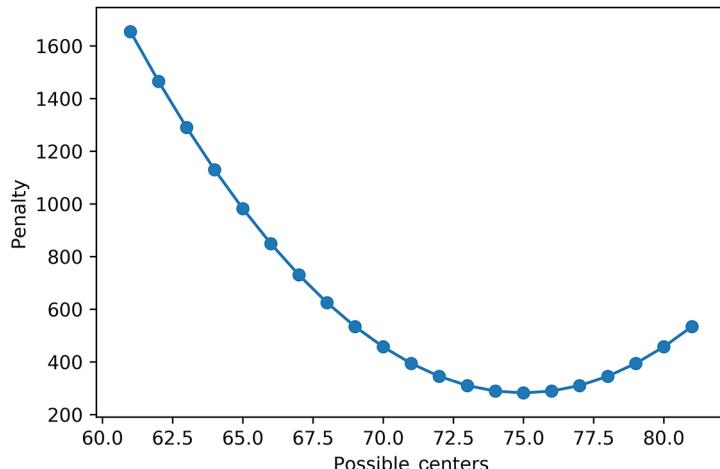


Figure 5.7 A plot of possible centers penalized based on the sum of their squared distances relative to all recorded temperatures. The minimum penalty occurs at 75 degrees.

Listing 5.14 Penalizing centers using the total sum of squared distances

```

def sum_of_squared_distances(value, measurements):
    return sum(squared_distance(value, m) for m in measurements)

```

```

penalties = [sum_of_squared_distances(center, measurements)
             for center in possible_centers]
plt.plot(possible_centers, penalties)
plt.scatter(possible_centers, penalties)
plt.xlabel('Possible Centers')
plt.ylabel('Penalty')
plt.show()

```

Based on our plot, the temperature of 75 degrees incurs the lowest penalty. We'll informally refer to this temperature value as our "least-penalized center." Let's demarcate it using a horizontal line on our temperature plot (figure 5.8).

Listing 5.15 Plotting the least-penalized temperature

```

least_penalized = 75
assert least_penalized == possible_centers[np.argmin(penalties)]

plt.plot(range(number_of_days), measurements)
plt.scatter(range(number_of_days), measurements)
plt.axhline(midpoint, color='k', linestyle='--', label='midpoint')
plt.axhline(median, color='g', linestyle='-.', label='median')
plt.axhline(least_penalized, color='r', linestyle='-', label='least penalized center')
plt.legend()
plt.ylabel('Temperature')
plt.show()

```

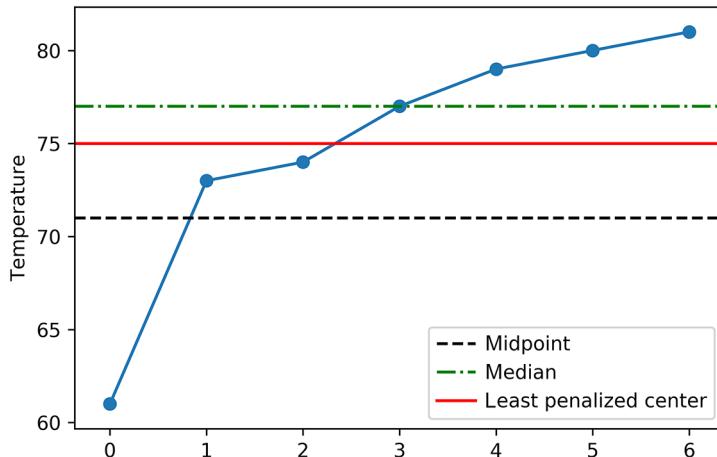


Figure 5.8 A plot containing seven sorted temperatures. The least-penalized center of 75 degrees splits the temperatures in a balanced manner.

The least-penalized center splits the measured temperatures fairly evenly: four measurements appear above it, and three measurements appear below it. Thus, this center

maintains a balanced data split while providing a closer distance to the coldest recorded temperature relative to the median.

The least-penalized center is a good measure of centrality. It minimizes all the penalties incurred for being too far from any given point, which leads to balanced distances between the center and every data point. Unfortunately, our computation of that center was very inefficient. Scanning all possible penalties is not a scalable solution. Is there a more efficient way to compute the center? Yes! Mathematicians have shown that the sum-of-squared-distances error is always minimized by the *average* value of a dataset. Thus, we can compute the least-penalized center directly. We simply need to sum all the elements in `measurements` and then divide that sum by the array size.

Listing 5.16 Computing the least-penalized center using an average value

```
assert measurements.sum() / measurements.size == least_penalized
```

A summed array of values divided by array size is formally called the *arithmetic mean*. Informally, the value is referred to as the *mean* or the average of the array. The mean can be computed by calling the `mean` method of a NumPy array. We can also compute the mean by calling the `np.mean` and `np.average` methods.

Listing 5.17 Computing the mean using NumPy

```
mean = measurements.mean()
assert mean == least_penalized
assert mean == np.mean(measurements)
assert mean == np.average(measurements)
```

The `np.average` method differs from the `np.mean` method because it takes as input an optional `weights` parameter. The `weights` parameter is a list of numeric weights that capture the importance of the measurements relative to each other. When all the weights are equal, the output of `np.average` is no different from `np.mean`. However, adjusting the weights leads to a difference in the outputs.

Listing 5.18 Passing weights into `np.average`

```
equal_weights = [1] * 7
assert mean == np.average(measurements, weights=equal_weights)

unequal_weights = [100] + [1] * 6
assert mean != np.average(measurements, weights=unequal_weights)
```

The `weights` parameter is useful for computing the mean across duplicate measurements. Suppose we analyze 10 temperature measurements where 75 degrees appears 9 times and 77 degrees appears just once. The full list of measurements is represented by `9 * [75] + [77]`. We can compute the mean by calling `np.mean` on that list. We can also compute the mean by calling `np.average([75, 77], weights=[9, 1])`; both computations are equal.

Listing 5.19 Computing the weighted mean of duplicate values

```
weighted_mean = np.average([75, 77], weights=[9, 1])
print(f"The mean is {weighted_mean}")
assert weighted_mean == np.mean(9 * [75] + [77]) == weighted_mean
```

The mean is 75.2

Computing the weighted mean serves as a shortcut for computing the regular mean when duplicates are present. In the computation, the relative ratio of unique measurement counts is represented by the ratio of the weights. Thus, even if we convert our absolute counts of 9 and 1 into relative weights of 900 and 100, the final value of `weighted_mean` should remain the same. This is also true if the weights are converted into relative probabilities of 0.9 and 0.1.

Listing 5.20 Computing the weighted mean of relative weights

```
assert weighted_mean == np.average([75, 77], weights=[900, 100])
assert weighted_mean == np.average([75, 77], weights=[0.9, 0.1])
```

We can treat probabilities as weights. Consequently, this allows us to compute the mean of any probability distribution.

5.2.1 Finding the mean of a probability distribution

At this point in the book, we are intimately familiar with the 20-coin-flip binomial distribution. The distribution's peak is symmetrically centered at 10 heads. How does that peak compare to the distribution's mean? Let's find out. We compute the mean by passing a `probabilities` array into the `weights` parameter of `np.average`. Then we plot the mean as a vertical line that cuts across the distribution (figure 5.9).

Listing 5.21 Computing the mean of a binomial distribution

```
num_flips = 20
interval_all_counts = range(num_flips + 1)
probabilities = stats.binom.pmf(interval_all_counts, 20, prob_head)
mean_binomial = np.average(interval_all_counts, weights=probabilities)
print(f"The mean of the binomial is {mean_binomial:.2f} heads")
plt.plot(interval_all_counts, probabilities)
plt.axvline(mean_binomial, color='k', linestyle='--')
```

The `axvline` method plots a vertical line at a specified x coordinate.

The mean of the binomial is 10.00 heads

The mean of the binomial is 10 heads. It cuts across the distribution's central peak and perfectly captures the binomial's centrality. For this reason, SciPy allows us to obtain the mean of any binomial simply by calling `stats.binom.mean`. The `stats.binom.mean`

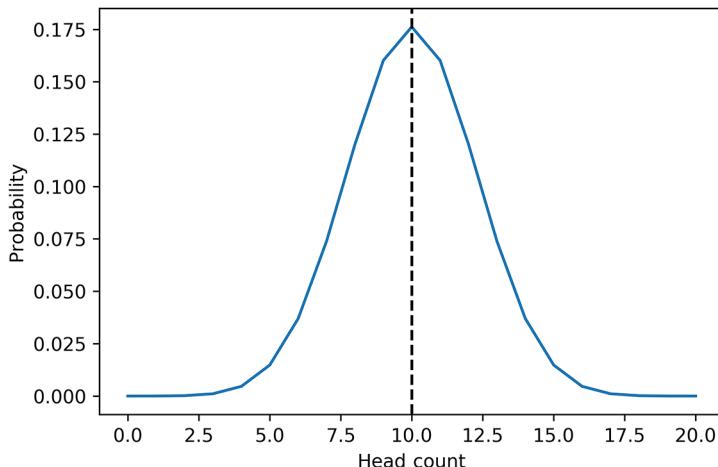


Figure 5.9 A 20-coin-flip binomial distribution bisected by its mean. The mean is positioned directly in the distribution's center.

method takes as input two parameters: the number of coin flips and the probability of heads.

Listing 5.22 Computing the binomial mean using SciPy

```
assert stats.binom.mean(num_flips, 0.5) == 10
```

Using the `stats.binom.mean` method, we can rigorously analyze the relationship between binomial centrality and coin-flip count. Let's plot the binomial mean across a range of coin-flip counts from 0 through 500 (figure 5.10).

Listing 5.23 Plotting multiple binomial means

```
means = [stats.binom.mean(num_flips, 0.5) for num_flips in range(500)]
plt.plot(range(500), means)
plt.xlabel('Coin Flips')
plt.ylabel('Mean')
plt.show()
```

The coin-flip count and mean share a linear relationship in which the mean is equal to half the coin-flip count. With this in mind, let's consider the mean of the single coin-flip binomial distribution (commonly called the *Bernoulli distribution*). The Bernoulli distribution has a coin-flip count of 1, so its mean is equal to 0.5. Not surprisingly, the probability of a fair coin landing on heads is equal to the Bernoulli mean.

Listing 5.24 Predicting the mean of a Bernoulli distribution

```
num_flips = 1
assert stats.binom.mean(num_flips, 0.5) == 0.5
```

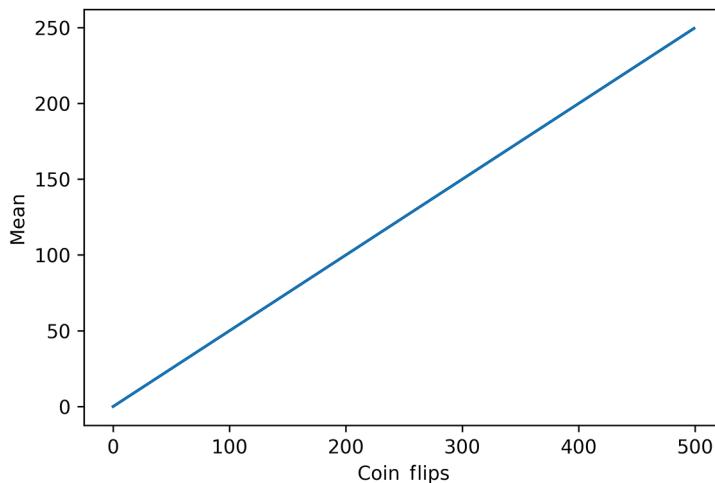


Figure 5.10 Coin-flip count plotted against binomial mean. The relationship is linear. The mean of each binomial is equal to half its coin-flip count.

We can use the observed linear relationship to predict the mean of a 1,000-coin-flip distribution. We expect that mean to equal 500 and be positioned in the distribution's center. Let's confirm that this is the case (figure 5.11).

Listing 5.25 Predicting the mean of a 1,000-coin-flip distribution

```
num_flips = 1000
assert stats.binom.mean(num_flips, 0.5) == 500

interval_all_counts = range(num_flips)
probabilities = stats.binom.pmf(interval_all_counts, num_flips, 0.5)
plt.axvline(500, color='k', linestyle='--')
plt.plot(interval_all_counts, probabilities)
plt.xlabel('Head-count')
plt.ylabel('Probability')
plt.show()
```

A distribution's mean serves as an excellent measure of centrality. Let's now explore the use of variance as a measure of dispersion.

5.3 Variance as a measure of dispersion

Dispersion is the scattering of data points around some central value. A smaller dispersion indicates more predictable data. A larger dispersion indicates greater data fluctuations. Consider a scenario where we measure summer temperatures in California and Kentucky. We gather three measurements for each state, at random locations. California is a huge state, with very diverse climates, so we expect to see fluctuations in our measurements. Our measured California temperatures are 52, 77, and 96 degrees.

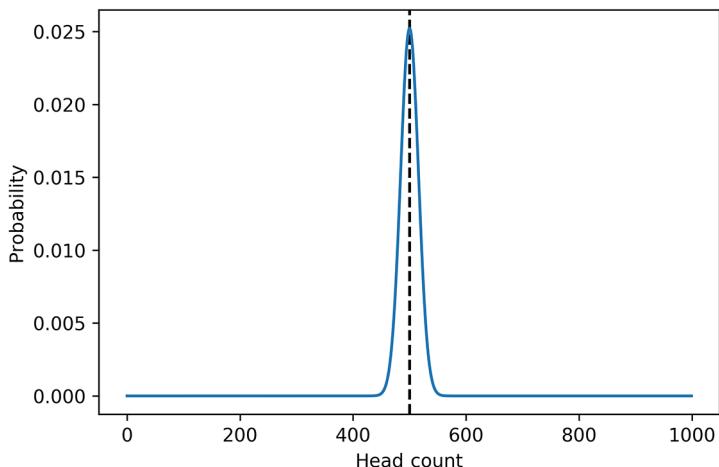


Figure 5.11 A 1,000-coin-flip binomial distribution bisected by its mean. The mean is positioned directly in the distribution's center.

Our measured Kentucky temperatures are 71, 75, and 79 degrees. We store these measured temperatures and compute their means.

Listing 5.26 Measuring the means of multiple temperature arrays

```
california = np.array([52, 77, 96])
kentucky = np.array([71, 75, 79])

print(f"Mean California temperature is {california.mean()}")
print(f"Mean Kentucky temperature is {kentucky.mean()}")

Mean California temperature is 75.0
Mean Kentucky temperature is 75.0
```

The means of the two measurement arrays both equal 75. California and Kentucky appear to share the same central temperature value. Despite this, the two measurement arrays are far from equal. The California temperatures are much more dispersed and unpredictable: they range from 52 to 96 degrees. Meanwhile, the stable Kentucky temperatures range from the low 70s to high 70s. They are more closely centered around the mean. We visualize this difference in dispersion by plotting the two measurement arrays (figure 5.12). Additionally, we demarcate the mean by plotting a horizontal line.

Listing 5.27 Visualizing the difference in dispersion

```
plt.plot(range(3), california, color='b', label='California')
plt.scatter(range(3), california, color='b')
plt.plot(range(3), kentucky, color='r', linestyle='-.', label='Kentucky')
plt.scatter(range(3), kentucky, color='r')
```

```
plt.axhline(75, color='k', linestyle='--', label='Mean')
plt.legend()
plt.show()
```

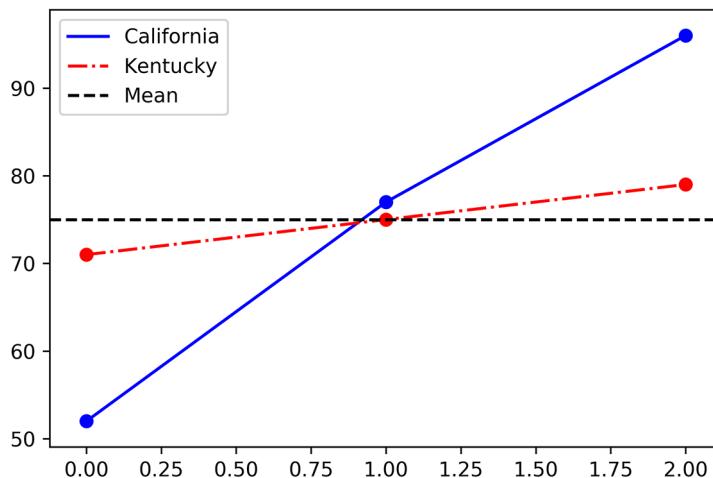


Figure 5.12 A plot of sorted temperatures for California and Kentucky. Temperatures in both states share a mean of 75 degrees. The California temperatures are more dispersed around that mean.

Within the plot, the three Kentucky temperatures nearly overlap with the flat mean. Meanwhile, the majority of California temperatures are noticeably more distant from the mean. We can quantify these observations if we penalize the California measurements for being too distant from their center. Previously, we computed such penalties using the sum-of-squared-distances function. Now we'll compute the sum of squared distances between the California measurements and their mean. Statisticians refer to the sum of squared distances from the mean as simply the *sum of squares*. We define a `sum_of_squares` function and then apply it to our California temperatures.

Listing 5.28 Computing California's sum of squares

```
def sum_of_squares(data):
    mean = np.mean(data)
    return sum(squared_distance(value, mean) for value in data)

california_sum_squares = sum_of_squares(california)
print(f"California's sum of squares is {california_sum_squares}")

California's sum of squares is 974.0
```

California's sum of squares is 974. We expect Kentucky's sum of squares to be noticeably lower. Let's confirm.

Listing 5.29 Computing Kentucky's sum of squares

```
kentucky_sum_squares = sum_of_squares(kentucky)
print(f"Kentucky's sum of squares is {kentucky_sum_squares}")
```

Kentucky's sum of squares is 32.0

Kentucky's sum of squares is 32. Thus, we see a thirtyfold difference between our California results and our Kentucky calculations. This isn't surprising, because the Kentucky data points are much less dispersed. The sum of squares helps measure that dispersion—however, the measurement is not perfect. Suppose we duplicate the temperatures in the California array by recording each temperature twice. The level of dispersion will remain the same even though the sum of squares will double.

Listing 5.30 Computing sum of squares after array duplication

```
california_duplicated = np.array(california.tolist() * 2)
duplicated_sum_squares = sum_of_squares(california_duplicated)
print(f"Duplicated California sum of squares is {duplicated_sum_squares}")
assert duplicated_sum_squares == 2 * california_sum_squares
```

Duplicated California sum of squares is 1948.0

The sum of squares is not a good measure of dispersion because it's influenced by the size of the inputted array. Fortunately, that influence is easy to eliminate if we divide the sum of squares by the array size. Dividing `california_sum_squares` by `california.size` produces a value equal to `duplicated_sum_squares / california_duplicated.size`.

Listing 5.31 Dividing sum of squares by array size

```
value1 = california_sum_squares / california.size
value2 = duplicated_sum_squares / california_duplicated.size
assert value1 == value2
```

Dividing the sum of squares by the number of measurements produces what statisticians call the *variance*. Conceptually, the variance is equal to the average squared distance from the mean.

Listing 5.32 Computing the variance from mean squared distance

```
def variance(data):
    mean = np.mean(data)
    return np.mean([squared_distance(value, mean) for value in data])

assert variance(california) == california_sum_squares / california.size
```

The variances for the `california` and `california_duplicated` arrays are equal since their levels of dispersion are identical.

Listing 5.33 Computing the variance after array duplication

```
assert variance(california) == variance(california_duplicated)
```

Meanwhile, the variances for the California and Kentucky arrays retain their thirty-fold ratio caused by a difference in dispersion.

Listing 5.34 Comparing the variances of California and Kentucky

```
california_variance = variance(california)
kentucky_variance = variance(kentucky)
print(f"California Variance is {california_variance}")
print(f"Kentucky Variance is {kentucky_variance}")
```

```
California Variance is 324.6666666666667
Kentucky Variance is 10.666666666666666
```

Variance is a good measure of dispersion. It can be computed by calling `np.var` on a Python list or NumPy array. The variance of a NumPy array can also be computed using the array's built-in `var` method.

Listing 5.35 Computing the variance using NumPy

```
assert california_variance == california.var()
assert california_variance == np.var(california)
```

Variance is dependent on the mean. If we compute a weighted mean, then we must also compute a weighted variance. Computing the weighted variance is easy: as stated earlier, the variance is simply the average of all the squared distances from the mean, so the weighted variance is the weighted average of all the squared distances from the weighted mean. Let's define a `weighted_variance` function that takes as input two parameters: a data list and weights. It then computes the weighted mean and uses the `np.average` method to compute the weighted average of the squared distances from that mean.

Listing 5.36 Computing the weighted variance using np.average

```
def weighted_variance(data, weights):
    mean = np.average(data, weights=weights)
    squared_distances = [squared_distance(value, mean) for value in data]
    return np.average(squared_distances, weights=weights)

assert weighted_variance([75, 77], [9, 1]) == np.var(9 * [75] + [77])
```

**weighted_variance lets us treat
duplicated elements as weights.**

The `weighted_variance` function can take as its input an array of probabilities. This allows us to compute the variance of any probability distribution.

5.3.1 Finding the variance of a probability distribution

Let's compute the variance of the binomial distribution associated with 20 fair coin flips. We run the computation by assigning a probabilities array to the weights parameter of `weighted_variance`.

Listing 5.37 Computing the variance of a binomial distribution

```
interval_all_counts = range(21)
probabilities = stats.binom.pmf(interval_all_counts, 20, prob_head)
variance_binomial = weighted_variance(interval_all_counts, probabilities)
print(f"The variance of the binomial is {variance_binomial:.2f} heads")
```

The variance of the binomial is 5.00 heads

The binomial's variance is 5, which is equal to half of the binomial's mean. That variance can be computed more directly using SciPy's `stats.binom.var` method.

Listing 5.38 Computing the binomial variance using SciPy

```
assert stats.binom.var(20, prob_head) == 5.0
assert stats.binom.var(20, prob_head) == stats.binom.mean(20, prob_head) / 2
```

Using the `stats.binom.var` method, we can rigorously analyze the relationship between binomial dispersion and coin-flip count. Let's plot the binomial variance across a range of coin-flip counts from 0 to 500 (figure 5.13).

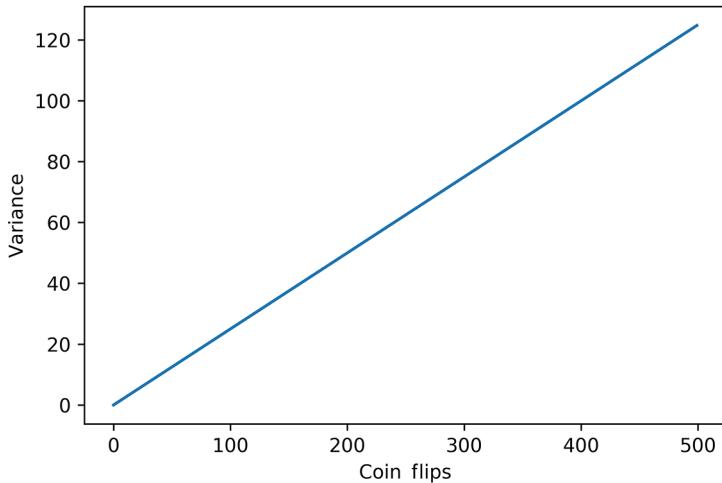


Figure 5.13 Coin-flip count plotted against binomial variance. The relationship is linear. The variance of each binomial is equal to one-fourth of its coin-flip count.

Listing 5.39 Plotting multiple binomial variances

```
variances = [stats.binom.var(num_flips, prob_head)
             for num_flips in range(500)]
plt.plot(range(500), variances)
plt.xlabel('Coin Flips')
plt.ylabel('Variance')
plt.show()
```

The binomial's variance, like its mean, is linearly related to the coin-flip count. The variance is equal to one-fourth of the coin-flip count. Thus, the Bernoulli distribution has a variance of 0.25 because its coin-flip count is 1. By this logic, we can expect a variance of 250 for a 1,000-coin-flip distribution.

Listing 5.40 Predicting binomial variances

```
assert stats.binom.var(1, 0.5) == 0.25
assert stats.binom.var(1000, 0.5) == 250
```

Common SciPy methods for binomial analysis

- `stats.binom.mean(num_flips, prob_heads)`—Returns the mean of a binomial where the flip count equals `num_flips` and the probability of heads equals `prob_heads`.
- `stats.binom.var(num_flips, prob_heads)`—Returns the variance of a binomial where the flip count equals `num_flips` and probability of heads equals `prob_heads`.
- `stats.binom.pmf(head_count_int, num_flips, prob_heads)`—Returns the probability of observing `head_count_int` heads out of `num_flips` coin flips. A single coin flip's probability of heads is set to `prob_heads`.
- `stats.binom.pmf(head_count_array, num_flips, prob_heads)`—Returns an array of binomial probabilities. These are obtained by executing `stats.binom.pmf(e, num_flips, prob_head)` on each element `e` of `head_count_array`.
- `stats.binom_test(head_count_int, num_flips, prob_heads)`—Returns the probability of `num_flips` coin flips generating at least `head_count_int` heads or `tail_count_int` tails. A single coin flip's probability of heads is set to `prob_heads`.

The variance is a powerful measure of data dispersion. However, statisticians often use an alternative measure, which they call the *standard deviation*. The standard deviation is equal to the square root of the variance. It can be computed by calling `np.std`. Squaring the output of `np.std` naturally returns the variance.

Listing 5.41 Computing the standard deviation

```
data = [1, 2, 3]
standard_deviation = np.std(data)
assert standard_deviation ** 2 == np.var(data)
```

We sometimes use standard deviation instead of variance to track units more easily. All measurements have units. For example, our temperatures were in units of degrees Fahrenheit. When we squared the distances of the temperature to their mean, we also squared their units; therefore, our variance was in units of degrees Fahrenheit squared. Such squared units are very tricky to conceptualize. Taking the square root converts the units back to degrees Fahrenheit: a standard deviation in units of degrees Fahrenheit is more easily interpretable than the variance.

The mean and standard deviation are incredibly useful values. They allow us to do the following:

- *Compare numeric datasets.* Suppose we're given two arrays of recorded temperatures for two consecutive summers. We can quantify the differences between these summer records using mean and standard deviation.
- *Compare probability distributions.* Suppose two climate research labs publish probability distributions. Each distribution captures all temperature probabilities across a standard summer day. We can summarize the differences between the two distributions by comparing their means and standard deviations.
- *Compare a numeric dataset to a probability distribution.* Suppose a well-known probability distribution captures a decade's worth of temperature probabilities. However, recently recorded summer temperatures appear to contradict these probability outputs. Is this a sign of climate change or simply a random anomaly? We can find out by juxtaposing the centrality and dispersion for the distribution and the temperature dataset.

The third use case underlies much of statistics. In the subsequent sections, we learn how to compare datasets to distribution likelihoods. Many of our comparisons focus on the normal distribution, which commonly arises in data analysis. Conveniently, that distribution's bell-shaped curve is a direct function of mean and standard deviation. We'll soon use SciPy, along with these two parameters, to better grasp the normal curve's significance.

Summary

- A *probability mass function* maps inputted integer values to their probability of occurrence.
- The probability mass function for the binomial distribution can be generated by calling `stats.binom.pmf`.
- *Mean* is a good measure of a dataset's centrality. It minimizes the *sum of squares* relative to the dataset. We can compute an unweighted mean by summing the

dataset values and dividing by the dataset size. We can also compute a weighted mean by inputting a weights array into `np.average`. The weighted mean of the binomial distribution increases linearly with the coin-flip count.

- *Variance* is a good measure of a dataset's dispersion. It equals the average squared distance of the data point from the mean. The weighted variance of the binomial distribution increases linearly with the coin-flip count.
- The *standard deviation* is an alternative measure of dispersion. It equals the square root of the variance. The standard deviation maintains the units used in a dataset.

Making predictions using the central limit theorem and SciPy

This section covers

- Analyzing the normal curve using the SciPy library
- Predicting mean and variance using the central limit theorem
- Predicting population properties using the central limit theorem

The *normal distribution* is a bell-shaped curve that we introduced in section 3. The curve arises naturally from random data sampling due to the central limit theorem. Previously, we noted how, according to that theorem, repeatedly sampled frequencies take the shape of a normal curve. Furthermore, the theorem predicts a narrowing of that curve as the size of each frequency sample goes up. In other words, the distribution's standard deviation should decrease as the sampling size increases.

The central limit theorem lies at the heart of all classic statistics. In this section, we probe the theorem in great detail using the computational power of SciPy. Eventually, we learn how to use the theorem to make predictions from limited data.

6.1 Manipulating the normal distribution using SciPy

In section 3, we showed how random coin-flip sampling produces a normal curve. Let's generate a normal distribution by plotting a histogram of coin-flip samples. Our input into the histogram will contain 100,000 head-count frequencies. Computing the frequencies will require us to sample a series of coin flips 100,000 times. Each sample will contain an array of 0s and 1s representing 10,000 flipped coins. We'll refer to the array length as our sample size. If we use the sample size to divide the sum of values in the sample, we will compute the observed head-count frequency. Conceptually, this frequency is equal to simply taking the sample's mean.

The following code computes the head-count frequency of a single random sample and confirms its relationship to the mean. Note that every data point in the sample is drawn from the Bernoulli distribution.

Listing 6.1 Computing head-count frequencies from the mean

```
np.random.seed(0)
sample_size = 10000
sample = np.array([np.random.binomial(1, 0.5) for _ in range(sample_size)])
head_count = sample.sum()
head_count_frequency = head_count / sample_size
assert head_count_frequency == sample.mean()    ← The head-count frequency is identical to the sample mean.
```

Of course, we can compute all 100,000 head-count frequencies in a single line of code, as discussed in section 3.

Listing 6.2 Computing 100,000 head-count frequencies

```
np.random.seed(0)
frequencies = np.random.binomial(sample_size, 0.5, 100000) / sample_size
```

Each sampled frequency equals the mean of 10,000 randomly flipped coins. Therefore, we rename our frequencies variable `sample_means`. We then visualize our `sample_means` data as a histogram (figure 6.1).

Listing 6.3 Visualizing sample means in a histogram

```
sample_means = frequencies
likelihoods, bin_edges, _ = plt.hist(sample_means, bins='auto',
                                      edgecolor='black', density=True)
plt.xlabel('Binned Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

The histogram is shaped like a normal distribution. Let's calculate the distribution's mean and standard deviation.

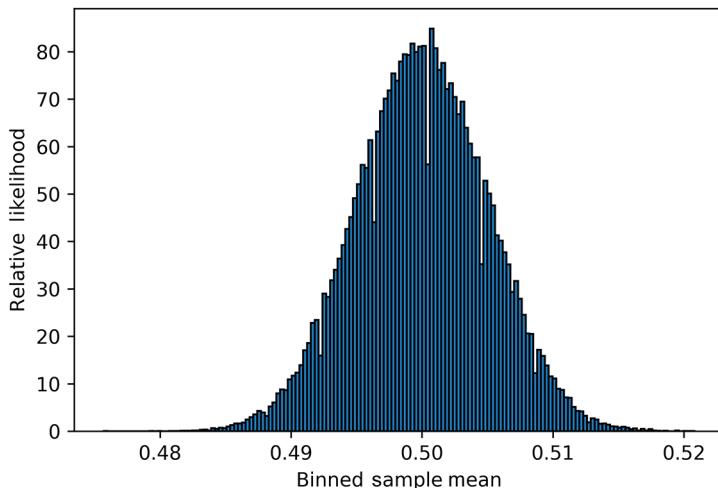


Figure 6.1 A histogram of 100,000 sampled means plotted against their relative likelihoods. The histogram resembles a bell-shaped normal distribution.

Listing 6.4 Computing the mean and standard deviation of a histogram

```
mean_normal = np.average(bin_edges[:-1], weights=likelihoods)
var_normal = weighted_variance(bin_edges[:-1], likelihoods)
std_normal = var_normal ** 0.5
print(f"Mean is approximately {mean_normal:.2f}")
print(f"Standard deviation is approximately {std_normal:.3f}")

Mean is approximately 0.50
Standard deviation is approximately 0.005
```

The distribution's mean is approximately 0.5, and its standard deviation is approximately 0.005. In a normal distribution, these values can be computed directly from the distribution's peak. We just need the peak's x-value and y-value coordinates. The x-value equals the distribution's mean, and the standard deviation is equal to the inverse of the y-value multiplied by $(2\pi)^{1/2}$. These properties are derived from the mathematical analysis of the normal curve. Let's recompute the mean and standard deviation using just the coordinates of the peak.

Listing 6.5 Computing mean and standard deviation from peak coordinates

```
import math
peak_x_value = bin_edges[likelihoods.argmax()]
print(f"Mean is approximately {peak_x_value:.2f}")
peak_y_value = likelihoods.max()
std_from_peak = (peak_y_value * (2* math.pi) ** 0.5) ** -1
print(f"Standard deviation is approximately {std_from_peak:.3f}")

Mean is approximately 0.50
Standard deviation is approximately 0.005
```

Additionally, we can compute the mean and standard deviation simply by calling `stats.norm.fit(sample_means)`. This SciPy method returns the two parameters required to re-create the normal distribution formed by our data.

Listing 6.6 Computing mean and standard deviation using `stats.norm.fit`

```
fitted_mean, fitted_std = stats.norm.fit(sample_means)
print(f"Mean is approximately {fitted_mean:.2f}")
print(f"Standard deviation is approximately {fitted_std:.3f}")

Mean is approximately 0.50
Standard deviation is approximately 0.005
```

The computed mean and standard deviation can be used to reproduce our normal curve. We can regenerate the curve by calling `stats.norm.pdf(bin_edges, fitted_mean, fitted_std)`. SciPy's `stats.norm.pdf` method represents the *probability density function* of a normal distribution. A probability density function is like a probability mass function but with one key difference: it does not return probabilities. Instead, it returns relative likelihoods. As discussed in section 2, relative likelihoods are the y-axis values of a curve whose total area equals 1.0. Unlike probabilities, these likelihoods can equal values that are greater than 1.0. Despite this, the total area beneath a plotted likelihood interval still equals the probability of observing a random value within that interval.

Let's compute the relative likelihoods using `stats.norm.pdf`. Then we plot the likelihoods together with the sampled coin-flip histogram (figure 6.2).

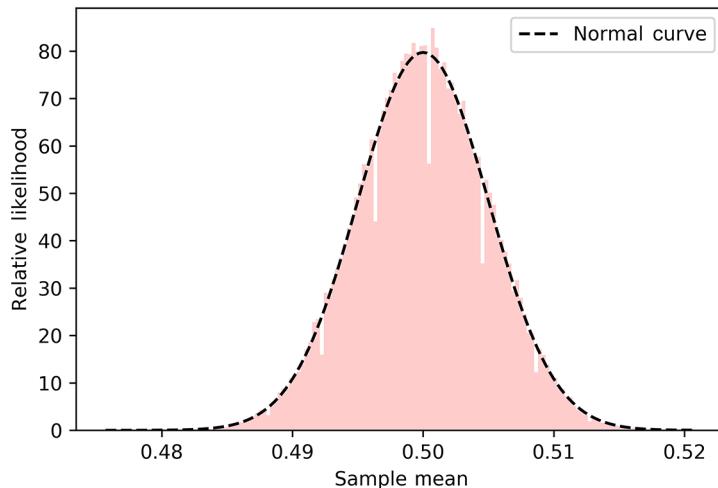


Figure 6.2 A histogram overlaid with a normal probability density function. The parameters defining the plotted normal curve were computed using SciPy. The plotted normal curve fits nicely over the histogram.

Listing 6.7 Computing normal likelihoods using stats.norm.pdf

```
normal_likelihoods = stats.norm.pdf(bin_edges, fitted_mean, fitted_std)
plt.plot(bin_edges, normal_likelihoods, color='k', linestyle='--',
         label='Normal Curve')
plt.hist(sample_means, bins='auto', alpha=0.2, color='r', density=True) ←
plt.legend()
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

The alpha parameter is used to make the histogram more transparent to better contrast the histogram with the plotted likelihood curve.

The plotted curve fits well over the histogram. The curve's peak sits at an x-axis position of 0.5 and rises to a y-axis position of approximately 80. As a reminder, the peak's x and y coordinates are a direct function of `fitted_mean` and `fitted_std`. To emphasize this important relationship, let's do a simple exercise: we'll shift the peak 0.01 units to the right while also doubling the peak's height (figure 6.3). How do we execute the shift? Well, the peak's axis is equal to the mean, so we adjust the input mean to `fitted_mean + 0.01`. Also, the peak's height is inversely proportional to the standard deviation. Therefore, inputting `fitted_std / 2` should double the height of the peak.

Listing 6.8 Manipulating a normal curve's peak coordinates

```
adjusted_likelihoods = stats.norm.pdf(bin_edges, fitted_mean + 0.01,
                                       fitted_std / 2)
plt.plot(bin_edges, adjusted_likelihoods, color='k', linestyle='--')
plt.hist(sample_means, bins='auto', alpha=0.2, color='r', density=True)
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

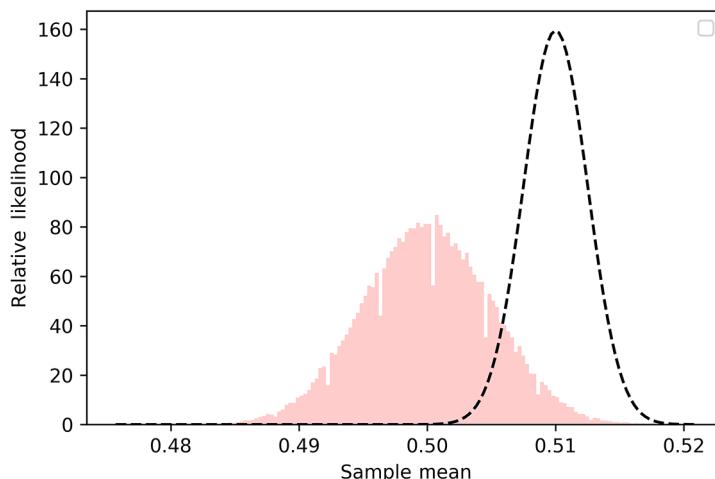


Figure 6.3 A modified normal curve whose center is .01 units to the right of the histogram. The peak of the curve is twice the height of the histogram's peak. These modifications were achieved by manipulating the histogram's mean and standard deviation.

6.1.1 Comparing two sampled normal curves

SciPy allows us to explore and adjust the shape of the normal distribution based on the inputted parameters. Also, the values of these input parameters depend on how we sample random data. Let's quadruple the coin-flip sample size to 40,000 and plot the resulting distribution changes. The following code compares the plotted shapes of the old and updated normal distributions, which we label A and B, respectively (figure 6.4).

Listing 6.9 Plotting two curves with different samples sizes

```
np.random.seed(0)
new_sample_size = 40000
new_head_counts = np.random.binomial(new_sample_size, 0.5, 100000)
new_mean, new_std = stats.norm.fit(new_head_counts / new_sample_size)
new_likelihoods = stats.norm.pdf(bin_edges, new_mean, new_std)
plt.plot(bin_edges, normal_likelihoods, color='k', linestyle='--',
          label='A: Sample Size 10K')
plt.plot(bin_edges, new_likelihoods, color='b', label='B: Sample Size 40K')
plt.legend()
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

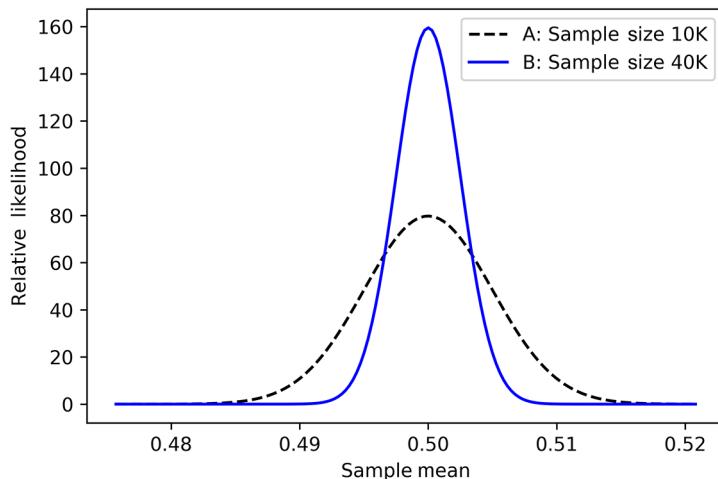


Figure 6.4 Two normal distributions generated using coin-flip data. Distribution A was derived using a sample size of 10,000 coin flips per sample. Distribution B was derived using a sample size of 40,000 coin flips per sample. Both distributions are centered around a mean value of 0.5. However, distribution B is much more narrowly dispersed around its center, and the peak of distribution B is twice as high as the peak of distribution A. Given the relationship between peak height and variance, we can infer that the variance of distribution B is one-fourth the variance of distribution A.

Both normal distributions are centered around the sample mean value of 0.5. However, the distribution with the larger sample size is more narrowly centered around its peak. This is consistent with what we saw in section 3. In that section, we observed that as the sample size increases, the peak location stays constant while the area around the peak contracts in width. The narrowing of the peak leads to a drop in the confidence interval range. A confidence interval represents the likely value range covering the true probability of heads. Previously, we used confidence intervals to estimate the probability of heads from the x-axis head-count frequencies. Now our x-axis represents the sample means, where every sample mean is identical to a head-count frequency. Thus, we can use our sample means to find the probability of heads. Also, as a reminder, all coin samples were drawn from the Bernoulli distribution. We've recently shown that the mean of the Bernoulli distribution equals the probability of heads, so (not surprisingly) each sample's mean serves as an estimate of the true Bernoulli mean. We can interpret the confidence interval as a likely value range covering the true Bernoulli mean.

Let's calculate the 95% confidence interval for the true Bernoulli mean using normal distribution B. Previously, we manually computed the 95% confidence interval by exploring the curve area around the peak. However, SciPy allows us to automatically extract that range by calling `stats.norm.interval(0.95, mean, std)`. The method returns an interval that covers 95% of the area beneath the normal distribution defined by `mean` and `std`.

Listing 6.10 Computing a confidence interval using SciPy

```
mean, std = new_mean, new_std
start, end = stats.norm.interval(0.95, mean, std)
print(f"The true mean of the sampled binomial distribution is between
      {start:.3f} and {end:.3f}")
```

The true mean of the sampled binomial distribution is between 0.495 and 0.505

We are 95% confident that the true mean of our sampled Bernoulli distribution is between 0.495 and 0.505. In fact, that mean is equal to exactly 0.5. We can confirm this using SciPy.

Listing 6.11 Confirming the Bernoulli mean

```
assert stats.binom.mean(1, 0.5) == 0.5
```

Let's now attempt to estimate the variance of the Bernoulli distribution based on the plotted normal curves. At first glance, this seems like a difficult task. Although the means of the two plotted distributions remain constant at 0.5, their variances shift noticeably. The relative shift in variance can be estimated by comparing peaks. The peak of distribution B is twice as high as the peak of distribution A. This height is inversely proportional to the standard deviation, so the standard deviation of distribution B is

Common SciPy methods for normal curve analysis

- `stats.norm.fit(data)`—Returns the mean and standard deviation required to fit a normal curve to data.
- `stats.norm.pdf(observation, mean, std)`—Returns the likelihood mapped to a single value of a normal curve defined by mean `mean` and standard deviation `std`.
- `stats.norm.pdf(observation_array, mean, std)`—Returns an array of normal likelihoods. These are obtained by executing `stats.norm.pdf(e, mean, std)` on each element `e` of `observation_array`.
- `stats.norm.interval(x_percent, mean, std)`—Returns the `x_percent` confidence interval defined by mean `mean` and standard deviation `std`.

half the standard deviation of distribution A. Since the standard deviation is the square root of the variance, we can infer that the variance of distribution B is one-fourth the variance of distribution A. Thus, increasing the sample size fourfold from 10,000 to 40,000 leads to a fourfold decrease in the variance.

Listing 6.12 Assessing shift in variance after increased sampling

```
variance_ratio = (new_std ** 2) / (fitted_std ** 2)
print(f"The ratio of variances is approximately {variance_ratio:.2f}")
```

The ratio of variances is approximately 0.25

It appears that variance is inversely proportional to sample size. If so, a fourfold decrease in sample size from 10,000 to 2,500 should generate a fourfold increase in the variance. Let's generate some head counts using a sample size of 2,500 and confirm if this is the case.

Listing 6.13 Assessing shift in variance after decreased sampling

```
np.random.seed(0)
reduced_sample_size = 2500
head_counts = np.random.binomial(reduced_sample_size, 0.5, 100000)
_, std = stats.norm.fit(head_counts / reduced_sample_size)
variance_ratio = (std ** 2) / (fitted_std ** 2)
print(f"The ratio of variances is approximately {variance_ratio:.1f}")
```

The ratio of variances is approximately 4.0

Yes! A fourfold decrease in the sample size leads to a fourfold increase in the variance. Thus, if we decrease the sample size from 10,000 to 1, we can expect a 10,000-fold increase in the variance. That variance for a sample size of 1 should be equal to `(fitted_std ** 2) * 10000`.

Listing 6.14 Predicting variance for a sample size of 1

```
estimated_variance = (fitted_std ** 2) * 10000
print(f"Estimated variance for a sample size of 1 is
      {estimated_variance:.2f}")
```

Estimated variance for a sample size of 1 is 0.25

Our estimated variance for a sample size of 1 is 0.25. However, if the sample size were 1, then our `sample_means` array would simply be a sequence of randomly recorded 1s and 0s. By definition, that array would represent the output of the Bernoulli distribution, so running `sample_means.var` would approximate the variance of the Bernoulli distribution. Thus our estimated variance for a sample size of 1 equals the variance of the Bernoulli distribution. In fact, the Bernoulli variance does equal 0.25.

Listing 6.15 Confirming the predicted variance for a sample size of 1

```
assert stats.binom.var(1, 0.5) == 0.25
```

We have just used the normal distribution to compute the variance and mean of the Bernoulli distribution from which we sampled. Let's review the chain of steps that led to our results:

- 1 We sampled random 1s and 0s from the Bernoulli distribution.
- 2 Each sequence of `sample_size` 1s and 0s was grouped into a single sample.
- 3 We computed a mean for every sample.
- 4 The sample means produced a normal curve. We found its mean and standard deviation.
- 5 The mean of the normal curve equaled the mean of the Bernoulli distribution.
- 6 The variance of the normal curve multiplied by the sample size equaled the variance of the Bernoulli distribution.

What if we had sampled from some other non-Bernoulli distribution? Would we still be able to estimate the mean and variance through random sampling? Yes, we would! According to the central limit theorem, sampling mean values from almost any distribution will produce a normal curve. This includes distributions such as the following:

- The *Poisson distribution* (`stats.poisson.pmf`). Commonly used to model
 - Number of customers who visit a store per hour
 - Number of clicks on an online ad per second
- The *Gamma distribution* (`scipy.stats.gamma.pdf`). Commonly used to model
 - Monthly rainfall in a region
 - Banking loan defaults based on loan size

- The *log-normal* distribution (`scipy.stats.lognorm.pdf`). Commonly used to model
 - Fluctuating stock prices
 - Incubation periods of infectious diseases
- Countless distributions occurring in nature that haven't yet been assigned a name

WARNING Under edge-case circumstances, sampling does not produce a normal curve. This is occasionally true of the Pareto distribution, which is used to model income inequality.

Once we've sampled a normal curve, we can use it to analyze the underlying distribution. The mean of the normal curve approximates the mean of the underlying distribution. Also, the variance of the normal curve multiplied by the sample size approximates the variance of the underlying distribution.

NOTE In other words, if we sample from a distribution with variance `var`, we obtain a normal curve with variance `sample_size / var`. As the sample size approaches infinity, the variance of the normal curve approaches zero. At zero variance, the normal curve collapses into a single vertical line positioned at the mean. This property can be used to derive the law of large numbers, which we introduced in section 2.

The relationship between a normal distribution produced by sampling and the properties of the underlying distribution serves as a foundation for all statistics. Using that relationship, we can use the normal curve to estimate both the mean and variance of almost any distribution through random sampling.

6.2 Determining the mean and variance of a population through random sampling

Suppose we are tasked with finding the average age of people living in a town. The town's population is exactly 50,000 people. The following code simulates the ages of the townsfolk using the `np.random.randint` method.

Listing 6.16 Generating a random population

```
np.random.seed(0)
population_ages = np.random.randint(1, 85, size=50000)
```

How do we compute the average age of the residents? One cumbersome approach would be to take a census of every resident in the town. We could record all 50,000 ages and then compute their mean. That exact mean would cover the entire population, which is why it's called the *population mean*. Furthermore, the variance of an entire population is referred to as the *population variance*. Let's quickly compute the population mean and population variance of our simulated town.

Listing 6.17 Computing the population mean and variance

```
population_mean = population_ages.mean()
population_variance = population_ages.var()
```

Computing the population mean is easy when we have simulated data. However, obtaining that data in real life would be incredibly time consuming. We would have to interview all 50,000 people. Without more resources, interviewing the whole town would be borderline impossible.

A simpler approach would be to interview 10 randomly chosen people in the town. We'd record the ages from this random sample and then compute the sample mean. Let's simulate the sampling process by drawing 10 random ages from the `np.random.choice` method. Executing `np.random.choice(age, size=sample_size)` returns an array of 10 randomly sampled ages. After sampling is complete, we will compute the mean of the resulting 10-element array.

Listing 6.18 Simulating 10 interviewed people

```
np.random.seed(0)
sample_size = 10
sample = np.random.choice(population_ages, size=sample_size)
sample_mean = sample.mean()
```

Of course, our sample mean is likely to be noisy and inexact. We can measure that noise by finding the percent difference between `sample_mean` and `population_mean`.

Listing 6.19 Comparing the sample mean to the population mean

```
percent_diff = lambda v1, v2: 100 * abs(v1 - v2) / v2
percent_diff_means = percent_diff(sample_mean, population_mean)
print(f"There is a {percent_diff_means:.2f} percent difference between means.")
```

There is a 27.59 percent difference between means

There is approximately a 27% difference between the sample mean and the population mean. Clearly, our sample is insufficient to estimate the mean—we need to gather more samples. Perhaps we should increase our sampling to cover 1,000 residents of the town. This seems like a reasonable objective that is preferable to surveying all 50,000 residents. Unfortunately, interviewing 1,000 people will still be very time consuming: even if we assume an idealistic interview rate of 2 people per minute, it will take us eight hours to reach our interview goal. Conceivably, we can optimize our time by parallelizing the interview process. We can post an ad in the local paper asking for 100 volunteers: each volunteer will survey 10 random people to sample their ages and then send us a computed sample mean. Thus we will receive 100 sample means representing 1,000 interviews total.

NOTE Each volunteer will send us a sample mean. Conceivably, the volunteers could send the full data instead. However, the sample means are preferable, for the following reasons. First, the means don't require as much memory storage as the full data. Second, the means can be plotted as a histogram to check the quality of our sample size. If that histogram does not approximate a normal curve, then additional samples will be required.

Let's simulate our surveying process.

Listing 6.20 Computing sample means across 1,000 people

```
np.random.seed(0)
sample_means = [np.random.choice(population_ages, size=sample_size).mean()
                for _ in range(100)]
```

According to the central limit theorem, a histogram of sample means should resemble the normal distribution. Furthermore, the mean of the normal distribution should approximate the population mean. We can confirm that this is the case by fitting the sample means to a normal distribution (figure 6.5).

Listing 6.21 Fitting sample means to a normal curve

```
likelihoods, bin_edges, _ = plt.hist(sample_means, bins='auto', alpha=0.2,
                                      color='r', density=True)
mean, std = stats.norm.fit(sample_means)
normal_likelihoods = stats.norm.pdf(bin_edges, mean, std)
plt.plot(bin_edges, normal_likelihoods, color='k', linestyle='--')
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

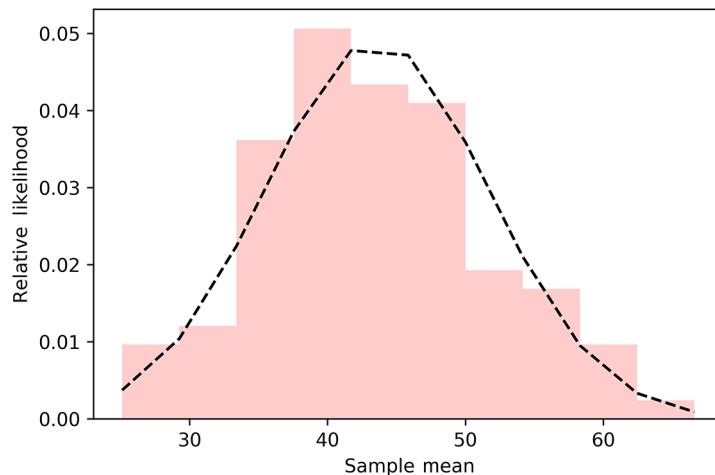


Figure 6.5 A histogram computed from 100 age samples. The histogram is overlaid with its associated normal distribution. The normal distribution's mean and standard deviation parameters were derived from the plotted histogram data.

Our histogram is not very smooth because we've only processed 100 data points. However, the histogram's shape still approximates a normal distribution. We print that distribution's mean and compare it to the population mean.

Listing 6.22 Comparing the normal mean to the population mean

```
print(f"Actual population mean is approximately {population_mean:.2f}")
percent_diff_means = percent_diff(mean, population_mean)
print(f"There is a {percent_diff_means:.2f}% difference between means.")
```

```
Actual population mean is approximately 42.53
There is a 2.17% difference between means.
```

Our estimated mean of the ages is roughly 43. The actual population mean is roughly 42.5. There is an approximately 2% difference between the estimated mean and the actual mean. Thus our result, while not perfect, is still a very good approximation of the actual average age within the town.

Now, we briefly turn our attention to the standard deviation computed from the normal distribution. Squaring the standard deviation produces the distribution's variance. According to the central limit theorem, we can use that variance to estimate the variance of ages in the town. We simply need to multiply the computed variance by the sample size.

Listing 6.23 Estimating the population variance

```
normal_variance = std ** 2
estimated_variance = normal_variance * sample_size
```

Let's compare the estimated variance to the population variance.

Listing 6.24 Comparing the estimated variance to the population variance

```
print(f"Estimated variance is approximately {estimated_variance:.2f}")
print(f"Actual population variance is approximately
      {population_variance:.2f}")
percent_diff_var = percent_diff(estimated_variance, population_variance)
print(f"There is a {percent_diff_var:.2f} percent difference between
      variances.")
```

```
Estimated variance is approximately 576.73
Actual population variance is approximately 584.33
There is a 1.30 percent difference between variances.
```

There is approximately a 1.3% difference between the estimated variance and the population variance. We've thus approximated the town's variance to a relatively accurate degree while sampling only 2% of the people living in the town. Our estimates may not be 100% perfect. However, the amount of time we saved more than makes up for that minuscule drop in accuracy.

So far, we've only used the central limit theorem to estimate the population mean and variance. However, the power of the theorem goes beyond the mere estimation of distribution parameters. We can use the central limit theorem to make predictions about people.

6.3 Making predictions using the mean and variance

Let's now consider a new scenario in which we analyze a fifth-grade classroom. Mrs. Mann is a brilliant fifth-grade teacher. She has spent 25 years inspiring a love of learning in her students. Her classroom holds 20 students. Thus, over the years, she has taught 500 students total.

NOTE We are assuming that each year, Mrs. Mann teaches exactly 20 students. Of course, in real life, classroom size might fluctuate from year to year.

Her students frequently outperform other fifth graders in the state. That performance is measured using scholastic assessment exams, which are administered to all fifth graders every year. These exams are graded from 0 to 100. All grades can be accessed by querying the state assessment database. However, due to poor database design, the queryable exam records do not specify the year when each exam was taken.

Imagine we're tasked with addressing the following question: Has Mrs. Mann ever taught a class that collectively aced the assessment exam? More specifically, has she ever taught a class of 20 students whose mean assessment grade was above 89%?

To answer that question, assume that we've queried the state database. We've obtained grades for all of Mrs. Mann's past students. Of course, a lack of temporal information prevents us from grouping the grades by year. Thus, we cannot simply scan the records for a yearly mean above 89%. However, we can still compute the mean and variance across the 500 total grades. Let's suppose the mean is equal to 84 and the variance is equal to 25. We'll refer to these values as the population mean and population variance, since they cover the entire population of students who've ever been taught by Mrs. Mann.

Listing 6.25 Population mean and variance of recorded grades

```
population_mean = 84
population_variance = 25
```

Let's model the yearly test results of Mrs. Mann's class as a collection of 20 grades randomly drawn from a distribution with mean `population_mean` and variance `population_variance`. This model is simplistic. It makes several extreme assumptions, such as these:

- Performance of each student in the class does not depend on any other student.

In real life, this assumption doesn't always hold. For instance, disruptive students can negatively impact the performance of others.

- Exams are equally difficult every year.
In real life, standardized exams can be adjusted by government officials.
- Local economic factors are negligible.
In real life, fluctuating economies impact school district budgets and as well as student home environments. These external factors can affect the quality of grades.

Our simplifications might impact prediction accuracy. However, given our limited data, we have little choice in the matter. Statisticians are frequently forced to make such compromises to address otherwise intractable problems. Most of the time, their simplified predictions still reasonably reflect real-world behaviors.

Given our simple model, we can sample a random batch of 20 grades. What is the probability that the grades will have a mean of at least 90? This probability can easily be computed using the central limit theorem. According to the theorem, the likelihood distribution of mean grades will resemble a normal curve. The mean of the normal curve will equal `population_mean`. The variance of the normal curve will equal `population_variance` divided by our sample size of 20 students. Taking the square root of that variance produces the standard deviation of the curve, which statisticians call the *standard error of the mean* (SEM). By definition, the SEM equals the population standard deviation divided by the square root of the sample size. We compute the curve parameters and plot the normal curve next (figure 6.6).

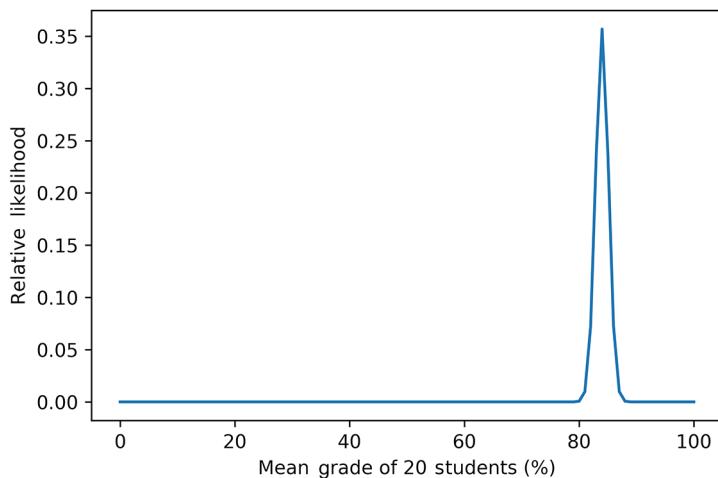


Figure 6.6 A normal distribution derived from the population mean and the standard error from the mean (SEM). The SEM is equal to the standard deviation divided by the square root of the sample size. The area beneath the plotted curve can be used to calculate probabilities.

Listing 6.26 Plotting a normal curve using the mean and SEM

```

mean = population_mean
population_std = population_variance ** 0.5
sem = population_std / (20 ** 0.5)
grade_range = range(101)
normal_likelihoods = stats.norm.pdf(grade_range, mean, sem)
plt.plot(grade_range, normal_likelihoods)
plt.xlabel('Mean Grade of 20 Students (%)')
plt.ylabel('Relative Likelihood')
plt.show()

```

The population standard deviation equals the square root of the population variance.

The SEM equals population_std divided by the square root of the sample size. Alternatively, we can compute the SEM by running (population_variance / 20) ** 0.5.

The area beneath the plotted curve approaches zero at values higher than 89%. That area is also equal to the probability of a given observation. Therefore, the probability of observing a mean grade that's at or above 90% is incredibly low. Still, to be sure, we need to compute the actual probability. Thus, we need to somehow accurately measure the area under the normal distribution.

6.3.1 Computing the area beneath a normal curve

In section 3, we computed areas under histograms. Determining these areas proved easy. All histograms, by definition, are made up of small rectangular units: we could sum the areas of rectangles composing a specified interval, and the total sum equaled the interval's area. Unfortunately, our smooth normal curve does not decompose into rectangles. So how do we find its area? One simple solution is to subdivide the normal curve into small, trapezoidal units. This ancient technique is referred to as the *trapezoidal rule*. A trapezoid is a four-sided polygon with two parallel sides; the trapezoid's area is equal to the sum of these parallel sides multiplied by half the distance between them. Summing over multiple consecutive trapezoid areas approximates the area over an interval, as shown in figure 6.7.

The trapezoidal rule is very easy to execute in just a few lines of code. Alternatively, we can utilize NumPy's `np.trapz` method to take the area of an inputted array. Let's apply the trapezoidal rule to our normal distribution. We want to test how well the rule approximates the total area covered by `normal_likelihoods`. Ideally, that area will approximate 1.0.

Listing 6.27 Approximating the area using the trapezoidal rule

The area of each trapezoid equals the sum of two consecutive likelihoods divided by 2. The x-coordinate distance between the trapezoid sides is 1, so it doesn't factor into our calculations.

```

total_area = np.sum([normal_likelihoods[i:i + 2].sum() / 2
                     for i in range(normal_likelihoods.size - 1)])
assert total_area == np.trapz(normal_likelihoods)
print(f"Estimated area under the curve is {total_area}")

```

Estimated area under the curve is 1.000000000384808

Note that NumPy executes the trapezoidal rule in a mathematically more efficient manner.

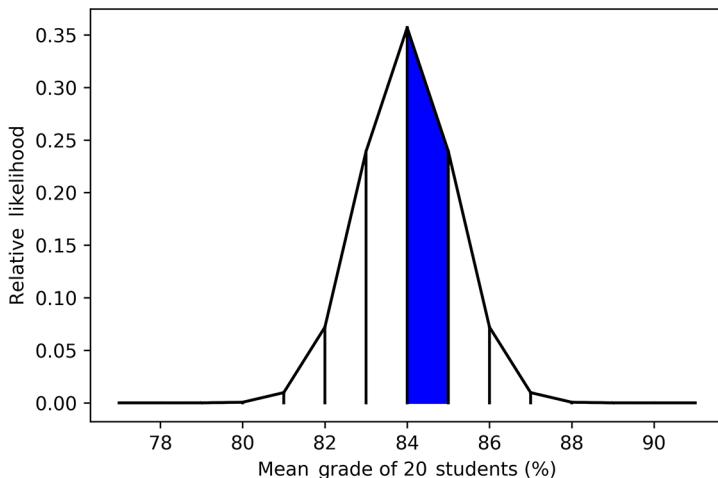


Figure 6.7 A normal distribution subdivided into trapezoidal regions. The lower-left corner of each trapezoid is located at an x coordinate of i . The parallel sides of each trapezoid are defined by `stats.norm.pdf(i)` and `stats.norm.pdf(i + 1)`. These parallel sides are 1 unit apart. The area of the trapezoid at position 84 has been shaded in. That area is equal to `(stats.norm.pdf(84) + stats.norm.pdf(85)) / 2`. Summing trapezoid areas across an interval range approximates the total area over that interval.

The estimated area is very close to 1.0, but it's not exactly equal to 1.0. In fact, it is slightly greater than 1.0. If we're willing to tolerate this minor imprecision, then our trapezoidal rule output is acceptable. Otherwise, we need a precise solution for the area of a normal distribution. That precision is provided by SciPy. We can access a mathematically exact solution using the `stats.norm.sf` method. This method represents the *survival function* of the normal curve. The survival function equals the distribution's area over an interval that's greater than some x . In other words, the survival function is the exact solution to the area approximated by `np.trapz(normal_likelihoods[x:])`. Thus, we can expect `stats.norm.sf(0, mean, sem)` to equal 1.0.

Listing 6.28 Computing the total area using SciPy

```
assert stats.norm.sf(0, mean, sem) == 1.0
```

Theoretically, the lower-bound x-value of a normal curve stretches into negative infinity. Therefore, this actual area is microscopically smaller than 1.0. However, the difference is so negligible that SciPy is unable to detect it. For our intents and purposes, we can treat the precise area as 1.0.

Similarly, we expect `stats.norm.sf(mean, mean, sem)` to equal 0.5, since the mean perfectly splits the normal curve into two equal halves (figure 6.8). Thus, the interval of values beyond the mean covers half the area of the normal curve. Meanwhile, we

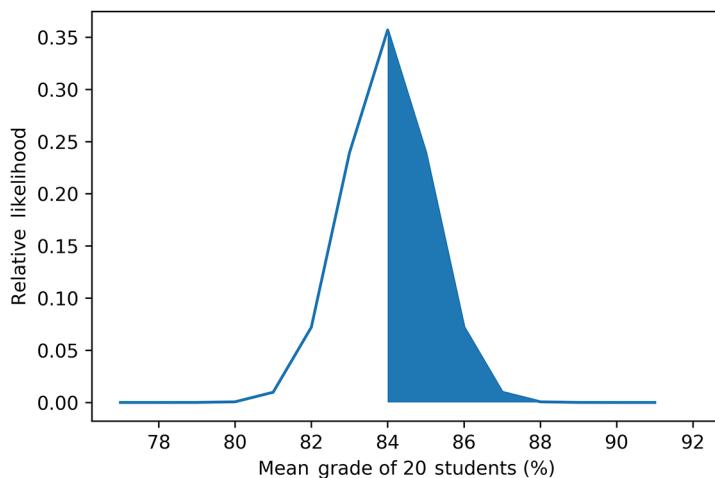


Figure 6.8 We've highlighted the area denoted by `stats.norm.sf(mean, mean, sem)`. That area covers an interval of values greater than or equal to the mean. The shaded area equals half the total area of the curve. Its exact value is 0.5.

expect `np.trapz(normal_likelihoods[:mean])` to approximate but not fully equal 0.5. Let's confirm.

Listing 6.29 Inputting the mean into the survival function

```
assert stats.norm.sf(mean, mean, sem) == 0.5
estimated_area = np.trapz(normal_likelihoods[:mean])
print(f"Estimated area beyond the mean is {estimated_area}")

Estimated area beyond the mean is 0.500000000192404
```

Common methods for measuring curve area

- `numpy.trapz(array)`—Executes the trapezoidal rule to estimate the area of array. The x-coordinate difference between the array elements is set to 1.
- `numpy.trapz(array, dx=dx)`—Executes the trapezoidal rule to estimate the area of array. The x-coordinate difference between the array elements is set to `dx`.
- `stats.norm.sf(x_value, mean, std)`—Returns the area beneath a normal curve, covering an interval that's greater than or equal to `x_value`. The mean and standard deviation of the normal curve are set to `mean` and `std`, respectively.
- `stats.norm.sf(x_array, mean, std)`—Returns an array of areas. These are obtained by executing `stats.norm.sf(e, mean, std)` on each element `e` of `x_array`.

Now, let's execute `stats.norm.sf(90, mean, sem)`. This returns the area over an interval of values lying beyond 90%. The area represents the likelihood of 20 students jointly acing an exam.

Listing 6.30 Computing the probability of a good collective grade

```
area = stats.norm.sf(90, mean, sem)
print(f"Probability of 20 students acing the exam is {area}")

Probability of 20 students acing the exam is 4.012555633463782e-08
```

As expected, the probability is low.

6.3.2 *Interpreting the computed probability*

The probability of all the students acing the exam is approximately 1 in 25 million. The exam is held just once a year, so it would take about 25 million years for a random arrangement of students to achieve that level of performance. Meanwhile, Mrs. Mann has been teaching for only 25 years. This represents a million-fold difference in magnitude. What are the odds of her presiding over a classroom with an average grade of at least 90%? Practically zero. We can conclude that such a classroom never existed!

NOTE The actual odds can be computed by running `1 - stats.binom.pmf(0, 25, stats.norm.sf(90, mean, sem))`. Can you figure out why?

Of course, we could be wrong. Perhaps a group of very talented fifth graders randomly wound up in the same classroom. This is highly unlikely, but nonetheless, it's possible. Also, our simple calculations didn't factor in changes to the exam. What if the exam gets easier every year? This would invalidate our treatment of the grades as a randomly drawn sample.

It seems our final conclusion is imperfect. We did the best we could, given what we knew, but some uncertainty remains. To eliminate that uncertainty, we'd need the missing dates for the graded exams. Unfortunately, that data was not provided. Quite commonly, statisticians are forced to make consequential decisions from limited records. Consider the following two scenarios:

- A coffee farm ships 500 tons of coffee beans per year in 5-pound bags. On average, 1% of the beans are moldy, with a standard deviation of 0.2%. The FDA permits a maximum of 3% moldy beans per bag. Does there exist a bag that violates the FDA's requirements?

We can apply the central limit theorem if we assume that mold growth is independent of time. However, mold could grow more rapidly in the humid summer months. Regrettably, we lack the records to confirm.

- A seaside town is building a seawall to defend against tsunamis. According to historical data, the average tsunami height is 23 feet, with a standard deviation

of 4 feet. The planned wall height is 33 feet. Is that height sufficient to protect the town?

It's tempting to assume that the tsunami average height will remain unchanged from year to year. However, certain studies indicate that climate change is causing sea levels to rise. Climate change might lead to more powerful tsunamis in the future. Regrettably, the scientific data is not conclusive enough to know for sure.

In both scenarios, we must make important decisions by relying on statistical techniques. These techniques depend on certain assumptions that might not hold. Consequently, we must exercise great caution when we draw conclusions from incomplete information. In the coming section, we continue to explore both the risks and advantages of making decisions based on limited data.

Summary

- A normal distribution's mean and standard deviation are determined by the position of its peak. The mean is equal to the x coordinate of the peak. Meanwhile, the standard deviation is equal to the inverse of the y coordinate multiplied by $(2\pi)^{1/2}$.
- A probability density function maps inputted float values to their likelihood weights. Taking the area underneath that curve produces a probability.
- Repeatedly sampling the mean from almost any distribution produces a normal curve. The mean of the normal curve approximates the mean of the underlying distribution. Also, the variance of the normal curve multiplied by the sample size approximates the variance of the underlying distribution.
- The *standard error of the mean* (SEM) equals the population standard deviation divided by the square root of the sample size. Consequently, dividing the population variance by the sample size and subsequently taking the square root also generates the SEM. The SEM, coupled with the population mean, allows us to compute the probability of observing certain sample combinations.
- The *trapezoidal rule* allows us to estimate the area under a curve by decomposing that curve into trapezoidal units. Then we simply sum over the areas of each trapezoid.
- A *survival function* measures a distribution's area over an interval that's greater than some x .
- We must cautiously consider our assumptions while making inferences from limited data.



Statistical hypothesis testing

This section covers

- Comparing sample means to population means
- Comparing means of two distinct samples
- What is statistical significance?
- Common statistical errors and how to avoid them

Many ordinary people are forced to make hard choices every day. This is especially true of jurors in the American justice system. Jurors preside over a defendant's fate during a trial. They consider the evidence and then decide between two competing hypotheses:

- The defendant is innocent.
- The defendant is guilty.

The two hypotheses are not weighted equally: the defendant is presumed to be innocent until proven guilty. Thus, the jurors assume that the innocence hypothesis is true. They can only reject the innocence hypothesis if the prosecution's evidence is convincing. Yet the evidence is rarely 100% conclusive, and some doubt of the defendant's guilt remains. That doubt is factored into the legal process. The jury is instructed to accept the innocence hypothesis if there is "reasonable doubt"

of the defendant's guilt. They can only reject the innocence hypothesis if the defendant appears guilty "beyond a reasonable doubt."

Reasonable doubt is an abstract concept that's hard to define precisely. Nonetheless, we can distinguish between reasonable and unreasonable doubt across a range of real-world scenarios. Consider the following two trial cases:

- DNA evidence links the defendant directly to the crime. There is a 1 in a billion chance that the DNA does not belong to the defendant.
- Blood-type evidence links the defendant directly to the crime. There is a 1 in 15 chance that the blood does not belong to the defendant.

In the first scenario, the jury cannot be 100% certain of the defendant's guilt. There is a 1 in a billion chance that an innocent defendant is on trial. Such circumstances, however, are incredibly unlikely. It's not reasonable to assume that this is the case. Thus, the jury should reject the innocence hypothesis.

Meanwhile, in the second scenario, the doubt is much more prevalent: 1 in 15 people share the same blood type as the defendant. It's reasonable to assume that someone else could have been present at the crime scene. While the jurors might doubt the defendant's innocence, they will also reasonably doubt the defendant's guilt. Thus, the jurors can't reject the innocence hypothesis unless additional proof of guilt is offered.

In our two scenarios, the jurors are carrying out a *statistical hypothesis test*. Such tests allow statisticians to choose between two competing hypotheses, both of which arise from uncertain data. One of the hypotheses is accepted or rejected based on a measured level of doubt. In this section, we explore several well-known statistical hypothesis testing techniques. We begin with a simple test to measure whether a sample mean noticeably deviates from an existing population.

7.1 Assessing the divergence between sample mean and population mean

In section 6, we used statistics to analyze a single fifth-grade classroom. Now, let's imagine a scenario where we analyze every fifth-grade classroom in North Dakota. One spring day, all fifth graders in the state are given the same assessment exam. The exam grades are fed into North Dakota's assessment database, and the population mean and variance are computed across all grades in the state. According to the records, the population mean is 80, and the population variance is 100. Let's quickly store these values for later use.

Listing 7.1 Population mean and variance of North Dakota grades

```
population_mean = 80
population_variance = 100
```

Next, suppose we travel to South Dakota and encounter a fifth-grade class whose mean exam grade equals 84%. This 18-student class has outperformed North Dakota's

population by 4 percentage points. Are fifth graders in South Dakota better educated than their North Dakota counterparts? If so, North Dakota should incorporate South Dakotan teaching methods into the curriculum. The curriculum adjustment would be costly, but the payoff to the students would be worth it. Of course, it's also possible that the observed exam difference is a mere statistical fluke. Which is it? We'll try to find out using hypothesis testing.

We face two rival possibilities. First, it's possible that the overall student population is identical across the neighboring states. In other words, a typical South Dakota classroom is no different from a typical North Dakota classroom. Under such circumstances, South Dakota's population mean and variance values would be indistinguishable from those of its neighbor. Statisticians refer to this hypothetical parameter equivalency as the *null hypothesis*. If the null hypothesis is true, then our high-performing South Dakota classroom is simply an outlier and doesn't represent the actual mean.

Alternatively, it's feasible that the classroom's high performance is representative of South Dakota's general population. Thus the state's mean and variance values would differ from North Dakota's population parameters. Statisticians call this the *alternative hypothesis*. If the alternative hypothesis is true, we'll update North Dakota's fifth-grade curriculum. However, the alternative hypothesis is only true when the null hypothesis is false (and vice versa). Therefore, to justify the curriculum overhaul, we must first show that the null hypothesis is unlikely to be true. We can measure this likelihood using the central limit theorem.

Let's temporarily assume that the null hypothesis is true and both Dakotas share the same population mean and variance. Consequently, we can model our 18-student classroom as a random sample taken from a normal distribution. That distribution's mean will equal `population_mean`, and its standard deviation will equal the standard error of the mean (SEM), defined as `(population_variance / 18) ** 0.5`.

Listing 7.2 Normal curve parameters if the null hypothesis is true

```
mean = population_mean
sem = (population_variance / 18) ** 0.5
```

If the null hypothesis is true, the probability of encountering an average exam grade of at least 84% is equal to `stats.norm.sf(84, mean, sem)`. Let's check that probability.

Listing 7.3 Finding the probability of a high-performance grade

```
prob_high_grade = stats.norm.sf(84, mean, sem)
print(f"Probability of an average grade >= 84 is {prob_high_grade}")
```

Probability of an average grade >= 84 is 0.044843010885182284

Under the null hypothesis, a random South Dakotan classroom will obtain an average grade of at least 84% with a probability of 0.044. This probability is low, and hence the

4% grade difference with the population mean appears extreme. But is it actually extreme? In section 1, we asked a similar question when we examined the likelihood of observing 8 heads out of 10 coin flips. In our coin analysis, we summed the probability of overperformance with the probability of underperformance. In other words, we summed the probability of observing eight or more heads with the probability of observing two heads or fewer. Here, our dilemma is identical. Analyzing exam overperformance is insufficient to evaluate extremeness; we must also consider the likelihood of an equally extreme underperformance. Therefore, we need to compute the probability of observing a sample mean that is at least four percentage points below the population mean of 80%.

We will now compute the probability of observing an exam average that's less than or equal to 76%. The calculation can be carried out with SciPy's `stats.norm.cdf` method, which computes the *cumulative distribution function* of the normal curve. A cumulative distribution function is the direct opposite of the survival function, as seen in figure 7.1. Applying `stats.norm.cdf` to x returns the area under a normal curve that ranges from negative infinity to x .

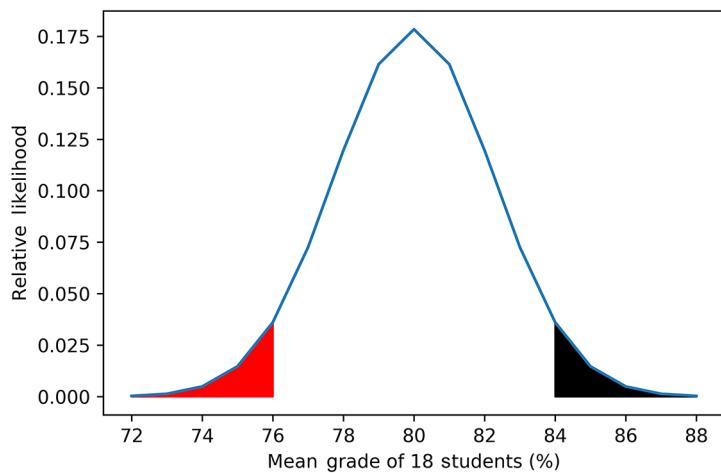


Figure 7.1 Two areas are highlighted beneath a normal curve. The leftmost area covers all x -values that are less than or equal to 76%. We can compute that area using the cumulative distribution function. To execute the function, we simply need to call `stats.norm.cdf(76, mean, sem)`. Meanwhile, the rightmost area covers all x -values that are at least 84%. We can compute that area using the survival function. To execute the function, we call `stats.norm.sf(84, mean, sem)`.

We now use `stats.norm.cdf` to find the probability of observing an unusually low average grade.

Listing 7.4 Finding the probability of a low-performance grade

```
prob_low_grade = stats.norm.cdf(76, mean, sem)
print(f"Probability of an average grade <= 76 is {prob_low_grade}")
```

```
Probability of an average grade <= 76 is 0.044843010885182284
```

It appears that `prob_low_grade` is equal to `prob_high_grade`. This equality arises from the symmetric shape of the normal curve. The cumulative distribution and the survival function are mirror images that are reflected across the mean. Thus, `stats.norm.sf(mean + x, mean, sem)` always equals `stats.norm.cdf(mean - x, mean, sem)` for any input `x`. Next, we visualize both functions to confirm their reflection across a vertically plotted mean (figure 7.2).

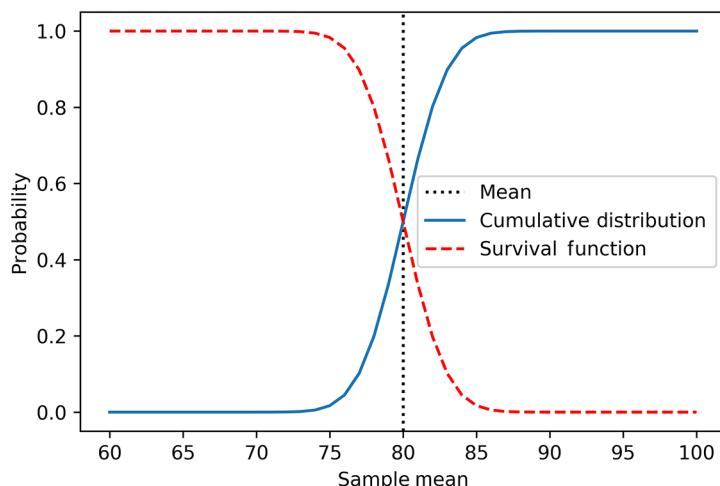


Figure 7.2 A cumulative distribution function of a normal distribution plotted together with the survival function. The cumulative distribution function and the survival function are mirror images. They are reflected across the normal curve's mean, which is plotted as a vertical line.

Listing 7.5 Comparing the survival and cumulative distribution functions

```
for x in range(-100, 100):
    sf_value = stats.norm.sf(mean + x, mean, sem)
    assert sf_value == stats.norm.cdf(mean - x, mean, sem)

plt.axvline(mean, color='k', label='Mean', linestyle=':')
x_values = range(60, 101)
plt.plot(x_values, stats.norm.cdf(x_values, mean, sem),
         label='Cumulative Distribution')
plt.plot(x_values, stats.norm.sf(x_values, mean, sem),
         label='Survival Function', linestyle='--', color='r')
```

```
plt.xlabel('Sample Mean')
plt.ylabel('Probability')
plt.legend()
plt.show()
```

Now we are ready to sum `prob_high_grade` and `prob_low_grade`. Due to symmetry, that sum equals $2 * \text{prob_high_grade}$. Conceptually, the sum represents the probability of observing an extreme deviation from the population mean when the null hypothesis is true. Statisticians refer to this null-hypothesis-driven probability as the *p-value*. Let's print the p-value arising from our data.

Listing 7.6 Computing the null-hypothesis-driven p-value

```
p_value = prob_low_grade + prob_high_grade
assert p_value == 2 * prob_high_grade
print(f"The p-value is {p_value}")
```

The p-value is 0.08968602177036457

Under the null hypothesis, there is approximately a 9% chance of observing the grade extreme at random. It's therefore plausible that the null hypothesis is true and the extreme test average is just a random fluctuation. We haven't definitively proved this, but our calculations raise serious doubts about restructuring North Dakota's fifth-grade curriculum. What if the average of the South Dakotan class had equaled 85%, not 84%? Let's check if that slight grade shift would have influenced our p-value.

Listing 7.7 Computing the p-value for an adjusted sample mean

```
def compute_p_value(observed_mean, population_mean, sem):
    mean_diff = abs(population_mean - observed_mean)
    prob_high = stats.norm.sf(population_mean + mean_diff, population_mean, sem)
    return 2 * prob_high

new_p_value = compute_p_value(85, mean, sem)
print(f"The updated p-value is {new_p_value}")
```

The updated p-value is 0.03389485352468927

A tiny increase in the average grade has caused a threefold decrease in the p-value. Now, under the null hypothesis, there's only a 3.3% chance of observing an average test grade that's at least as extreme as 85%. This likelihood is low, and we might therefore be tempted to reject the null hypothesis. Should we accept the alternative hypothesis and invest our time and money in revamping North Dakota's school system?

This is not an easy question to answer. Generally, statisticians tend to reject the null hypothesis if the p-value is less than or equal to 0.05. The threshold of 0.05 is called the *significance level*, and p-values below that threshold are deemed to be *statistically significant*. However, 0.05 is just an arbitrary cutoff intended to heuristically uncover

interesting data, not to make critical decisions. The threshold was first introduced in 1935 by famed statistician Ronald Fisher; later, Fisher said that the significance level should not remain static and should be manually adjusted based on the nature of the analysis. Regrettably, by then it was too late: the 0.05 cutoff had been adopted as our standard measure of significance. Today, most statisticians agree that a p-value below 0.05 implies an interesting signal in the data, so a p-value of 0.033 is sufficient to temporarily reject the null hypothesis and get one's data published in a scientific journal. Unfortunately, the threshold of 0.05 doesn't actually arise from the laws of mathematics and statistics: it's an ad hoc value chosen by the academic community as a requirement for research publication. As a consequence, many research journals are flooded with *type I errors*. A type I error is defined as an erroneous rejection of the null hypothesis. Such errors occur when random data fluctuations are interpreted as genuine deviations from the population mean. Scientific articles containing type I errors falsely assert a difference between means where none exists.

How do we limit type I errors? Well, some scientists believe that a threshold of 0.05 is unreasonably high and that we should only reject the null hypothesis if the p-value is much lower. But there is currently no consensus on whether using a lower threshold is appropriate, since doing so would lead to an increase in *type II errors*, in which we wrongly reject the alternative hypothesis. When scientists commit a type II error, they fail to notice a legitimate discovery.

Selecting an optimal significance level is difficult. Nevertheless, let's temporarily set the significance level to a very stringent value of 0.001. What would be the minimum grade average that would fall below this threshold? Let's find out. We loop through all grade averages above 80%, computing the p-value as we go. We stop when we encounter a p-value that's less than or equal to 0.001.

Listing 7.8 Scanning for a stringent p-value result

```
for grade in range(80, 100):
    p_value = compute_p_value(grade, mean, sem)
    if p_value < 0.001:
        break

print(f"An average grade of {grade} leads to a p-value of {p_value}")

An average grade of 88 leads to a p-value of 0.0006885138966450773
```

Given the new threshold, we would require an average grade of at least 88% to reject the null hypothesis. Thus, an average grade of 87% would not be considered statistically significant, even though it's noticeably higher than the population mean. Our lowering of the cutoff has inevitably exposed us to an increased risk of type II errors. Consequently, in this book, we maintain the commonly accepted p-value cutoff of 0.05. But we also proceed with excessive caution to avoid erroneously rejecting the null hypothesis. In particular, we do our best to minimize the most common cause of type I errors and the topic of the next subsection: data dredging.

7.2 Data dredging: Coming to false conclusions through oversampling

Sometimes, statistics students utilize the p-value incorrectly. Consider the following simple scenario. Two roommates pour out a bag of candy. The bag contains multiple candy pieces in five different colors. There are more blue candies in the bag than any other individual color. The first roommate assumes that blue is the dominant color in any candy bag. The second roommate disagrees: she computes the p-value based on the null hypothesis that all colors occur with equal likelihood. That p-value is greater than 0.05. However, the first roommate refuses to back down. He opens another bag of candy. The p-value is recomputed from the contents of that bag. This time, the p-value is equal to 0.05. The first roommate claims victory: he asserts that given the low p-value, the null hypothesis is probably false. Yet he is wrong.

The first roommate fundamentally misconstrued the meaning of the p-value. He wrongly assumed it represents the probability of the null hypothesis being true. In fact, the p-value represents the probability of observing deviations if the null hypothesis is true. The difference between the definitions is subtle but very important: the first definition implies that the null hypothesis is likely to be false if the p-value is low; but the second definition guarantees that we'll eventually observe a low p-value by repeatedly counting candies, even when the null hypothesis is true. Furthermore, the frequency of low p-value observations will equal the p-value itself. Hence, if we open 100 bags of candy, we should expect to observe a p-value of 0.05 approximately five times. By taking random measurements repeatedly, we will eventually obtain a statistically significant result, even if no statistical significance exists!

Running the same experiment too many times increases our risk of type I errors. Let's explore this notion in the context of our fifth-grade exam analysis. Suppose that North Dakota's statewide test performance does not diverge from the exam results in the other 49 states. More precisely, we'll assume that the national mean and variance equal North Dakota's `population_mean` and `population_variance` exam-grade results. Thus, the null hypothesis is true for all the states in the United States.

Furthermore, let's assume we don't yet know that the null hypothesis is always true. The only things we know for sure are North Dakota's population mean and variance. We set out on a road trip in search of a state whose grade distribution differs from North Dakota's distribution. Unfortunately, our search is bound to be futile because no such state exists.

Our first stop is Montana. There, we choose a random fifth-grade classroom of 18 students. We then compute the classroom's average grade. Since the null hypothesis is secretly true, we can simulate the value of that average grade by sampling from a normal distribution defined by `mean` and `sem`. Let's simulate the exam performance of the class by calling `np.random.normal(mean, sem)`. The method call samples from a normal distribution defined by the inputted variables.

Listing 7.9 Randomly sampling Montana's exam performance

```
np.random.seed(0)
random_average_grade = np.random.normal(mean, sem)
print(f"Average grade equals {random_average_grade:.2f}")
```

Average grade equals 84.16

The average exam grade in the class equals approximately 84.16. We can determine if that average is statistically significant by checking if its p-value is less than or equal to 0.05.

Listing 7.10 Testing the significance of Montana's exam performance

```
if compute_p_value(random_average_grade, mean, sem) <= 0.05:
    print("The observed result is statistically significant")
else:
    print("The observed result is not statistically significant")
```

The observed result is not statistically significant

The average grade is not statistically significant. We will continue our journey and visit a single 18-student classroom in each of the remaining 48 states, computing the grade average for each classroom. The p-value will also be computed. Once we discover a statistically significant p-value, our journey will end.

The following code simulates our travels. It iterates through the remaining 48 states, randomly drawing a grade average for each state. Once a statistically significant grade average is discovered, the iteration loop will stop.

Listing 7.11 Randomly searching for a significant state result

```
np.random.seed(0)
for i in range(1, 49):
    print(f"We visited state {i + 1}")
    random_average_grade = np.random.normal(mean, sem)
    p_value = compute_p_value(random_average_grade, mean, sem)
    if p_value <= 0.05:
        print("We found a statistically significant result.")
        print(f"The average grade was {random_average_grade:.2f}")
        print(f"The p-value was {p_value}")
        break

if i == 48:
    print("We visited every state and found no significant results.")

We visited state 2
We visited state 3
We visited state 4
We visited state 5
We found a statistically significant result.
The average grade was 85.28
The p-value was 0.025032993883401307
```

The fifth state that we visit produces a statistically significant result! A classroom in the state has a grade average of 85.28. The associated p-value of 0.025 falls below our 0.05 cutoff. It appears we can reject the null hypothesis! However, this conclusion is erroneous since the null hypothesis is true. What went wrong? Well, as stated earlier, the frequency of low p-value observations will equal the p-value itself. Therefore, we expect to encounter a p-value of 0.025 approximately 2.5% of the time, even if the null hypothesis is true. Since we are traveling across 49 states, and 2.5% of 49 is 1.225, we should expect to visit approximately one state with a random p-value of roughly 0.025.

Our quest to find a statistically significant result was doomed from the start because we have misused statistics. We have indulged in the cardinal statistical sin of *data dredging*, also known as *data fishing* or *p-hacking*. In data dredging, experiments are repeated over and over until a statistically significant result is found. Then the statistically significant result is presented to others, while the remaining failed experiments are discarded. Data dredging is the most common cause of type I errors in scientific publications. Sadly, sometimes researchers formulate a hypothesis and repeat an experiment until the particular false hypothesis is validated as true. For instance, a researcher might hypothesize that certain candies cause cancer in mice. The researcher proceeds to feed a specific candy brand to a group of mice, but no cancer link is found. The researcher then switches the brand of candy and runs the experiment again. And again. And again. Years later, a brand of candy linked to cancer is finally found. Of course, the actual experiment outcome is borderline fraudulent. No real statistical link exists between cancer and candy—the researcher has simply run the experiment too many times, until a low p-value was randomly measured.

Avoiding data dredging is not difficult: we must simply choose in advance a finite number of experiments to run. Then we set our significance level to 0.05 divided by the planned experiment count. This simple technique is known as the *Bonferroni correction*. Let's repeat our analysis of US exam performance using the Bonferroni correction. The analysis requires us to visit 49 states to evaluate 49 classrooms, so our significance level should be set to $0.05 / 49$.

Listing 7.12 Using the Bonferroni correction to adjust significance

```
num_planned_experiments = 49
significance_level = .05 / num_planned_experiments
```

We rerun our analysis, which will terminate if we encounter a p-value that's less than or equal to `significance_level`.

Listing 7.13 Rerunning an analysis using an adjusted significance level

```
np.random.seed(0)
for i in range(49):
    random_average_grade = np.random.normal(mean, sem)
    p_value = compute_p_value(random_average_grade, mean, sem)
```

```

if p_value <= significance_level:
    print("We found a statistically significant result.")
    print(f"The average grade was {random_average_grade:.2f}")
    print(f"The p-value was {p_value}")
    break

if i == 48:
    print("We visited every state and found no significant results.")

We visited every state and found no significant results.

```

We've visited 49 states and found no statistically significant deviations from North Dakota's population mean and variance. The Bonferroni correction has allowed us to avoid a type I error.

As a final word of caution, the Bonferroni correction only works if we divide 0.05 by the number of planned experiments. It is not effective if we divide by the count of completed experiments. For instance, if we plan to run 1,000 experiments, but the p-value of our very first experiment equals 0.025, we should not alter our significance level to $0.05 / 1$. Similarly, if the p-value of the second completed experiment equals 0.025, we should maintain a significance level of $0.05 / 1000$ rather than adjust it to $0.05 / 2$. Otherwise, we risk wrongly biasing our conclusions toward our first few experimental outcomes. All experiments must be treated equally for us to draw a fair, correct conclusion.

The Bonferroni correction is a useful technique for more accurate hypothesis testing. It can be applied to all kinds of statistical hypothesis tests beyond just simple tests that exploit both population mean and variance. This is fortunate because statistical tests vary in their levels of complexity. In the next subsection, we explore a more complicated test that does not depend on knowing the population variance.

7.3 Bootstrapping with replacement: Testing a hypothesis when the population variance is unknown

We are easily able to compute a p-value using the population mean and variance. Regrettably, in many real-life circumstances, the population variance is not known. Consider the following scenario, in which we own a very large aquarium. It holds 20 tropical fish of lengths varying from 2 cm to nearly 120 cm. The average fish length equals 27 cm. We represent these fish lengths using the `fish_lengths` array.

Listing 7.14 Defining lengths of fish in an aquarium

```

fish_lengths = np.array([46.7, 17.1, 2.0, 19.2, 7.9, 15.0, 43.4,
                      8.8, 47.8, 19.5, 2.9, 53.0, 23.5, 118.5,
                      3.8, 2.9, 53.9, 23.9, 2.0, 28.2])
assert fish_lengths.mean() == 27

```

Does our aquarium accurately capture the distributed lengths of real tropical fish? We would like to find out. A trusted source informs us that the population mean length of wild tropical fish equals 37 cm. There is a sizable 10 cm difference between the

population mean and our sample mean. That difference feels significant, but feelings have no place in rigorous statistics. We must determine if the difference is statistically significant in order to draw a valid conclusion.

Thus far, we have measured statistical significance using our `compute_p_value` function. However, we cannot apply this function to our fish data since we don't know the population variance! Without the population variance, we cannot compute the SEM, which is a variable required to run `compute_p_value`. How do we find the standard error of the mean when the population variance is not known?

At first glance, it appears we have no way of finding the SEM. We could naively treat our sample variance as an estimate of the population variance by executing `fish_lengths.var()`. Unfortunately, small samples are prone to random variance fluctuations, so any such estimate is highly unreliable. Thus, we are stuck. We face a seemingly impenetrable problem and must rely on a seemingly impossible solution: *bootstrapping with replacement*. The term *bootstrapping* originates from the phrase “pull yourself up by your bootstraps.” The phrase refers to lifting yourself into the air by pulling on the laces of your boots. Of course, doing so is impossible. In bootstrapping with replacement, we'll attempt something equally impossible by computing a p-value directly from our limited data! Despite this seemingly ludicrous solution, we will be successful in our efforts.

We begin the bootstrapping procedure by removing a random fish from the aquarium. The length of the selected fish is measured for later use.

Listing 7.15 Sampling a random fish from the aquarium

```
np.random.seed(0)
random_fish_length = np.random.choice(fish_lengths, size=1)[0]
sampled_fish_lengths = [random_fish_length]
```

Now we place the chosen fish back into the aquarium. This replacement step is where bootstrapping with replacement gets its name. After we return the fish, we reach into the aquarium again and choose another fish at random. There is a 1 in 20 chance that we'll select the same fish as before, which is perfectly acceptable. We record the length of the chosen fish and place it back into the water. Then we repeat the procedure 18 more times until 20 random fish lengths have been measured.

Listing 7.16 Sampling 20 random fish with repetition

```
np.random.seed(0)
for _ in range(20):
    random_fish_length = np.random.choice(fish_lengths, size=1)[0]
    sampled_fish_lengths.append(random_fish_length)
```

The `sampled_fish_lengths` list contains 20 measurements, all taken from the 20-element `fish_lengths` array. However, the elements of `fish_lengths` and `sampled_fish_lengths` are not identical. Due to random sampling, the mean values of the array and the list are likely to differ.

Listing 7.17 Comparing the sample mean to the aquarium mean

```
sample_mean = np.mean(sampled_fish_lengths)
print(f"Mean of sampled fish lengths is {sample_mean:.2f} cm")
```

Mean of sampled fish lengths is 26.03 cm

The mean of the sampled fish lengths is 26.03 cm. It deviates from our original mean by 0.97 cm. Thus, sampling with replacement has introduced some variance into our observations. If we sample another 20 measurements from the aquarium, we can expect the subsequent sample mean to also deviate from 27 cm. Let's confirm by repeating our sampling using a single line of code: `np.random.choice(fish_lengths, size=20, replace=True)`. Setting the `replace` parameter to True ensures that we sample with replacement from the `fish_lengths` array.

Listing 7.18 Sampling with replacement using NumPy

As a side note, the `replace` parameter is currently set to True by default within the function.

```
np.random.seed(0)
new_sampled_fish_lengths = np.random.choice(fish_lengths, size=20,
                                             replace=True)
new_sample_mean = new_sampled_fish_lengths.mean()
print(f"Mean of the new sampled fish lengths is {new_sample_mean:.2f} cm")
```

Mean of the new sampled fish lengths is 26.16 cm

The new sample mean equals 26.16 cm. Our mean values will fluctuate when we sample with replacement: fluctuation implies randomness, and thus our mean values are randomly distributed. Let's explore the shape of this random distribution by repeating our sampling process 150,000 times. During iteration, we compute the mean of 20 random fish; then we plot a histogram of the 150,000 sampled means (figure 7.3).

Listing 7.19 Plotting the distribution of 150,000 sampled means

```
np.random.seed(0)
sample_means = [np.random.choice(fish_lengths,
                                 size=20,
                                 replace=True).mean()
                for _ in range(150000)]
likelihoods, bin_edges, _ = plt.hist(sample_means, bins='auto',
                                      edgecolor='black', density=True)
plt.xlabel('Binned Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

The histogram we've generated is not a normal curve. The shape is not symmetric: its left side rises more steeply than its right side. Mathematicians refer to this asymmetry as a *skew*. We can confirm the skew in our histogram by calling `stats.skew(sample_means)`. The `stats.skew` method returns a nonzero value when the inputted data is asymmetric.

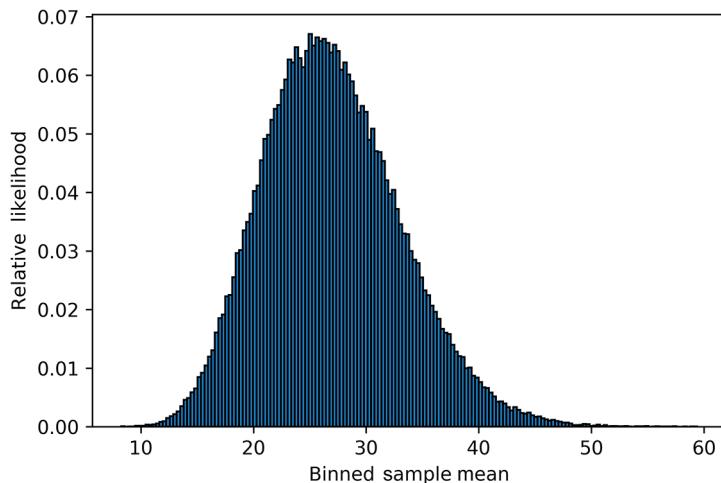


Figure 7.3 A histogram of sample means computed using sampling with replacement. The histogram is not bell shaped; it's asymmetric.

Listing 7.20 Computing the skew of an asymmetric distribution

```
assert abs(stats.skew(sample_means)) > 0.4
```

No data is ever perfectly symmetric, and the skew is rarely 0.0, even if the data is sampled from a normal curve. However, normal data tends to have a skew that is exceedingly close to 0.0. Any data with a skew whose absolute value is greater than 0.04 is very unlikely to come from a normal distribution.

Our asymmetric histogram cannot be modeled using a normal distribution. Nevertheless, the histogram represents a continuous probability distribution. Like all continuous distributions, the histogram can be mapped to a probability density function, a cumulative distribution function, and a survival function. Knowing the function outputs would be useful. For instance, the survival function would give us the probability of observing a sample mean that's greater than our population mean. We could obtain the function outputs by manually writing code that computes the curve area using the `bin_edges` and `likelihoods` arrays.

Alternatively, we can just use SciPy, which provides us with a method for obtaining all three functions from the histogram. That method is `stats.rv_histogram`, which takes as input a tuple defined by the `bin_edges` and `likelihoods` arrays. Calling `stats.rv_histogram((likelihoods, bin_edges))` returns a `random_variable` SciPy object containing `pdf`, `cdf`, and `sf` methods, just like `stats.norm`. The `random_variable.pdf` method outputs the probability density for the histogram. Likewise, the `random_variable.cdf` and `random_variable.sf` methods output the cumulative distribution function and the survival function, respectively.

The following code computes the `random_variable` object arising from the histogram. Then we plot the probability density function by calling `random_variable.pdf(bin_edges)` (figure 7.4).

Listing 7.21 Fitting data to a generic distribution using SciPy

```
random_variable = stats.rv_histogram((likelihoods, bin_edges))
plt.plot(bin_edges, random_variable.pdf(bin_edges))
plt.hist(sample_means, bins='auto', alpha=0.1, color='r', density=True)
plt.xlabel('Sample Mean')
plt.ylabel('Relative Likelihood')
plt.show()
```

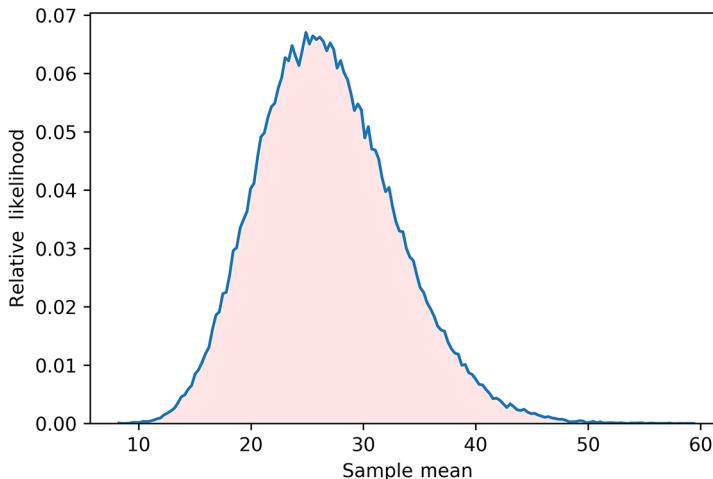


Figure 7.4 An asymmetric histogram overlaid with its probability density function. We used SciPy to learn the probability density function from the histogram.

As expected, the probability density function perfectly resembles the histogram shape. Let's now plot both the cumulative distribution function and the survival function associated with `random_variable`. We should anticipate that the two plotted functions will not be symmetric around the mean. To check for this asymmetry, we plot the distribution's mean using a vertical line. We obtain that mean by calling `random_variable.mean()` (figure 7.5).

Listing 7.22 Plotting the mean and interval areas for a generic distribution

```
rv_mean = random_variable.mean()
print(f"Mean of the distribution is approximately {rv_mean:.2f} cm")

plt.axvline(random_variable.mean(), color='k', label='Mean', linestyle=':')
plt.plot(bin_edges, random_variable.cdf(bin_edges),
         label='Cumulative Distribution')
```

```

plt.plot(bin_edges, random_variable.sf(bin_edges),
         label='Survival', linestyle='--', color='r')
plt.xlabel('Sample Mean')
plt.ylabel('Probability')
plt.legend()
plt.show()

```

Mean of the distribution is approximately 27.00 cm

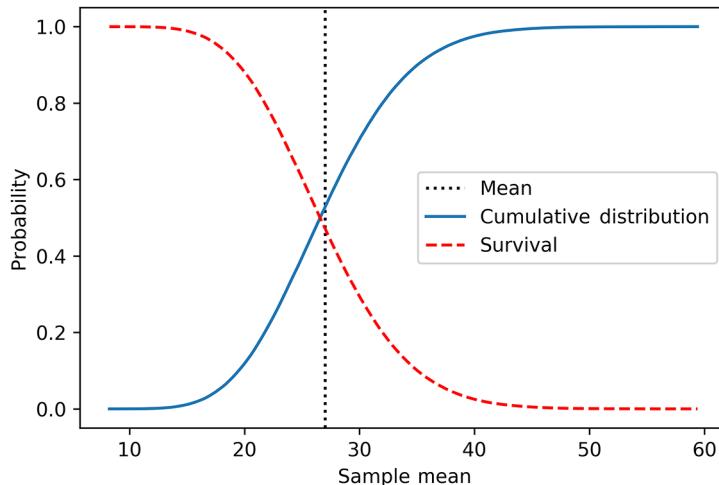


Figure 7.5 A cumulative distribution function of an asymmetric distribution plotted together with the survival function. The two functions no longer symmetrically reflect across the mean as they did in our normal-curve analysis. Therefore, we can no longer compute the p-value simply by doubling the survival function output.

The mean of the distribution is approximately 27 cm, which is also the mean length of the fish in our aquarium. A random fish sample is likely to produce a value that is close to the aquarium's mean. However, sampling with replacement sometimes produces a value greater than 37 cm or less than 17 cm. The probabilities of observing these extremes can be computed from our two plotted functions. Let's examine these two functions in more detail.

Based on our plot, the cumulative distribution function and the survival function are not mirror images. Nor do they intersect directly at the mean, as they did in our normal-curve analysis. Our distribution doesn't behave like a symmetric normal curve, which leads to certain consequences. Using the symmetric curve, we could compute the p-value by doubling the survival function. In our asymmetric distribution, the survival function by itself is insufficient for computing tail-end probabilities. Fortunately, we can use both the survival function and the cumulative distribution function to uncover probabilities of extreme observations. Using these probabilities, we can evaluate the statistical significance.

We can measure significance by answering this question: what is the probability that 20 sampled (with replacement) fish produce a mean as extreme as the population mean? As a reminder, the population mean is 37 cm, which is 10 cm greater than our distribution mean. Therefore, *extremeness* is defined as a sampled output that's at least 10 cm away from `rv_mean`. Based on our previous discussions, the problem can be broken down into computing two distinct values. First we must compute the probability of observing a sample mean that's at least 37 cm, and then we must compute the probability of observing a sample mean that's less than or equal to 17 cm. The former probability equals `random_variable.sf(37)`, while the latter equals `random_variable.cdf(17)`. Summing these two values will provide us with our answer.

Listing 7.23 Computing the probability of an extreme sample mean

```
prob_extreme= random_variable.sf(37) + random_variable.cdf(17)
print("Probability of observing an extreme sample mean is approximately "
      f"{prob_extreme:.2f}")
```

Probability of observing an extreme sample mean is approximately 0.10

The probability of observing an extreme value from our sampling is approximately 0.10. In other words, one-tenth of random aquarium samplings will produce a mean that's at least as extreme as the population mean. Our population mean is not as far from the aquarium mean as we thought. In fact, a mean discrepancy of 10 cm or more will appear in 10% of sampled fish outputs. Thus, the difference between our sample mean of 27 cm and our population mean of 37 cm is not statistically significant.

By now, all this should seem familiar. The `prob_extreme` value is just the p-value in disguise. When the null hypothesis is true, the difference between the sample mean and population mean will be at least 10 cm in 10% of sampled cases. This p-value of 0.1 is greater than our cutoff of 0.05. So, we cannot reject the null hypothesis. There is no statistically significant difference between our sample mean and population mean.

We've computed a p-value in a roundabout way. Some readers may be suspicious of our methods—after all, sampling from our limited collection of 20 fish seems like a strange way to draw statistical insights. Nevertheless, the described technique is legitimate. Bootstrapping with replacement is a reliable procedure for extracting the p-value, especially when dealing with limited data.

Useful methods for bootstrapping with replacement

- `rv = stats.rv_histogram((likelihoods, bin_edges))`—Creates a random variable object `rv` based on the histogram output of `likelihoods, bin_edges = np.hist(data)`.
- `p_value = rv.sf(head_extreme) + random_variable.cdf(tail_extreme)`—Computes a p-value from a random variable object based on the survival output and the cumulative distribution output of the head extreme and tail extreme, respectively.

- `z = np.random.choice(x, size=y, replace=True)`—Samples y elements from array x with replacement. The samples are stored in array z. In bootstrapping with replacement, `y == x.size`.

The bootstrapping technique has been rigorously studied for more than four decades. Statisticians have uncovered multiple variations of this technique for accurate p-value computation. We've just reviewed one such variation; now we will briefly introduce another. It has been shown that sampling with replacement approximates a dataset's SEM. Basically, the standard deviation of the sampled distribution is equal to the SEM when the null hypothesis is true. Thus, if the null hypothesis is true, our missing SEM is equal to `random_variable.std`. This gives us yet another way of finding the p-value. We simply need to execute `compute_p_value(27, 37, random_variable.std)`; that computed p-value should equal approximately 0.1. Let's confirm.

Listing 7.24 Using bootstrapping to estimate the SEM

```
estimated_sem = random_variable.std()
p_value = compute_p_value(27, 37, estimated_sem)
print(f"P-value computed from estimated SEM is approximately {p_value:.2f}")

P-value computed from estimated SEM is approximately 0.10
```

As expected, the computed p-value is approximately 0.1. We've shown how bootstrapping with replacement provides us with two divergent approaches for computing the p-value. The first approach requires us to do the following:

- 1 Sample with replacement from the data. Repeat tens of thousands of times to obtain a list of sample means.
- 2 Generate a histogram from the sample means.
- 3 Convert the histogram to a distribution using the `stats.rv_histogram` method.
- 4 Take the area beneath the left and right extremes of the distribution curve using the survival function and the cumulative distribution function.

Meanwhile, the second approach appears to be slightly simpler:

- 1 Sample with replacement from the data. Repeat tens of thousands of times to obtain a list of sample means.
- 2 Compute the standard deviation of the means to approximate the SEM.
- 3 Use the estimated SEM to carry out basic hypothesis testing using our `compute_p_value` function.

Let's briefly discuss a third approach, which is even easier to implement. This approach does not require a histogram, nor does it rely on a custom `compute_value_` function. Instead, the technique uses the law of large numbers introduced in section 2. According to that law, the frequency of observed events approximates the probability of event occurrence if the sample count is sufficiently large. Thus, we can estimate the

p-value simply by computing the frequency of extreme observations. Let's quickly apply this technique to `sample_means` by counting means that do not fall between 17 cm and 37 cm. We will divide the count by `len(sample_means)` in order to compute the p-value.

Listing 7.25 Computing the p-value from direct counts

```
number_extreme_values = 0
for sample_mean in sample_means:
    if not 17 < sample_mean < 37:
        number_extreme_values += 1

p_value = number_extreme_values / len(sample_means)
print(f"P-value is approximately {p_value:.2f}")

P-value is approximately 0.10
```

Bootstrapping with replacement is a simple but powerful technique for making inferences from limited data. However, the technique still presupposes the knowledge of a population mean. Unfortunately, in real-life situations, the population mean is rarely known. For instance, in this case study, we are required to analyze an online ad-click table that does not include a population mean. This missing information will not stop us: in the next subsection, we learn how to compare collected samples when both the population mean and the population variance are unknown.

7.4 Permutation testing: Comparing means of samples when the population parameters are unknown

Sometimes, in statistics, we need to compare two distinct sample means while the population parameters remain unknown. Let's explore one such scenario.

Suppose our neighbor also owns an aquarium. Her aquarium contains 10 fish whose average length is 46 cm. We represent these new fish lengths using the `new_fish_lengths` array.

Listing 7.26 Defining lengths of fish in a new aquarium

```
new_fish_lengths = np.array([51, 46.5, 51.6, 47, 54.4, 40.5, 43, 43.1,
                            35.9, 47.0])
assert new_fish_lengths.mean() == 46
```

We want to compare the contents of our neighbor's aquarium with our own. We begin by measuring the difference between `new_fish_lengths.mean()` and `fish_lengths.mean()`.

Listing 7.27 Computing the difference between two sample means

```
mean_diff = abs(new_fish_lengths.mean() - fish_lengths.mean())
print(f"There is a {mean_diff:.2f} cm difference between the two means")

There is a 19.00 cm difference between the two means
```

There is a 19 cm difference between the two aquarium means. That difference is substantial, but is it statistically significant? We want to find out. However, all our previous analyses have relied on a population mean. Currently, we have two sample means but no population mean. This makes it difficult to evaluate the null hypothesis, which assumes that fish from both aquariums share a population mean. This presumed shared value is now unknown. What should we do?

We need to reframe the null hypothesis so that it doesn't directly depend on the population mean. If the null hypothesis is true, then the 20 fish in the first aquarium and the 10 fish in the second aquarium are all drawn from the same population. Under the hypothesis, it doesn't really matter which 20 fish wind up in aquarium A and which 10 fish wind up in aquarium B. The arrangements of fish between the two aquariums will have little effect. Random rearrangements of the fish will cause the `mean_diff` variable to fluctuate, but that difference between means should fluctuate in a predictable manner.

Hence, we don't need to know the sample mean to evaluate the null hypothesis. Instead, we can focus on the random permutations of fish between the two aquariums. This will allow us to carry out a *permutation test*, where `mean_diff` is used to compute statistical significance. Like bootstrapping with replacement, the permutation test relies on random sampling of data.

We begin the permutation test by placing all 30 fish into a single aquarium. The unification of our fish can be modeled using the `np.hstack` method. The method takes as input a list of NumPy arrays, which are then merged together into a single NumPy array.

Listing 7.28 Merging two arrays using `np.hstack`

```
total_fish_lengths = np.hstack([fish_lengths, new_fish_lengths])
assert total_fish_lengths.size == 30
```

Once the fish are grouped together, we allow them to swim in random directions. This fully randomizes the positions of the fish in the aquarium. We use the `np.random.shuffle` method to shuffle the positions of the fish.

Listing 7.29 Shuffling the positions of merged fish

```
np.random.seed(0)
np.random.shuffle(total_fish_lengths)
```

Next, we choose 20 of our randomly shuffled fish. These 20 fish will be moved to a separate aquarium. The other 10 fish will remain. Once more, we'll have 20 fish in aquarium A and 10 fish in aquarium B. However, the mean lengths of the fish in each aquarium will probably differ from `fish_lengths.mean()` and `new_fish_lengths.mean()`, so the difference between mean fish lengths will also change. Let's confirm.

Listing 7.30 Computing the difference between two random sample means

```
random_20_fish_lengths = total_fish_lengths[:20]
random_10_fish_lengths = total_fish_lengths[20:]
mean_diff = random_20_fish_lengths.mean() - random_10_fish_lengths.mean()
print(f"The new difference between mean fish lengths is {mean_diff:.2f}")
```

The new difference between mean fish lengths is 14.33

The sampled difference between fish lengths is no longer 19 cm: now it is 14.33 cm. As expected, `mean_diff` is a fluctuating random variable, so we can find its distribution through random sampling. Next, we repeat our fish-shuffling procedure 30,000 times to obtain a histogram of `mean_diff` values (figure 7.6).

Listing 7.31 Plotting the fluctuating difference between means

```
np.random.seed(0)
mean_diffs = []
for _ in range(30000):
    np.random.shuffle(total_fish_lengths)
    mean_diff = total_fish_lengths[:20].mean() -
        total_fish_lengths[20:].mean()
    mean_diffs.append(mean_diff)

likelihoods, bin_edges, _ = plt.hist(mean_diffs, bins='auto',
                                     edgecolor='black', density=True)
plt.xlabel('Binned Mean Difference')
plt.ylabel('Relative Likelihood')
plt.show()
```

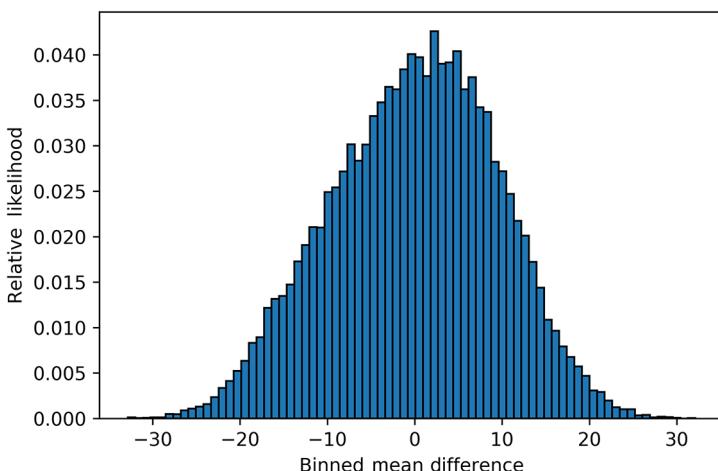


Figure 7.6 A histogram of sample mean differences computed using random rearrangements of samples into two distinct groups

Next, we fit the histogram to a random variable using the `stats.rv_histogram` method.

Listing 7.32 Fitting the histogram to a random variable

```
random_variable = stats.rv_histogram((likelihoods, bin_edges))
```

Finally, we use the `random_variable` object to carry out hypothesis testing. We want to know the probability of observing an extreme value when the null hypothesis is true. We define extremeness as a difference between means whose absolute value is at least 19 cm. Thus, our p-value will equal `random_variable.cdf(-19) + random_variable.sf(19)`.

Listing 7.33 Computing the permutation p-value

```
p_value = random_variable.sf(19) + random_variable.cdf(-19)
print(f"P-value is approximately {p_value:.2f}")
```

```
P-value is approximately 0.04
```

The p-value is approximately 0.04, which falls below our significance threshold of 0.05. Hence, the mean difference between fish lengths is statistically significant. The fish in the two aquariums do not originate from a shared distribution.

As an aside, we can simplify our permutation test by using the law of large numbers. We simply need to compute the frequency of extreme recorded samples, just as we did with bootstrapping with replacement. Let's use this alternative method to recompute our p-value of approximately 0.04.

Listing 7.34 Computing the permutation p-value from direct counts

```
number_extreme_values = 0.0
for min_diff in mean_diffs:
    if not -19 < min_diff < 19:
        number_extreme_values += 1

p_value = number_extreme_values / len(mean_diffs)
print(f"P-value is approximately {p_value:.2f}")
```

```
P-value is approximately 0.04
```

The permutation test allows us to statistically compare differences between two lists of collected samples. The nature of these samples isn't important; they could be fish lengths, or they could be ad-click counts. Hence, the permutation test could be very useful when we compare our recorded ad-click counts to uncover optimal ad colors during the case study resolution.

Summary

- Statistical hypothesis testing requires us to choose between two competing hypotheses. According to the *null hypothesis*, a pair of populations are identical. According to the *alternative hypothesis*, the pair of populations are not identical.
- To evaluate the null hypothesis, we must compute a *p-value*. The p-value equals the probability of observing our data when the null hypothesis is true. The null hypothesis is rejected if the p-value is lower than a specified *significance level* threshold. Typically, the significance level is set to 0.05.
- If we reject the null hypothesis, and the null hypothesis is true, we commit a *type I error*. If we fail to reject the null hypothesis and the alternative hypothesis is true, we commit a type II error.
- *Data dredging* increases our risk of type I errors. In data dredging, an experiment is repeated until the p-value falls below the significance level. We can minimize data dredging by carrying out a *Bonferroni correction*, in which the significance level is divided by the experiment count.
- We can compare a sample mean to a population mean and variance by relying on the central limit theorem. The population variance is needed to compute the SEM. If we're not provided with the population variance, we can estimate the SEM using *bootstrapping with replacement*.
- We can compare the means of two distinct samples by running a *permutation test*.

Analyzing tables using Pandas

This section covers

- Storing 2D tables using the Pandas library
- Summarizing 2D table content
- Manipulating row and column content
- Visualizing tables using the Seaborn library

The ad-click data for case study 2 is saved in a two-dimensional table. Data tables are commonly used to store information. The tables may be stored in different formats: some tables are saved as spreadsheets in Excel, and others are text-based CSV files in which the columns are separated by commas. The formatting of a table isn't important. What is important is its structure. All tables have structural features in common: every table contains horizontal rows and vertical columns, and quite often, column headers also hold explicit column names.

8.1 Storing tables using basic Python

Let's define a sample table in Python. The table stores measurements for various species of fish, in centimeters. Our measurement table contains three columns: Fish, Length, and Width. The Fish column stores a labeled species of fish, and the Length and Width columns specify the length and width of each fish species. We represent this table as a dictionary. The column names serve as dictionary keys, and these keys map to lists of column values.

Listing 8.1 Storing a table using Python data structures

```
fish_measures = {'Fish': ['Angelfish', 'Zebrafish', 'Killifish', 'Swordtail'],
                 'Length':[15.2, 6.5, 9, 6],
                 'Width': [7.7, 2.1, 4.5, 2]}
```

Suppose we want to know the length of a zebrafish. To obtain the length, we must first access the index of the 'Zebrafish' element in `fish_measures['Fish']`. Then we need to check that index in `fish_measures['Length']`. The process is slightly convoluted, as illustrated in the following code.

Listing 8.2 Accessing table columns using a dictionary

```
zebrafish_index = fish_measures['Fish'].index('Zebrafish')
zebrafish_length = fish_measures['Length'][zebrafish_index]
print(f"The length of a zebrafish is {zebrafish_length:.2f} cm")
```

The length of a zebrafish is 6.50 cm

Our dictionary representation is functional but also difficult to use. A better solution is provided by the Pandas library, which is designed for table manipulation.

8.2 Exploring tables using Pandas

Let's install the Pandas library. Once Pandas is installed, we will import it as `pd` using common Pandas usage convention.

NOTE Call `pip install pandas` from the command line terminal to install the Pandas library.

Listing 8.3 Importing the Pandas library

```
import pandas as pd
```

We now load our `fish_measures` tables into Pandas by calling `pd.DataFrame(fish_measures)`. That method call returns a Pandas DataFrame object. The term *data frame* is a common synonym for *table* in statistical software jargon. Basically, the DataFrame object will convert our dictionary into a two-dimensional table. According to convention, Pandas DataFrame objects are assigned to a variable `df`. Here, we execute `df = pd.DataFrame(fish_measures)` and then print the contents of `df`.

Listing 8.4 Loading a table into Pandas

```
df = pd.DataFrame(fish_measures)
print(df)

      Fish  Length  Width
0  Angelfish    15.2    7.7
1  Zebrafish     6.5    2.1
2  Killifish     9.0    4.5
3  Swordtail     6.0    2.0
```

The alignments between table rows and columns are clearly visible in the printed output. Our table is small and therefore easy to display. However, for larger tables, we might prefer to print only the first few rows. Calling `print(df.head(x))` prints just the first `x` rows in a table. Let's print the first two rows by calling `print(df.head(2))`.

Listing 8.5 Accessing the first two rows of a table

```
print(df.head(2))

      Fish  Length  Width
0  Angelfish    15.2    7.7
1  Zebrafish     6.5    2.1
```

A better way to summarize a larger Pandas table is to execute `pd.describe()`. By default, this method generates statistics for all numeric columns in the table. The statistical output includes minimum and maximum column values as well as the mean and standard deviation. When we print `pd.describe()`, we should expect to see information for the numeric `Length` and `Width` columns but not for the string-based `Fish` column.

Listing 8.6 Summarizing the numeric columns

```
print(df.describe())

      Length      Width
count    4.000000   4.000000
mean    9.175000   4.075000
std    4.225616   2.678775
min    6.000000   2.000000
25%    6.375000   2.075000
50%    7.750000   3.300000
75%   10.550000   5.300000
max   15.200000   7.700000
```

Outputs of the Pandas describe method

- `count`—The number of elements in each column.
- `mean`—The mean of the elements in each column.
- `std`—The standard deviation of the elements in each column.

(continued)

- min—The minimum value in each column.
- 25%—25% of the column elements fall below this value.
- 50%—50% of the column elements fall below this value. The value is identical to the median.
- 75%—75% of the column elements fall below this value.
- max—The maximum value in each column.

According to the summary, the mean of Length is 9.175 cm and the mean of Width is 4.075 cm. Additional statistical information is also included in the output. Sometimes that other information is not very useful—if all we care about is the mean, we can omit the other outputs by calling `df.mean()`.

Listing 8.7 Computing the column mean

```
print(df.mean())

Length    9.175
Width     4.075
dtype: float64
```

The `df.describe()` method is intended to run on numeric columns. However, we can force it to process strings by calling `df.describe(include=[np.object])`. Setting the `include` parameter to `[np.object]` instructs Pandas to search for table columns built on top of NumPy string arrays. Since we cannot run statistical analysis on strings, the resulting output does not contain statistical information. Instead, the description counts the total number of unique strings and the frequency with which the most common string occurs. The most frequent string is also included. Our Fish column contains four unique strings, and each string is mentioned only once. Therefore, we expect the most frequent string to be chosen at random with a frequency of 1.

Listing 8.8 Summarizing the string columns

```
print(df.describe(include=[np.object]))
```

count	4	↑ The number of strings in each column
unique	4	↑ The number of unique strings in each column
top	Zebrafish	↑ The most frequently occurring string in each column
freq	1	↑ The frequency with which the most frequent string occurs

Pandas summarization methods

- `df.head()`—Returns the first five rows in data frame `df`
- `df.head(x)`—Returns the first `x` rows in data frame `df`
- `df.describe()`—Returns statistics relating to numeric columns in `df`
- `df.describe(include=[np.object])`—Returns statistics relating to string columns in `df`
- `df.mean()`—Returns the mean of all numeric columns in `df`

As mentioned, the `Fish` column is built on top of a NumPy string array. In fact, the entire data frame is built on top of a two-dimensional NumPy array. Pandas stores all the data in NumPy for quick manipulation. We can retrieve the underlying NumPy array by accessing `df.values`.

Listing 8.9 Retrieving the table as a 2D NumPy array

```
print(df.values)
assert type(df.values) == np.ndarray

[['Angelfish' 15.2 7.7]
 ['Zebrafish' 6.5 2.1]
 ['Killifish' 9.0 4.5]
 ['Swordtail' 6.0 2.0]]
```

8.3 Retrieving table columns

Let's turn our attention to retrieving individual columns, which can be accessed using their column names. We can output all the column names by calling `print(df.columns)`.

Listing 8.10 Accessing all column names

```
print(df.columns)

Index(['Fish', 'Length', 'Width'], dtype='object')
```

Now let's print all of the data stored in the `Fish` column by accessing `df.Fish`.

Listing 8.11 Accessing an individual column

```
print(df.Fish)

0    Angelfish
1    Zebrafish
2    Killifish
3    Swordtail
Name: Fish, dtype: object
```

Note that the printed output is not a NumPy array. Rather, `df.Fish` is a Pandas object that represents a one-dimensional array. To print a NumPy array, we must run `print(df.Fish.values)`.

Listing 8.12 Retrieving a column as a NumPy array

```
print(df.Fish.values)
assert type(df.Fish.values) == np.ndarray

['Angelfish' 'Zebrafish' 'Killifish' 'Swordtail']
```

We've accessed the Fish column using df.Fish. We can also access Fish using a dictionary-style bracket representation by printing df['Fish'].

Listing 8.13 Accessing a column using brackets

```
print(df['Fish'])

0    Angelfish
1    Zebrafish
2    Killifish
3    Swordtail
Name: Fish, dtype: object
```

The bracket representation allows us to retrieve multiple columns by running df[name_list], where name_list is a list of column names. Suppose we want to retrieve both the Fish column and the Length column. Running df[['Fish', 'Length']] returns a truncated table containing only those two columns.

Listing 8.14 Accessing multiple columns using brackets

```
print(df[['Fish', 'Length']])

   Fish  Length
0  Angelfish    15.2
1  Zebrafish     6.5
2  Killifish     9.0
3  Swordtail     6.0
```

We can analyze data stored in df a variety of ways. We can, for instance, sort our rows based on a value of a single column. Calling df.sort_values('Length') returns a new table whose rows are sorted based on length.

Listing 8.15 Sorting rows by column value

```
print(df.sort_values('Length'))

   Fish  Length  Width
3  Swordtail    6.0    2.0
1  Zebrafish     6.5    2.1
2  Killifish     9.0    4.5
0  Angelfish    15.2    7.7
```

Furthermore, we can use values in columns to filter out unwanted rows. For example, calling df[df.Width >= 3] returns a table whose rows contain a width of at least 3 cm.

Listing 8.16 Filtering rows by column value

```
print(df[df.Width >= 3])

   Fish  Length  Width
0  Angelfish     15.2    7.7
2  Killifish      9.0    4.5
```

Pandas column-retrieval methods

- `df.columns`—Returns the column names in data frame `df`
- `df.x`—Returns column `x`
- `df[x]`—Returns column `x`
- `df[[x,y]]`—Returns columns `x` and `y`
- `df.x.values`—Returns column `x` as a NumPy array
- `df.sort_values(x)`—Returns a data frame sorted by the values in column `x`
- `df[df.x > y]`—Returns a data frame filtered by the values in column `x` that are `> y`

8.4 Retrieving table rows

Now let's turn our attention to retrieving rows in `df`. Unlike columns, our rows do not have preassigned label values. To compensate, Pandas assigns a special index to each row. These indices appear on the leftmost side of the printed table. Based on the printed output, the index for the `Angelfish` row is 0 and the index for the `Swordtail` row is 3. We can access these rows by calling `df.loc[[0, 3]]`. As a general rule, executing `df.loc[[index_list]]` locates all the rows whose indices appear in `index_list`. Let's now locate the rows that align with the `Swordtail` and `Angelfish` indices.

Listing 8.17 Accessing rows by index

```
print(df.loc[[0, 3]])

   Fish  Length  Width
0  Angelfish     15.2    7.7
3  Swordtail      6.0    2.0
```

Suppose we wish to retrieve rows using species names and not numeric indices. More precisely, we want to retrieve those rows whose `Fish` column contains either '`Angelfish`' or '`Swordtail`'. In Pandas, that retrieval process is a bit tricky: we need to execute `df[booleans]`, where `booleans` is a list of Booleans that are `True` if they match a row of interest. Basically, the indices of `True` values must correspond to rows that match either '`Angelfish`' or '`Whitefish`'. How do we obtain the `booleans` list? One naive approach is to iterate over `df.Fish`, returning `True` if a column value appears in `['Angelfish', 'Swordtail']`. Let's run the naive approach next.

Listing 8.18 Accessing rows by column value

```
booleans = [name in ['Angelfish', 'Swordtail']
            for name in df.Fish]
print(df[booleans])

      Fish  Length  Width
0  Angelfish     15.2    7.7
3  Swordtail      6.0    2.0
```

We can more concisely locate rows of interest using the `isin` method. Calling `df.Fish.isin(['Angelfish', 'Swordtail'])` returns an analogue of our previously computed `booleans` list. Thus, we can retrieve all the rows in a single line of code by running `df[df.Fish.isin(['Angelfish', 'Swordtail'])]`.

Listing 8.19 Accessing rows by column value using isin

```
print(df[df.Fish.isin(['Angelfish', 'Swordtail'])])

      Fish  Length  Width
0  Angelfish     15.2    7.7
3  Swordtail      6.0    2.0
```

The `df` table stores two measurements across four species of fish. We can easily access measurements in the columns; unfortunately, accessing rows by species is harder since the row indices don't equal the species names. Let's remedy the situation by replacing the row indices with species. We swap numbers for species names using the `df.set_index` method. Calling `df.set_index('Fish', inplace=True)` sets our indices to equal the species in the `Fish` column. The `inplace=True` parameter modifies the indices internally rather than returning a modified copy of `df`.

Listing 8.20 Swapping row indices for column values

```
df.set_index('Fish', inplace=True)
print(df)

      Fish  Length  Width
Angelfish     15.2    7.7
Zebrafish      6.5    2.1
Killifish      9.0    4.5
Swordtail      6.0    2.0
```

The leftmost index column is no longer numeric: the numbers have been replaced with species names. We can now access the `Angelfish` and `Swordtail` columns by running `df.loc[['Angelfish', 'Swordtail']]`.

Listing 8.21 Accessing rows by string index

```
print(df.loc[['Angelfish', 'Swordtail']])

Fish    Length   Width
Angelfish      15.2     7.7
Swordtail       6.0      2.0
```

Pandas row-retrieval methods

- df.loc[[x, y]]—Returns the rows located at indices x and y
- df[booleans]—Returns the rows where booleans[i] is True for column i
- df[name in array for name in df.x]—Returns rows where column name x is present in array
- df[df.x.isin(array)])—Returns rows where column name x is present in array
- df.set_index('x', inplace=True)—Swaps numeric row indices for the column values in column x

8.5 Modifying table rows and columns

Currently, each table row contains the length and width of a specified fish. What will happen if we swap our rows and columns? We can find out by running df.T. The T stands for *transpose*: in a transpose operation, the elements of a table are flipped around its diagonal so that the rows and columns are switched. Let's transpose our table and print the results.

Listing 8.22 Swapping rows and columns

```
df_transposed = df.T
print(df_transposed)

Fish      Angelfish   Zebrafish   Killifish   Swordtail
Length      15.2        6.5        9.0        6.0
Width       7.7        2.1        4.5        2.0
```

We've modified the table: each column now refers to an individual species of fish, and each row refers to a particular measurement type. The first row holds length, and the second row holds width. Thus, calling print(df_transposed.Swordtail) will print the swordtail's length and width.

Listing 8.23 Printing a transposed column

```
print(df_transposed.Swordtail)

Length      6.0
Width       2.0
Name: Swordtail, dtype: float64
```

Let's modify our table by adding clownfish measurements to `df_transposed`. The length and width of a clownfish are 10.6 cm and 3.7 cm, respectively. We add these measurements by running `df_transposed['Clownfish'] = [10.6, 3.7]`.

Listing 8.24 Adding a new column

```
df_transposed['Clownfish'] = [10.6, 3.7]
print(df_transposed)

Fish      Angelfish   Zebrafish   Killifish   Swordtail   Clownfish
Length      15.2        6.5        9.0        6.0        10.6
Width       7.7        2.1        4.5        2.0        3.7
```

Alternatively, we can assign new columns using the `df_transposed.assign` method. The method lets us add multiple columns by passing in more than one column name. For instance, calling `df_transposed.assign(Clownfish2=[10.6, 3.7], Clownfish3=[10.6, 3.7])` returns a table with two new columns; `Clownfish2` and `Clownfish3`. Note that the `assign` method never adds new columns directly to a table—instead, it returns a copy of the table containing the new data.

Listing 8.25 Adding multiple new columns

```
df_new = df_transposed.assign(Clownfish2=[10.6, 3.7], Clownfish3=[10.6, 3.7])
assert 'Clownfish2' not in df_transposed.columns
assert 'Clownfish2' in df_new.columns
print(df_new)

Fish      Angelfish   Zebrafish   Killifish   Swordtail   Clownfish   Clownfish2 \
Length      15.2        6.5        9.0        6.0        10.6        10.6
Width       7.7        2.1        4.5        2.0        3.7        3.7

Fish      Clownfish3
Length      10.6
Width       3.7
```

Our newly added columns are redundant. We delete these columns by calling `df_new.drop(columns=['Clownfish2', 'Clownfish3'], inplace=True)`. The `df_new.drop` method drops all specified columns from a table.

Listing 8.26 Deleting multiple columns

```
df_new.drop(columns=['Clownfish2', 'Clownfish3'], inplace=True)
print(df_new)

Fish      Angelfish   Zebrafish   Killifish   Swordtail   Clownfish
Length      15.2        6.5        9.0        6.0        10.6
Width       7.7        2.1        4.5        2.0        3.7
```

We now utilize the stored measurements to compute the surface area of each fish. We can treat every fish as an ellipse with an area of `math.pi * length * width / 4`. To find

each area, we must iterate over the values in every column. Iterating over columns in a data frame is just like iterating over elements in a dictionary: we simply execute `df_new.items()`. Doing so returns an iterable of tuples containing column names and column values. Let's iterate over the columns in `df_new` to get the area of every fish.

Listing 8.27 Iterating over column values

```
areas = []
for fish_species, (length, width) in df_new.items():
    area = math.pi * length * width / 4
    print(f"Area of {fish_species} is {area}")
    areas.append(area)

Area of Angelfish is 91.92300104403735
Area of Zebrafish is 10.720684930375171
Area of Killifish is 31.808625617596654
Area of Swordtail is 9.42477796076938
Area of Clownfish is 30.80331596844792
```

Let's add the computed areas to our table. We can augment a new `Area` row by executing `df_new.loc['Area'] = areas`. Then we need to run `df_new.reindex()` to update the row indices with the added `Area` name.

Listing 8.28 Adding a new row

```
df_new.loc['Area'] = areas
df_new.reindex()
print(df_new)

Fish      Angelfish   Zebrafish   Killifish   Swordtail   Clownfish
Length    15.200000    6.500000    9.000000    6.000000    10.600000
Width     7.700000    2.100000    4.500000    2.000000    3.700000
Area      91.923001    10.720685   31.808626   9.424778    30.803316
```

Our updated table contains three rows and five columns. We can confirm by accessing `df_new.shape`.

Listing 8.29 Checking the table shape

```
row_count, column_count = df_new.shape
print(f"Our table contains {row_count} rows and {column_count} columns")
```

Our table contains 3 rows and 5 columns

Modifying data frames in Pandas

- `df.T`—Returns a transposed data frame, where rows and columns are swapped.
- `df[x] = array`—Creates a new column `x`. `df.x` maps to values in `array`.

(continued)

- `df.assign(x=array)`—Returns a data frame containing all the elements of `df` and a new column `x`. `df.x` maps to values in `array`.
- `df.assign(x=array, y=array2)`—Returns a data frame containing two new columns, `x` and `y`.
- `df.drop(columns=[x, y])`—Returns a data frame in which columns `x` and `y` have been deleted.
- `df.drop(columns=[x, y], inplace=True)`—Deletes columns `x` and `y` in place, thus modifying `df`.
- `df.loc[x] = array`—Adds a new row at index `x`. We need to run `df.reindex()` for that row to be accessible.

8.6 Saving and loading table data

We've finished making changes to the table. Let's store the table for later use. Calling `df_new.to_csv('Fish_measurements.csv')` saves the table to a CSV file in which the columns are delimited by commas.

Listing 8.30 Saving a table to a CSV file

```
df_new.to_csv('Fish_measurements.csv')
with open('Fish_measurements.csv') as f:
    print(f.read())

,Angelfish,Zebrafish,Killifish,Swordtail,Clownfish
Length,15.2,6.5,9.0,6.0,10.6
Width,7.7,2.1,4.5,2.0,3.7
Area,91.92300104403735,10.720684930375171,31.808625617596654,9.42477796076938
,30.80331596844792
```

The CSV file can be loaded into Pandas using the `pd.read_csv` method. Calling `pd.read_csv('Fish_measurements.csv', index_col=0)` returns a data frame containing all our table information. The optional `index_col` parameter specifies which column holds the row index names. If no column is specified, numeric row indices are automatically assigned.

Listing 8.31 Loading a table from a CSV file

```
df = pd.read_csv('Fish_measurements.csv', index_col=0)
print(df)
print("\nRow index names when column is assigned:")
print(df.index.values)

df_no_assign = pd.read_csv('Fish_measurements.csv')
print("\nRow index names when no column is assigned:")
print(df_no_assign.index.values)
```

```

      Angelfish  Zebrafish  Killifish  Swordtail  Clownfish
Length    15.200000    6.500000    9.000000   6.000000   10.600000
Width     7.700000    2.100000    4.500000   2.000000   3.700000
Area     91.923001   10.720685   31.808626   9.424778   30.803316

```

Row index names when column is assigned:
['Length' 'Width' 'Area']

Row index names when no column is assigned:
[0 1 2]

Using pd.csv, we can load the case study ad-click table into Pandas. Then we'll be able to efficiently analyze that table.

Saving and loading data frames in Pandas

- pd.DataFrame(dictionary)—Converts the data in dictionary to a data frame.
- pd.read_csv(filename)—Converts a CSV file to a data frame.
- pd.read_csv(filename, index_col=i)—Converts a CSV file to a data frame. The *i*th column provides row index names.
- df.to_csv(filename)—Saves the contents of df to a CSV file.

8.7 Visualizing tables using Seaborn

We can view the contents of a Pandas table using a simple print command. However, some numeric tables are too large to be viewed as printed output. Such tables are more easily displayed using heatmaps. A *heatmap* is a graphical representation of a table in which numeric cells are colored by value; the color shades shift continuously depending on the value size. The end result is a bird's-eye view of value differences in the table.

The easiest way to create a heatmap is to use the external Seaborn library. Seaborn is a visualization library built on top of Matplotlib and is closely integrated with Pandas data frames. Let's install the library and then import Seaborn as sns.

NOTE Call pip install seaborn from the command line terminal to install the Seaborn library.

Listing 8.32 Importing the Seaborn library

```
import seaborn as sns
```

Now we visualize our data frame as a heatmap by calling sns.heatmap(df) (figure 8.1).

Listing 8.33 Visualizing a heatmap using Seaborn

```
sns.heatmap(df)
plt.show()
```

We've plotted a heatmap of fish measurements. The displayed colors correspond with measurement values. The mappings between color shades and values are shown in a legend



Figure 8.1 A heatmap of fish measurements. Its color legend specifies the mapping between measurements and colors. Darker colors correspond to lower measurement values. Lighter colors correspond to higher measurement values.

to the right of the plot. Lighter colors map to higher measurement values. Thus, we can immediately tell that the area of an angelfish is the largest measurement in the plot.

We can alter the color palette in the heatmap plot by passing in a `cmap` parameter. The following code executes `sns.heatmap(df, cmap='YlGnBu')` to create a heatmap where the color shades transition from yellow to green and then to blue (figure 8.2).

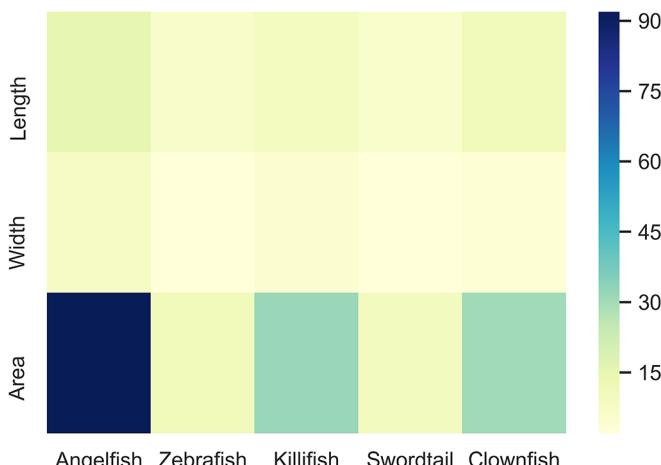


Figure 8.2 A heatmap of fish measurements. Darker colors correspond to higher measurement values. Lighter colors correspond to lower measurement values.

Listing 8.34 Adjusting heatmap colors

```
sns.heatmap(df, cmap='YlGnBu')
plt.show()
```

In the updated heatmap, the color tones have flipped: now darker colors correspond to higher measurements. We can confirm this by annotating the plot with the actual measurement values. We annotate the heatmap by passing `annot=True` into the `sns.heatmap` method (figure 8.3).



Figure 8.3 A heatmap of fish measurements. The actual measurement values are included in the plot.

Listing 8.35 Annotating the heatmap

```
sns.heatmap(df, cmap='YlGnBu', annot=True)
plt.show()
```

As mentioned previously, the Seaborn library is built on top of Matplotlib. Consequently, we can use Matplotlib commands to modify the elements of the heatmap. For example, calling `plt.yticks(rotation=0)` rotates the y-axis measurement labels, which makes them easier to read (figure 8.4).

Listing 8.36 Rotating heatmap labels using Matplotlib

```
sns.heatmap(df, cmap='YlGnBu', annot=True)
plt.yticks(rotation=0)
plt.show()
```

Finally, we should note that the `sns.heatmap` method can also process 2D lists and arrays. Thus, running `sns.heatmap(df.values)` also creates a heatmap plot, but the



Figure 8.4 A heatmap of fish measurements. The y-axis measurement labels have been rotated horizontally for easier viewing.

y-axis and x-axis labels will be missing. To specify the labels, we need to pass the `xticklabels` and `yticklabels` parameters into the method. The following code uses our table's array representation to replicate the contents of figure 8.4.

Listing 8.37 Visualizing a heatmap from a NumPy array

Seaborn heatmap visualization commands

- `sns.heatmap(array)`—Generates a heatmap from the contents of the 2D array.
 - `sns.heatmap(array, xticklabels=x, yticklabels=y)`—Generates a heatmap from the contents of the 2D array. The x-labels and y-labels are set to equal `x` and `y`, respectively.
 - `sns.heatmap(df)`—Generates a heatmap from the contents of the data frame `df`. The x-labels and y-labels are automatically set to equal `df.columns` and `df.index`, respectively.
 - `sns.heatmap(df, cmap=m)`—Generates a heatmap where the color scheme is specified by `m`.
 - `sns.heatmap(df, annot=True)`—Generates a heatmap where the annotated values are included in the plot.

Summary

- 2D table structures can easily be processed using Pandas. We can load the data into Pandas using dictionaries or external files.
- Pandas stores each table in a data frame built on top of a NumPy array.
- Columns in a data frame have a name; we can use these names to access the columns. Meanwhile, the rows in a data frame are assigned numeric indices by default; we can use these indices to access the rows. It is also possible to swap the numeric row indices for string names.
- We can summarize the contents of a data frame using the `describe` method. The method returns valuable statistics such as the mean and standard deviation.
- We can visualize the contents of a data frame using a colored *heatmap*.

Case study 2 solution

This section covers

- Measuring statistical significance
- Permutation testing
- Manipulating tables using Pandas

We've been asked to analyze the online ad-click data collected by our buddy Fred. His advertising data table monitors ad clicks across 30 different colors. Our aim is to discover an ad color that generates significantly more clicks than blue. We will do so by following these steps:

- 1 Load and clean our advertising data using Pandas.
- 2 Run a permutation test between blue and the other recorded colors.
- 3 Check the computed p-values for statistical significance using a properly determined significance level.

WARNING Spoiler alert! The solution to case study 2 is about to be revealed. I strongly encourage you to try to solve the problem before reading the solution. The original problem statement is available for reference at the beginning of the case study.

9.1 Processing the ad-click table in Pandas

Let's begin by loading our ad-click table into Pandas. Then we check the number of rows and columns in the table.

Listing 9.1 Loading the ad-click table into Pandas

```
df = pd.read_csv('colored_ad_click_table.csv')
num_rows, num_cols = df.shape
print(f"Table contains {num_rows} rows and {num_cols} columns")
```

```
Table contains 30 rows and 41 columns
```

Our table contains 30 rows and 41 columns. The rows should correspond to clicks per day and views per day associated with individual colors. Let's confirm by checking the column names.

Listing 9.2 Checking the column names

```
print(df.columns)

Index(['Color', 'Click Count: Day 1', 'View Count: Day 1',
       'Click Count: Day 2', 'View Count: Day 2', 'Click Count: Day 3',
       'View Count: Day 3', 'Click Count: Day 4', 'View Count: Day 4',
       'Click Count: Day 5', 'View Count: Day 5', 'Click Count: Day 6',
       'View Count: Day 6', 'Click Count: Day 7', 'View Count: Day 7',
       'Click Count: Day 8', 'View Count: Day 8', 'Click Count: Day 9',
       'View Count: Day 9', 'Click Count: Day 10', 'View Count: Day 10',
       'Click Count: Day 11', 'View Count: Day 11', 'Click Count: Day 12',
       'View Count: Day 12', 'Click Count: Day 13', 'View Count: Day 13',
       'Click Count: Day 14', 'View Count: Day 14', 'Click Count: Day 15',
       'View Count: Day 15', 'Click Count: Day 16', 'View Count: Day 16',
       'Click Count: Day 17', 'View Count: Day 17', 'Click Count: Day 18',
       'View Count: Day 18', 'Click Count: Day 19', 'View Count: Day 19',
       'Click Count: Day 20', 'View Count: Day 20'],
      dtype='object')
```

The columns are consistent with our expectations: the first column contains all of the analyzed colors, and the remaining 40 columns hold the click counts and view counts for each day of the experiment. As a sanity check, let's examine the quality of the data stored in our table. We start by outputting the analyzed color names.

Listing 9.3 Checking the color names

```
print(df.Color.values)

['Pink' 'Gray' 'Sapphire' 'Purple' 'Coral' 'Olive' 'Navy' 'Maroon' 'Teal'
 'Cyan' 'Orange' 'Black' 'Tan' 'Red' 'Blue' 'Brown' 'Turquoise' 'Indigo'
 'Gold' 'Jade' 'Ultramarine' 'Yellow' 'Viridian' 'Violet' 'Green'
 'Aquamarine' 'Magenta' 'Silver' 'Bronze' 'Lime']
```

30 common colors are present in the *Color* column. The first letter of every color name is capitalized. Thus, we can confirm that the color blue is present by executing assert 'Blue' in df.Color.

Listing 9.4 Checking for blue

```
assert 'Blue' in df.Color.values
```

The string-based *Color* column looks good. Let's turn our attention to the remaining 40 numeric columns. Outputting all 40 columns would lead to an overwhelming amount of data. Instead, we'll examine columns for the first day of the experiment: Click Count: Day 1 and View Count: Day 1. We select these two columns and use describe() to summarize their contents.

Listing 9.5 Summarizing day 1 of the experiment

```
selected_columns = ['Color', 'Click Count: Day 1', 'View Count: Day 1']
print(df[selected_columns].describe())
```

	Click Count: Day 1	View Count: Day 1
count	30.000000	30.0
mean	23.533333	100.0
std	7.454382	0.0
min	12.000000	100.0
25%	19.250000	100.0
50%	24.000000	100.0
75%	26.750000	100.0
max	49.000000	100.0

The values in the Click Count: Day 1 column range from 12 to 49 clicks. Meanwhile, the minimum and maximum values in View Count: Day 1 are both equal to 100 views. Therefore, all the values in that column are equal to 100 views. This behavior is expected. We were specifically informed that each color receives 100 daily views. Let's confirm that all the daily views equal 100.

Listing 9.6 Confirming equivalent daily views

```
view_columns = [column for column in df.columns if 'View' in column]
assert np.all(df[view_columns].values == 100) ← Efficient NumPy code to ensure that
                                                values in a NumPy array equal 100
```

All view counts equal 100. Therefore, all 20 View Count columns are redundant. We can delete them from our table.

Listing 9.7 Deleting view counts from the table

```
df.drop(columns=view_columns, inplace=True)
print(df.columns)
```

```
Index(['Color', 'Click Count: Day 1', 'Click Count: Day 2',
       'Click Count: Day 3', 'Click Count: Day 4', 'Click Count: Day 5',
       'Click Count: Day 6', 'Click Count: Day 7', 'Click Count: Day 8',
       'Click Count: Day 9', 'Click Count: Day 10', 'Click Count: Day 11',
       'Click Count: Day 12', 'Click Count: Day 13', 'Click Count: Day 14',
       'Click Count: Day 15', 'Click Count: Day 16', 'Click Count: Day 17',
       'Click Count: Day 18', 'Click Count: Day 19', 'Click Count: Day 20'],
      dtype='object')
```

The redundant columns have been removed. Only the color and click-count data remain. Our 20 Click Count columns correspond to the number of clicks per 100 daily views, so we can treat these counts as percentages. Effectively, the color in each row is mapped to the percentage of daily ad clicks. Let's summarize the percentage of daily ad clicks for blue ads. To generate that summary, we index each row by color and then call df.T.Blue.describe().

Listing 9.8 Summarizing daily blue-click statistics

```
df.set_index('Color', inplace=True)
print(df.T.Blue.describe())

count    20.000000
mean     28.350000
std      5.499043
min     18.000000
25%    25.750000
50%    27.500000
75%    30.250000
max     42.000000
Name: Blue, dtype: float64
```

The daily click percentages for blue range from 18% to 42%. The mean percent of clicks is 28.35%: on average, 28.35% of blue ads receive a click per view. This average click rate is pretty good. How does it compare to the other 29 colors? We are ready to find out.

9.2 Computing *p*-values from differences in means

Let's start by filtering the data. We delete blue, leaving behind the other 29 colors. Then we transpose our table to access colors by column name.

Listing 9.9 Creating a no-blue table

```
df_not_blue = df.T.drop(columns='Blue')
print(df_not_blue.head(2))
```

Color	Pink	Gray	Sapphire	Purple	Coral	Olive	Navy	Maroon	
Click Count: Day 1	21	27	30	26	26	26	38	21	
Click Count: Day 2	20	27	32	21	24	19	29	29	

```

Color          Teal   Cyan   ... Ultramarine  Yellow  Viridian  Violet  \
Click Count: Day 1    25     24   ...           49      14      27      15
Click Count: Day 2    25     22   ...           41      24      23      22

Color          Green  Aquamarine  Magenta  Silver  Bronze  Lime
Click Count: Day 1     14          24       18      26      19      20
Click Count: Day 2     25          28       21      24      19      19

[2 rows x 29 columns]

```

Our df_not_blue table contains the percent clicks for 29 colors. We would like to compare these percentages to our blue percentages. More precisely, we want to know if there exists a color whose mean click rate is statistically different from the mean click rate of blue. How do we compare these means? The sample mean for every color is easily obtainable, but we do not have a population mean. Thus, our best option is to run a permutation test. To run the test, we need to define a reusable permutation test function. The function will take as input two NumPy arrays and return a p-value as its output.

Listing 9.10 Defining a permutation test function

```

def permutation_test(data_array_a, data_array_b):
    data_mean_a = data_array_a.mean()
    data_mean_b = data_array_b.mean()
    extreme_mean_diff = abs(data_mean_a - data_mean_b) ← | Observed difference
    total_data = np.hstack([data_array_a, data_array_b]) | between sample means
    number_extreme_values = 0.0
    for _ in range(30000):
        np.random.shuffle(total_data)
        sample_a = total_data[:data_array_a.size]
        sample_b = total_data[data_array_a.size:]
        if abs(sample_a.mean() - sample_b.mean()) >= extreme_mean_diff: ← |
            number_extreme_values += 1 | The difference between
    p_value = number_extreme_values / 30000 | resampled means is
    return p_value | extremely large.

```

We'll run a permutation test between blue and the other 29 colors. Then we'll sort these colors based on their p-value results. Our outputs are visualized as a heatmap (figure 9.1), to better emphasize the differences between p-values.

Listing 9.11 Running a permutation test across colors

```

np.random.seed(0)
blue_clicks = df.T.Blue.values
color_to_p_value = {}
for color, color_clicks in df_not_blue.items():
    p_value = permutation_test(blue_clicks, color_clicks)
    color_to_p_value[color] = p_value
sorted_colors, sorted_p_values = zip(*sorted(color_to_p_value.items(), ← |
                                         key=lambda x: x[1])) | Efficient Python code to

```

sort a dictionary and
return two lists: a list of
sorted values and a list of
associated keys. Each
sorted p-value at position
i aligns with the color in
sorted_colors[i].

```
→ plt.figure(figsize=(3, 10))
sns.heatmap([p_value] for p_value in sorted_p_values],           ←
            cmap='YlGnBu', annot=True, xticklabels=['p-value'],
            yticklabels=sorted_colors)
plt.show()
```

Adjusts the width and height of the plotted heatmap to 3 inches and 10 inches, respectively. These adjustments improve the quality of our heatmap visualization.

The `sns.heatmap` method takes as its input a 2D table. Thus, we transform our 1D list of *p*-values into a 2D table containing 29 rows and 1 column.

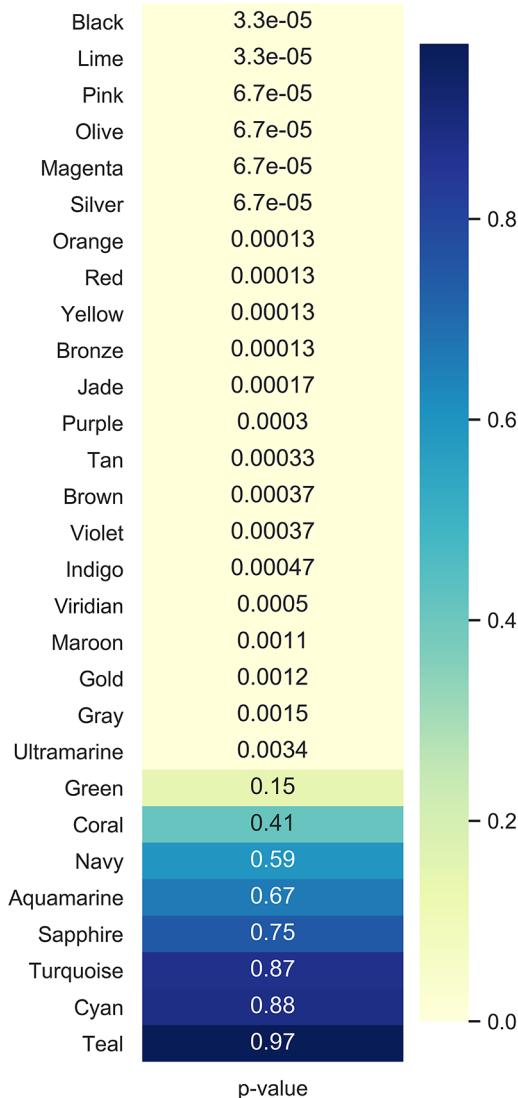


Figure 9.1 A heatmap of *p*-value/color pairs returned by the permutation test: 21 of the colors map to a *p*-value lower than 0.05.

The majority of colors generate a *p*-value that is noticeably lower than 0.05. Black has the lowest *p*-value: its ad-click percentages must deviate significantly from blue. But

from a design perspective, black is not a very clickable color. Text links usually are not black, because black links are hard to distinguish from regular text. Something suspicious is going on here: what exactly is the difference between recorded clicks for black and blue? We can check by printing `df_not_blue.Black.mean()`.

Listing 9.12 Finding the mean click rate of black

```
mean_black = df_not_blue.Black.mean()
print(f"Mean click-rate of black is {mean_black}")
```

Mean click-rate of black is 21.6

The mean click rate of black is 21.6. This value is significantly lower than the blue mean of 28.35. Hence, the statistical difference between the colors is caused by fewer people clicking black. Perhaps other low p-values are also caused by inferior click rates. Let's filter out those colors whose mean is less than the mean of blue and then print the remaining colors.

Listing 9.13 Filtering colors with inferior click rates

```
remaining_colors = df[df.T.mean().values > blue_clicks.mean()].index
size = remaining_colors.size
print(f"{size} colors have on average more clicks than Blue.")
print("These colors are:")
print(remaining_colors.values)

5 colors have on average more clicks than Blue.
These colors are:
['Sapphire' 'Navy' 'Teal' 'Ultramarine' 'Aquamarine']
```

Efficient one-line code to filter the colors. First, the code creates a Boolean array. The array specifies which colors contain a mean greater than blue. The Boolean array is fed into `df` for filtering. The indices of the filtered result specify the remaining color names.

Only five colors remain. Each of these colors is a different shade of blue. Let's print the sorted p-values for the five remaining colors; we also print the mean clicks for easier analysis.

Listing 9.14 Printing the five remaining colors

```
for color, p_value in sorted(color_to_p_value.items(), key=lambda x: x[1]):
    if color in remaining_colors:
        mean = df_not_blue[color].mean()
        print(f"{color} has a p-value of {p_value} and a mean of {mean}")

Ultramarine has a p-value of 0.0034 and a mean of 34.2
Navy has a p-value of 0.5911666666666666 and a mean of 29.3
Aquamarine has a p-value of 0.6654666666666667 and a mean of 29.2
Sapphire has a p-value of 0.7457666666666667 and a mean of 28.9
Teal has a p-value of 0.9745 and a mean of 28.45
```

9.3 Determining statistical significance

Four of the colors have large p-values. Only one color has a p-value that's small. That color is ultramarine: a special shade of blue. Its mean of 34.2 is greater than blue's mean of 28.35. Ultramarine's p-value is 0.0034. Is that p-value statistically significant? Well, it's more than 10 times lower than the standard significance level of 0.05. However, that significance level does not take into account our comparisons between blue and 29 other colors. Each comparison is an experiment testing whether a color differs from blue. If we run enough experiments, then we are guaranteed to encounter a low p-value sooner or later. The best way to correct for this is to execute a Bonferroni correction—otherwise, we will fall victim to p-value hacking. To carry out a Bonferroni correction, we lower the significance level to $0.05 / 29$.

Listing 9.15 Applying the Bonferroni correction

```
significance_level = 0.05 / 29
print(f"Adjusted significance level is {significance_level}")
if color_to_p_value['Ultramarine'] <= significance_level:
    print("Our p-value is statistically significant")
else:
    print("Our p-value is not statistically significant")

Adjusted significance level is 0.001724137931034483
Our p-value is not statistically significant
```

Our p-value is not statistically significant—Fred carried out too many experiments for us to draw a meaningful conclusion. Not all of these experiments were necessary. There is no valid reason to expect that black, brown, or gray would outperform blue. Perhaps if Fred had disregarded some of these colors, our analysis would have been more fruitful. Conceivably, if Fred had simply compared blue to the other five variants of blue, we might have obtained a statistically significant result. Let's explore the hypothetical situation where Fred instigates five experiments and ultramarine's p-value remains unchanged.

Listing 9.16 Exploring a hypothetical significance level

```
hypothetical_sig_level = 0.05 / 5
print(f"Hypothetical significance level is {hypothetical_sig_level}")
if color_to_p_value['Ultramarine'] <= hypothetical_sig_level:
    print("Our hypothetical p-value would have been statistically significant")
else:
    print("Our hypothetical p-value would not have been statistically significant")

Hypothetical significance level is 0.01
Our hypothetical p-value would have been statistically significant
```

Under these hypothetical conditions, our results would be statistically significant. Sadly, we can't use the hypothetical conditions to lower our significance level. We have

no guarantee that rerunning the experiments would reproduce a p-value of 0.0034. P-values fluctuate, and superfluous experiments increase the chance of untrustworthy fluctuations. Given Fred's high experiment count, we simply cannot draw a statistically significant conclusion.

However, all is not lost. Ultramarine still represents a promising substitute for blue. Should Fred carry out that substitution? Perhaps. Let's consider our two alternative scenarios. In the first scenario, the null hypothesis is true. If that's the case, then both blue and ultramarine share the same population mean. Under these circumstances, swapping ultramarine for blue will not affect the ad click rate. In the second scenario, the higher ultramarine click rate is actually statistically significant. If that's the case, then swapping ultramarine for blue will yield more ad clicks. Therefore, Fred has everything to gain and nothing to lose by setting all his ads to ultramarine.

From a logical standpoint, Fred should definitely swap blue for ultramarine. But if he carries out the swap, some uncertainty will remain; Fred will never know if ultramarine truly returns more clicks than blue. What if Fred's curiosity gets the best of him? If he really wants an answer, his only choice is to run another experiment. In that experiment, half the displayed ads would be blue and the other displayed ads would be ultramarine. Fred's software would exhibit the advertisements while recording all the clicks and views. Then we could recompute the p-value and compare it to the appropriate significance level, which would remain at 0.05. The Bonferroni correction would not be necessary because only a single experiment would be run. After the p-value comparison, Fred would finally know whether ultramarine outperforms blue.

9.4 **41 shades of blue: A real-life cautionary tale**

Fred assumed that analyzing every single color would yield more impactful results, but he was wrong. More data isn't necessarily better: sometimes more data leads to more uncertainty.

Fred is not a statistician. He can be forgiven for failing to comprehend the consequences of overanalysis. The same cannot be said of certain quantitative experts operating in business today. Take, for example, a notorious incident that occurred at a well-known corporation. The corporation needed to select a color for the web links on its site. The chief designer chose a visually appealing shade of blue, but a top-level executive distrusted this decision. Why did the designer choose this shade of blue and not another?

The executive came from a quantitative background and insisted that link color should be selected scientifically via a massive analytic test that would supposedly determine the perfect shade of blue. 41 shades of blue were assigned to company web links completely at random, and millions of clicks were recorded. Eventually, the "optimal" shade of blue was selected based on maximum clicks per view.

The executive proceeded to make the methodology public. Worldwide, statisticians cringed. The executive's decisions revealed an ignorance of basic statistics, and that ignorance embarrassed both the executive and the company.

Summary

- More data isn't always better. Running a pointless surplus of analytic tests increases the chance of anomalous results.
- It's worth taking the time to think about a problem before running an analysis. If Fred had carefully considered the 31 colors, he would have realized that it was pointless to test them all. Many colors make ugly links. Colors like black are very unlikely to yield more clicks than blue. Filtering the color set would have led to a more informative test.
- Even though Fred's experiment was flawed, we still managed to extract a useful insight. Ultramarine might prove to be a reasonable substitute for blue, though more testing is required. Occasionally, data scientists are presented with flawed data, but good insights may still be possible.

Case study 3

Tracking disease outbreaks using news headlines

Problem statement

Congratulations! You have just been hired by the American Institute of Health. The Institute monitors disease epidemics in both foreign and domestic lands. A critical component of the monitoring process is analyzing published news data. Each day, the Institute receives hundreds of news headlines describing disease outbreaks in various locations. The news headlines are too numerous to be analyzed by hand.

Your first assignment is as follows: You will process the daily quota of news headlines and extract locations that are mentioned. You will then cluster the headlines based on their geographic distribution. Finally, you will review the largest clusters within and outside the United States. Any interesting findings should be reported to your immediate superior.

Dataset description

The file `headlines.txt` contains the hundreds of headlines that you must analyze. Each headline appears on a separate line in the file.

Overview

To address the problem at hand, we need to know how to do the following:

- Cluster datasets using multiple techniques and distance measures.
- Measure distances between locations on a spherical globe.
- Visualize locations on a map.
- Extract location coordinates from headline text.

10

Clustering data into groups

This section covers

- Clustering data by centrality
- Clustering data by density
- Trade-offs between clustering algorithms
- Executing clustering using the scikit-learn library
- Iterating over clusters using Pandas

Clustering is the process of organizing data points into conceptually meaningful groups. What makes a given group “conceptually meaningful”? There is no easy answer to that question. The usefulness of any clustered output is dependent on the task we’ve been assigned.

Imagine that we’re asked to cluster a collection of pet photos. Do we cluster fish and lizards in one group and fluffy pets (such as hamsters, cats, and dogs) in another? Or should hamsters, cats, and dogs be assigned three separate clusters of their own? If so, perhaps we should consider clustering pets by breed. Thus, Chihuahuas and Great Danes fall into diverging clusters. Differentiating between dog breeds will not be easy. However, we can easily distinguish between Chihuahuas and Great Danes based on breed size. Maybe we should compromise: we’ll cluster on both fluffiness and size, thus bypassing the distinction between the Cairn Terrier and the similar-looking Norwich Terrier.

Is the compromise worth it? It depends on our data science task. Suppose we work for a pet food company, and our aim is to estimate demand for dog food, cat food, and lizard food. Under these conditions, we must distinguish between fluffy dogs, fluffy cats, and scaly lizards. However, we won't need to resolve differences between separate dog breeds. Alternatively, imagine an analyst at a vet's office who's trying to group pet patients by their breed. This second task requires a much more granular level of group resolution.

Different situations require different clustering techniques. As data scientists, we must choose the correct clustering solution. Over the course of our careers, we will cluster thousands (if not tens of thousands) of datasets using a variety of clustering techniques. The most commonly used algorithms rely on some notion of centrality to distinguish between clusters.

10.1 Using centrality to discover clusters

In section 5, we learned how the centrality of data can be represented using the mean. Later, in section 7, we computed the mean length of a single group of fish. Eventually, we compared two separate sets of fish by analyzing the difference between their means. We utilized that difference to determine whether all the fish belonged to the same group. Intuitively, all data points in a single group should cluster around one central value. Meanwhile, the measurements in two divergent groups should cluster around two different means. Thus, we can utilize centrality to distinguish between two divergent groups. Let's explore this notion in concrete detail.

Suppose we take a field trip to a lively local pub and see two dartboards hanging side by side. Each of the dartboards is covered in darts, and darts also protrude from the walls. The tipsy players in the pub aim for the bull's-eye of one board or the other. Frequently, they miss, which leads to the observed scattering of darts centered around the two bull's-eyes.

Let's simulate the scattering numerically. We'll treat each bull's-eye location as a 2D coordinate. Darts are randomly flung at that coordinate. Consequently, the 2D positions of the darts are randomly distributed. The most appropriate distribution for modeling dart positions is the normal distribution, for the following reasons:

- A typical dart thrower aims at the bull's-eye, not at the edge of the dartboard. Thus, each dart is more likely to strike close to the center of the board. This behavior is consistent with random normal samples, in which values closer to the mean occur more frequently than other, more distant values.
- We expect the darts to strike the board symmetrically relative to the center. Darts will strike 3 inches left of center and 3 inches right of center with equal frequency. This symmetry is captured by the bell-shaped normal curve.

Suppose the first bull's-eye is located at coordinate $[0, 0]$. A dart is thrown at that coordinate. We'll model the x and y positions of the dart using two normal distributions. These distributions share a mean of 0, and we also assume that they share a variance of 2. The following code generates the random coordinates of the dart.

Listing 10.1 Modeling dart coordinates using two normal distributions

```
import numpy as np
np.random.seed(0)
mean = 0
variance = 2
x = np.random.normal(mean, variance ** 0.5)
y = np.random.normal(mean, variance ** 0.5)
print(f"The x coordinate of a randomly thrown dart is {x:.2f}")
print(f"The y coordinate of a randomly thrown dart is {y:.2f}")
```

The x coordinate of a randomly thrown dart is 2.49
The y coordinate of a randomly thrown dart is 0.57

NOTE We can more efficiently model dart positions using the `np.random.multivariate_normal` method. This method selects a single random point from a *multivariate normal distribution*. The multivariate normal curve is simply a normal curve that is extended to more than one dimension. Our 2D multivariate normal distribution will resemble a round hill whose summit is positioned at [0, 0].

Let's simulate 5,000 random darts tossed at the bull's-eye positioned at [0, 0]. We also simulate 5,000 random darts tossed at a second bull's-eye, positioned at [0, 6]. Then we generate a scatter plot of all the random dart coordinates (figure 10.1).

Listing 10.2 Simulating randomly thrown darts

```
import matplotlib.pyplot as plt
np.random.seed(1)
bulls_eye1 = [0, 0]
bulls_eye2 = [6, 0]
bulls_eyes = [bulls_eye1, bulls_eye2]
x_coordinates, y_coordinates = [], []
for bulls_eye in bulls_eyes:
    for _ in range(5000):
        x = np.random.normal(bulls_eye[0], variance ** 0.5)
        y = np.random.normal(bulls_eye[1], variance ** 0.5)
        x_coordinates.append(x)
        y_coordinates.append(y)

plt.scatter(x_coordinates, y_coordinates)
plt.show()
```

NOTE Listing 10.2 includes a nested five-line for loop beginning with `for _ in range(5000)`. It's possible to use NumPy to execute this loop in just one line of code: running `x_coordinates, y_coordinates = np.random.multivariate_normal(bulls_eye, np.diag(2 * [variance]), 5000).T` returns 5,000 x and y coordinates sampled from the multivariate normal distribution.

Two overlapping dart groups appear in the plot. The two groups represent 10,000 darts. Half the darts were aimed at the bull's-eye on the left, and the rest were aimed

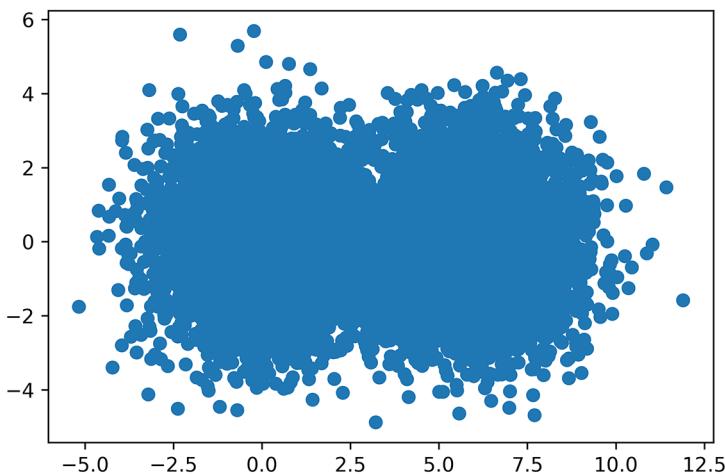


Figure 10.1 A simulation of darts randomly scattered around two bull's-eye targets

toward the right. Each dart has an intended target, which we can estimate by looking at the plot. Darts closer to $[0, 0]$ were probably aimed at the bull's-eye on the left. We'll incorporate this assumption into our dart plot.

Let's assign each dart to its nearest bull's-eye. We start by defining a `nearest_bulls_eye` function that takes as input a dart list holding a dart's x and y positions. The function returns the index of the bull's-eye that is most proximate to dart. We measure dart proximity using *Euclidean distance*, which is the standard straight-line distance between two points.

NOTE Euclidean distance arises from the Pythagorean theorem. Suppose we examine a dart at position $[x_{\text{dart}}, y_{\text{dart}}]$ relative to a bull's-eye at position $[x_{\text{bull}}, y_{\text{bull}}]$. According to the Pythagorean theorem, $\text{distance}^2 = (x_{\text{dart}} - x_{\text{bull}})^2 + (y_{\text{dart}} - y_{\text{bull}})^2$. We can solve for distance using a custom Euclidean function. Alternatively, we can use the `scipy.spatial.distance.euclidean` function provided by SciPy.

The following code defines `nearest_bulls_eye` and applies it to darts $[0, 1]$ and $[6, 1]$.

Listing 10.3 Assigning darts to the nearest bull's-eye

```
from scipy.spatial.distance import euclidean
def nearest_bulls_eye(dart):
    distances = [euclidean(dart, bulls_e) for bulls_e in bulls_eyes]
    return np.argmin(distances)
```

Obtains the Euclidean distance between the dart and each bull's-eye using the `euclidean` function imported from SciPy

darts = [[0,1], [6, 1]]
for dart in darts:

Returns the index matching the shortest bull's-eye distance in the array

```

index = nearest_bulls_eye(dart)
print(f"The dart at position {dart} is closest to bulls-eye {index}")

The dart at position [0, 1] is closest to bulls-eye 0
The dart at position [6, 1] is closest to bulls-eye 1

```

Now we apply `nearest_bulls_eye` to all our computed dart coordinates. Each dart point is plotted using one of two colors to distinguish between the two bull's-eye assignments (figure 10.2).

Listing 10.4 Coloring darts based on the nearest bull's-eye

```

Selects the darts most proximate
to bulls_eyes[bs_index]           ← Helper function that plots the colored elements of
def color_by_cluster(darts):      an inputted darts list. Each dart in darts serves as
    nearest_bulls_eyes = [nearest_bulls_eye(dart) for dart in darts]
    for bs_index in range(len(bulls_eyes)):
        selected_darts = [darts[i] for i in range(len(darts))
                            if bs_index == nearest_bulls_eyes[i]]           ←
        x_coordinates, y_coordinates = np.array(selected_darts).T
        plt.scatter(x_coordinates, y_coordinates,
                    color=['g', 'k'][bs_index])
    plt.show()

darts = [[x_coordinates[i], y_coordinates[i]]
         for i in range(len(x_coordinates))]           ←
color_by_cluster(darts)           ← Separates the x and
                                  y coordinates of each dart
                                  by transposing an array of
                                  selected darts. As discussed
                                  in section 8, the transpose
                                  swaps the row and column
                                  positions within a 2D
                                  data structure.

                                  Combines the separate
                                  coordinates of each dart into a
                                  single list of x and y coordinates.

```

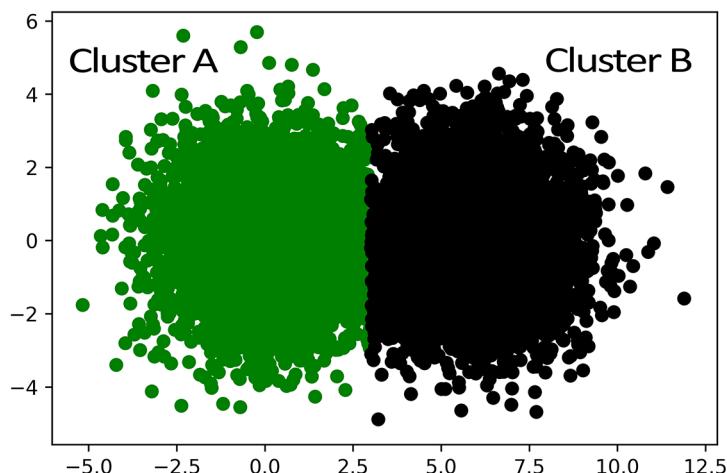


Figure 10.2 Darts colored based on proximity to the nearest bull's-eye. Cluster A represents all points closest to the left bull's-eye, and cluster B represents all points closest to the right bull's-eye.

The colored darts split sensibly into two even clusters. How would we identify such clusters if no central coordinates were provided? Well, one primitive strategy is to simply guess the location of the bull's-eyes. We can pick two random darts and hope these darts are somehow relatively close to each of the bull's-eyes, although the likelihood of that happening is incredibly low. In most cases, coloring darts based on two randomly chosen centers will not yield good results (figure 10.3).

Listing 10.5 Assigning darts to randomly chosen centers

```
bulls_eyes = np.array(darts[:2])      ← Randomly selects the first two darts
color_by_cluster(darts)
```

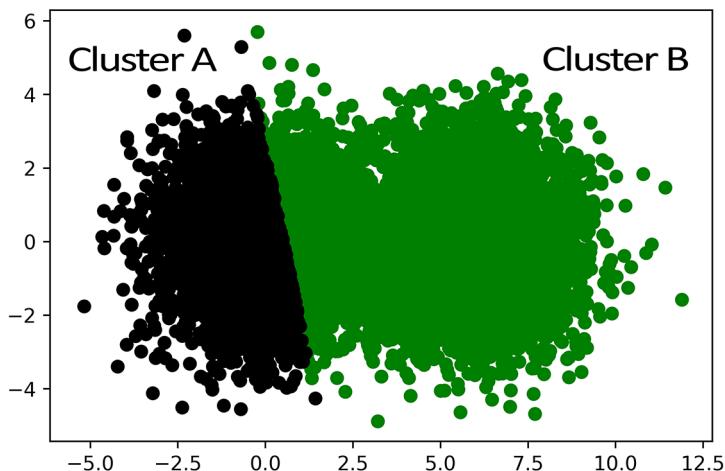


Figure 10.3 Darts colored based on proximity to randomly selected centers.
Cluster B is stretched too far to the left.

Our indiscriminately chosen centers feel wrong qualitatively. For instance, cluster B on the right seems to be stretching way too far to the left. The arbitrary center we've assigned doesn't appear to match its actual bull's-eye point. But there's a way to remedy our error: we can compute the mean coordinates of all the points in the stretched right clustered group and then utilize these coordinates to adjust our estimation of the group's center. After assigning the cluster's mean coordinates to the bull's-eye, we can reapply our distance-based grouping technique to adjust the rightmost cluster's boundaries. In fact, for maximum effectiveness, we will also reset the leftmost cluster's center to its mean prior to rerunning our centrality-based clustering (figure 10.4).

NOTE When we compute the mean of a 1D array, we return a single value. We are now extending that definition to encompass multiple dimensions. When we compute the mean of a 2D array, we return the mean of all x coordinates and also the mean of all y coordinates. The final output is a 2D array containing means across the x-axis and y-axis.

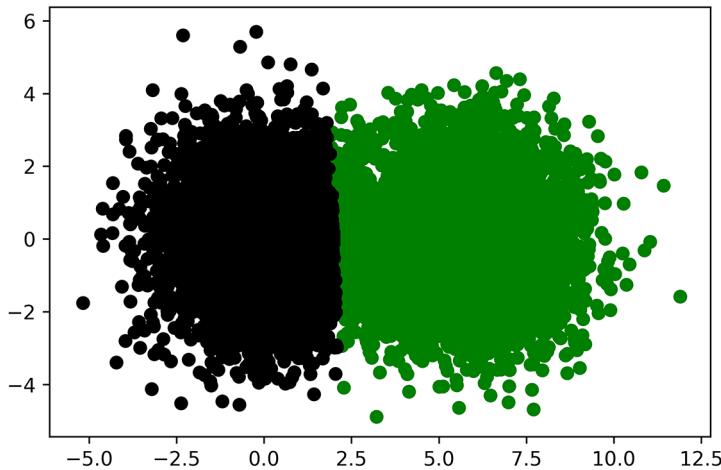


Figure 10.4 Darts colored based on proximity to recomputed centers. The two clusters now appear to be more even.

Listing 10.6 Assigning darts to centers based on means

```
def update_bulls_eyes(darts):
    updated_bulls_eyes = []
    nearest_bulls_eyes = [nearest_bulls_eye(dart) for dart in darts]
    for bs_index in range(len(bulls_eyes)):
        selected_darts = [darts[i] for i in range(len(darts))
                           if bs_index == nearest_bulls_eyes[i]]
        x_coordinates, y_coordinates = np.array(selected_darts).T
        mean_center = [np.mean(x_coordinates), np.mean(y_coordinates)] ←
    updated_bulls_eyes.append(mean_center)
    Takes the mean of the x and y
    coordinates for all the darts assigned
    to a given bull's-eye. These average
    coordinates are then used to update our
    estimated bull's-eye position. We can more
    efficiently run this calculation by executing
    np.mean(selected_darts, axis=0).

    return updated_bulls_eyes

bulls_eyes = update_bulls_eyes(darts)
color_by_cluster(darts)
```

The results are already looking better, although they're not quite as effective as they could be. The cluster's centers still appear a little off. Let's remedy the results by repeating the mean-based centrality adjustment over 10 additional iterations (figure 10.5).

Listing 10.7 Adjusting bull's-eye positions over 10 iterations

```
for i in range(10):
    bulls_eyes = update_bulls_eyes(darts)

    color_by_cluster(darts)
```

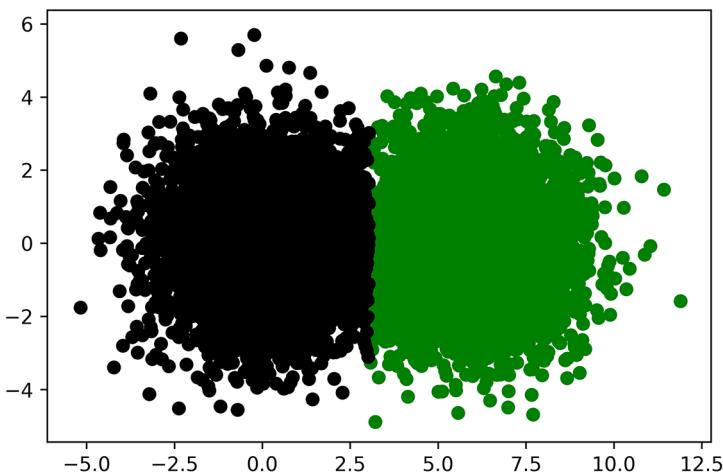


Figure 10.5 Darts colored based on proximity to iteratively recomputed centers

The two sets of darts are now perfectly clustered! We have essentially replicated the *K-means* clustering algorithm, which organizes data using centrality.

10.2 K-means: A clustering algorithm for grouping data into K central groups

The K-means algorithm assumes that inputted data points swirl around K different centers. Each central coordinate is like a hidden bull's-eye surrounded by scattered data points. The purpose of the algorithm is to uncover these hidden central coordinates.

We initialize K-means by first selecting K , which is the number of central coordinates we will search for. In our dartboard analysis, K was set to 2, although generally K can equal any whole number. The algorithm chooses K data points at random. These data points are treated as though they are true centers. Then the algorithm iterates by updating the chosen central locations, which data scientists call *centroids*. During a single iteration, every data point is assigned to its closest center, leading to the formation of K groups. Next, the center of each group is updated. The new center equals the mean of the group's coordinates. If we repeat the process long enough, the group means will converge to K representative centers (figure 10.6). The convergence is mathematically guaranteed. However, we cannot know in advance the number of iterations required for the convergence to take place. A common trick is to halt the iterations when none of the newly computed centers deviate significantly from their predecessors.

K-means is not without its limitations. The algorithm is predicated on our knowledge of K : the number of clusters to look for. Frequently, such knowledge is not available. Also, while K-means commonly finds reasonable centers, it's not mathematically

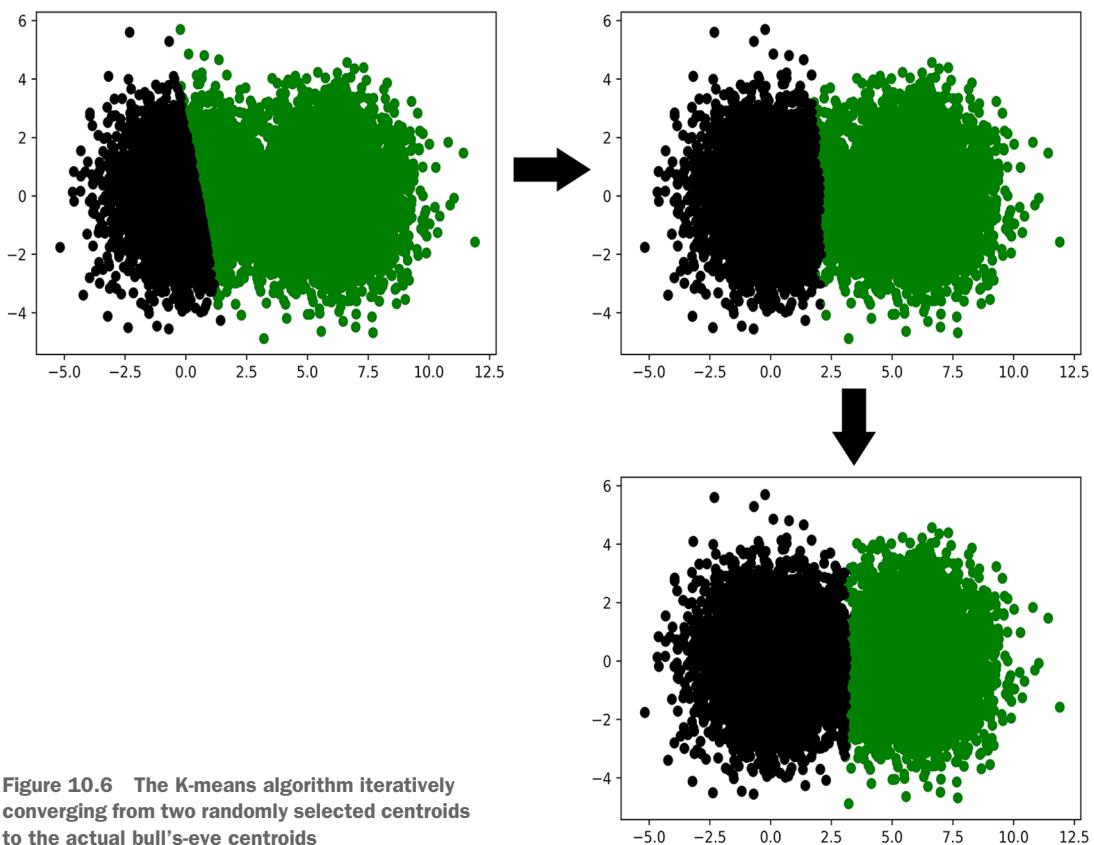


Figure 10.6 The K-means algorithm iteratively converging from two randomly selected centroids to the actual bull's-eye centroids

guaranteed to find the best possible centers in the data. Occasionally, K-means returns unintuitive or suboptimal groups due to poor selection of random centroids at the initialization step of the algorithm. Finally, K-means presupposes that the clusters in the data actually swirl around K central locations. But as we learn later in the section, this supposition does not always hold.

10.2.1 K-means clustering using scikit-learn

The K-means algorithm runs in a reasonable time if it is implemented efficiently. A speedy implementation of the algorithm is available through the external scikit-learn library. Scikit-learn is an extremely popular machine learning toolkit built on top of NumPy and SciPy. It features a variety of core classification, regression, and clustering algorithms—including, of course, K-means. Let's install the library. Then we import scikit-learn's KMeans clustering class.

NOTE Call `pip install scikit-learn` from the command line terminal to install the scikit-learn library.

Listing 10.8 Importing KMeans from scikit-learn

```
from sklearn.cluster import KMeans
```

Applying KMeans to our darts data is easy. First, we need to run `KMeans(n_clusters=2)`, which will create a `cluster_model` object capable of finding two bull's-eye centers. Then, we can execute K-means by running `cluster_model.fit_predict(darts)`. That method call will return an `assigned_bulls_eyes` array that stores the bull's-eye index of each dart.

Listing 10.9 K-means clustering using scikit-learn

```
cluster_model = KMeans(n_clusters=2)
assigned_bulls_eyes = cluster_model.fit_predict(darts) ← Creates a
print("Bull's-eye assignments:")
print(assigned_bulls_eyes) ← Optimizes two centers
                           using the K-means
                           algorithm and returns
                           the assigned cluster
                           for each dart
```

Bull's-eye assignments:
[0 0 0 ... 1 1 1]

Let's color our darts based on their clustering assignments to verify the results (figure 10.7).

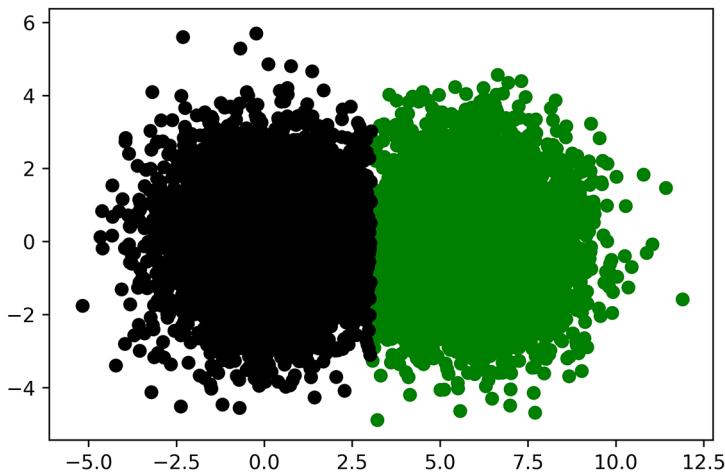


Figure 10.7 The K-means clustering results returned by scikit-learn are consistent with our expectations.

Listing 10.10 Plotting K-means cluster assignments

```
for bs_index in range(len(bulls_eyes)):
    selected_darts = [darts[i] for i in range(len(darts))
                      if bs_index == assigned_bulls_eyes[i]]
```

```

x_coordinates, y_coordinates = np.array(selected_darts).T
plt.scatter(x_coordinates, y_coordinates,
            color=['g', 'k'][bs_index])
plt.show()

```

Our clustering model has located the centroids in the data. Now we can reuse these centroids to analyze new data points that the model has not seen before. Executing `cluster_model.predict([x, y])` assigns a centroid to a data point defined by `x` and `y`. We use the `predict` method to cluster two new data points.

Listing 10.11 Using `cluster_model` to cluster new data

```

new_darts = [[500, 500], [-500, -500]]
new_bulls_eye_assignments = cluster_model.predict(new_darts)
for i, dart in enumerate(new_darts):
    bulls_eye_index = new_bulls_eye_assignments[i]
    print(f"Dart at {dart} is closest to bull's-eye {bulls_eye_index}")

Dart at [500, 500] is closest to bull's-eye 0
Dart at [-500, -500] is closest to bull's-eye 1

```

10.2.2 Selecting the optimal K using the elbow method

K-means relies on an inputted K . This can be a serious hindrance when the number of authentic clusters in the data isn't known in advance. We can, however, estimate an appropriate value for K using a technique known as the *elbow method*.

The elbow method depends on a calculated value called *inertia*, which is the sum of the squared distances between each point and its closest K-means center. If K is 1, then the inertia equals the sum of all squared distances to the dataset's mean. This value, as discussed in section 5, is directly proportional to the variance. Variance, in turn, is a measure of dispersion. Thus, if K is 1, the inertia is an estimate of dispersion. This property holds true even if K is greater than 1. Basically, inertia estimates total dispersion around our K computed means.

By estimating dispersion, we can determine whether our K value is too high or too low. For example, imagine that we set K to 1. Potentially, many of our data points will be positioned too far from one center. Our dispersion will be large, and our inertia will be large. As we increase K toward a more sensible number, the additional centers will cause the inertia to decrease. Eventually, if we go overboard and set K equal to the total number of points, each data point will fall into its own private cluster. Dispersion will be eliminated, and inertia will drop to zero (figure 10.8).

Some inertia values are too large. Others are too low. Somewhere in between might lie a value that's just right. How do we find it?

Let's work out a solution. We begin by plotting the inertia of our dartboard dataset over a large range of K values (figure 10.9). Inertia is automatically computed for each scikit-learn `KMeans` object. We can access this stored value through the model's `inertia_` attribute.

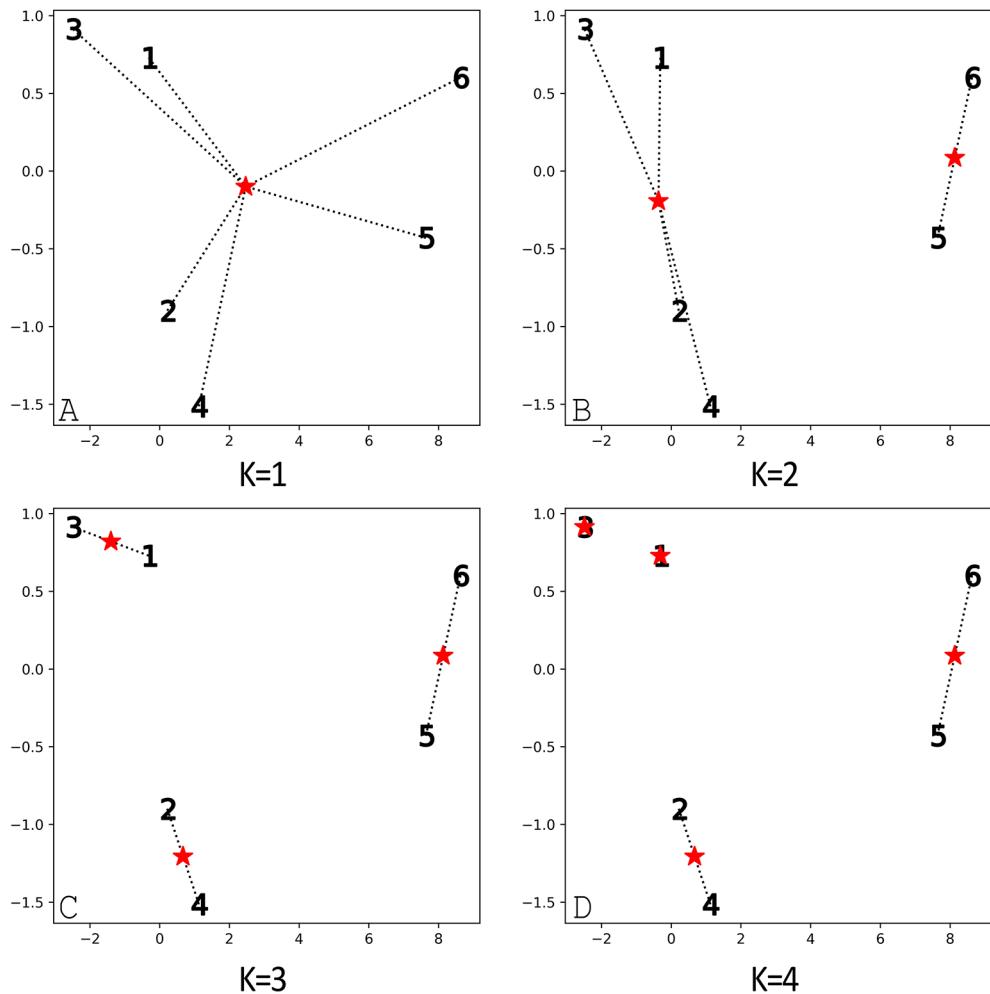


Figure 10.8 Six points, numbered 1 through 6, are plotted in 2D space. The centers, marked by stars, are computed across various values of K . A line is drawn from every point to its nearest center. Inertia is computed by summing the squared lengths of the six lines. (A) $K = 1$. All six lines stretch out from a single center. The inertia is quite large. (B) $K = 2$. Points 5 and 6 are very close to a second center. The inertia is reduced. (C) $K = 3$. Points 1 and 3 are substantially closer to a newly formed center. Points 2 and 4 are also substantially closer to a newly formed center. The inertia has radically decreased. (D) $K = 4$. Points 1 and 3 now overlap with their centers. Their contribution to the inertia has shifted from a very low value to zero. The distances between the remaining four points and their associated centers remain unchanged. Thus, increasing K from 3 to 4 caused a very small decrease in inertia.

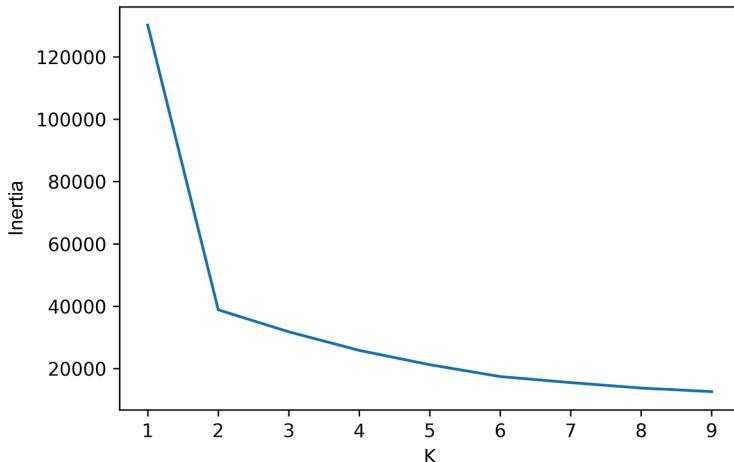


Figure 10.9 An inertia plot for a dartboard simulation containing two bull's-eye targets. The plot resembles an arm bent at the elbow. The elbow points directly to a K of 2.

Listing 10.12 Plotting the K-means inertia

```
k_values = range(1, 10)
inertia_values = [KMeans(k).fit(darts).inertia_
                  for k in k_values]

plt.plot(k_values, inertia_values)
plt.xlabel('K')
plt.ylabel('Inertia')
plt.show()
```

The generated plot resembles an arm bent at the elbow, and the elbow points at a K value of 2. As we already know, this K accurately captures the two centers we have pre-programmed into the dataset.

Will the approach still hold if the number of present centers is increased? We can find out by adding an additional bull's-eye to our dart-throwing simulation. After we increase the cluster count to 3, we regenerate our inertia plot (figure 10.10).

Listing 10.13 Plotting inertia for a 3-dartboard simulation

```
new_bulls_eye = [12, 0]
for _ in range(5000):
    x = np.random.normal(new_bulls_eye[0], variance ** 0.5)
    y = np.random.normal(new_bulls_eye[1], variance ** 0.5)
    darts.append([x, y])

inertia_values = [KMeans(k).fit(darts).inertia_
                  for k in k_values]
```

```
plt.plot(k_values, inertia_values)
plt.xlabel('K')
plt.ylabel('Inertia')
plt.show()
```

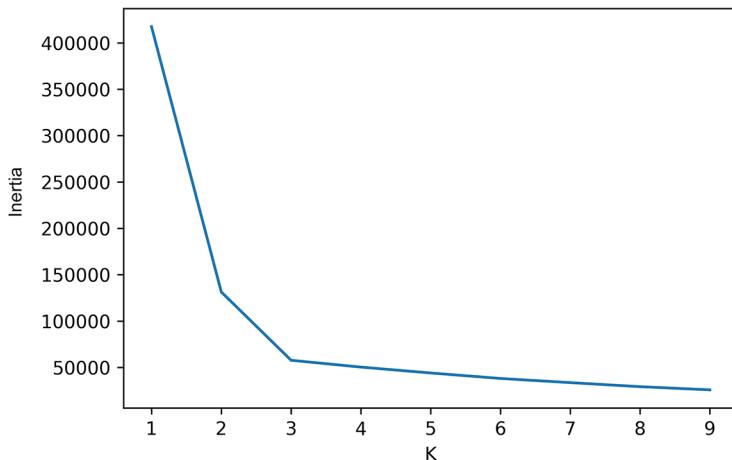


Figure 10.10 An inertia plot for a dartboard simulation containing three bull's-eye targets. The plot resembles an arm bent at the elbow. The lowermost portion of the elbow points to a K of 3.

Adding a third center leads to a new elbow whose lowermost inclination points to a K value of 3. Essentially, our elbow plot traces the dispersion captured by each incremental K . A rapid decrease in inertia between consecutive K values implies that scattered data points have been assigned to a tighter cluster. The reduction in inertia incrementally loses its impact as the inertia curve flattens out. This transition from a vertical drop to a gentler angle leads to the presence of an elbow shape in our plot. We can use the position of the elbow to select a proper K in the K-means algorithm.

The elbow method selection criterion is a useful heuristic, but it is not guaranteed to work in every case. Under certain conditions, the elbow levels off slowly over multiple K values, making it difficult to choose a single valid cluster count.

NOTE There exist more powerful K -selection methodologies, such as the *silhouette score*, which captures the distance of each point to neighboring clusters. A thorough discussion of the silhouette score is beyond the scope of this book. However, you're encouraged to explore the score on your own, using the `sklearn.metrics.silhouette_score` method.

The elbow method isn't perfect, but it performs reasonably well if the data is centered on K distinct means. Of course, this assumes that our data clusters differ due to centrality. However, in many instances, data clusters differ due to the density of the data

K-means clustering methods

- `k_means_model = KMeans(n_clusters=K)`—Creates a K-means model to search for K different centroids. We need to fit these centroids to inputted data.
- `clusters = k_means_model.fit_predict(data)`—Executes K-means on inputted data using an initialized `KMeans` object. The returned `clusters` array contains cluster IDs ranging from 0 to K . The cluster ID of `data[i]` is equal to `clusters[i]`.
- `clusters = KMeans(n_clusters=K).fit_predict(data)`—Executes K-means in a single line of code, and returns the resulting clusters.
- `new_clusters = k_means_model.predict(new_data)`—Finds the nearest centroids to previously unseen data using the existing centroids in a data-optimized `KMeans` object.
- `inertia = k_means_model.inertia_`—Returns the inertia associated with a data-optimized `KMeans` object.
- `inertia = KMeans(n_clusters=K).fit(data).inertia_`—Executes K-means in a single line of code, and returns the resulting inertia.

points in space. Let's explore the concept of density-driven clusters, which are not dependent on centrality.

10.3 Using density to discover clusters

Suppose an astronomer discovers a new planet at the far-flung edge of the solar system. The planet, much like Saturn, has multiple rings spinning in constant orbits around its center. Each ring is formed from thousands of rocks. We'll model these rocks as individual points defined by `x` and `y` coordinates. Let's generate three rock rings composed of many rocks, using scikit-learn's `make_circles` function (figure 10.11).

Listing 10.14 Simulating rings around a planet

```
from sklearn.datasets import make_circles

x_coordinates = []
y_coordinates = []
for factor in [.3, .6, 0.99]:
    rock_ring, _ = make_circles(n_samples=800, factor=factor,
                                 noise=.03, random_state=1)
    for rock in rock_ring:
        x_coordinates.append(rock[0])
        y_coordinates.append(rock[1])

plt.scatter(x_coordinates, y_coordinates)
plt.show()
```

The `make_circles` function creates two concentric circles in 2D. The scale of the smaller circle's radius relative to the larger circle is determined by the `factor` parameter.

Three ring groups are clearly present in the plot. Let's search for these three clusters using K-means by setting K to 3 (figure 10.12).

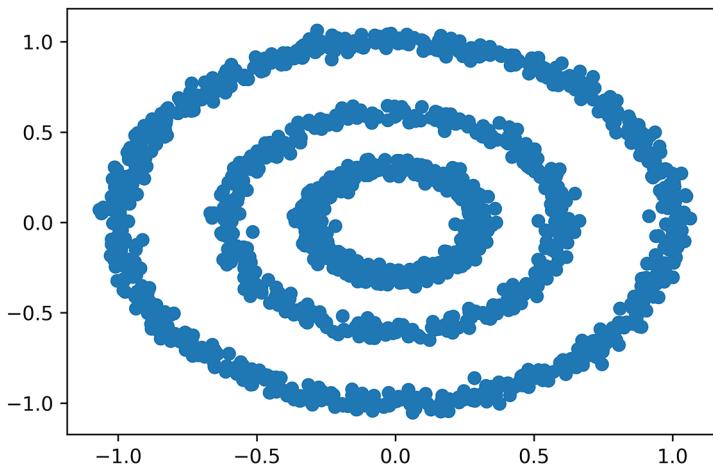


Figure 10.11 A simulation of three rock rings positioned around a central point

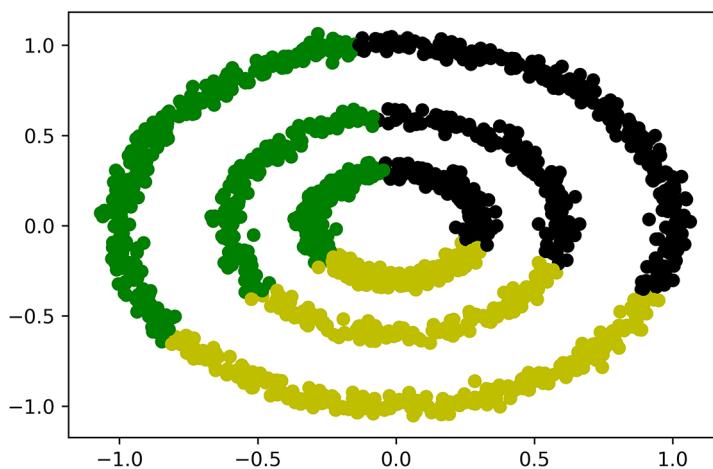


Figure 10.12 K-means clustering fails to properly identify the three distinct rock rings.

Listing 10.15 Using K-means to cluster rings

```
rocks = [[x_coordinates[i], y_coordinates[i]]  
         for i in range(len(x_coordinates))]  
rock_clusters = KMeans(3).fit_predict(rocks)  
  
colors = [['g', 'y', 'k'][cluster] for cluster in rock_clusters]  
plt.scatter(x_coordinates, y_coordinates, color=colors)  
plt.show()
```

The output is an utter failure! K-means dissects the data into three symmetric segments, and each segment spans multiple rings. The solution doesn't align with our intuitive expectation that each ring should fall into its own distinct group. What went wrong? Well, K-means assumed that the three clusters were defined by three unique centers, but the actual rings spin around a single central point. The difference between clusters is driven not by centrality, but by density. Each ring is constructed from a dense collection of points, with empty areas of sparsely populated space serving as the boundaries between rings.

We need to design an algorithm that clusters data in dense regions of space. Doing so requires that we define whether a given region is dense or sparse. One simple definition of *density* is as follows: a point is in a dense region only if it's located within a distance X of Y other points. We'll refer to X and Y as `epsilon` and `min_points`, respectively. The following code sets `epsilon` to 0.1 and `min_points` to 10. Thus, our rocks are present in a dense region of space if they're within a 0.1 radius of at least 10 other rocks.

Listing 10.16 Specifying density parameters

```
epsilon = 0.1
min_points = 10
```

Let's analyze the density of the first rock in our `rocks` list. We begin by searching for all other rocks within `epsilon` units of `rocks[0]`. We store the indices of these neighboring rocks in a `neighbor_indices` list.

Listing 10.17 Finding the neighbors of `rocks[0]`

```
neighbor_indices = [i for i, rock in enumerate(rocks[1:])
                     if euclidean(rocks[0], rock) <= epsilon]
```

Now we compare the number of neighbors to `min_points` to determine whether `rocks[0]` lies in a dense region of space.

Listing 10.18 Checking the density of `rocks[0]`

```
num_neighbors = len(neighbor_indices)
print(f"The rock at index 0 has {num_neighbors} neighbors.")

if num_neighbors >= min_points:
    print("It lies in a dense region.")
else:
    print("It does not lie in a dense region.)
```

```
The rock at index 0 has 40 neighbors.
It lies in a dense region.
```

The rock at index 0 lies in a dense region of space. Do the neighbors of `rocks[0]` also share that dense region of space? This is a tricky question to answer. After all, it's

possible that every neighbor has fewer than `min_points` neighbors of its own. Under our rigorous density definition, we wouldn't consider these neighbors to be dense points. However, this would lead to a ludicrous situation in which the dense region is composed of just a single point: `rocks[0]`. We can avoid such absurd outcomes by updating our density definition. Let's formally define `density` as follows:

- If a point is located within `epsilon` distance of `min_points` neighbors, then that point is in a dense region of space.
- Every neighbor of a point in a dense region of space also clusters in that space.

Based on our updated definition, we can combine `rocks[0]` and its neighbors into a single dense cluster.

Listing 10.19 Creating a dense cluster

```
dense_region_indices = [0] + neighbor_indices
dense_region_cluster = [rocks[i] for i in dense_region_indices]
dense_cluster_size = len(dense_region_cluster)
print(f"We found a dense cluster containing {dense_cluster_size} rocks")

We found a dense cluster containing 41 rocks
```

The rock at index 0 and its neighbors form a single 41-element dense cluster. Do any neighbors of the neighbors belong to a dense region of space? If so, then by our updated definition, these rocks also belong to the dense cluster. Thus, by analyzing additional neighboring points, we can expand the size of `dense_region_cluster`.

Listing 10.20 Expanding a dense cluster

Converts `dense_region_indices` into a set. This allows us to update the set with additional indices without worrying about duplicates.

```
dense_region_indices = set(dense_region_indices) ←
for index in neighbor_indices:
    point = rocks[index]
    neighbors_of_neighbors = [i for i, rock in enumerate(rocks)
                             if euclidean(point, rock) <= epsilon]
    if len(neighbors_of_neighbors) >= min_points:
        dense_region_indices.update(neighbors_of_neighbors)

dense_region_cluster = [rocks[i] for i in dense_region_indices]
dense_cluster_size = len(dense_region_cluster)
print(f"We expanded our cluster to include {dense_cluster_size} rocks")
```

We expanded our cluster to include 781 rocks

We've iterated over neighbors of neighbors and expanded our dense cluster nearly twentyfold. Why stop there? We can expand our cluster even further by analyzing the density of newly encountered neighbors. Iteratively repeating our analysis will increase the breadth of our cluster boundary. Eventually, the boundary will spread to completely encompass one of our rock rings. Then, with no new neighbors to absorb, we

can repeat the iterative analysis on a `rocks` element that has not been analyzed thus far. The repetition will lead to the clustering of additional dense rings.

The procedure just described is known as DBSCAN. The DBSCAN algorithm organizes data based on its spatial distribution.

10.4 DBSCAN: A clustering algorithm for grouping data based on spatial density

DBSCAN is an acronym that stands for *density-based spatial clustering of applications with noise*. This is a ridiculously long name for what essentially is a very simple technique:

- 1 Select a random point coordinate from a data list.
- 2 Obtain all neighbors within `epsilon` distance of that point.
- 3 If fewer than `min_points` neighbors are discovered, repeat step 1 using a different random point. Otherwise, group point and its neighbors into a single cluster.
- 4 Iteratively repeat steps 2 and 3 across all newly discovered neighbors. All neighboring dense points are merged into the cluster. Iterations terminate after the cluster stops expanding.
- 5 After extracting the entire cluster, repeat steps 1-4 on all data points whose density hasn't yet been analyzed.

The DBSCAN procedure can be programmed in less than 20 lines of code. However, any basic implementation will run very slowly on our `rocks` list. Programming a fast implementation requires some very nuanced optimizations that improve neighbor traversal speed and are beyond the scope of this book. Fortunately, there's no need for us to rebuild the algorithm from scratch: scikit-learn provides a speedy DBSCAN class, which we can import from `sklearn.cluster`. Let's import and initialize the class by assigning `epsilon` and `min_points` using the `eps` and `min_samples` parameters. Then we utilize DBSCAN to cluster our three rings (figure 10.13).

Listing 10.21 Using DBSCAN to cluster rings

Creates a `cluster_model` object to carry out density clustering. An `epsilon` value of 0.1 is passed in using the `eps` parameter. A `min_points` value of 10 is passed in using the `min_samples` parameter.

```
from sklearn.cluster import DBSCAN
cluster_model = DBSCAN(eps=epsilon, min_samples=min_points)
rock_clusters = cluster_model.fit_predict(rocks)
colors = [['g', 'y', 'k'][cluster] for cluster in rock_clusters]
plt.scatter(x_coordinates, y_coordinates, color=colors)
plt.show()
```

Clusters the rock rings based on density, and returns the assigned cluster for each rock

DBSCAN has successfully identified the three rock rings. The algorithm succeeded where K-means failed.

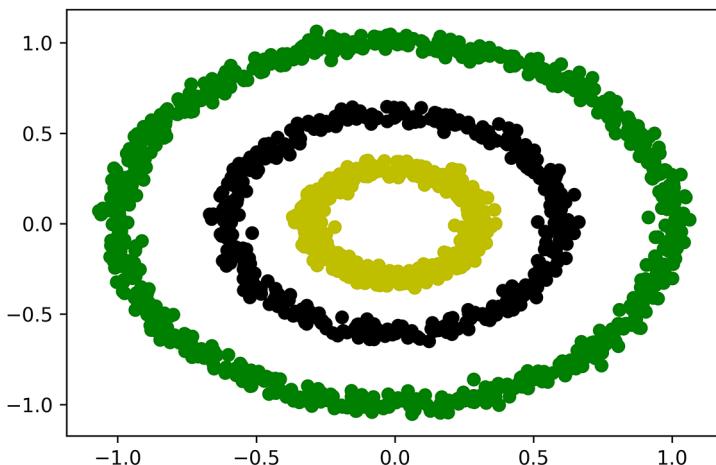


Figure 10.13 DBSCAN clustering accurately identifies the three distinct rock rings.

10.4.1 Comparing DBSCAN and K-means

DBSCAN is an advantageous algorithm for clustering data composed of curving and dense shapes. Also, unlike K-means, the algorithm doesn't require an approximation of the cluster count before execution. Additionally, DBSCAN can filter random outliers located in sparse regions of space. For example, if we add an outlier located beyond the boundary of the rings, DBSCAN will assign it a cluster ID of -1 . The negative value indicates that the outlier cannot be clustered with the rest of the dataset.

NOTE Unlike K-means, a fitted DBSCAN model cannot be reapplied to brand-new data. Instead, we need to combine new and old data and execute the clustering from scratch. This is because computed K-means centers can easily be compared to additional data points. However, the additional data points could influence the density distribution of previously seen data, which forces DBSCAN to recompute all clusters.

Listing 10.22 Finding outliers using DBSCAN

```
noisy_data = rocks + [[1000, -1000]]
clusters = DBSCAN(eps=epsilon,
                  min_samples=min_points).fit_predict(noisy_data)
assert clusters[-1] == -1
```

Another advantage of the DBSCAN technique is that it does not depend on the mean. Meanwhile, the K-means algorithm requires us to compute the mean coordinates of grouped points. As we discussed in section 5, these mean coordinates minimize the sum of squared distances to the center. The minimization property holds only if the squared distances are Euclidean. Thus, if our coordinates are not Euclidean, the

mean is not very useful, and the K-means algorithm should not be applied. However, the Euclidean distance is not the only metric for gauging separation between points— infinite metrics exist for defining distance. We explore a few of them in the subsequent subsection. In the process, we learn how to integrate these metrics into our DBSCAN clustering output.

10.4.2 Clustering based on non-Euclidean distance

Suppose that we are visiting Manhattan and wish to know the walking distance from the Empire State Building to Columbus Circle. The Empire State Building is located at the intersection of 34th Street and Fifth Avenue. Meanwhile, Columbus Circle is located at the intersection of 57th Street and Eighth Avenue. The streets and avenues in Manhattan are always perpendicular to each other. This lets us represent Manhattan as a 2D coordinate system, where streets are positioned on the x-axis and avenues are positioned on the y-axis. Under this representation, the Empire State Building is located at coordinate (34, 5), and Columbus Circle is located at coordinate (57, 8). We can easily calculate a straight-line Euclidean distance between the two coordinate points. However, that final length would be impassable because towering steel buildings occupy the area outlined by every city block. A more correct solution is limited to a path across the perpendicular sidewalks that form the city's grid. Such a route requires us to walk 3 blocks between Fifth Avenue and Third Avenue and then 23 blocks between 34th Street and 57th Street, for a total distance of 26 blocks. Manhattan's average block length is 0.17 miles, so we can estimate the walking distance as 4.42 miles. Let's compute that walking distance directly using a generalized `manhattan_distance` function.

Listing 10.23 Computing the Manhattan distance

```
def manhattan_distance(point_a, point_b):
    num_blocks = np.sum(np.absolute(point_a - point_b))
    return 0.17 * num_blocks

x = np.array([34, 5])
y = np.array([57, 8])
distance = manhattan_distance(x, y)           ←
print(f"Manhattan distance is {distance} miles")
```

We can also generate this output
by importing `cityblock` from
`scipy.spatial.distance` and then
running `0.17 * cityblock(x, y)`.

Manhattan distance is 4.42 miles

Now, suppose we wish to cluster more than two Manhattan locations. We'll assume each cluster holds a point that is within a one-mile walk of three other clustered points. This assumption lets us apply DBSCAN clustering using scikit-learn's `DBSCAN` class. We set `eps` to 1 and `min_samples` to 3 during DBSCAN's initialization. Furthermore, we pass `metric=manhattan_distance` into the initialization method. The `metric` parameter swaps Euclidean distance for our custom distance metric, so the clustering distance correctly reflects the grid-based constraints within the city. The following

code clusters Manhattan coordinates and plots them on a grid along with their cluster designations (figure 10.14).

Listing 10.24 Clustering using Manhattan distance

```
points = [[35, 5], [33, 6], [37, 4], [40, 7], [45, 5]]
clusters = DBSCAN(eps=1, min_samples=3,
                  metric='manhattan_distance').fit_predict(points)

for i, cluster in enumerate(clusters):
    point = points[i]
    if cluster == -1:
        print(f"Point at index {i} is an outlier")
        plt.scatter(point[0], point[1], marker='x', color='k') ←
    else:
        print(f"Point at index {i} is in cluster {cluster}")
        plt.scatter(point[0], point[1], color='g') ←

plt.grid(True, which='both', alpha=0.5) ←
plt.minorticks_on()

plt.show()

Point at index 0 is in cluster 0
Point at index 1 is in cluster 0
Point at index 2 is in cluster 0
Point at index 3 is an outlier
Point at index 4 is an outlier
```

The manhattan_distance function is passed into DBSCAN through the metric parameter.

Outliers are plotted using x-shaped markers.

The grid method displays the rectangular grid across which we compute Manhattan distance.

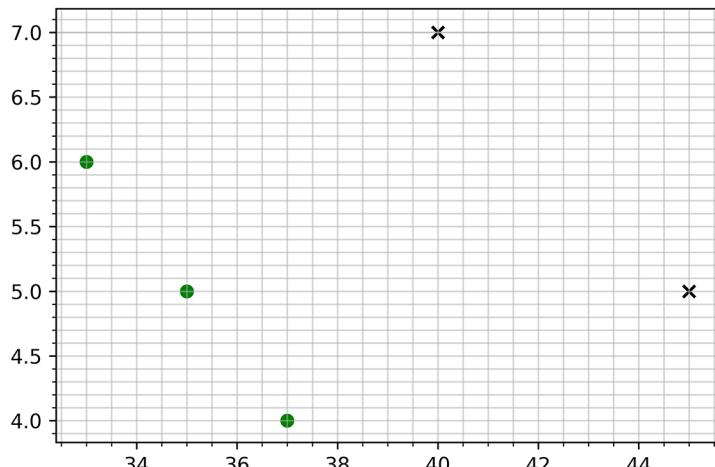


Figure 10.14 Five points in a rectangular grid have been clustered using the Manhattan distance. The three points in the lower-left corner of the grid fall within a single cluster. The remaining two points are outliers, marked by an x.

The first three locations fall within a single cluster, and the remaining points are outliers. Could we have detected that cluster using the K-means algorithm? Perhaps. After all, our Manhattan block coordinates can be averaged out, making them compatible with a K-means implementation. What if we swap Manhattan distance for a different metric where average coordinates are not so easily obtained? Let's define a nonlinear distance metric with the following properties: two points are 0 units apart if all their elements are negative, 2 units apart if all their elements are non-negative, and 10 units apart otherwise. Given this ridiculous measure of distance, can we compute the mean of any two arbitrary points? We can't, and K-means cannot be applied. A weakness of the algorithm is that it depends on the existence of an average distance. Unlike K-means, the DBSCAN algorithm does not require our distance function to be linearly divisible. Thus, we can easily run DBSCAN clustering using our ridiculous distance metric.

Listing 10.25 Clustering using a ridiculous measure of distance

```

def ridiculous_measure(point_a, point_b):
    is_negative_a = np.array(point_a) < 0
    is_negative_b = np.array(point_b) < 0
    if is_negative_a.all() and is_negative_b.all():
        return 0
    elif is_negative_a.any() or is_negative_b.any():
        return 10
    else:
        return 2

points = [[-1, -1], [-10, -10], [-1000, -13435], [3, 5], [5, -7]]

clusters = DBSCAN(eps=.1, min_samples=2,
                  metric=ridiculous_measure).fit_predict(points)

for i, cluster in enumerate(clusters):
    point = points[i]
    if cluster == -1:
        print(f"{point} is an outlier")
    else:
        print(f"{point} falls in cluster {cluster}")

[-1, -1] falls in cluster 0
[-10, -10] falls in cluster 0
[-1000, -13435] falls in cluster 0
[3, 5] is an outlier
[5, -7] is an outlier

```

Running DBSCAN with our `ridiculous_measure` metric leads to the clustering of negative coordinates into a single group. All other coordinates are treated as outliers. These results are not conceptually practical, but the flexibility with regard to metric selection is much appreciated. We are not constrained in our metric choice! We could, for instance, set the metric to compute traversal distance based on the

curvature of the Earth. Such a metric would be particularly useful for clustering geographic locations.

DBSCAN clustering methods

- `dbscan_model = DBSCAN(eps=epsilon, min_samples=min_points)`—Creates a DBSCAN model to cluster by density. A dense point is defined as having at least `min_points` neighbors within a distance of `epsilon`. The neighbors are considered to be part of the same cluster as the point.
- `clusters = dbscan_model.fit_predict(data)`—Executes DBSCAN on inputted data using an initialized DBSCAN object. The `clusters` array contains cluster IDs. The cluster ID of `data[i]` is equal to `clusters[i]`. Unclustered outlier points are assigned an ID of `-1`.
- `clusters = DBSCAN(eps=epsilon, min_samples=min_points).fit_predict(data)`—Executes DBSCAN in a single line of code, and returns the resulting clusters.
- `dbscan_model = DBSCAN(eps=epsilon, min_samples=min_points, metric=metric_function)`—Creates a DBSCAN model where the distance metric is defined by a custom metric function. The `metric_function` distance metric does not need to be Euclidean.

DBSCAN does have certain drawbacks. The algorithm is intended to detect clusters with similar point-density distributions. However, real-world data varies in density. For instance, pizza shops in Manhattan are distributed more densely than pizza shops in Orange County, California. Thus, we might have trouble choosing density parameters that will let us cluster shops in both locations. This highlights another limitation of the algorithm: DBSCAN requires meaningful values for the `eps` and `min_samples` parameters. In particular, varying `eps` inputs will greatly impact the quality of clustering. Unfortunately, there is no one reliable procedure for estimating the appropriate `eps`. While certain heuristics are occasionally mentioned in the literature, their benefit is minimal. Most of the time, we must rely on our gut-level understanding of the problem to assign practical inputs to the two DBSCAN parameters. For example, if we were to cluster a set of geographic locations, our `eps` and `min_samples` values would depend on whether the locations are spread out across the entire globe or whether they are constrained to a single geographic region. In each instance, our understanding of density and distance would vary. Generally speaking, if we are clustering random cities spread out across the Earth, we can set the `min_samples` and `eps` parameters to equal three cities and 250 miles, respectively. This assumes each cluster holds a city within 250 miles of at least three other clustered cities. For a more regional location distribution, a lower `eps` value is required.

10.5 Analyzing clusters using Pandas

So far, we have kept our data inputs and clustering outputs separate. For instance, in our rock ring analysis, the input data is in the `rocks` list and the clustering output is in a `rock_clusters` array. Tracking both the coordinates and the clusters requires us to map indices between the input list and the output array. Thus, if we wish to extract all the rocks in cluster 0, we must obtain all instances of `rocks[i]` where `rock_clusters[i] == 0`. This index analysis is convoluted. We can more intuitively analyze clustered rocks by combining the coordinates and the clusters together in a single Pandas table.

The following code creates a Pandas table with three columns: `X`, `Y`, and `Cluster`. Each *i*th row in the table holds the `x` coordinate, the `y` coordinate, and the cluster of the rock located at `rocks[i]`.

Listing 10.26 Storing clustered coordinates in a table

```
import pandas as pd
x_coordinates, y_coordinates = np.array(rocks).T
df = pd.DataFrame({'X': x_coordinates, 'Y': y_coordinates,
                    'Cluster': rock_clusters})
```

Our Pandas table lets us easily access the rocks in any cluster. Let's plot the rocks that fall into cluster 0, using techniques described in section 8 (figure 10.15).

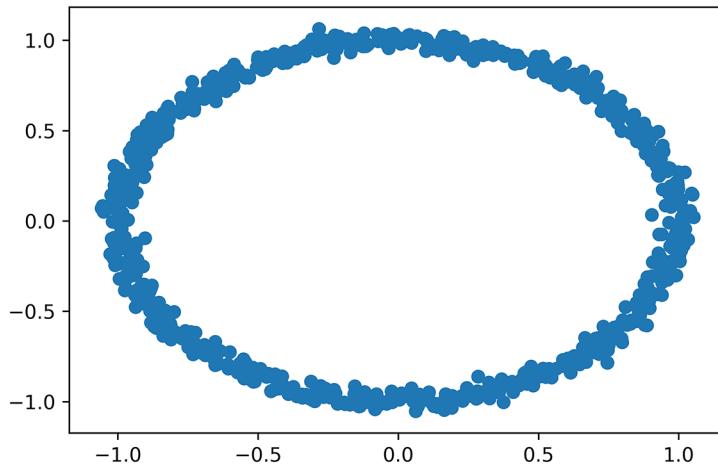


Figure 10.15 Rocks that fall into cluster 0

Listing 10.27 Plotting a single cluster using Pandas

```
df_cluster = df[df.Cluster == 0]
plt.scatter(df_cluster.X, df_cluster.Y)
plt.show()
```

Plots the X and Y columns of the selected rows.
Note that we can also execute the scatter plot by
running `df_cluster.plot.scatter(x='X', y='Y')`.

Select just those
rows where the
Cluster column
equals 0

Pandas allows us to obtain a table containing elements from any single cluster. Alternatively, we might want to obtain multiple tables, where each table maps to a cluster ID. In Pandas, this is done by calling `df.groupby('Cluster')`. The `groupby` method will create three tables: one for each cluster. It will return an iterable over the mappings between cluster IDs and tables. Let's use the `groupby` method to iterate over our three clusters. We'll subsequently plot the rocks in cluster 1 and cluster 2, but not the rocks in cluster 0 (figure 10.16).

NOTE Calling `df.groupby('Cluster')` returns more than just an iterable: it returns a `DataFrameGroupBy` object, which provides additional methods for cluster filtering and analysis.

Listing 10.28 Iterating over clusters using Pandas

```
for cluster_id, df_cluster in df.groupby('Cluster'):
    if cluster_id == 0:
        print(f"Skipping over cluster {cluster_id}")
        continue

    print(f"Plotting cluster {cluster_id}")
    plt.scatter(df_cluster.X, df_cluster.Y)

plt.show()
```

Skipping over cluster 0
 Plotting cluster 1
 Plotting cluster 2

Each element of the iterable returned by `df.groupby('Cluster')` is a tuple. The first element of the tuple is the cluster ID obtained from `df.Cluster`. The second element is a table composed of all rows where `df.Cluster` equals the cluster ID.

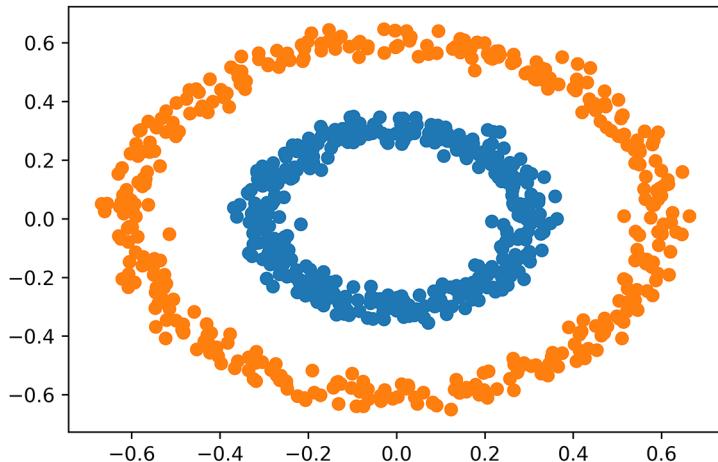


Figure 10.16 Rocks that fall into clusters 1 and 2

The Pandas `groupby` method lets us iteratively examine different clusters. This could prove useful in our case study 3 analysis.

Summary

- The *K-means* algorithm clusters inputted data by searching for *K centroids*. These centroids represent the mean coordinates of the discovered data groups. K-means is initialized by selecting *K* random centroids. Each data point is then clustered based on its nearest centroid, and the centroids are iteratively recomputed until they converge on stable locations.
- K-means is guaranteed to converge to a solution. However, that solution may not be optimal.
- K-means requires Euclidean distances to distinguish between points. The algorithm is not intended to cluster non-Euclidean coordinates.
- After executing K-means clustering, we can compute the *inertia* of the result. Inertia equals the sum of the squared distances between each data point and its closest center.
- Plotting the inertia across a range of *K* values generates an *elbow plot*. The elbow component in the elbow-shaped plot should point downward to a reasonable *K* value. Using the elbow plot, we can heuristically select a meaningful *K* input for K-means.
- The *DBSCAN* algorithm clusters data based on density. Density is defined using the *epsilon* and *min_points* parameters. If a point is located within *epsilon* distance of *min_points* neighbors, then that point is in a dense region of space. Every neighbor of a point in a dense region of space also clusters in that space. DBSCAN iteratively expands the boundaries of a dense region of space until a complete cluster is detected.
- Points in non-dense regions are not clustered by the DBSCAN algorithm. They are treated as outliers.
- DBSCAN is an advantageous algorithm for clustering data composed of curving and dense shapes.
- DBSCAN can cluster using arbitrary, non-Euclidean distances.
- There is no reliable heuristic for choosing appropriate *epsilon* and *min_points* parameters. However, if we wish to cluster global cities, we can set the two parameters to 250 miles and three cities, respectively.
- Storing clustered data in a Pandas table allows us to intuitively iterate over clusters with the *groupby* method.

11

Geographic location visualization and analysis

This section covers

- Computing the distance between geographic locations
- Plotting locations on a map using the Cartopy library
- Extracting geo-coordinates from location names
- Finding location names in text using regular expressions

People have relied on location information since before the dawn of recorded history. Cave dwellers once carved maps of hunting routes into mammoth tusks. Such maps evolved as civilizations flourished. The ancient Babylonians fully mapped the borders of their vast empire. Much later, in 3000 BC, Greek scholars improved cartography using mathematical innovations. The Greeks discovered that the Earth was round and accurately computed the planet's circumference. Greek mathematicians laid the groundwork for measuring distances across the Earth's curved surface. Such measurements required the creation of a geographic coordinate system: a rudimentary system based on latitude and longitude was introduced in 2000 BC.

Combining cartography with latitude and longitude helped revolutionize maritime navigation. Sailors could more freely travel the seas by checking their positions on a map. Roughly speaking, maritime navigation protocols followed these three steps:

- 1 *Data observation*—A sailor recorded a series of observations including wind direction, the position of the stars, and (after approximately AD 1300) the northward direction of a compass.
- 2 *Mathematical and algorithmic analysis of data*—A navigator analyzed all of the data to estimate the ship's position. Sometimes the analysis required trigonometric calculations. More commonly, the navigator consulted a series of rule-based measurement charts. By algorithmically adhering to the rules in the charts, the navigator could figure out the ship's coordinates.
- 3 *Visualizing and decision making*—The captain examined the computed location on a map relative to the expected destination. Then the captain would give orders to adjust the ship's orientation based on the visualized results.

This navigation paradigm perfectly encapsulates the standard data science process. As data scientists, we are offered raw observations. We algorithmically analyze that data. Then, we visualize the results to make critical decisions. Thus, data science and location analysis are linked. That link has only grown stronger through the centuries. Today, countless corporations analyze locations in ways the ancient Greeks could never have imagined. Hedge funds study satellite photos of farmlands to make bets on the global soybean market. Transport-service providers analyze vast traffic patterns to efficiently route fleets of cars. Epidemiologists process newspaper data to monitor the global spread of disease.

In this section, we explore a variety of techniques for analyzing and visualizing geographic locations. We begin with the simple task of calculating the distance between two geographic points.

11.1 *The great-circle distance: A metric for computing the distance between two global points*

What is the shortest travel distance between any pair of points on Earth? The distance cannot be a straight line since direct linear travel would require burrowing deep through the Earth's crust. A much more realistic path entails traveling along our spherical planet's curved surface. This direct path between two points along the surface of a sphere is called the *great-circle distance* (figure 11.1).

We can compute the great-circle distance given a sphere and two points on that sphere. Any point on the sphere's surface can be represented using *spherical coordinates* x and y , where x and y measure the angles of the point relative to the x -axis and y -axis (figure 11.2).

Let's define a basic `great_circle_distance` function that takes as input two pairs of spherical coordinates. For simplicity's sake, we will assume that the coordinates are present on a unit sphere with a radius of 1. This simplification allows us to define

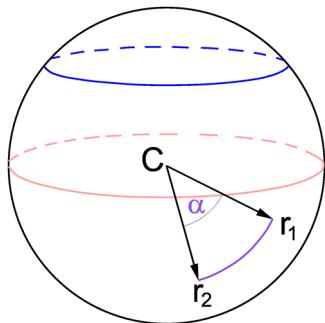


Figure 11.1 Visualizing the great-circle distance between two points on the surface of a sphere. These points are labeled r_1 and r_2 . A curved arc designates the traveling distance between them. The arc length is equal to the radius of the sphere multiplied by α , where α is the angle between points relative to the sphere's center at C.

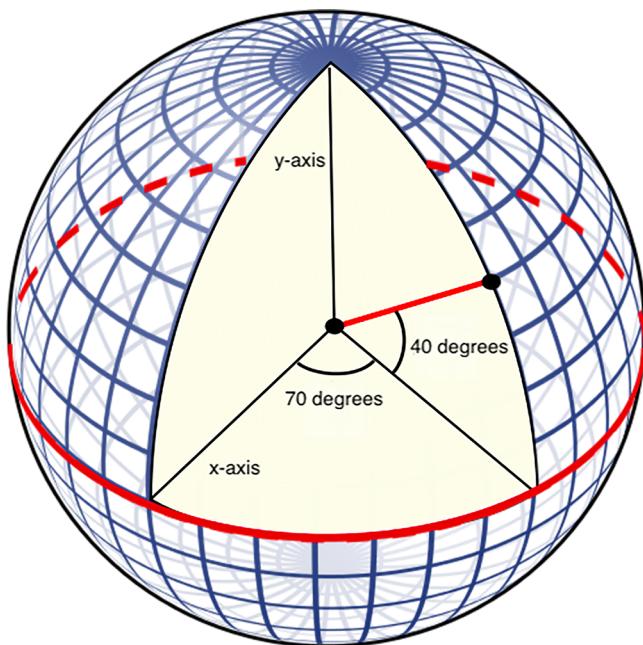


Figure 11.2 Representing a point on the surface of a sphere using spherical coordinates. The point is reached as we rotate 70 degrees away from the x-axis and 40 degrees toward the y-axis. Hence, its spherical coordinates are (70, 40).

`great_circle_distance` in just four lines of code. The function depends on a series of well-known trigonometric operations; a detailed derivation of these operations is beyond the scope of this book.

Listing 11.1 Defining a great-circle distance function

```
Imports three common trigonometric
functions from Python's math module
from math import cos, sin, asin
def great_circle_distance(x1, y1, x2, y2):
    delta_x, delta_y = x2 - x1, y2 - y1
    Computes the angular
    difference between the
    two pairs of spherical
    coordinates
```

```

haversin = sin(delta_x / 2) ** 2 + np.product([cos(x1), cos(x2),
                                              sin(delta_y / 2) ** 2])
return 2 * asin(haversin ** 0.5)

```

Executes a series of well-known trigonometric operations to obtain the great-circle distance on a unit sphere. The `np.product` function multiplies together three of the trigonometric values.

Python's trigonometric functions assume that the input angle is in radians, where 0 degrees equal 0 radians and 180 degrees equal π radians. Let's calculate the great-circle distance between two points that lie 180 degrees apart relative to both the x-axis and the y-axis.

NOTE Radians measure the length of a unit circle arc relative to an angle. The maximum arc length equals the unit-circle circumference of 2π . Traversing the circumference of a circle requires a 360-degree angle. Thus, 2π radians equal 360 degrees, and a single degree equals $\pi / 180$ radians.

Listing 11.2 Computing the great-circle distance

```

from math import pi
distance = great_circle_distance(0, 0, 0, pi)
print(f"The distance equals {distance} units")

```

The distance equals 3.141592653589793 units

The points are exactly π units apart, half the distance required to circumnavigate a unit circle. That value is the longest possible distance we can travel between two spherical points. This is akin to traveling between the North and South Poles of any planet. We'll confirm by analyzing the latitudes and longitudes of Earth's North Pole and South Pole. Terrestrial latitudes and longitudes are spherical coordinates measured in degrees. Let's begin by recording the known coordinates of each pole.

Listing 11.3 Defining the coordinates of Earth's poles

```

latitude_north, longitude_north = (90.0, 0)
latitude_south, longitude_south = (-90.0, 0)

```

Technically speaking, the North Pole and South Pole do not have an official longitude coordinate. However, we're mathematically justified in assigning a zero longitude to each pole.

Latitudes and longitudes measure spherical coordinates in degrees, not radians. We'll thus convert to radians from degrees using the `np.radians` function. The function takes as input a list of degrees and returns a radian array. This result can subsequently be inputted into `great_circle_distance`.

Listing 11.4 Computing the great-circle distance between poles

```

to_radians = np.radians([latitude_north, longitude_north,
                        latitude_south, longitude_south])

```

```
distance = great_circle_distance(*to_radians.tolist())
print(f"The unit-circle distance between poles equals {distance} units")
```

The unit-circle distance between poles equals 3.141592653589793 units

As a reminder, running `func(*[arg1, arg2])` is a
Python shortcut for executing `func(arg1, arg2)`.

As expected, the distance between poles on a unit sphere is π . Now, let's measure the distance between two poles here on Earth. The radius of Earth is not 1 hypothetical unit but rather 3956 actual miles, so we must multiply distance by 3956 to obtain a terrestrial measurement.

Listing 11.5 Computing the travel distance between Earth's poles

```
earth_distance = 3956 * distance
print(f"The distance between poles equals {earth_distance} miles")
```

The distance between poles equals 12428.14053760122 miles

The distance between the two poles is approximately 12,400 miles. We were able to compute it by converting the latitudes and longitudes to radians, calculating their unit-sphere distance, and then multiplying that value by the radius of Earth. We can now create a general `travel_distance` function to calculate the travel mileage between any two terrestrial points.

Listing 11.6 Defining a travel distance function

```
def travel_distance(lat1, lon1, lat2, lon2):
    to_radians = np.radians([lat1, lon1, lat2, lon2])
    return 3956 * great_circle_distance(*to_radians.tolist())

assert travel_distance(90, 0, -90, 0) == earth_distance
```

Our `travel_distance` function is a non-Euclidean metric for measuring distances between locations. As discussed in the previous section, we can pass such metrics into the DBSCAN clustering algorithm, so we can use `travel_distance` to cluster locations based on their spatial distributions. Then we can visually validate the clusters by plotting the locations on a map. This map plot can be executed using the external Cartopy visualization library.

11.2 Plotting maps using Cartopy

Visualizing geographic data is a common data science task. One external library used to map such data is Cartopy: a Matplotlib-compatible tool for generating maps in Python. Unfortunately, Cartopy can be a little tricky to install. Every other library in this book can be installed with the one-line `pip install` command. This calls the `pip` package-management system, which then connects to an external server of Python libraries. Pip subsequently installs the selected library along with all its Python dependencies, which represent additional library requirements.

NOTE For example, NumPy is a dependency of Matplotlib. Calling `pip install matplotlib` automatically installs NumPy on a local machine, if NumPy has not been installed already.

Pip works well when the dependencies are all written in Python. However, Cartopy has a dependency that's written in C++. The GEOS library is a geo-spatial engine that underlies Cartopy's visualizations. It cannot be installed using pip, so Cartopy also cannot be installed directly using pip. We're left with two options:

- Manually installing GEOS and Cartopy
- Installing the Cartopy library using the Conda package manager

Let's discuss the pros and cons of each approach.

NOTE For a deeper dive into Python dependencies, see Manning's "Managing Python Dependencies" liveVideo: www.manning.com/livevideo/talk-python-managing-python-dependencies.

11.2.1 Manually installing GEOS and Cartopy

The GEOS installation varies based on the operating system. On macOS, it can be installed by calling `brew install proj geos` from the command line; and on Linux, it can be installed by calling `apt-get` instead of `brew`. Additionally, Windows users can download and install the library from <https://trac.osgeo.org/geos>. Once GEOS is installed, Cartopy and its dependencies can be added with the following sequential pip commands:

- 1 `pip install --upgrade cython numpy pyshp six`
This installs all Python dependencies except the Shapely shape-rendering library.
- 2 `pip install shapely --no-binary shapely`
The Shapely library must be compiled from scratch so that it links to GEOS.
The `no-binary` command ensures a fresh compilation.
- 3 `pip install cartopy`
Now that the dependencies are ready, we call Cartopy using pip.

Manual installation can be cumbersome. Our alternative is to utilize the Conda package manager.

11.2.2 Utilizing the Conda package manager

Conda, like pip, is a package manager that can download and install external libraries. Unlike pip, Conda can easily handle non-Python dependencies. Also unlike pip, Conda does not come preinstalled on most machines: it must be downloaded and installed from <https://docs.conda.io/en/latest/miniconda.html>. Then we can easily install the Cartopy library by running `conda install -c conda-forge cartopy`.

Unfortunately, using Conda has some trade-offs. When Conda installs a new Python library, it does so in an isolated environment called a *virtual environment*. The

virtual environment has its own version of Python, which is separated from the main version of Python that resides on a user's machine. Consequently, the Cartopy library is installed in the virtual environment but not the main environment. This can cause confusion when importing Cartopy, especially in a Jupyter notebook, because Jupyter points to the main environment by default. To add the Conda environment to Jupyter, we must run the following two commands:

- 1 conda install -c anaconda ipykernel
- 2 python -m ipykernel install --user --name=base

Doing so ensures that the Jupyter notebook can interact with a Conda environment called base, which is the default name of the environment created by Conda.

Now we can select the base environment from Jupyter's drop-down menu when creating a new notebook (figure 11.3). Then we'll be able to import Cartopy in the notebook.

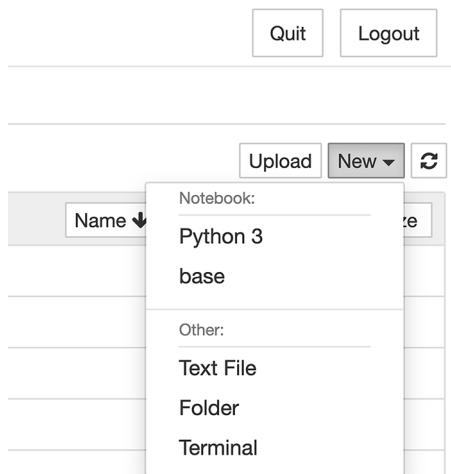


Figure 11.3 Selecting the environment when creating a new notebook. Conda's base environment can be selected from the drop-down menu. Choosing base allows us to import the installed Cartopy library.

NOTE Conda's default virtual environment is called base. However, Conda allows us to create and track multiple environments. To create a new virtual environment called new_env, we need to execute `conda create -n new_env` from the command line. Then we can switch to the new environment by running `conda activate new_env`. Running `conda activate base` switches back to base, where Cartopy is installed. Additionally, the `conda deactivate` command switches to our machine's default Python settings. We can also check the current environment's name by running `conda info`.

Let's confirm installation by running `import cartopy` from within a Jupyter notebook.

Listing 11.7 Importing the Cartopy library

```
import cartopy
```

Cartopy installation can be confusing, but the confusion is worth it. Cartopy is the best, most commonly used map visualization tool for Python. Let's plot some maps.

11.2.3 Visualizing maps

A geographic map is a 2D representation of a 3D surface on a globe. Flattening the spherical globe is carried out using a process called *projection*. There are many different types of map projections: the simplest involves superimposing the globe on an unrolled cylinder, which yields a 2D map whose (x, y) coordinates perfectly correspond with longitude and latitude.

NOTE In most other projections, the 2D grid coordinates don't equal the spherical coordinates. Hence, they require conversion from one coordinate system to another. We encounter this issue later in the section.

This technique is called the *equidistant cylindrical projection* or *plate carrée projection*. We utilize this standard projection in our plots by importing PlateCarree from cartopy.crs.

NOTE The cartopy.crs module includes many other projection types. We can, for instance, import Orthographic: doing so returns an *orthographic projection* in which the Earth is represented from the perspective of a viewer in the outer reaches of the galaxy.

Listing 11.8 Importing the plate carrée projection

```
from cartopy.crs import PlateCarree
```

The PlateCarree class can be used in conjunction with Matplotlib to visualize the Earth. For instance, running plt.axes(projection=PlateCarree()).coastlines() plots the outlines of the Earth's seven continents. More precisely, plt.axes(projection=PlateCarree()) initializes a custom Matplotlib axis capable of visualizing maps. Subsequently, the coastlines method call draws the coastline boundaries of the continents (figure 11.4).

Listing 11.9 Visualizing the Earth using Cartopy

```
plt.axes(projection=PlateCarree()).coastlines()  
plt.show()
```

Our plotted map is a bit small. We can increase the map size using Matplotlib's plt.figure function. Calling plt.figure(figsize=(width, height)) creates a figure that is width inches wide and height inches high. Listing 11.10 increases the figure size to 12 x 8 inches before generating the world map (figure 11.5).

NOTE The actual dimensions of the figure in the book are not 12 x 8 inches due to image formatting.



Figure 11.4 A standard map of the Earth on which the coastlines of the continents have been plotted

Listing 11.10 Visualizing a larger map of the Earth

```
plt.figure(figsize=(12, 8))  
plt.axes(projection=PlateCarree()).coastlines()  
plt.show()
```

Creates a larger figure
that is 12 inches wide
and 8 inches high

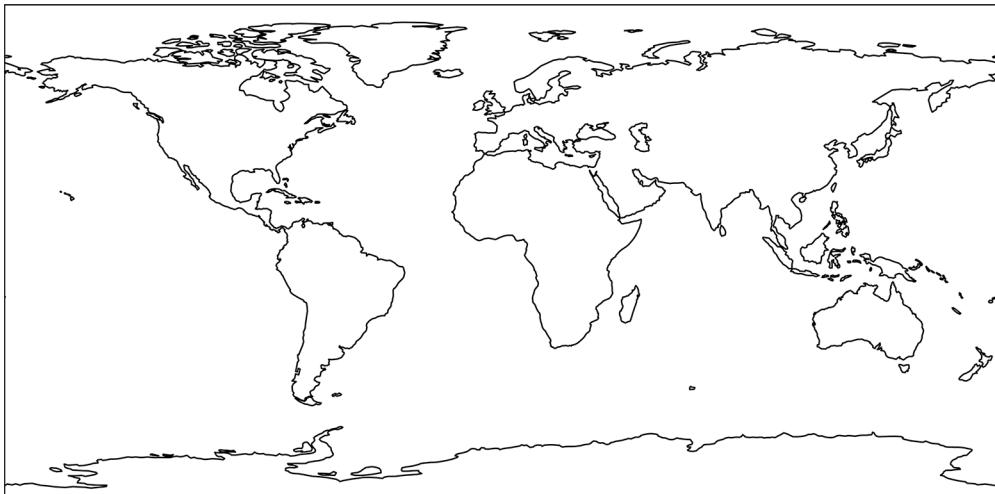


Figure 11.5 A standard map of the Earth on which the coastlines of the continents have been plotted. The map's size has been increased using Matplotlib's plt.figure function.

So far, our map looks sparse and uninviting. We can improve the quality by calling `plt.axes(projection=PlateCarree()).stock_img()`. The method call colors the map using topographic information: oceans are colored blue, and forested regions are colored green (figure 11.6).

Listing 11.11 Coloring a map of the Earth

```
fig = plt.figure(figsize=(12, 8))
plt.axes(projection=PlateCarree()).stock_img()
plt.show()
```

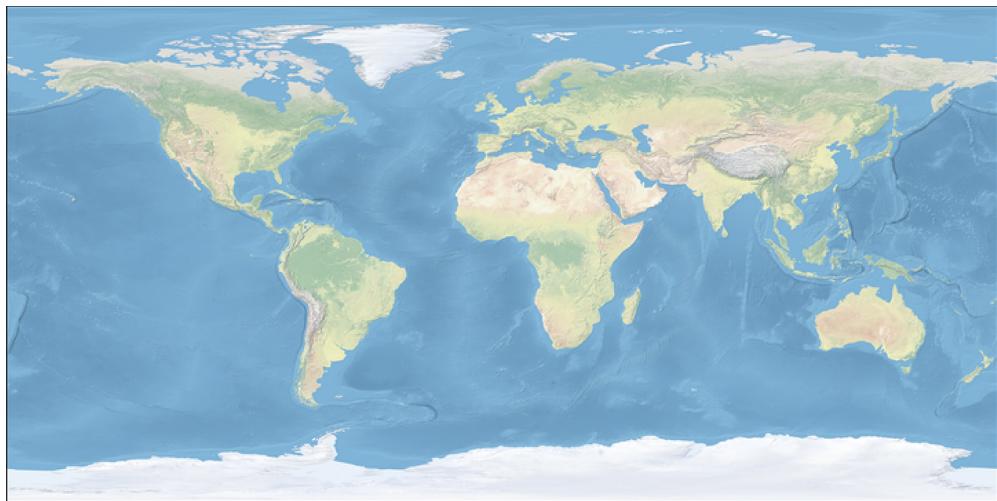


Figure 11.6 A standard map of the Earth that has been colored to display oceanographic and topographic details

Our colored map does not include the border lines demarcating coastal boundaries. Adding these boundaries will improve the map's quality. However, we are unable to add both color and boundaries in a single line of code. Instead, we need to execute the following three lines (figure 11.7):

1 `ax = plt.axes(projection=PlateCarree())`

This line initializes a custom Matplotlib axis capable of visualizing maps. Per standard convention, the axis is assigned to the `ax` variable.

2 `ax.coastlines()`

This line adds the coastlines to the plot.

3 `ax.stock_img()`

This line adds the topographic colors to the plot.

Let's run these steps to generate a crisp, colorful map.

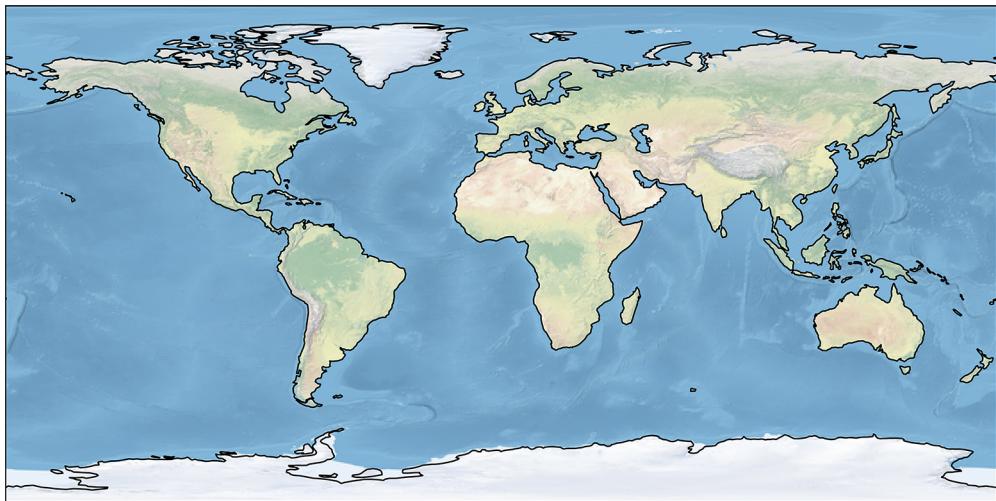


Figure 11.7 A standard map of the Earth that has been colored to display oceanographic and topographic details. Furthermore, plotted coastlines provide crisp details for the continental boundaries.

Listing 11.12 Plotting coastlines together with map colors

```
plt.figure(figsize=(12, 8))
ax = plt.axes(projection=PlateCarree())
ax.coastlines()
ax.stock_img()
plt.show()
```

Note that `ax.stock_img()` relies on a saved stock image of the Earth to color the map. This image renders poorly when a user zooms in on the map (which we'll do shortly). Alternatively, we can color the oceans and continents using the `ax.add_feature` method, which displays special Cartopy features stored in the `cartopy.feature` module. For example, calling `ax.add_feature(cartopy.feature.OCEAN)` colors all the oceans blue, and inputting `cartopy.feature.LAND` colors all land masses beige. Let's utilize these features to color the map (figure 11.8).

Listing 11.13 Adding colors with the feature module

```
plt.figure(figsize=(12, 8))
ax = plt.axes(projection=PlateCarree())
ax.coastlines()
ax.add_feature(cartopy.feature.OCEAN)
ax.add_feature(cartopy.feature.LAND)
plt.show()
```

We continue to display the coastlines to add crispness to the image.

Currently, national borders are missing from the plot. Cartopy treats these borders as a feature in the feature module. We can incorporate country borders by calling `ax.add_feature(cartopy.feature.BORDERS)` (figure 11.9).

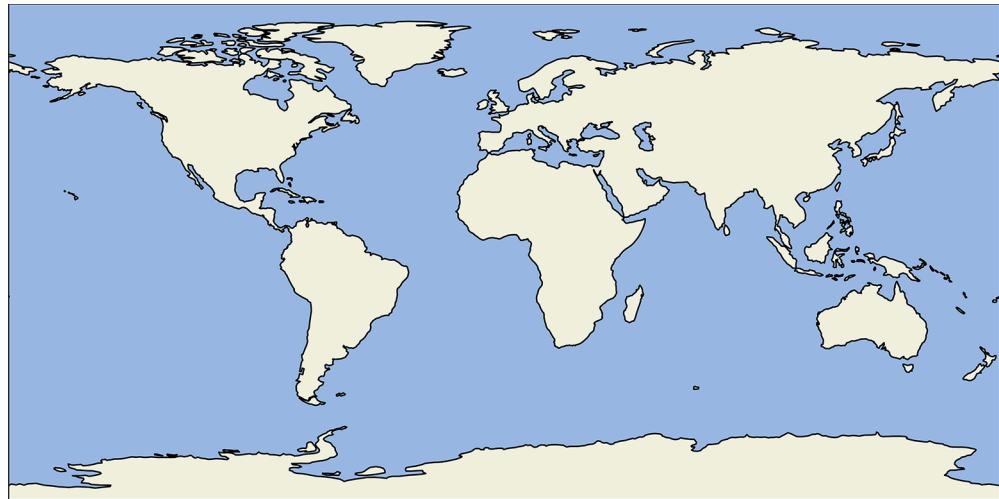


Figure 11.8 A standard map of the Earth that has been colored using the `feature` module

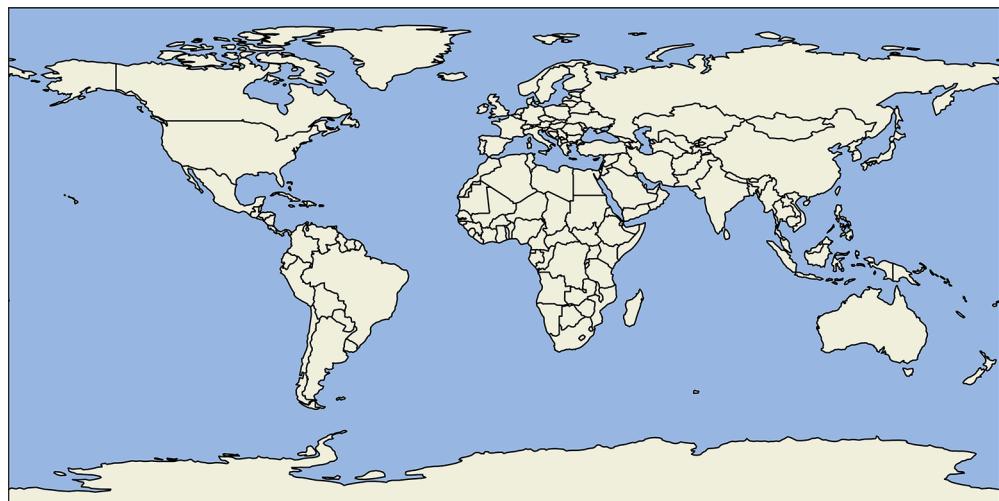


Figure 11.9 A standard map of the Earth including national borderlines

Listing 11.14 Adding national borders to the plot

```
plt.figure(figsize=(12, 8))
ax = plt.axes(projection=PlateCarree())
ax.coastlines()
ax.add_feature(cartopy.feature.BORDERS)
ax.add_feature(cartopy.feature.OCEAN)
ax.add_feature(cartopy.feature.LAND)
plt.show()
```

Suppose we are given a list of locations defined by pairs of latitudes and longitudes. We can plot these locations on a global map as a standard scatter plot by calling `ax.scatter(longitudes, latitudes)`. However, Matplotlib zooms in on the scattered points by default, making the mapped image incomplete. We can prevent this by calling `ax.set_global()`, which extends the plotted image to all four edges of the globe. Listing 11.15 plots some geographic points; for simplicity, we limit our map content to the coastal boundaries (figure 11.10).

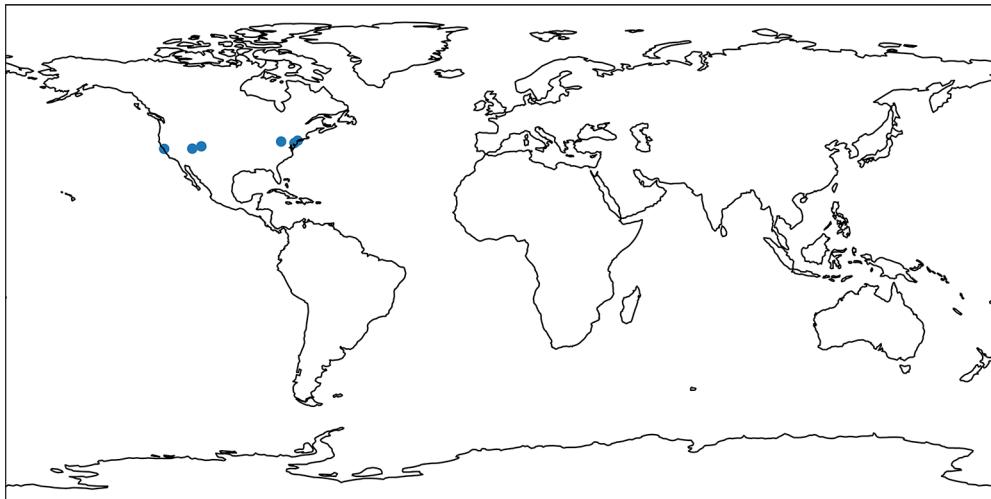


Figure 11.10 A standard map of the Earth with plotted latitude and longitude coordinates

NOTE As previously mentioned, the plate carrée projection yields a 2D grid in which longitudes and latitudes can be plotted directly on the axes. For other projections, this is not the case: they require a transformation of longitudes and latitudes before the scatter plot is generated. Shortly, we discuss how to properly handle that transformation.

Listing 11.15 Plotting coordinates on a map

```
plt.figure(figsize=(12, 8))
coordinates = [(39.9526, -75.1652), (37.7749, -122.4194),
                (40.4406, -79.9959), (38.6807, -108.9769),
                (37.8716, -112.2727), (40.7831, -73.9712)]

latitudes, longitudes = np.array(coordinates).T
ax = plt.axes(projection=PlateCarree())
ax.scatter(longitudes, latitudes)
ax.set_global()
ax.coastlines()
plt.show()
```

The plotted points all fall within the borders of North America. We can simplify the map by zooming in on that continent. However, first we need to adjust the *map extent*, which is the geographic area shown on a map. The extent is determined by a rectangle whose corners are positioned on the minimum and maximum latitude and longitude coordinates on display. In Cartopy, these corners are defined by a four-element tuple of the form `(min_lon, max_lon, min_lat, max_lat)`. Passing that list into `ax.set_extent` adjusts the boundaries of the map.

We now assign a common North American extent to a `north_america_extent` variable. Then we utilize the `ax.set_extent` method to zoom in on North America. We regenerate our scatter plot, this time adding color by passing `color='r'` into `ax.scatter`; we also utilize the `feature` module to color the map while adding national borders (figure 11.11).

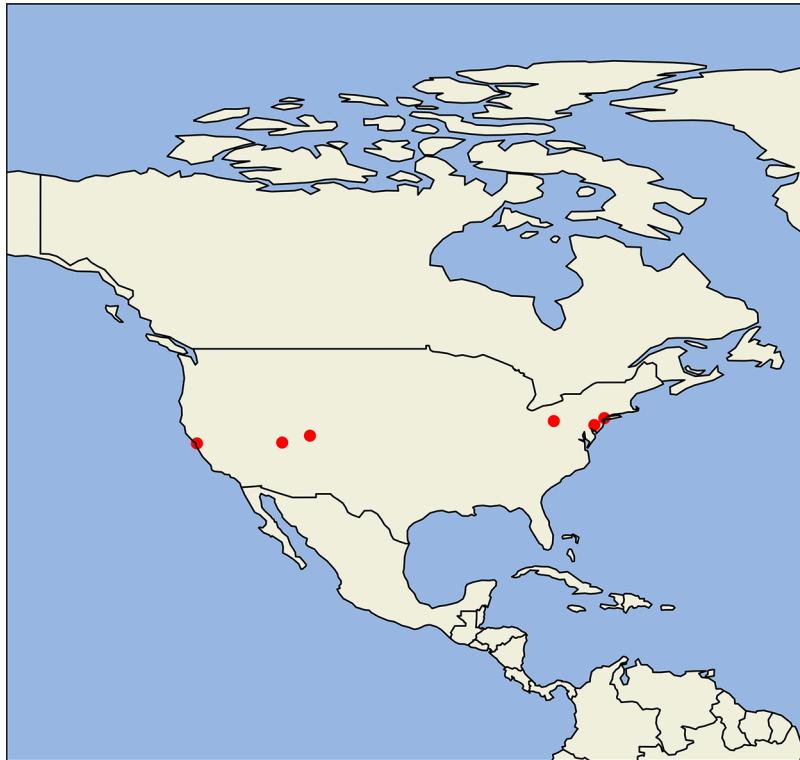


Figure 11.11 A map of North America with plotted latitude and longitude coordinates

Listing 11.16 Plotting North American coordinates

```
plt.figure(figsize=(12, 8))
ax = plt.axes(projection=PlateCarree())
```

```

north_america_extent = (-145, -50, 0, 90)
ax.set_extent(north_america_extent)
ax.scatter(longitudes, latitudes, color='r')

def add_map_features():
    ax.coastlines()
    ax.add_feature(cartopy.feature.BORDERS)
    ax.add_feature(cartopy.feature.OCEAN)
    ax.add_feature(cartopy.feature.LAND)

add_map_features()
plt.show()

```

The North American extent occurs between -145 and -50 degrees longitude and between 0 and 90 degrees latitude.

This function adds common features to a map. It is reused elsewhere in this section.

We successfully zoomed in on North America. Now we'll zoom in further, to the United States. Unfortunately, the plate carrée projection is insufficient for this purpose: that technique distorts the map if we zoom in too close on any country.

Instead, we will rely on the *Lambert conformal conic projection*. In this projection, a cone is placed on top of the spherical Earth. The cone's circular base covers the region we intend to map. Then, coordinates in the region are projected onto the surface of the cone. Finally, the cone is unrolled to create a 2D map. However, that map's 2D coordinates don't directly equal longitude and latitude.

Cartopy includes a `LambertConformal` class in the `crs` module. Executing `plt.axes(projection=LambertConformal())` yields axes corresponding to the Lambert conformal coordinate system. Subsequently, passing the US extent into `ax.set_extent` will zoom the map onto the United States. Listing 11.17 defines `us_extent` and passes it into the method. We'll also plot our geographic data, but first we need to transform longitudes and latitudes into coordinates that are compatible with `LambertConformal`: in other words, we must transform the data from PlateCarree-compatible coordinates to something different. This can be done by passing `transform=PlateCarree()` into `ax.scatter()`. Let's run this transformation to visualize our points on a US map (figure 11.12).

NOTE When this code is first run, Cartopy downloads and installs the Lambert conformal projection. Hence, an internet connection is required to execute the code for the very first time.

Listing 11.17 Plotting US coordinates

```

from cartopy.crs import LambertConformal
plt.figure(figsize=(12, 8))
ax = plt.axes(projection=LambertConformal())
us_extent = (-120, -75, 20, 50)
ax.set_extent(us_extent)

ax.scatter(longitudes, latitudes, color='r',
           transform=PlateCarree(),
           s=100)

```

Imports the Lambert conformal conic projection

The ax axis corresponds with LambertConformal coordinates.

The US extent occurs between -120 and -75 degrees longitude and between 20 and 50 degrees latitude.

The s parameter specifies the plotted marker size. We increase that size for better visibility.

```
add_map_features()  
plt.show()
```

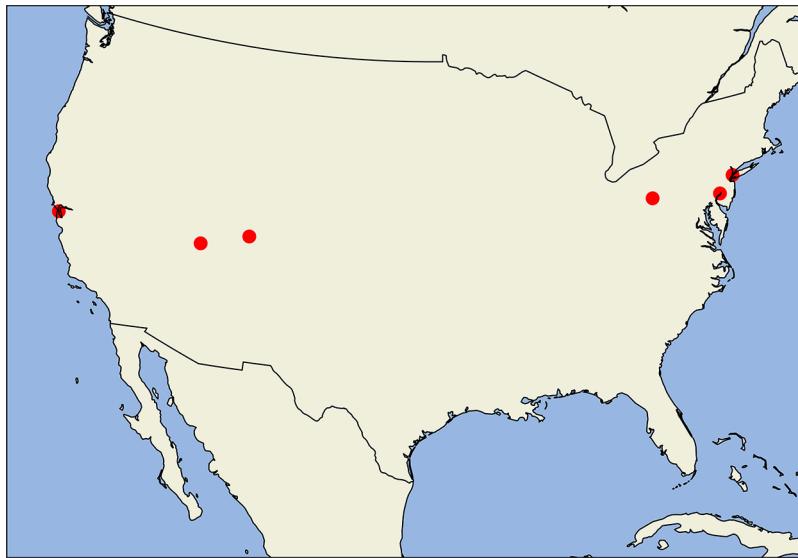


Figure 11.12 A Lambert conformal view of the United States with plotted latitude and longitude coordinates

Our map of the United States is looking a little sparse. Let's add state borders by calling `ax.add_feature(cartopy.feature.STATES)` (figure 11.13).

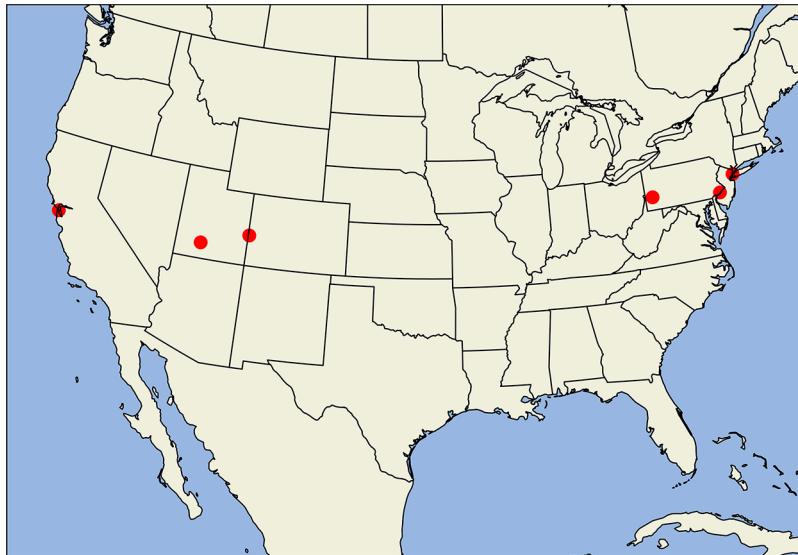


Figure 11.13 A Lambert conformal view of the United States including state borders

Listing 11.18 Plotting a US map including state borders

```
fig = plt.figure(figsize=(12, 8))
ax = plt.axes(projection=LambertConformal())
ax.set_extent(us_extent)

ax.scatter(longitudes, latitudes, color='r',
           transform=PlateCarree(),
           s=100)

ax.add_feature(cartopy.feature.STATES)
add_map_features()
plt.show()
```

Common Cartopy methods

- ax = plt.axes(projection=PlateCarree())—Creates a custom Matplotlib axis for generating a map using a plate carrée projection
- ax = plt.axes(projection=LambertConformal())—Creates a custom Matplotlib axis for generating a map using a Lambert conformal conic projection
- ax.coastlines()—Plots continental coastlines on a map
- ax.add_feature(cartopy.feature.BORDERS)—Plots national boundaries on a map
- ax.add_feature(cartopy.feature.STATES)— Plots US state boundaries on a map
- ax.stock_img()—Colors a plotted map using topographic information
- ax.add_feature(cartopy.feature.OCEAN)—Colors all the oceans blue on the map
- ax.add_feature(cartopy.feature.LAND)—Colors all the land masses beige on the map
- ax.set_global()—Extends the plotted image to all four edges of the globe
- ax.set_extent(min_lon, max_lon, min_lat, max_lat)—Adjusts the plotted map extent, which is the geographic area shown on a map, using minimum and maximum latitudes and longitudes
- ax.scatter(longitudes, latitudes)—Plots latitude and longitude coordinates on a map
- ax.scatter(longitudes, latitudes, transform=PlateCarree())—Plots latitude and longitude coordinates on a map while transforming the data from PlateCarree-compatible coordinates to something different (such as Lambert conformal conic coordinates)

Cartopy allows us to plot any location on a map. All we need is the location's latitude and longitude. Of course, we must know these geographic coordinates before plotting them on a map, so we need a mapping between location names and their geographic properties. That mapping is provided by the GeoNamesCache location-tracking library.

11.3 Location tracking using GeoNamesCache

The GeoNames database (<http://geonames.org>) is an excellent resource for obtaining geographic data. GeoNames contains over 11 million place names spanning all the countries in the world. In addition, GeoNames stores valuable information such as latitude and longitude. Thus, we can use the database to determine the precise geographic locations of cities and countries discovered in text.

How do we access the GeoNames data? Well, we could manually download the GeoNames data dump (<http://download.geonames.org/export/dump>), parse it, and then store the output data structure. That would take a lot of work. Fortunately, someone has already done the hard work for us by creating the GeoNamesCache library.

GeoNamesCache is designed to efficiently retrieve data about continents, countries, cities, and US counties and states. The library provides six easy-to-use methods to support access to location data: `get_continents`, `get_countries`, `get_cities`, `get_countries_by_name`, `get_cities_by_name`, and `get_us_counties`. Let's install the library and explore its usage in more detail. We begin by initializing a `GeonamesCache` location-tracking object.

NOTE Call `pip install geonamescache` from the command line terminal to install the `GeoNamesCache` library.

Listing 11.19 Initializing a `GeonamesCache` object

```
from geonamescache import GeonamesCache  
gc = GeonamesCache()
```

Let's use our `gc` object to explore the seven continents. We run `gc.get_continents()` to retrieve a dictionary of continent-related information. Then we investigate the dictionary's structure by printing out its keys.

Listing 11.20 Fetching all seven continents from `GeoNamesCache`

```
continents = gc.get_continents()  
print(continents.keys())  
  
dict_keys(['AF', 'AS', 'EU', 'NA', 'OC', 'SA', 'AN'])
```

The dictionary keys represent shorthand encoding of continent names in which *Africa* is transformed into 'AF' and *North America* is transformed into 'NA'. Let's check the values mapped to every key by passing in the code for *North America*.

NOTE `continents` is a nested dictionary. Thus, the seven top-level keys map to content-specific dictionary structures. Listing 11.21 outputs the content-specific keys contained in the `continents['NA']` dictionary.

Listing 11.21 Fetching North America from GeoNamesCache

```
north_america = continents['NA']
print(north_america.keys())

dict_keys(['lng', 'geonameId', 'timezone', 'bbox', 'toponymName',
'asciiName', 'astergdem', 'fcl', 'population', 'wikipediaURL',
'adminName5', 'srtm3', 'adminName4', 'adminName3', 'alternateNames',
'cc2', 'adminName2', 'name', 'fclName', 'fcdeName', 'adminName1',
'lat', 'fcode', 'continentCode'])
```

Many of the north_america data elements represent various naming schemes for the North American continent. Such information is not very useful.

Listing 11.22 Printing North America's naming schemes

```
for name_key in ['name', 'asciiName', 'toponymName']:
    print(north_america[name_key])

North America
North America
North America
```

However, other elements hold more value. For example, the 'lat' and 'lng' keys map to the latitude and longitude of the most central location in North America. Let's visualize this location on a map (figure 11.14).

Listing 11.23 Mapping North America's central coordinates

```
latitude = float(north_america['lat'])
longitude = float(north_america['lng'])

plt.figure(figsize=(12, 8))
ax = plt.axes(projection=PlateCarree())
ax.set_extent(north_america_extent)
ax.scatter([longitude], [latitude], s=200)
add_map_features()
plt.show()
```

The lat and lng keys map to North America's central latitude and longitude.

11.3.1 Accessing country information

The ability to access continental data is useful, although our primary concern is analyzing cities and countries. We can analyze countries using the `get_countries` method. It returns a dictionary whose two-character keys encode the names of 252 different countries. As with the continents, the country codes capture the abbreviated country names. For example, the code for *Canada* is 'CA', and the code for *United States* is 'US'. Accessing `gc.get_countries()['US']` returns a dictionary containing useful US data.



Figure 11.14 The central North American latitude and longitude plotted on a map of North America

Listing 11.24 Fetching US data from GeoNamesCache

```
countries = gc.get_countries()
num_countries = len(countries)
print(f"GeonamesCache holds data for {num_countries} countries.")

us_data = countries['US']
print("The following data pertains to the United States:")
print(us_data)

GeonamesCache holds data for 252 countries.
The following data pertains to the United States:
{'geonameid': 6252001,
'name': 'United States',
'iso': 'US',
'iso3': 'USA',
'isocode': 840,
'fips': 'US',
'continentcode': 'NA',
'capital': 'Washington',
'areakm2': 9629091,
```

US continent code
Capital of the US
US area, in square kilometers

```
'population': 310232863,           ← US population
'tld': '.us',
'currencycode': 'USD',
'currencyname': 'Dollar',          ← Currency of the US
'phone': '1',
'postalcodeeregex': '^\\d{5}(-\\d{4})?$', ← Common spoken
'languages': 'en-US,es-US,haw,fr',   languages in the US
'neighbours': 'CA,MX,CU'}          ← US neighboring territories
```

The outputted country data includes many useful elements, such as the country's capital, currency, area, spoken languages, and population. Regrettably, GeoNamesCache fails to provide the central latitude and longitude associated with the country's area. However, as we shortly discover, a country's centrality can be estimated using city coordinates.

Additionally, there is valuable information in each country's 'neighbours' element (the spelling is written in British English). The 'neighbours' key maps to a comma-delimited string of country codes that signify neighboring territories. We can obtain more details about each neighbor by splitting the string and passing the codes into the 'countries' dictionary.

Listing 11.25 Fetching neighboring countries

```
us_neighbors = us_data['neighbours']
for neighbor_code in us_neighbors.split(',') :
    print(countries[neighbor_code] ['name'])

Canada
Mexico
Cuba
```

According to GeoNamesCache, the immediate neighbors of the United States are Canada, Mexico, and Cuba. We can all agree on the first two locations, although whether Cuba is a neighbor remains questionable. Cuba does not directly border the United States. Also, if the Caribbean island nation is really a neighbor, why isn't Haiti included in that list? More importantly, how did Cuba get included in the first place? Well, GeoNames is a collaborative project run by a community of editors (like a location-focused Wikipedia). At some point, an editor decided that Cuba is a neighbor of the United States. Some might disagree with this decision, so it is important to remember that GeoNames is not a gold standard repository of location information. Instead, it is a tool for quickly accessing large quantities of location data. Some of that data may be imprecise, so please be cautious when using GeoNamesCache.

The `get_countries` method requires a country's two-character code. However, for most countries, we will not know the code. Fortunately, we can query all countries by name using the `get_countries_by_names` method, which returns a dictionary whose elements are country names rather than codes.

Listing 11.26 Fetching countries by name

```
result = gc.get_countries_by_names() ['United States']
assert result == countries['US']
```

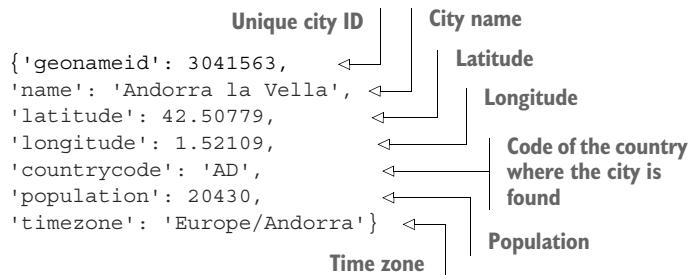
11.3.2 Accessing city information

Now, let's turn our attention to analyzing cities. The `get_cities` method returns a dictionary whose keys are unique IDs mapping back to city data. The following code outputs that data for a single city.

Listing 11.27 Fetching cities from GeoNamesCache

```
cities = gc.get_cities()
num_cities = len(cities)
print(f"GeoNamesCache holds data for {num_cities} total cities")
city_id = list(cities.keys())[0]
print(cities[city_id])
```

`cities` is a dictionary mapping a unique `city_id` to geographic information.



The data for each city contains the city name, its latitude and longitude, its population, and the reference code for the country where that city is located. By utilizing the country code, we can create a new mapping between a country and all of its territorial cities. Let's isolate and count all US cities stored in `GeoNamesCache`.

NOTE As we've discussed, `GeoNames` is not perfect. Certain US cities may be missing from the database. Over time, these cities will be added. Thus, the observed city count may increase with every library update.

Listing 11.28 Fetching US cities from GeoNamesCache

```
us_cities = [city for city in cities.values()
             if city['countrycode'] == 'US']
num_us_cities = len(us_cities)
print(f"GeoNamesCache holds data for {num_us_cities} US cities.")

GeoNamesCache holds data for 3248 US cities
```

`GeoNamesCache` contains information about more than 3,000 US cities. Each city's data dictionary contains a latitude and a longitude. Let's find the average US latitude and longitude, which will approximate the central coordinates of the United States.

Note that the approximation is not perfect. The calculated average does not take into account the curvature of the Earth and is inappropriately weighted by city location. A disproportionate number of US cities are located near the Atlantic Ocean, and thus the approximation is skewed toward the East. In the following code, we approximate and plot the US center while remaining fully aware that our approximation is not ideal (figure 11.15).

Listing 11.29 Approximating US central coordinates

```
center_lat = np.mean([city['latitude']
                     for city in us_cities])
center_lon = np.mean([city['longitude']
                     for city in us_cities])

fig = plt.figure(figsize=(12, 8))
ax = plt.axes(projection=LambertConformal())
ax.set_extent(us_extent)
ax.scatter([center_lon], [center_lat], transform=PlateCarree(), s=200)
ax.add_feature(cartopy.feature.STATES)
add_map_features()
plt.show()
```



Figure 11.15 The central location of the United States is approximated by averaging the coordinates of every US city in GeoNamesCache. The approximation is slightly skewed toward the east.

The `get_cities` method is suitable for iterating over city information but not querying cities by name. To search by name, we must rely on `get_cities_by_name`. This

method takes as an input a city name and returns a list of data outputs for all cities with that name.

Listing 11.30 Fetching cities by name

```
matched_cities_by_name = gc.get_cities_by_name('Philadelphia')
print(matched_cities_by_name)

[{'4560349': {'geonameid': 4560349, 'name': 'Philadelphia',
'latitude': 39.95233, 'longitude': -75.16379, 'countrycode': 'US',
'population': 1567442, 'timezone': 'America/New_York'}}]
```

The `get_cities_by_name` method may return more than one city because city names are not always unique. For example, GeoNamesCache contains six different instances of the city name San Francisco in five different countries. Calling `gc.get_cities_by_name('San Francisco')` returns data for each of these San Francisco instances. Let's iterate over that data and print the country where each San Francisco is found.

Listing 11.31 Fetching multiple cities with a shared name

```
matched_cities_list = gc.get_cities_by_name('San Francisco')

for i, san_francisco in enumerate(matched_cities_list):
    city_info = list(san_francisco.values())[0]
    country_code = city_info['countrycode']
    country = countries[country_code]['name']
    print(f"The San Francisco at index {i} is located in {country}")
```

```
The San Francisco at index 0 is located in Argentina
The San Francisco at index 1 is located in Costa Rica
The San Francisco at index 2 is located in Philippines
The San Francisco at index 3 is located in Philippines
The San Francisco at index 4 is located in El Salvador
The San Francisco at index 5 is located in United States
```

Multiple cities commonly share an identical name, and choosing among such cities can be difficult. Suppose, for instance, that someone queries a search engine for the "weather in Athens." The search engine must then choose between Athens, Ohio and Athens, Greece. Additional context is required to correctly disambiguate between the locations. Is the user from Ohio? Are they planning a trip to Greece? Without that context, the search engine must guess. Usually, the safest guess is the city with the largest population. From a statistical standpoint, the more populous cities are more likely to be referenced in everyday conversation. Choosing the most-populated city isn't guaranteed to work all the time, but it's still better than making a completely random choice. Let's see what happens when we plot the most populated San Francisco location (figure 11.16).

Listing 11.32 Mapping the most populous San Francisco

```
best_sf = max(gc.get_cities_by_name('San Francisco'),  
             key=lambda x: list(x.values())[0]['population'])  
sf_data = list(best_sf.values())[0]  
sf_lat = sf_data['latitude']  
sf_lon = sf_data['longitude']  
  
plt.figure(figsize=(12, 8))  
ax = plt.axes(projection=LambertConformal())  
ax.set_extent(us_extent)  
ax.scatter(sf_lon, sf_lat, transform=PlateCarree(), s=200)  
add_map_features()  
ax.text(sf_lon + 1, sf_lat, ' San Francisco', fontsize=16,  
       transform=PlateCarree())  
plt.show()
```

The `ax.text` method allows us to write "San Francisco" at the specified longitude and latitude. We slightly shift the longitude to the right to avoid overlapping the scatter plot dot. Also, on this map, the state borders are not plotted to better display the written text.



Figure 11.16 Among the six San Franciscos stored in `GeoNamesCache`, the city with the largest population is in California, as expected.

Selecting the San Francisco with the largest population returns the well-known Californian city rather than any of the lesser-known locations outside of the United States.

Common GeoNamesCache methods

- `gc = GeonamesCache()`—Initializes a `GeonamesCache` object
- `gc.get_continents()`—Returns a dictionary mapping continent IDs to continent data
- `gc.get_countries()`—Returns a dictionary mapping country IDs to country data
- `gc.get_countries_by_names()`—Returns a dictionary mapping country names to country data
- `gc.get_cities()`—Returns a dictionary mapping city IDs to city data
- `gc.get_cities_by_name(city_name)`—Returns a list of cities that share the name `city_name`

11.3.3 Limitations of the `GeoNamesCache` library

`GeoNamesCache` is a useful tool, but it does have some significant flaws. First, the library's record of cities is far from complete. Certain sparsely populated locations in rural areas (whether the rural United States or rural China) are missing from the stored database records. Furthermore, the `get_cities_by_name` method maps only one version of a city's name to its geographic data. This poses a problem for cities like New York that have more than one commonly referenced name.

Listing 11.33 Fetching New York City from `GeoNamesCache`

```
for ny_name in ['New York', 'New York City']:
    if not gc.get_cities_by_name(ny_name):
        print(f"'{ny_name}' is not present in the GeoNamesCache database")
    else:
        print(f"'{ny_name}' is present in the GeoNamesCache database")

'New York' is not present in the GeoNamesCache database
'New York City' is present in the GeoNamesCache database
```

The single name-to-city mapping is particularly problematic due to the presence of diacritics in city names. *Diacritics* are accent marks that designate the proper pronunciation of non-English-sounding words. They are commonly found in city names: for example, Cañon City, Colorado; and Hagåtña, Guam.

Listing 11.34 Fetching accented cities from `GeoNamesCache`

```
print(gc.get_cities_by_name(u'Cañon City'))
print(gc.get_cities_by_name(u'Hagåtña'))

[{'geonameid': 5416005, 'name': 'Cañon City',
'latitude': 38.44098, 'longitude': -105.24245, 'countrycode': 'US',
'population': 16400, 'timezone': 'America/Denver'}]}
[{'geonameid': 4044012, 'name': 'Hagåtña',
'latitude': 13.47567, 'longitude': 144.74886, 'countrycode': 'GU',
'population': 1051, 'timezone': 'Pacific/Guam'}]
```

How many of the cities stored in GeoNamesCache contain diacritics in their name? We can find out using the unidecode function from the external Unidecode library. The function strips all accent marks out of input text. By checking for differences between the input text and output text, we should be able to detect all city names containing accent marks.

NOTE Call pip install Unidecode from the command line terminal to install the Unidecode library.

Listing 11.35 Counting all accented cities in GeoNamesCache

```
from unidecode import unidecode
accented_names = [city['name'] for city in gc.get_cities().values()
                  if city['name'] != unidecode(city['name'])]
num_accented_cities = len(accented_names)

print(f"An example accented city name is '{accented_names[0]}'")
print(f"{num_accented_cities} cities have accented names")

An example accented city name is 'Khawr Fakkān'
4896 cities have accented names
```

Approximately 5,000 stored cities have diacritics in their names. These cities are commonly referenced without an accent in published text data. One way to ensure that we match all such cities is to create a dictionary of alternative city names; in it, the accent-free unidecode output maps back to the original accented names.

Listing 11.36 Stripping accents from alternative city names

```
alternative_names = {unidecode(name): name
                     for name in accented_names}
print(gc.get_cities_by_name(alternative_names['Hagatna']))

[{'4044012': {'geonameid': 4044012, 'name': 'Hagåtña',
  'latitude': 13.47567, 'longitude': 144.74886, 'countrycode': 'GU',
  'population': 1051, 'timezone': 'Pacific/Guam'}}]
```

We can now match the stripped dictionary keys against all inputted text by passing the accented dictionary values into GeoNamesCache whenever a key match is found.

Listing 11.37 Finding accent-free city names in text

```
text = 'This sentence matches Hagatna'
for key, value in alternative_names.items():
    if key in text:
        print(gc.get_cities_by_name(value))
        break

[{'4044012': {'geonameid': 4044012, 'name': 'Hagåtña',
  'latitude': 13.47567, 'longitude': 144.74886, 'countrycode': 'GU',
  'population': 1051, 'timezone': 'Pacific/Guam'}}]
```

GeoNamesCache allows us to easily track locations along with their geographical coordinates. Using the library, we can also search for mentioned location names within any inputted text. However, finding names in text is not a trivial process. If we wish to match location names appropriately, we must learn proper Python text-matching techniques while also avoiding common pitfalls.

NOTE The final subsection is intended for readers who are unfamiliar with basic string matching and regular expressions. If you are already familiar with these techniques, feel free to skip ahead.

11.4 Matching location names in text

In Python, we can easily determine whether one string is a substring of another or if the start of a string contains some predefined text.

Listing 11.38 Basic string matching

```
assert 'Boston' in 'Boston Marathon'  
assert 'Boston Marathon'.startswith('Boston')  
assert 'Boston Marathon'.endswith('Boston') == False
```

Unfortunately, Python's basic string syntax is quite limited. For example, there is no direct string method for executing a case-insensitive substring comparison. Furthermore, Python's string methods can't directly distinguish between sub-characters in a string and sub-phrases in a sentence. So if we wish to determine whether the phrase 'in a' is present in a sentence, we cannot safely rely on basic matching. Otherwise, we run the risk of incorrectly matching character sequences such as 'sin apple' or 'win attached'.

Listing 11.39 Basic substring matching errors

```
assert 'in a' in 'sin apple'  
assert 'in a' in 'win attached'
```

To overcome these limitations, we must rely on Python's built-in regular expression processing library, `re`. A *regular expression* (or *regex* for short) is a string-encoded pattern that can be compared against some text. Coded regex patterns range from simple string copies to incredibly complex formulations that very few people can decipher. In this subsection, we focus on simple regex composition and matching.

Most regex matching in Python can be executed with the `re.search` function. This function takes two inputs: a regex pattern and the text against which the pattern will be matched. It returns a `Match` object if a match is found or `None` otherwise. The `Match` object contains a `start` method and an `end` method; these methods return the start index and end index of the matched string in the text.

Listing 11.40 String matching using regexes

```
import re
regex = 'Boston'
random_text = 'Clown Patty'
match = re.search(regex, random_text)
assert match is None

matchable_text = 'Boston Marathon'
match = re.search(regex, matchable_text)
assert match is not None
start, end = match.start(), match.end()
matched_string = matchable_text[start: end]
assert matched_string == 'Boston'
```

Additionally, case-insensitive string matching is a breeze with `re.search`. We simply pass `re.IGNORECASE` as an added flags parameter.

Listing 11.41 Case-insensitive matching using regexes

```
for text in ['BOSTON', 'boston', 'BoSTOn']:
    assert re.search(regex, text, flags=re.IGNORECASE) is not None
```

We can achieve the same result by passing flags=re.I into re.search.

Regexes also allow us to match exact words using word boundary detection. Adding the `\b` pattern to a regex string captures the start and end points of words (as defined by whitespaces and punctuation). However, because the backslash is a special character in the standard Python lexicon, we must take measures to ensure that the backslash is interpreted like a regular raw character. We do this by either adding another backslash to the backslash (a rather cumbersome approach) or preceding the string with an `r` literal. The latter solution ensures that the regex is treated as a raw string during analysis.

Listing 11.42 Word boundary matching using regexes

```
for regex in ['\\bin a\\b', r'\bin a\b']:
    for text in ['sin apple', 'win attached']:
        assert re.search(regex, text) is None

    text = 'Match in a string'
    assert re.search(regex, text) is not None
```

Now, let's carry out a more complicated match. We match against the sentence `f'I visited {city} yesterday`, where `{city}` represents one of three possible locations: Boston, Philadelphia, or San Francisco. The correct regex syntax for executing the match is `r'I visited \b(Boston|Philadelphia|San Francisco)\b yesterday'`.

NOTE The pipe | is an *Or* condition. It requires the regex to match from one of the three cities in our list. Furthermore, the parentheses limit the scope of the matched cities. Without them, the matched text range would stretch beyond 'San Francisco', all the way to 'San Francisco yesterday'.

Listing 11.43 Multicity matching using regexes

```
regex = r'I visited \b(Boston|Philadelphia|San Francisco)\b yesterday.'
assert re.search(regex, 'I visited Chicago yesterday.') is None

cities = ['Boston', 'Philadelphia', 'San Francisco']
for city in cities:
    assert re.search(regex, f'I visited {city} yesterday.') is not None
```

Finally, let's discuss how to run a regex search efficiently. Suppose we want to match a regex against 100 strings. For every match, `re.search` transforms the regex into a Python `PatternObject`. Each such transformation is computationally costly. We're better off executing the transformation only once using `re.compile`, which returns a compiled `PatternObject`. Then we can use the object's built-in search method while avoiding any additional compilation.

NOTE If we intend to use the compiled pattern for case-independent matching, we must pass `flags=re.IGNORECASE` into `re.compile`.

Listing 11.44 String matching using compiled regexes

```
compiled_re = re.compile(regex)
text = 'I visited Boston yesterday.'
for i in range(1000):
    assert compiled_re.search(text) is not None
```

Common regex matching techniques

- `match = re.search(regex, text)`—Returns a `Match` object if `regex` is present in `text` or `None` otherwise.
- `match = re.search(regex, text, flags=re.IGNORECASE)`—Returns a `Match` object if `regex` is present in `text` or `None` otherwise. Matching is carried out independent of case.
- `match.start()`—Returns the start index of a regex matched to an input text.
- `match.end()`—Returns an end index of a regex matched to an input text.
- `compiled_regex = re.compile(regex)`—Transforms the `regex` string into a compiled pattern-matching object.
- `match = compiled_regex.search(text)`—Uses the compiled object's built-in search method to match a regex against `text`.
- `re.compile('Boston')`—Compiles a regex to match the string 'Boston' against the text.

(continued)

- `re.compile('Boston', flags=re.IGNORECASE)`—Compiles a regex to match the string 'Boston' against the text. The matching is independent of text case.
- `re.compile('\\bBoston\\b')`—Compiles a regex to match the word 'Boston' against the text. Word boundaries are used to execute an exact word match.
- `re.compile(r'\bBoston\b')`—Compiles a regex to match the word 'Boston' against the text. The inputted regex is treated as a raw string because of the `r` literal. Thus, we don't need to add additional backslashes to our `\b` word boundary delimiters.
- `re.compile(r'\b(Boston|Chicago)\b')`—Compiles a regex to match either the word 'Boston' or the word 'Chicago' to the text.

Regex matching allows us to find location names in text. Thus, the `re` module will prove invaluable for solving case study 3.

Summary

- The shortest travel distance between terrestrial points is along our planet's spherical surface. This *great-circle distance* can be computed using a series of well-known trigonometric operations.
- The latitude and longitude are *spherical coordinates*. These coordinates measure the angular position of a point on the surface of the Earth relative to the x-axis and y-axis.
- We can plot a latitude and longitude on a map using the Cartopy library. The library can visualize mapped data using multiple projection types. Our choice of projection is dependent on the plotted data. If the data spans the globe, we can use the standard *plate carrée*. If the data is confined to North America, we might consider using the *orthographic projection*. If the data points are located in the continental United States, we should use the *Lambert conformal conic projection*.
- We can obtain latitudes and longitudes from location names using the GeoNamesCache library. GeoNamesCache maps city names to latitudes and longitudes. It also maps country names to cities. Thus, given a country name, we can approximate its central coordinates by averaging the latitudes and longitudes of its cities. However, that approximation will not be perfect due to city bias and the curved shape of the Earth.
- Multiple cities commonly share an identical name. Thus, GeoNamesCache can map multiple coordinates to a single city name. Given only a city name without any other context, it is advisable to return the coordinates of the most populous city with that name.

- GeoNamesCache maps coordinates to accented versions of each city name. We can strip out these accents using the `unidecode` function from the external `Unidecode` library.
- *Regular expressions* can find location names in text. By combining GeoNamesCache with Cartopy and regular expressions, we can plot locations mentioned in text.

Case study 3 solution

This section covers

- Extracting and visualizing locations
- Cleaning data
- Clustering locations

Our goal is to extract locations from disease-related headlines to uncover the largest active epidemics within and outside of the United States. We will do as follows:

- 1 Load the data.
- 2 Extract locations from the text using regular expressions and the GeoNames-Cache library.
- 3 Check the location matches for errors.
- 4 Cluster the locations based on geographic distance.
- 5 Visualize the clusters on a map, and remove any errors.
- 6 Output representative locations from the largest clusters to draw interesting conclusions.

WARNING Spoiler alert! The solution to case study 3 is about to be revealed. I strongly encourage you to try to solve the problem before reading the solution. The original problem statement is available for reference at the beginning of the case study.

12.1 Extracting locations from headline data

We begin by loading the headline data.

Listing 12.1 Loading headline data

```
headline_file = open('headlines.txt', 'r')
headlines = [line.strip()
            for line in headline_file.readlines()]
num_headlines = len(headlines)
print(f"{num_headlines} headlines have been loaded")

650 headlines have been loaded
```

We have loaded 650 headlines. Now we need a mechanism for extracting city and country names from the headline text. One naive solution is to match the locations in GeoNamesCache against each and every headline. However, this approach will fail to match locations whose capitalization and accent marks diverge from the stored GeoNamesCache data. For more optimal matching, we should transform each location name into a case-independent and accent-independent regular expression. We can execute these transformations using a custom `name_to_regex` function. That function takes a location name as input and returns a compiled regular expression capable of identifying any location of our choosing.

Listing 12.2 Converting names to regexes

```
def name_to_regex(name):
    decoded_name = unidecode(name)
    if name != decoded_name:
        regex = fr'\b({name} | {decoded_name})\b'
    else:
        regex = fr'\b{name}\b'
    return re.compile(regex, flags=re.IGNORECASE)
```

Using `name_to_regex`, we can create a mapping between regular expressions and the original names in GeoNamesCache. Let's create two dictionaries, `country_to_name` and `city_to_name`, which map regular expressions to country names and city names, respectively.

Listing 12.3 Mapping names to regexes

```
countries = [country['name']
            for country in gc.get_countries().values()]
country_to_name = {name_to_regex(name): name
                  for name in countries}

cities = [city['name'] for city in gc.get_cities().values()]
city_to_name = {name_to_regex(name): name for name in cities}
```

Next, we use our mappings to define a function that looks for location names in text. The function takes as input both a headline and a location dictionary. It iterates over each regex key in the dictionary, returning the associated value if the regex pattern matches the headline.

Listing 12.4 Finding locations in text

```
def get_name_in_text(text, dictionary):
    for regex, name in sorted(dictionary.items(),
                               key=lambda x: x[1]): ←
        if regex.search(text):
            return name
    return None
```

Iterating over dictionaries gives us a nondeterministic sequence of results. A change in sequence order could alter which locations are matched to the inputted text. This is especially true if multiple locations are present in the text. Sorting by location name ensures that function output does not change from run to run.

We utilize `get_name_in_text` to discover the cities and countries mentioned in the headlines list. Then we store the results in a Pandas table for easier analysis.

Listing 12.5 Finding locations in headlines

```
import pandas as pd

matched_countries = [get_name_in_text(headline, country_to_name)
                     for headline in headlines]
matched_cities = [get_name_in_text(headline, city_to_name)
                  for headline in headlines]
data = {'Headline': headlines, 'City': matched_cities,
        'Country': matched_countries}
df = pd.DataFrame(data)
```

Let's explore our location table. We start by summarizing the contents of `df` using the `describe` method.

Listing 12.6 Summarizing the location data

```
summary = df[['City', 'Country']].describe()
print(summary)
```

	City	Country
count	619	15
unique	511	10
top	Of	Brazil
freq	45	3

NOTE Multiple countries in the data share the top occurrence frequency of 3. Pandas does not have a deterministic method for selecting one top country over another. Depending on your local settings, a country other than Brazil could be returned as a top country, but it will still have a frequency of 3.

The table contains 619 mentions of cities representing 511 unique city names. It also contains just 15 countries representing 10 unique country names. The most frequently mentioned country is Brazil, which appears in three headlines.

The most frequently mentioned city is apparently “Of,” Turkey. That doesn’t seem right! The 45 instances of “Of” are more likely to match the preposition than the rarely referenced Turkish location. We will output some instances of “Of” to confirm the error.

Listing 12.7 Fetching cities named "Of"

```
of_cities = df[df.City == 'Of'][['City', 'Headline']]
ten_of_cities = of_cities.head(10)
print(ten_of_cities.to_string(index=False))
```

City	Headline
Of	Case of Measles Reported in Vancouver
Of	Authorities are Worried about the Spread of Br...
Of	Authorities are Worried about the Spread of Ma...
Of	Rochester authorities confirmed the spread of ...
Of	Tokyo Encounters Severe Symptoms of Meningitis
Of	Authorities are Worried about the Spread of In...
Of	Spike of Pneumonia Cases in Springfield
Of	The Spread of Measles in Spokane has been Conf...
Of	Outbreak of Zika in Panama City
Of	Urbana Encounters Severe Symptoms of Meningitis

Converts df to a string in which the row indices have been removed. This leads to more concise printed output.

Yes, our matches to “Of” are definitely erroneous. We can fix the error by ensuring that all matches are capitalized. However, the observed bug is a symptom of a much bigger issue: in all the wrongly matched headlines, we matched to “Of” but not to the actual city name. This occurred because we didn’t account for multiple matches in a headline. How frequently do headlines contain more than one match? Let’s find out. We’ll track the list of all matched cities in a headline using an additional `Cities` column.

Listing 12.8 Finding multicity headlines

```
def get_cities_in_headline(headline):
```

cities_in_headline = set()	Returns a list of all unique cities in a headline
for regex, name in city_to_name.items():	Makes sure the first letter of the city name is capitalized
match = regex.search(headline)	
if match:	Adds a Cities column to the table by using the apply method, which applies an inputted function to all elements of a column to create a brand new column
if headline[match.start()].isupper():	
cities_in_headline.add(name)	
return list(cities_in_headline)	

```
df['Cities'] = df['Headline'].apply(get_cities_in_headline)
```

df['Num_cities'] = df['Cities'].apply(len)	
df_multiple_cities = df[df.Num_cities > 1]	Filters out rows that do not contain multiple city matches

Adds a column counting the number of cities in a headline

```
num_rows, _ = df_multiple_cities.shape
print(f"{num_rows} headlines match multiple cities")
67 headlines match multiple cities
```

← The city count may increase with data updates to the GeoNamesCache library.

We find that 67 headlines contain more than one city, which represents approximately 10% of the data. Why do so many headlines match against multiple locations? Perhaps exploring some sample matches will yield an answer.

Listing 12.9 Sampling multicity headlines

```
ten_cities = df_multiple_cities[['Cities', 'Headline']].head(10)
print(ten_cities.to_string(index=False))

          Cities                  Headline
[0] [York, New York City] Could Zika Reach New York City?
[1] [Miami Beach, Miami]   First Case of Zika in Miami Beach
[2] [San Juan, San]       San Juan reports 1st U.S. Zika-related death
[3]         amid outbreak
[4] [Los Angeles, Los Ángeles] New Los Angeles Hairstyle goes Viral
[5] [Bay, Tampa]           Tampa Bay Area Zika Case Count Climbs
[6] [Ho Chi Minh City, Ho] Zika cases in Vietnam's Ho Chi Minh City
[7]         surge
[8] [San, San Diego]        Key Zika Findings in San Diego Institute
[9] [H?t, Kuala Lumpur]    Kuala Lumpur is Hit By Zika Threat
[10] [San, San Francisco]  Zika Virus Reaches San Francisco
[11] [Salvador, San, San Salvador] Zika worries in San Salvador
```

It appears that short, invalid city names are being matched to the headlines along with the longer, correct location names. For example, the city of 'San' is always returned along with more legitimate city names like 'San Francisco' and 'San Salvador'. How do we fix this error? One solution is to just return the longest city name whenever more than one matched city is found.

Listing 12.10 Selecting the longest city names

```
def get_longest_city(cities):
    if cities:
        return max(cities, key=len)
    return None

df['City'] = df['Cities'].apply(get_longest_city)
```

As a sanity check, we'll output rows that contain a short city name (four characters or fewer) to ensure that no erroneous short name is assigned to one of our headlines.

Listing 12.11 Printing the shortest city names

```
short_cities = df[df.City.str.len() <= 4][['City', 'Headline']]
print(short_cities.to_string(index=False))
```

City	Headline
Lima	Lima tries to address Zika Concerns
Pune	Pune woman diagnosed with Zika
Rome	Authorities are Worried about the Spread of Ma...
Molo	Molo Cholera Spread Causing Concern
Miri	Zika arrives in Miri
Nadi	More people in Nadi are infected with HIV ever...
Baud	Rumors about Tuberculosis Spreading in Baud ha...
Kobe	Chikungunya re-emerges in Kobe
Waco	More Zika patients reported in Waco
Erie	Erie County sets Zika traps
Kent	Kent is infested with Rabies
Reno	The Spread of Gonorrhea in Reno has been Conf...
Sibu	Zika symptoms spotted in Sibu
Baku	The Spread of Herpes in Baku has been Confirmed
Bonn	Contaminated Meat Brings Trouble for Bonn Farmers
Jaen	Zika Troubles come to Jaen
Yuma	Zika seminars in Yuma County
Lyon	Mad Cow Disease Detected in Lyon
Yiwu	Authorities are Worried about the Spread of He...
Suva	Suva authorities confirmed the spread of Rotav...

The results appear to be legitimate. Let's now shift our attention from cities to countries. Only 15 of the total headlines contain actual country information. The count is low enough for us to manually examine all of these headlines.

Listing 12.12 Fetching headlines with countries

```
df_countries = df[df.Country.notnull()][['City',           ←
                                         'Country',
                                         'Headline']]
```

```
print(df_countries.to_string(index=False))
```

The `df.Country.notnull()` method returns a list of Booleans. Each Boolean equals True only if a country is present in the associated row.

City	Country	Headline
Recife	Brazil	Mystery Virus Spreads in Recife, Brazil
Ho Chi Minh City	Vietnam	Zika cases in Vietnam's Ho Chi Minh City surge
Bangkok	Thailand	Thailand-Zika Virus in Bangkok
Piracicaba	Brazil	Zika outbreak in Piracicaba, Brazil
Klang	Malaysia	Zika surfaces in Klang, Malaysia
Guatemala City	Guatemala	Rumors about Meningitis spreading in Guatemala...
Belize City	Belize	Belize City under threat from Zika
Campinas	Brazil	Student sick in Campinas, Brazil
Mexico City	Mexico	Zika outbreak spreads to Mexico City
Kota Kinabalu	Malaysia	New Zika Case in Kota Kinabalu, Malaysia
Johor Bahru	Malaysia	Zika reaches Johor Bahru, Malaysia
Hong Kong	Hong Kong	Norovirus Exposure in Hong Kong
Panama City	Panama	Outbreak of Zika in Panama City
Singapore	Singapore	Zika cases in Singapore reach 393
Panama City	Panama	Panama City's first Zika related death

All of the country-bearing headlines also contain city information. Thus, we can assign a latitude and longitude without relying on the country's central coordinates. Consequently, we can disregard the country names from our analysis.

Listing 12.13 Dropping countries from the table

```
df.drop('Country', axis=1, inplace=True)
```

We are nearly ready to add latitudes and longitudes to our table. However, we first need to consider the rows where no locations were detected. Let's count the number of unmatched headlines and then print a subset of that data.

Listing 12.14 Exploring unmatched headlines

```
df_unmatched = df[df.City.isnull()]
num_unmatched = len(df_unmatched)
print(f"{num_unmatched} headlines contain no city matches.")
print(df_unmatched.head(10)[['Headline']].values)

39 headlines contain no city matches.
[['Louisiana Zika cases up to 26'],
 ['Zika infects pregnant woman in Cebu'],
 ['Spanish Flu Sighted in Antigua'],
 ['Zika case reported in Oton'],
 ['Hillsborough uses innovative trap against Zika 20 minutes ago'],
 ['Maka City Experiences Influenza Outbreak'],
 ['West Nile Virus Outbreak in Saint Johns'],
 ['Malaria Exposure in Sussex'],
 ['Greenwich Establishes Zika Task Force'],
 ['Will West Nile Virus vaccine help Parsons?']]
```

Approximately 6% of the headlines do not match any cities. Some of these headlines mention legitimate cities, which GeoNamesCache failed to identify. How should we treat the missing cities? Well, given their low frequency, perhaps we should delete the missing mentions. The price for those deletions is a slight reduction in data quality, but that loss will not significantly impact our results because our coverage of matched cities is quite high.

Listing 12.15 Dropping unmatched headlines

```
df = df[~df.City.isnull()][['City', 'Headline']]
```

The `~` symbol reverses the Booleans in the list returned by the `df.City.isnull()` method. Thus, each reversed Boolean equals True only if a city is present in the associated row.

12.2 Visualizing and clustering the extracted location data

All the rows in our table contain a city name. Now we can assign a latitude and longitude to each row. We utilize `get_cities_by_name` to return the coordinates of the most populated city bearing the extracted city name.

Listing 12.16 Assigning geographic coordinates to cities

```
latitudes, longitudes = [], []
for city_name in df.City.values:
    city = max(gc.get_cities_by_name(city_name),
               key=lambda x: list(x.values())[0]['population']) ← Chooses the
    city = list(city.values())[0] ← matched city
    latitudes.append(city['latitude']) ← with the largest
    longitudes.append(city['longitude']) ← population

df = df.assign(Latitude=latitudes, Longitude=longitudes) ← Adds Latitude and
                                                               Longitude columns
                                                               to our table
```

Extracts city latitudes and longitudes

With latitudes and longitudes assigned, we can attempt to cluster the data. Let's execute K-means across our set of 2D coordinates. We use the elbow method to choose a reasonable value for K (figure 12.1).

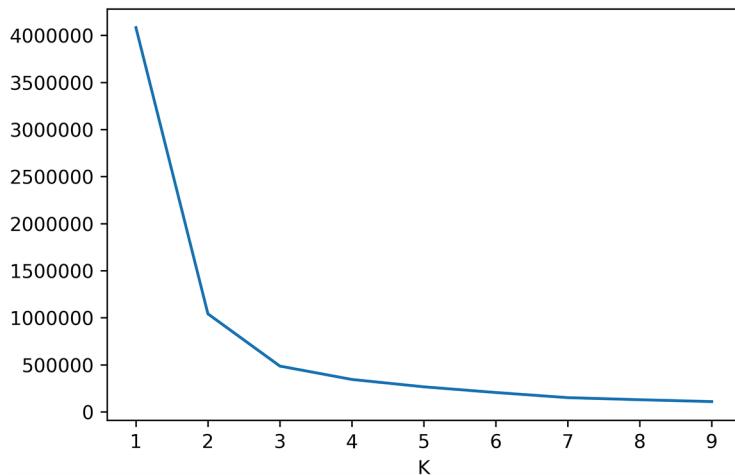


Figure 12.1 A geographic elbow curve points to a K of 3

Listing 12.17 Plotting a geographic elbow curve

```
coordinates = df[['Latitude', 'Longitude']].values
k_values = range(1, 10)
inertia_values = []
for k in k_values:
    inertia_values.append(KMeans(k).fit(coordinates).inertia_)
```

```
plt.plot(range(1, 10), inertia_values)
plt.xlabel('K')
plt.ylabel('Inertia')
plt.show()
```

The “elbow” in our elbow plot points to a K of 3. That K value is very low, limiting our scope to at most three different geographic territories. Still, we should maintain some faith in our analytic methodology. We cluster the locations into three groups and plot them on a map (figure 12.2).

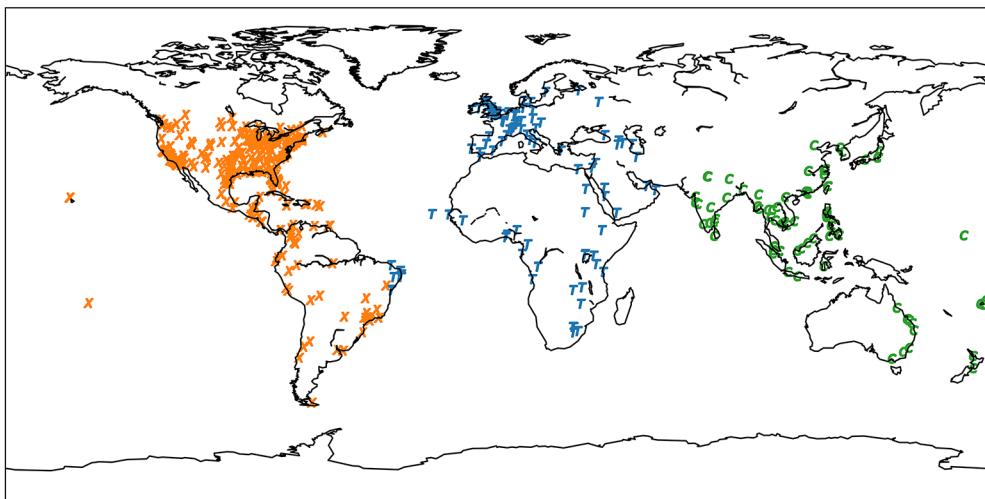


Figure 12.2 Mapped K-means city clusters. K is set to 3. The three clusters are spread thin across six continents.

Listing 12.18 Using K-means to cluster cities into three groups

```
def plot_clusters(clusters, longitudes, latitudes):
    plt.figure(figsize=(12, 10))
    ax = plt.axes(projection=PlateCarree())
    ax.coastlines()
    ax.scatter(longitudes, latitudes, c=clusters)
    ax.set_global()
    plt.show()

df['Cluster'] = KMeans(3).fit_predict(coordinates)
plot_clusters(df.Cluster, df.Longitude, df.Latitude)
```

This function will be reused to plot clusters throughout the rest of our analysis.

NOTE The marker shapes in figures 12.1 through 12.5 have been manually adjusted to discriminate among clusters in the black-and-white print version of the book.

The results look pretty ridiculous. Our three clusters cover

- North and South America
- Africa and Europe
- Asia and Australia

These continental categories are too broad to be useful. Furthermore, all South American cities on the eastern coast awkwardly cluster with African and European locations (despite the fact that an entire ocean lies between them). These clusters are not helpful for understanding the data. Perhaps our K was too low after all. Let's disregard our elbow analysis and double the size of K to 6 (figure 12.3).

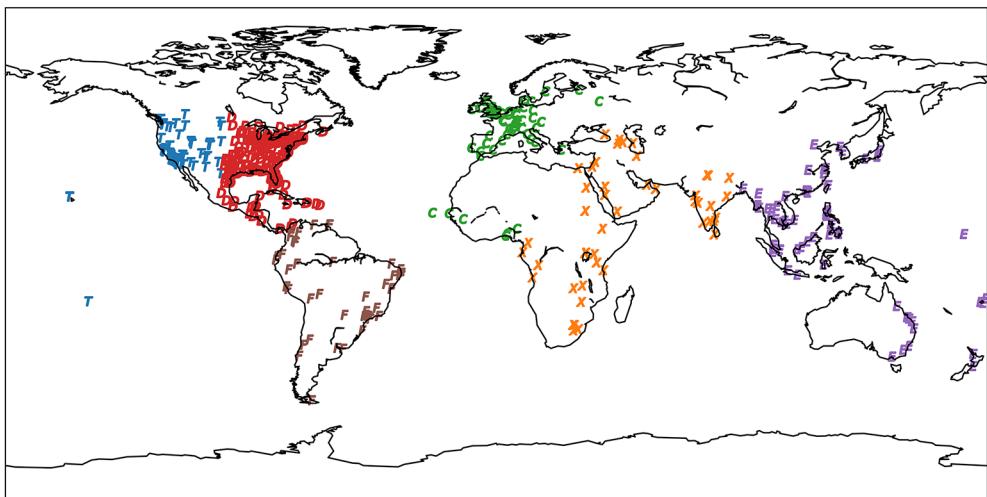


Figure 12.3 Mapped K-means city clusters. K is set to 6. Africa's clustered points are incorrectly split between the European and Asian continents.

Listing 12.19 Using K-means to cluster cities into six groups

```
df['Cluster'] = KMeans(6).fit_predict(coordinates)
plot_clusters(df.Cluster, df.Longitude, df.Latitude)
```

Increasing K improves clustering in North America and South America. South America now falls in its own separate cluster, and North America is split between two Western and Eastern cluster groups. However, on the other side of the Atlantic, the clustering quality remains low. Africa's geolocations are incorrectly split between Europe and Asia. K-mean's sense of centrality is unable to properly distinguish between Africa, Europe, and Asia. Perhaps the algorithm's reliance on Euclidean distance prevents it from capturing relationships between points distributed on our planet's curved surface.

As an alternate approach, we can attempt to execute DBSCAN clustering. The DBSCAN algorithm takes as input any distance metric of our choosing, allowing us to cluster on the great-circle distance between points. We start by coding a great-circle distance function whose inputs are a pair of NumPy arrays.

Listing 12.20 Defining a NumPy-based great-circle metric

```
def great_circle_distance(coord1, coord2, radius=3956):
    if np.array_equal(coord1, coord2):
        return 0.0

    coord1, coord2 = np.radians(coord1), np.radians(coord2)
    delta_x, delta_y = coord2 - coord1
    haversin = sin(delta_x / 2) ** 2 + np.product([
        cos(coord1[0]),
        cos(coord2[0]),
        sin(delta_y / 2) ** 2])
    return 2 * radius * asin(haversin ** 0.5)
```

radius is preset to the radius of the Earth in miles.

We've defined our distance metric and are nearly ready to run the DBSCAN algorithm. However, first we need to choose reasonable values for the `eps` and `min_samples` parameters. Let's assume the following: a global city cluster contains at least three cities that are on average no more than 250 miles apart. Based on these assumptions, we input values of 250 and 3 into `eps` and `min_samples`, respectively.

Listing 12.21 Using DBSCAN to cluster cities

```
metric = great_circle_distance
dbscan = DBSCAN(eps=250, min_samples=3, metric=metric)
df['Cluster'] = dbscan.fit_predict(coordinates)
```

DBSCAN assigns `-1` to outlier data points that do not cluster. Let's remove these outliers from our table and then plot the remaining results (figure 12.4).

Listing 12.22 Plotting non-outlier DBSCAN clusters

```
df_no_outliers = df[df.Cluster > -1]
plot_clusters(df_no_outliers.Cluster, df_no_outliers.Longitude,
              df_no_outliers.Latitude)
```

DBSCAN has done a decent job generating discrete clusters in parts of South America, Asia, and southern Africa. The eastern United States, however, falls into a single overly dense cluster. Why is this the case? It is partially due to a certain narrative bias in Western media, which means American events are more likely to get coverage. This leads to a denser spread of mentioned locations. One way to overcome the geographic bias is to recluster US cities using a more rigorous epsilon parameter. Such a strategy seems sensible in the context of our problem statement, which asks for separate top clusters from American and globally grouped headlines. So, we'll cluster US locations

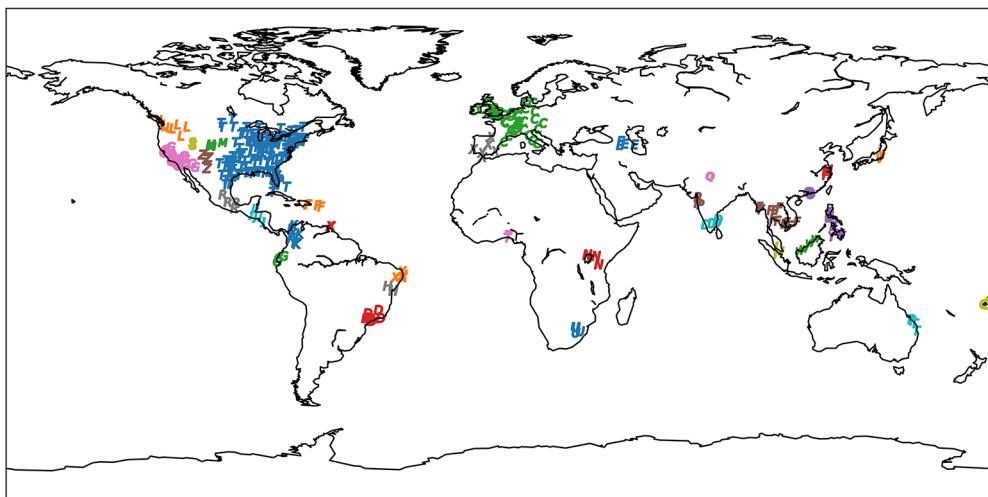


Figure 12.4 Mapped DBSCAN city clusters computed using the great-circle distance metric

independently from the rest of the world. To do so, we first assign country codes across each of our cities.

Listing 12.23 Assigning country codes to cities

```
def get_country_code(city_name):
    city = max(gc.get_cities_by_name(city_name),
               key=lambda x: list(x.values())[0]['population'])
    return list(city.values())[0]['countrycode']

df['Country code'] = df.City.apply(get_country_code)
```

The country codes allow us to separate the data into two distinct DataFrame objects. The first object, `df_us`, holds the US locations. The second object, `df_not_us`, holds all the remaining global cities.

Listing 12.24 Separating US and global cities

```
df_us = df[df.Country_code == 'US']
df_not_us = df[df.Country_code != 'US']
```

We've separated US and non-US cities. Now we need to recluster the coordinates in the two separated tables. Reclustering `df_not_us` is unavoidable due to density changes caused by deleting all the US locations. However, we maintain `eps` of 250 while clustering that table. Meanwhile, we reduce `eps` for `df_us` by half (to 125) to acknowledge the tighter density of US locations. Finally, all outliers are deleted after we recluster.

Listing 12.25 Reclustering extracted cities

```
def re_cluster(input_df, eps):
    input_coord = input_df[['Latitude', 'Longitude']].values
    dbscan = DBSCAN(eps=eps, min_samples=3,
                    metric=great_circle_distance)
    clusters = dbscan.fit_predict(input_coord)
    input_df = input_df.assign(Cluster=clusters)
    return input_df[input_df.Cluster > -1]

df_not_us = re_cluster(df_not_us, 250)
df_us = re_cluster(df_us, 125)
```

12.3 Extracting insights from location clusters

Let's investigate the clustered data in the `df_not_us` table. We start by grouping the results using the Pandas `groupby` method.

Listing 12.26 Grouping cities by cluster

```
groups = df_not_us.groupby('Cluster')
num_groups = len(groups)
print(f"{num_groups} Non-US clusters have been detected")

31 Non-US clusters have been detected
```

31 global clusters have been detected. Let's sort these groups by size and count the headlines in the largest cluster.

Listing 12.27 Finding the largest cluster

```
sorted_groups = sorted(groups, key=lambda x: len(x[1]),
                      reverse=True)
group_id, largest_group = sorted_groups[0]
group_size = len(largest_group)
print(f"Largest cluster contains {group_size} headlines")

Largest cluster contains 51 headlines
```

The largest cluster contains 51 total headlines. Reading all these headlines individually will be a time-consuming process. We can save time by outputting just those headlines that represent the most central locations in the cluster. Centrality can be captured by calculating the average latitude and longitude of a group. Then we can compute the distance between every location and the average coordinates. Lower distances indicate higher centrality.

NOTE As we discussed in section 11, the average latitude and longitude merely approximate the center since they do not consider the curvature of the Earth.

Next, we define a `compute_centrality` function that assigns a `Distance_to_center` column to an inputted group.

Listing 12.28 Computing cluster centrality

```
def compute_centrality(group):
    group_coords = group[['Latitude', 'Longitude']].values
    center = group_coords.mean(axis=0)
    distance_to_center = [great_circle_distance(center, coord)
                           for coord in group_coords]
    group['Distance_to_center'] = distance_to_center
```

We can now sort all headlines by centrality. Let's print the five most central headlines in our largest cluster.

Listing 12.29 Finding the central headlines in the largest cluster

```
def sort_by_centrality(group):
    compute_centrality(group)
    return group.sort_values(by=['Distance_to_center'], ascending=True)

largest_group = sort_by_centrality(largest_group)
for headline in largest_group.Headline.values[:5]:
    print(headline)

Mad Cow Disease Disastrous to Brussels
Scientists in Paris to look for answers
More Livestock in Fontainebleau are infected with Mad Cow Disease
Mad Cow Disease Hits Rotterdam
Contaminated Meat Brings Trouble for Bonn Farmers
```

The central headlines in `largest_group` focus on an outbreak of mad cow disease in various European cities. We can confirm that the cluster's locale is centered in Europe by outputting the top countries associated with cities in the cluster.

Listing 12.30 Finding the top three countries in the largest cluster

```
from collections import Counter
def top_countries(group):
    countries = [gc.get_countries() [country_code] ['name']
                 for country_code in group.Country_code.values]
    return Counter(countries).most_common(3) ←
print(top_countries(largest_group))
```

The Counter class tracks the most-repeated elements in a list, along with their counts.

```
[('United Kingdom', 19), ('France', 7), ('Germany', 6)]
```

The most frequently mentioned cities in `largest_group` are located in the United Kingdom, France, and Germany. The majority of locations in `largest_group` are definitely in Europe.

Let's repeat this analysis across the four next-largest global clusters. The following code helps determine whether any other disease epidemics are currently threatening the globe.

Listing 12.31 Summarizing content in the largest clusters

```

for _, group in sorted_groups[1:5]:
    sorted_group = sort_by_centrality(group)
    print(top_countries(sorted_group))
    for headline in sorted_group.Headline.values[:5]:
        print(headline)
    print('\n')

[('Philippines', 16)]
Zika afflicts patient in Calamba
Hepatitis E re-emerges in Santa Rosa
More Zika patients reported in Indang
Batangas Tourism Takes a Hit as Virus Spreads
Spreading Zika reaches Bacoor

[('El Salvador', 3), ('Honduras', 2), ('Nicaragua', 2)]
Zika arrives in Tegucigalpa
Santa Barbara tests new cure for Hepatitis C
Zika Reported in Ilopango
More Zika cases in Soyapango
Zika worries in San Salvador

[('Thailand', 5), ('Cambodia', 3), ('Vietnam', 2)]
More Zika patients reported in Chanthaburi
Thailand-Zika Virus in Bangkok
Zika case reported in Phetchabun
Zika arrives in Udon Thani
More Zika patients reported in Kampong Speu

[('Canada', 10)]
Rumors about Pneumonia spreading in Ottawa have been refuted
More people in Toronto are infected with Hepatitis E every year
St. Catharines Patient in Critical Condition after Contracting Dengue
Varicella has Arrived in Milton
Rabies Exposure in Hamilton

```

Oh no! Zika is spreading through the Philippines! There are also Zika outbreaks in Southeast Asia and in Central America. The Canadian cluster, however, contains a mix of random disease headlines, which implies that no dominant outbreak is occurring in that northern territory.

Let's turn our attention to the US clusters. We start by visualizing the clusters on a map of the United States (figure 12.5).

Listing 12.32 Plotting US DBSCAN clusters

```

plt.figure(figsize=(12, 10))
ax = plt.axes(projection=LambertConformal())
ax.set_extent(us_extent)

```

```

ax.scatter(df_us.Longitude, df_us.Latitude, c=df_us.Cluster,
           transform=PlateCarree())
ax.coastlines()
ax.add_feature(cartopy.feature.STATES)
plt.show()

```

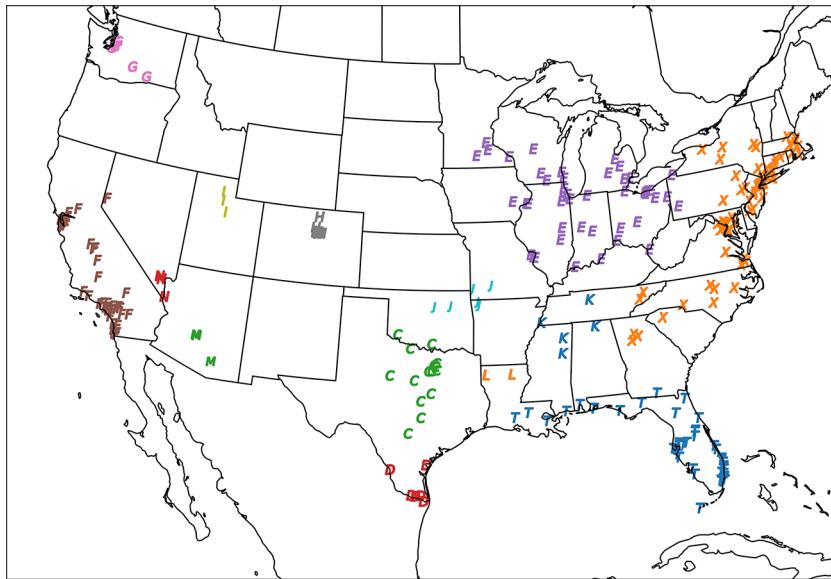


Figure 12.5 Mapped DBSCAN location clusters within the boundaries of the United States

The visualized map yields reasonable outputs. The eastern states no longer fall into a single dense cluster. We'll analyze the top five US clusters by printing their centrality-sorted headlines.

Listing 12.33 Summarizing content within the largest US clusters

```

us_groups = df_us.groupby('Cluster')
us_sorted_groups = sorted(us_groups, key=lambda x: len(x[1]),
                           reverse=True)
for _, group in us_sorted_groups[:5]:
    sorted_group = sort_by_centrality(group)
    for headline in sorted_group.Headline.values[:5]:
        print(headline)
    print('\n')

```

Schools in Bridgeton Closed Due to Mumps Outbreak
 Philadelphia experts track pandemic
 Vineland authorities confirmed the spread of Chlamydia
 Baltimore plans for Zika virus
 Will Swine Flu vaccine help Annapolis?

Bradenton Experiences Zika Troubles
 Tampa Bay Area Zika Case Count Climbs
 Zika Strikes St. Petersburg
 New Zika Case Confirmed in Sarasota County
 Zika spreads to Plant City

Rhinovirus Hits Bakersfield
 Schools in Tulare Closed Due to Mumps Outbreak
 New medicine wipes out West Nile Virus in Ventura
 Hollywood Outbreak Film Premieres
 Zika symptoms spotted in Hollywood

How to Avoid Hepatitis E in South Bend
 Hepatitis E Hits Hammond
 Chicago's First Zika Case Confirmed
 Rumors about Hepatitis C spreading in Darien have been refuted
 Rumors about Rotavirus Spreading in Joliet have been Refuted

More Zika patients reported in Fort Worth
 Outbreak of Zika in Stephenville
 Zika symptoms spotted in Arlington
 Dallas man comes down with case of Zika
 Zika spreads to Lewisville

The Zika epidemic has hit both Florida and Texas! This is very troubling. However, no discernible disease patterns are present in the other top clusters. Currently, the spreading Zika outbreak is confined to the southern United States. We will immediately report this to our superiors so they can take appropriate action. As we prepare to present our findings, let's plot one final image, which will appear on the front page of our report (figure 12.6). This image summarizes the menacing scope of the spreading Zika epidemic: it displays all US and global clusters where Zika is mentioned in more than 50% of article headlines.

Listing 12.34 Plotting Zika clusters

```
def count_zika_mentions(headlines):           ← Counts the number of
    zika_regex = re.compile(r'\bzika\b',       ← times Zika is mentioned
                           flags=re.IGNORECASE)   in a list of headlines
    zika_count = 0
    for headline in headlines:
        if zika_regex.search(headline):
            zika_count += 1
    return zika_count                         ← Regex that matches an
                                              instance of the word "Zika"
                                              in a headline. The match is
                                              case insensitive.

fig = plt.figure(figsize=(15, 15))
ax = plt.axes(projection=PlateCarree())
```

```

for _, group in sorted_groups + us_sorted_groups:
    headlines = group.Headline.values
    zika_count = count_zika_mentions(headlines)
    if float(zika_count) / len(headlines) > 0.5:
        ax.scatter(group.Longitude, group.Latitude)

ax.coastlines()
ax.set_global()
plt.show()

```

Iterates over both US and global clusters

Plots clusters where Zika is mentioned in more than 50% of article headlines

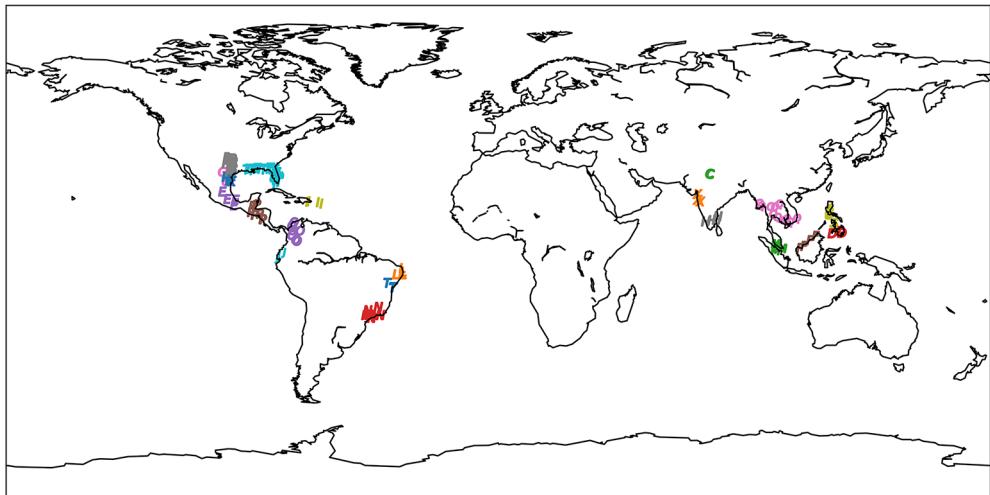


Figure 12.6 Mapped DBSCAN location clusters where Zika is mentioned in more than 50% of article headlines

We have successfully clustered our headlines by location and plotted those clusters where the word *Zika* is dominant. This relationship between our clusters and their textual content leads to an interesting question: is it possible to cluster the headlines based on text similarity rather than geographic distance? In other words, can we group our headlines by text overlap so that all the references to *Zika* automatically appear in a single cluster? Yes, we can! In the subsequent case study, we learn how to measure similarity between texts to group documents by topic.

Summary

- Data science tools can fail in unexpected ways. When we ran GeoNamesCache on our news headlines, the library incorrectly matched short city names (such as “Of” and “San”) to the inputted text. Through data exploration, we were able to account for these mistakes. If, instead, we had blindly clustered the locations, our final output would have been junky. We must diligently explore our data prior to serious analysis.

- Sometimes, problematic data points are present in an otherwise good dataset. In our case, less than 6% of headlines incorrectly lacked a city assignment. Correcting for these headlines would have been difficult. Instead, we chose to delete the headlines from the dataset. Occasionally, it's okay to delete problematic examples if their impact on the dataset is minimal. However, we should weigh the pros and cons of the deletion before making a final decision.
- The elbow method heuristically picks K for K-means clustering. Heuristic tools are not guaranteed to work correctly every time. In our analysis, an elbow plot returned a K of 3. Obviously, this value was too low. Thus, we intervened and attempted to choose a different K . If we had indiscriminately trusted the elbow output, our final clustering would have been worthless.
- Common sense should dictate our analysis of clustering outputs. Earlier, we examined a K-means output where K equaled 6. We observed the clustering of Central African and European cities. This result was clearly wrong—Europe and Central Africa are very different locations. So, we transitioned to a different clustering approach. When common sense dictates that the clustering is wrong, we should try an alternate approach.
- Sometimes it's acceptable to break a dataset into parts and analyze each part individually. In our initial DBSCAN analysis, the algorithm failed to correctly cluster US cities. Most eastern US cities fell into a single cluster. We could have abandoned our DBSCAN approach. Instead, we clustered the US cities separately, using more appropriate parameters. Analyzing the dataset in two separate parts led to better clustering results.

Case study 4

Using online job postings to improve your data science resume

Problem statement

We're ready to expand our data science career. Six months from now, we'll apply for a new job. In preparation, we begin to draft our resume. The early draft is rough and incomplete. It doesn't yet cover our career goals or education. Nonetheless, the resume covers the first four case studies in this book, including this one, which we'll complete before seeking new employment.

Our resume draft is far from perfect. It's possible that certain vital data science skills are not yet represented. If so, what are those missing skills? We decide to find out analytically. After all, we are data scientists! We fill in gaps in knowledge using rigorous analysis, so why shouldn't we apply that rigorous analysis to ourselves?

First we need some data. We go online and visit a popular job-search site. The website offers millions of searchable job listings, posted by understaffed employers. A built-in search engine allows us to filter the jobs by keyword, such as *analyst* or *data scientist*. Additionally, the search engine can match jobs to uploaded documents. This feature is intended to search postings based on resume content. Unfortunately, our resume is still a work in progress. So instead, we search

on the table of contents of this book! We copy and paste the first 15 listed sections of the table of contents into a text file.

Next, we upload the file to the job-search site. Material from the first four case studies is compared against millions of job listings, and thousands of job postings are returned. Some of these postings may be more relevant than others; we can't vouch for the search engine's overall quality, but the data is appreciated. We download the HTML from every posting.

Our goal is to extract common data science skills from the downloaded data. We'll then compare these skills to our resume to determine which skills are missing. To reach our goal, we'll proceed like this:

- 1 Parse out all the text from the downloaded HTML files.
- 2 Explore the parsed output to learn how job skills are commonly described in online postings. Perhaps specific HTML tags are more commonly used to underscore job skills.
- 3 Try to filter out any irrelevant job postings from our dataset. The search engine isn't perfect. Perhaps some irrelevant postings were erroneously downloaded. We can evaluate relevance by comparing the postings with our resume and the table of contents.
- 4 Cluster the job skills within the relevant postings, and visualize the clusters.
- 5 Compare the clustered skills to our resume content. We'll then make plans to update our resume with any missing data science skills.

Dataset description

Our rough draft of the resume is stored in the file resume.txt. The full text of that draft is as follows:

Experience

1. Developed probability simulations using NumPy
2. Assessed online ad clicks for statistical significance using permutation testing
3. Analyzed disease outbreaks using common clustering algorithms

Additional Skills

1. Data visualization using Matplotlib
2. Statistical analysis using SciPy
3. Processing structured tables using Pandas
4. Executing K-means clustering and DBSCAN clustering using scikit-learn
5. Extracting locations from text using GeoNamesCache
6. Location analysis and visualization using GeoNamesCache and Cartopy
7. Dimensionality reduction with PCA and SVD using scikit-learn
8. NLP analysis and text topic detection using scikit-learn

We'll learn skills 7 and 8 in the subsequent sections of this case study.

Our preliminary draft is short and incomplete. To compensate for any missing material, we also use the partial table of contents of this book, which is stored in the file `table_of_contents.txt`. It covers the first 15 sections of the book, as well as all the top-level subsection headers. The table of contents file has been utilized to search for thousands of relevant job postings that were downloaded and stored in a `job_postings` directory. Each file in the directory is an HTML file associated with an individual posting. These files can be viewed locally in your web browser.

Overview

To address the problem at hand, we need to know how to do the following:

- Measure similarity between texts
- Efficiently cluster large text datasets
- Visually display multiple text clusters
- Parse HTML files for text content

13

Measuring text similarities

This section covers

- What is natural language processing?
- Comparing texts based on word overlap
- Comparing texts using one-dimensional arrays called vectors
- Comparing texts using two-dimensional arrays called matrices
- Efficient matrix computation using NumPy

Rapid text analysis can save lives. Let's consider a real-world incident when US soldiers stormed a terrorist compound. In the compound, they discovered a computer containing terabytes of archived data. The data included documents, text messages, and emails pertaining to terrorist activities. The documents were too numerous to be read by any single human being. Fortunately, the soldiers were equipped with special software that could perform very fast text analysis. The software allowed the soldiers to process all of the text data without even having to leave the compound. The onsite analysis immediately revealed an active terrorist plot in a nearby neighborhood. The soldiers instantly responded to the plot and prevented a terrorist attack.

This swift defensive response would not have been possible without *natural language processing* (NLP) techniques. NLP is a branch of data science that focuses on speedy text analysis. Typically, NLP is applied to very large text datasets. NLP use cases are numerous and diverse and include the following:

- Corporate monitoring of social media posts to measure the public's sentiment toward a company's brand
- Analyzing transcribed call center conversations to monitor common customer complaints
- Matching people on dating sites based on written descriptions of shared interests
- Processing written doctors' notes to ensure proper patient diagnosis

These use cases depend on fast analysis. Delayed signal extraction could be costly. Unfortunately, the direct handling of text is an inherently slow process. Most computational techniques are optimized for numbers, not text. Consequently, NLP methods depend on a conversion from pure text to a numeric representation. Once all words and sentences have been replaced with numbers, the data can be analyzed very rapidly.

In this section, we focus on a basic NLP problem: measuring the similarity between two texts. We will quickly discover a feasible solution that is not computationally efficient. We will then explore a series of numerical techniques for rapidly computing text similarities. These computations will require us to transform our input texts into 2D numeric tables for full efficiency.

13.1 Simple text comparison

Many NLP tasks depend on the analysis of similarities and differences between texts. Suppose we want to compare three simple texts:

- `text1`—*She sells seashells by the seashore.*
- `text2`—“*Seashells! The seashells are on sale! By the seashore.*”
- `text3`—*She sells 3 seashells to John, who lives by the lake.*

Our goal is to determine whether `text1` is more similar to `text2` or to `text3`. We start by assigning the texts to three variables.

Listing 13.1 Assigning texts to variables

```
text1 = 'She sells seashells by the seashore.'
text2 = '"Seashells! The seashells are on sale! By the seashore."'
text3 = 'She sells 3 seashells to John, who lives by the lake.'
```

Now we need to quantify the differences between texts. One basic approach is to simply count the words shared between each pair of texts. This requires us to split each text into a list of words. Text splitting in Python can be carried out using the built-in string `split` method.

NOTE The process of splitting text into individual words is commonly called *tokenization*.

Listing 13.2 Splitting texts into words

```
words_lists = [text.split() for text in [text1, text2, text3]]
words1, words2, words3 = words_lists

for i, words in enumerate(words_lists, 1):
    print(f"Words in text {i}:")
    print(f"{words}\n")

Words in text 1
['She', 'sells', 'seashells', 'by', 'the', 'seashore.']

Words in text 2
['"Seashells!', 'The', 'seashells', 'are', 'on', 'sale!', 'By', 'the',
 'seashore."']

Words in text 3
['She', 'sells', '3', 'seashells', 'to', 'John,', 'who', 'lives', 'by',
 'the', 'lake.']}
```

Even though we've split the texts, an accurate word comparison is not immediately possible for the following reasons:

- *Capitalization inconsistency*—The words *she* and *seashells* are capitalized in some texts but not others, making a direct comparison difficult.
- *Punctuation inconsistency*—For example, an exclamation point and a quotation mark are attached to *seashells* in *text2* but not in the other texts.

We can eliminate the capitalization inconsistency by calling the built-in `lower` string method, which converts a string to lowercase. Furthermore, we can strip out punctuation from a word string by calling `word.replace(punctuation, '')`, where `punctuation` is set to `'!'` or `"!"`. Let's use these built-in string methods to eliminate all inconsistencies. We define a `simplify_text` function, which converts text to lowercase and removes all common punctuation.

Listing 13.3 Removing case sensitivity and punctuation

```
def simplify_text(text):
    for punctuation in ['.', ',', '!', '?', '"']:
        text = text.replace(punctuation, '')
    return text.lower()                                     ←

for i, words in enumerate(words_lists, 1):
    for j, word in enumerate(words):
        words[j] = simplify_text(word)

    print(f"Words in text {i}:")
    print(f"{words}\n")
```

Strips out common punctuation from a string and converts the string to lowercase

As a reminder, our immediate goal is to

- 1 Count all unique words in `text1` that are also present in `text2`.
- 2 Count all unique words in `text1` that are also present in `text3`.
- 3 Use the counts to determine whether `text2` or `text3` is more similar to `text1`.

Currently, we're just interested in comparing unique words. Therefore, duplicate words (like `seashore`, which appears twice in `text2`) will only be counted once. Thus, we can eliminate all duplicate words by converting each word list into a set.

Listing 13.4 Converting word lists to sets

```
words_sets = [set(words) for words in words_lists]
for i, unique_words in enumerate(words_sets, 1):
    print(f"Unique Words in text {i}")
    print(f"{unique_words}\n")

Unique Words in text 1
{'sells', 'seashells', 'by', 'seashore', 'the', 'she'}

Unique Words in text 2
{'by', 'on', 'are', 'sale', 'seashore', 'the', 'seashells'}

Unique Words in text 3
{'to', 'sells', 'seashells', 'lake', 'by', 'lives', 'the', 'john', '3',
 'who', 'she'}
```

Given two Python sets `set_a` and `set_b`, we can extract all overlapping elements by running `set_a & set_b`. Let's use the `&` operator to count overlapping words between text pairs (`text1`, `text2`) and (`text1`, `text3`).

NOTE Formally, the set of overlapping elements is called the *intersection* of two sets.

Listing 13.5 Extracting overlapping words between two texts

```
words_set1 = words_sets[0]
for i, words_set in enumerate(words_sets[1:], 2):
    shared_words = words_set1 & words_set
    print(f"Texts 1 and {i} share these {len(shared_words)} words:")
    print(f"{shared_words}\n")

Texts 1 and 2 share these 4 words:
{'seashore', 'by', 'the', 'seashells'}

Texts 1 and 3 share these 5 words:
{'sells', 'by', 'she', 'the', 'seashells'}
```

Texts 1 and 2 share four words, while texts 1 and 3 share five words. Does this mean `text1` is more similar to `text3` than to `text2`? Not necessarily. While texts 1 and 3

share five overlapping words, they also contain diverging words that appear in one text but not the other. Let's count all the diverging words between text pairs (`text1`, `text2`) and (`text1`, `text3`). We use the `^` operator to extract diverging elements between each pair of word sets.

Listing 13.6 Extracting diverging words between two texts

```
for i, words_set in enumerate(words_sets[1:], 2):
    diverging_words = words_set1 ^ words_set
    print(f"Texts 1 and {i} don't share these {len(diverging_words)} words:")
    print(f"{diverging_words}\n")

Texts 1 and 2 don't share these 5 words:
{'are', 'sells', 'sale', 'on', 'she'}

Texts 1 and 3 don't share these 7 words:
{'to', 'lake', 'lives', 'seashore', 'john', '3', 'who'}
```

Texts 1 and 3 contain two more diverging words than texts 1 and 2. Thus, texts 1 and 3 show significant word overlap and also significant divergence. To combine their overlap and divergence into a single similarity score, we must first combine all overlapping and diverging words between the texts. This aggregation, which is called a *union*, will contain all the unique words across the two texts. Given two Python sets `set_a` and `set_b`, we can compute their union by running `set_a | set_b`.

The differences between word divergence, intersection, and union are illustrated in figure 13.1. Here, the unique words in texts 1 and 2 are displayed in three rectangular boxes. The leftmost box and the rightmost box represent the diverging words between texts 1 and 2, respectively. Meanwhile, the middle box contains all the shared words at the intersection of texts 1 and 2. Together, the three boxes represent the union of all words in the two texts.

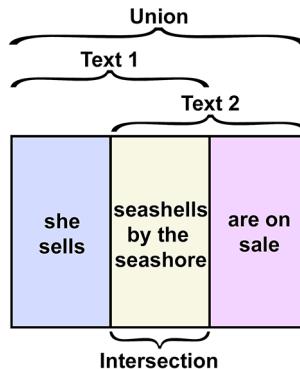


Figure 13.1 A visualized representation of the union, intersection, and divergence between two texts.

Common Python set operations

- `set_a & set_b`—Returns all overlapping elements between `set_a` and `set_b`
- `set_a ^ set_b`—Returns all diverging elements between `set_a` and `set_b`
- `set_a | set_b`—Returns the union of all elements between `set_a` and `set_b`
- `set_a - set_b`—Returns all elements in `set_a` that are not in `set_b`

Let's utilize the `|` operator to count the total unique words across text pairs (`text1`, `text2`) and (`text1`, `text3`).

Listing 13.7 Extracting the union of words between two texts

```
for i, words_set in enumerate(words_sets[1:], 2):
    total_words = words_set1 | words_set
    print(f"Together, texts {i} contain {len(total_words)} "
          f"unique words. These words are:\n {total_words}\n")
```

Together, texts 1 and 2 contain 9 unique words. These words are:
{'sells', 'seashells', 'by', 'on', 'are', 'sale', 'seashore', 'the', 'she'}

Together, texts 1 and 3 contain 12 unique words. These words are:
{'sells', 'lake', 'by', 'john', 'the', 'she', 'to', 'lives', 'seashore',
 '3', 'who', 'seashells'}

Together, `text1` and `text3` contain 12 unique words. Five of these words overlap, and seven diverge. Accordingly, both overlap and divergence represent complementary percentages of the total unique word count across texts. Let's output these percentages for text pairs (`text1`, `text2`) and (`text1`, `text3`).

Listing 13.8 Extracting the percentage of shared words between two texts

```
for i, words_set in enumerate(words_sets[1:], 2):
    shared_words = words_set1 & words_set
    diverging_words = words_set1 ^ words_set
    total_words = words_set1 | words_set
    assert len(total_words) == len(shared_words) + len(diverging_words)
    percent_shared = 100 * len(shared_words) / len(total_words)
    percent_diverging = 100 * len(diverging_words) / len(total_words)
```

Percent of total words
shared with text 1

```
    print(f"Together, texts {i} contain {len(total_words)} "
          f"unique words. \n{percent_shared:.2f}% of these words are "
          f"shared. \n{percent_diverging:.2f}% of these words diverge.\n")
```

Percent of
total words
that diverge
from text 1

Together, texts 1 and 2 contain 9 unique words.
44.44% of these words are shared.
55.56% of these words diverge.

Together, texts 1 and 3 contain 12 unique words.
41.67% of these words are shared.
58.33% of these words diverge.

Texts 1 and 3 share 41.67% of total words. The remaining 58.33% of words diverge. Meanwhile, texts 1 and 2 share 44.44% of total words. That percentage is higher, and thus we can infer that `text1` is more similar to `text2` than to `text3`.

We've essentially developed a simple metric for assessing similarities between texts. The metric works as follows:

- 1 Given two texts, extract a list of words from each text.
- 2 Count the unique words that are shared between the texts.
- 3 Divide the shared word count by the total unique words across both texts. Our output is a fraction of the total words shared between texts.

This similarity metric is referred to as the *Jaccard similarity*, or the *Jaccard index*.

The Jaccard similarity between texts 1 and 2 is illustrated in figure 13.2, where the texts are represented as two circles. The left circle corresponds to text 1, and the right circle corresponds to text 2. Each circle contains the words in its corresponding text. The two circles intersect, and their intersection contains all words that are shared between the texts. The Jaccard similarity equals the fraction of total words that are present in the intersection. Four of the nine words in the diagram appear in the intersection. Therefore, the Jaccard similarity is equal to 4 / 9.

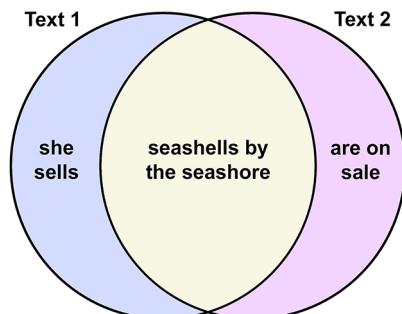


Figure 13.2 A visualized representation of the Jaccard similarity between two texts

13.1.1 Exploring the Jaccard similarity

The Jaccard similarity is a reasonable measure of text resemblance for the following reasons:

- The similarity takes into account both text overlap and text divergence.
- The fractional similarity is always between 0 and 1. The fraction is easy to interpret: 0 indicates that no words are shared, 0.5 indicates that half the words are shared, and 1 indicates that all the words are shared.
- The similarity is simple to implement.

Let's define a function to compute the Jaccard similarity.

Listing 13.9 Computing the Jaccard similarity

```
def jaccard_similarity(text_a, text_b):
    word_set_a, word_set_b = [set(simplify_text(text).split())
                                for text in [text_a, text_b]]
    num_shared = len(word_set_a & word_set_b)
    num_total = len(word_set_a | word_set_b)
    return num_shared / num_total

for text in [text2, text3]:
    similarity = jaccard_similarity(text1, text)
    print(f"The Jaccard similarity between '{text1}' and '{text}' "
          f>equals {similarity:.4f}.\n")
```

The Jaccard similarity between 'She sells seashells by the seashore.' and
 "Seashells! The seashells are on sale! By the seashore." equals 0.4444.

The Jaccard similarity between 'She sells seashells by the seashore.' and
 'She sells 3 seashells to John, who lives by the lake.' equals 0.4167.

Our implementation of the Jaccard similarity is functional but not very efficient. The function executes two set-comparison operations: `word_set_a & word_set_b` and `word_set_a | word_set_b`. These operations compare and contrast all words between two sets. In Python, such comparisons are computationally costlier than streamlined numerical analysis.

How do we make the function more efficient? Well, we can start by eliminating the union computation `word_set_a | word_set_b`. We take the union in order to count the unique words between the sets, but there's a simpler way to obtain that count. Consider the following:

- 1 Adding `len(word_set_a)` and `len(word_set_b)` yields a word count where the shared words are counted twice.
- 2 Subtracting `len(word_set_a & word_set_b)` from that sum eliminates the double count. The final result equals `len(word_set_a | word_set_b)`.

We can replace the union computation with `len(word_set_a) + len(word_set_b) - num_shared`, thus making our function more efficient. Let's modify the function while ensuring that our Jaccard output remains the same.

Listing 13.10 Efficiently computing the Jaccard similarity

```
def jaccard_similarity_efficient(text_a, text_b):
    word_set_a, word_set_b = [set(simplify_text(text).split())
                                for text in [text_a, text_b]]
    num_shared = len(word_set_a & word_set_b)
    num_total = len(word_set_a) + len(word_set_b) - num_shared ←
    return num_shared / num_total

for text in [text2, text3]:
    similarity = jaccard_similarity_efficient(text1, text)
    assert similarity == jaccard_similarity(text1, text)
```

Unlike our previous `jaccard_similarity` function, here we compute `num_total` without executing any set-comparison operations.

We've improved our Jaccard function. Unfortunately, the function still won't scale: it might run efficiently on hundreds of sentences but not on thousands of multisentence documents. The inefficiency is caused by our remaining set comparison, `word_set_a & word_set_b`. The operation is too slow to execute across thousands of complicated texts. Perhaps we can speed up the computation by somehow running it using NumPy. However, NumPy is intended to process numbers, not words, so we cannot use the library unless we replace all words with numeric values.

13.1.2 Replacing words with numeric values

Can we swap out words for numbers? Yes! We simply need to iterate over all words in all texts and assign each unique *i*th word a value of *i*. The mapping between words and their numeric values can be stored in a Python dictionary. We'll refer to this dictionary as our *vocabulary*. Let's build a vocabulary that covers all the words in our three texts. We'll also create a complementary `value_to_word` dictionary, which maps the numeric values back to words.

NOTE Essentially, we're numbering all the words in the union of the texts. We iteratively choose a word and assign it a number, starting with zero. However, the order in which we choose the words is not important—we might as well reach blindly into a bag of words and pull the words out by random. That is why this technique is commonly referred to as the *bag-of-words* technique.

Listing 13.11 Assigning words to numbers in a vocabulary

```
words_set1, words_set2, words_set3 = words_sets
total_words = words_set1 | words_set2 | words_set3
vocabulary = {word : i for i, word in enumerate(total_words)}
value_to_word = {value: word for word, value in vocabulary.items()}
print(f"Our vocabulary contains {len(vocabulary)} words. "
      f"This vocabulary is:\n{vocabulary}")
```

```
Our vocabulary contains 15 words. This vocabulary is:
{'sells': 0, 'seashells': 1, 'to': 2, 'lake': 3, 'who': 4, 'by': 5,
 'on': 6, 'lives': 7, 'are': 8, 'sale': 9, 'seashore': 10, 'john': 11,
 '3': 12, 'the': 13, 'she': 14}
```

NOTE The order of the words in the `total_words` variable in listing 13.11 may vary based on the installed version of Python. That order change will slightly alter certain figures used to display the texts later in this section. Setting `total_words` to equal `['sells', 'seashells', 'to', 'lake', 'who', 'by', 'on', 'lives', 'are', 'sale', 'seashore', 'john', '3', 'the', 'she']` will ensure consistency in the outputs.

Given our vocabulary, we can convert any text into a one-dimensional array of numbers. Mathematically, a 1D numeric array is called a *vector*. Hence, the process of converting text into a vector is called *text vectorization*.

NOTE Array dimensionality is different from data dimensionality. A data point has d dimensions if d coordinates are required to spatially represent that point. Meanwhile, an array has d dimensions if d values are required to describe the array's shape. Imagine that we have recorded five data points, each with three coordinates. Our data is three-dimensional because it can be plotted in 3D space. Additionally, we can store the data in a table containing five rows and three columns. That table has a two-element shape of $(5, 3)$ and is therefore two-dimensional. Thus, we store our 3D data in a 2D array.

The simplest way to vectorize text is to create a vector of binary elements. Each index of that vector corresponds to a word in the vocabulary. Hence, the vector size equals the vocabulary size, even if some vocabulary words are missing from the associated text. If the word at index i is missing from the text, the i th vector element is set to 0. Otherwise, it is set to 1. Consequently, each vocabulary index in the vector maps to either 0 or 1.

For example, in `vocabulary`, the word *john* maps to a value of 11. Also, the word *john* is not present in `text1`. Thus, the vectorized representation of `text1` has a 0 at index 11. Meanwhile, the word *john* is present in `text3`. Consequently, the vectorized representation of `text3` has a 1 at index 11 (figure 13.3). In this manner, we can convert any text into a binary vector of 0s and 1s.

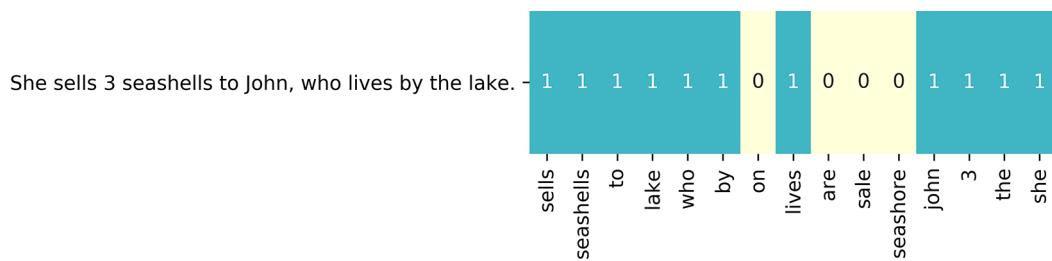


Figure 13.3 `Text3` is converted into a binary vector. Each index in the vector corresponds to a word in the vocabulary. For example, index 0 corresponds to *sells*. This word is present in our text, so the first element of the vector is set to 1. Meanwhile, the words *on*, *are*, *sale*, and *seashore* are not present in the text. Their corresponding elements are thus set to 0 in the vector.

Let's use binary vectorization to convert all texts into NumPy arrays. We'll store the computed vectors in a 2D `vectors` list, which can be treated like a table. The table's rows will map to texts, and its columns will map to the vocabulary. Figure 13.4 visualizes the table as a heatmap using techniques discussed in section 8.

NOTE As discussed in section 8, heatmaps are best visualized using the `Seaborn` library.

Listing 13.12 Transforming words into binary vectors

```

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

vectors = []
for i, words_set in enumerate(words_sets, 1):
    vector = np.array([0] * len(vocabulary))
    for word in words_set:
        vector[vocabulary[word]] = 1
    vectors.append(vector)

sns.heatmap(vectors, annot=True, cmap='YlGnBu',
            xticklabels=vocabulary.keys(),
            yticklabels=['Text 1', 'Text 2', 'Text 3'])
plt.yticks(rotation=0)
plt.show()

```

Generates an array of 0s. We can also generate this array by running `np.zeros(len(vocabulary))`.

As of Python 3.6, the dictionary keys method returns dictionary keys based on their order of insertion. In `vocabulary`, the order of insertion is equivalent to the word index.

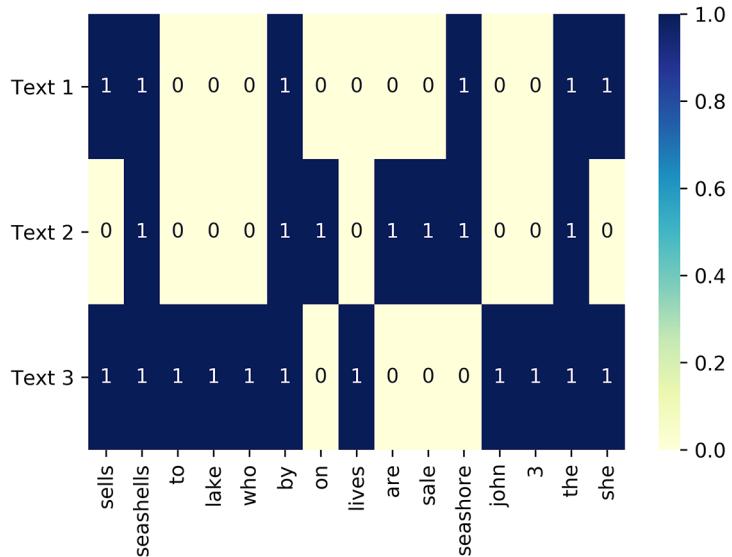


Figure 13.4 A table of vectorized texts. Rows correspond to labeled texts. Columns correspond to labeled words. Binary table elements are either 0 or 1. A nonzero value indicates the presence of a specified word in the specified text. Glancing at the table, we can immediately tell which words are shared across which texts.

Using our table, we can easily tell which words are shared between which texts. Take, for example, the word *sells*, which is tracked in the first column of the table. In that column, *sells* is assigned a 1 in the first and third rows of the table. These rows correspond to `text1` and `text3`. Hence, we know that *sells* is shared between `text1` and `text3`. More formally, the word is shared between the texts because `vectors[0][0] == 1`

and `vectors[2][0] == 1`. Furthermore, since both elements equal 1, their product must also equal 1. Consequently, the texts share a word in column `i` if the product of `vectors[0][i]` and `vectors[2][i]` is equal to 1.

Our binary vector representation allows us to extract shared words numerically. Suppose we wish to know whether the word in column `i` is present both in `text1` and `text2`. If the associated vectors are labeled `vector1` and `vector2`, then the word is present in both texts if `vector1[i] * vector2[i] == 1`. Here, we use pairwise vector multiplication to find all words shared by `text1` and `text2`.

Listing 13.13 Finding shared words using vector arithmetic

```
vector1, vector2 = vectors[:2]
for i in range(len(vocabulary)):
    if vector1[i] * vector2[i]:
        shared_word = value_to_word[i]
        print(f"'{shared_word}' is present in both texts 1 and 2")

'seashells' is present in both texts 1 and 2
'by' is present in both texts 1 and 2
'seashore' is present in both texts 1 and 2
'the' is present in both texts 1 and 2
```

We've outputted all four words shared between `text1` and `text2`. That shared word count is equal to the sum of every nonzero instance of `vector1[i] * vector2[i]`. Meanwhile, the sum of every zero instance equals 0. Therefore, we can compute the shared word count merely by summing the pairwise product of `vector1[i]` and `vector2[i]` across every possible `i`. In other words, `sum(vector1[i] * vector2[i] for i in range(len(vocabulary)))` equals `len(words_set1 & words_set2)`.

Listing 13.14 Counting shared words using vector arithmetic

```
shared_word_count = sum(vector1[i] * vector2[i]
                       for i in range(len(vocabulary)))
assert shared_word_count == len(words_set1 & words_set2)
```

The sum of the pairwise products across all vector indices is called the *dot product*. Given two NumPy arrays `vector_a` and `vector_b`, we can compute their dot product by running `vector_a.dot(vector_b)`. We can also compute the dot product using the `@` operator by running `vector_a @ vector_b`. In our example, that dot product equals the number of shared words between texts 1 and 2, which of course also equals their intersection size. Thus, running `vector1 @ vector2` produces a value that is equal to `len(words_set1 & words_set2)`.

Listing 13.15 Computing a vector dot product using NumPy

```
assert vector1.dot(vector2) == shared_word_count
assert vector1 @ vector2 == shared_word_count
```

The dot product of `vector1` and `vector2` equals the shared word count between `text1` and `text2`. Suppose that, instead, we take the dot product of `vector1` with itself. That output should equal the number of words that `text1` shares with `text1`. Stated more concisely, `vector1 @ vector1` should equal the number of unique words in `text1`, which is also equal to `len(words_set1)`. Let's confirm.

Listing 13.16 Counting total words using vector arithmetic

```
assert vector1 @ vector1 == len(words_set1)
assert vector2 @ vector2 == len(words_set2)
```

We are able to compute both shared word count and total unique word count using vector dot products. Essentially, we can compute the Jaccard similarity using only vector operations. This vectorized implementation of Jaccard is called the *Tanimoto similarity*.

Useful NumPy vector operations

- `vector_a.dot(vector_b)`—Returns the dot product between `vector_a` and `vector_b`. Equivalent to running `sum(vector_a[i] * vector_b[i] for i in range(vector_a.size))`.
- `vector_b @ vector_b`—Returns the dot product between `vector_a` and `vector_b` using the `@` operator.
- `binary_text_vector_a @ binary_text_vector_b`—Returns the number of shared words between `text_a` and `text_b`.
- `binary_text_vector_a @ binary_text_vector_a`—Returns the number of unique words in `text_a`.

Let's define a `tanimoto_similarity` function. The function takes as input two vectors, `vector_a` and `vector_b`. Its output is equal to `jaccard_similarity(text_a, text_b)`.

Listing 13.17 Computing text similarity using vector arithmetic

```
def tanimoto_similarity(vector_a, vector_b):
    num_shared = vector_a @ vector_b
    num_total = vector_a @ vector_a + vector_b @ vector_b - num_shared
    return num_shared / num_total

for i, text in enumerate([text2, text3], 1):
    similarity = tanimoto_similarity(vector1, vectors[i])
    assert similarity == jaccard_similarity(text1, text)
```

Our `tanimoto_similarity` function was intended to compare binary vectors. What would happen if we inputted two arrays with values other than 0 or 1? Technically, the function should return a similarity, but would that similarity make sense? For instance, vectors [5, 3] and [5, 2] are nearly identical. We expect their similarity

to be nearly equal to 1. Let's test our expectations by inputting the vectors into `tanimoto_similarity`.

Listing 13.18 Computing the similarity of non-binary vectors

```
non_binary_vector1 = np.array([5, 3])
non_binary_vector2 = np.array([5, 2])
similarity = tanimoto_similarity(non_binary_vector1, non_binary_vector2)
print(f"The similarity of 2 non-binary vectors is {similarity}")
```

The similarity of 2 non-binary vectors is 0.96875

The outputted value is nearly equal to 1. Thus, `tanimoto_similarity` has successfully measured the similarity between two nearly identical vectors. The function can analyze non-binary inputs. This means we can use non-binary techniques to vectorize our texts before comparing their contents.

There are benefits to vectorizing texts in a non-binary way. Let's discuss these benefits in more detail.

13.2 Vectorizing texts using word counts

Binary vectorization captures the presence and absence of words in a text, but it doesn't capture word counts. This is unfortunate since word counts can provide a differentiating signal between texts. For example, suppose we're contrasting two texts: A and B. Text A mentions *Duck* 61 times and *Goose* twice. Text B mentions *Goose* 71 times and *Duck* only once. Based on the counts, we can infer that the two texts are rather different relative to the discussion of ducks and geese. That difference is not captured by binary vectorization, which assigns a 1 to the *Duck* index and *Goose* index of both texts. What if we replace all binary values with actual word counts? For instance, we can assign values of 61 and 2 to the *Duck* and *Goose* indices of vector A, while assigning 1 and 71 to the corresponding indices of vector B.

These assignments will produce vectors of word counts. A vector of word counts is commonly referred to as a *term-frequency vector*, or a *TF vector* for short. Let's compute the TF vectors of A and B using a two-element vocabulary `{'duck': 0, 'goose': 1}`. As a reminder, each word in the vocabulary maps to a vector index. Given the vocabulary, we can convert the texts into TF vectors `[61, 2]` and `[1, 71]`. Then we print the Tanimoto similarity of the two vectors.

Listing 13.19 Computing TF vector similarity

```
similarity = tanimoto_similarity(np.array([61, 2]), np.array([1, 71]))
print(f"The similarity between texts is approximately {similarity:.3f}")
```

The similarity between texts is approximately 0.024

The TF vector similarity between the texts is very low. Let's compare it to the binary-vector similarity of the two texts. Each text has a binary-vector representation of `[1, 1]`, and thus the binary similarity should equal 1.

Listing 13.20 Assessing identical vector similarity

```
assert tanimoto_similarity(np.array([1, 1]), np.array([1, 1])) == 1
```

Replacing binary values with word counts can greatly impact our similarity output. What will happen if we vectorize `text1`, `text2`, and `text3` based on their word counts? Let's find out. We start by computing TF vectors for each of the three texts using the word lists stored in `words_lists`. These vectors are visualized in figure 13.5 using a heatmap.

Listing 13.21 Computing TF vectors from word lists

```
tf_vectors = []
for i, words_list in enumerate(words_lists, 1):
    tf_vector = np.array([0] * len(vocabulary))
    for word in words_list:
        word_index = vocabulary[word]
        tf_vector[word_index] += 1      ← Updates the word
                                         count using the word's
                                         vocabulary index
    tf_vectors.append(tf_vector)

sns.heatmap(tf_vectors, cmap='YlGnBu', annot=True,
            xticklabels=vocabulary.keys(),
            yticklabels=['Text 1', 'Text 2', 'Text 3'])
plt.yticks(rotation=0)
plt.show()
```

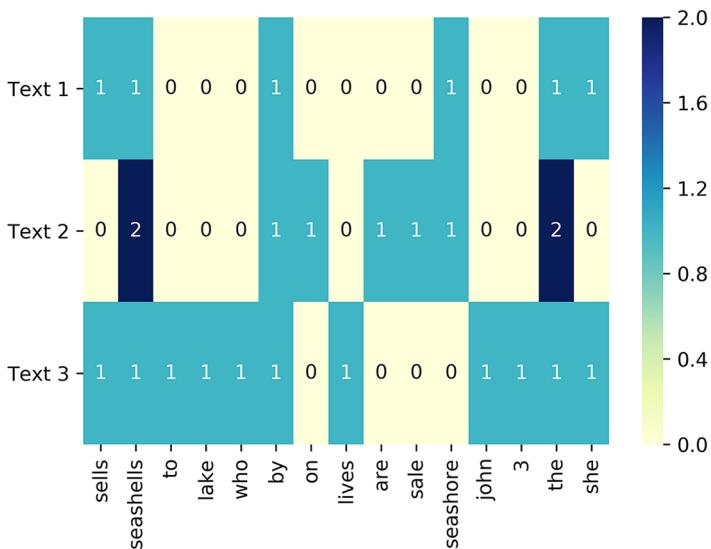


Figure 13.5 A table of TF vectors. Rows correspond to labeled texts. Columns correspond to labeled words. Each value indicates the count of a specified word in the specified text. Two words in the table are mentioned twice; all other words are mentioned no more than once.

The TF vectors of texts 1 and 3 are identical to previously seen binary vector outputs. However, the TF vector of text 2 is no longer binary since two words are mentioned more than once. How will this affect the similarity between `text1` and `text2`? Let's find out. The following code computes the TF vector similarity between `text1` and the other two texts. It also outputs the original binary vector similarity for comparison. Based on our observations, the similarity between `text1` and `text2` should shift, while the similarity between `text1` and `text3` should remain the same.

Listing 13.22 Comparing metrics of vector similarity

```
tf_vector1 = tf_vectors[0]
binary_vector1 = vectors[0]

for i, tf_vector in enumerate(tf_vectors[1:], 2):
    similarity = tanimoto_similarity(tf_vector1, tf_vector)
    old_similarity = tanimoto_similarity(binary_vector1, vectors[i - 1])
    print(f"The recomputed Tanimoto similarity between texts 1 and {i} is"
          f" {similarity:.4f}.")
    print(f"Previously, that similarity equaled {old_similarity:.4f} " "\n")

The recomputed Tanimoto similarity between texts 1 and 2 is 0.4615.
Previously, that similarity equaled 0.4444

The recomputed Tanimoto similarity between texts 1 and 3 is 0.4167.
Previously, that similarity equaled 0.4167
```

As expected, the similarity between `text1` and `text3` has stayed the same, while the similarity between `text1` and `text2` has increased. Thus, TF vectorization has made the affinity of the two texts more pronounced.

TF vectors yield improved comparisons because they're sensitive to count differences between texts. This sensitivity is useful. However, it can also be detrimental when comparing texts of different lengths. In the following subsection, we examine a flaw associated with TF vector comparison. Then we apply a technique called *normalization* to eliminate this flaw.

13.2.1 Using normalization to improve TF vector similarity

Imagine that you are testing a very simple search engine. The search engine takes a query and compares it to document titles stored in a database. The query's TF vector is compared to every vectorized title. Titles with a nonzero Tanimoto similarity are returned and ranked based on their similarity score.

Suppose you run a query for “Pepperoni Pizza” and the following two titles are returned:

- *Title A*—“Pepperoni Pizza! Pepperoni Pizza! Pepperoni Pizza!”
- *Title B*—“Pepperoni”

NOTE These titles are purposefully oversimplified for easier visualization. Most real document titles are more complicated.

Which of our two titles best matches the query? Most data scientists would agree that title A is a better match than title B. Both title A and the query mention *pepperoni pizza*. Meanwhile, title B mentions only *pepperoni*. There is no indication that the associated document actually discusses pizza in any context.

Let's check whether title A ranks higher than title B relative to the query. We start by constructing TF vectors from a two-element vocabulary {pepperoni: 0, pizza: 1}.

Listing 13.23 Simple search engine vectorization

```
query_vector = np.array([1, 1])
title_a_vector = np.array([3, 3])
title_b_vector = np.array([1, 0])
```

We now compare the query to the titles and sort the titles based on the Tanimoto similarity.

Listing 13.24 Ranking titles by query similarity

```
titles = ["A: Pepperoni Pizza! Pepperoni Pizza! Pepperoni Pizza!",
          "B: Pepperoni"]
title_vectors = [title_a_vector, title_b_vector]
similarities = [tanimoto_similarity(query_vector, title_vector)
               for title_vector in title_vectors]

for index in sorted(range(len(titles)), key=lambda i: similarities[i],
                     reverse=True):
    title = titles[index]
    similarity = similarities[index]
    print(f"'{title}' has a query similarity of {similarity:.4f}")

'B: Pepperoni' has a query similarity of 0.5000
'A: Pepperoni Pizza! Pepperoni Pizza! Pepperoni Pizza!' has a query
similarity of 0.4286
```

Unfortunately, title A outranks title B. This discrepancy in rankings is caused by text size. Title A has three times as many words as the query, while title B and the query differ by just a single word. Superficially, this difference can be used to distinguish texts by size. However, in our search engine, the size signal leads to faulty rankings. We need to subdue the influence of text size on ranked results. One naive approach is to just divide title_a_vector by 3. The division yields an output that is equal to query_vector. Therefore, running tanimoto_similarity(query_vector, title_a_vector / 3) should return a similarity of 1.

Listing 13.25 Eliminating size differences through division

```
assert np.array_equal(query_vector, title_a_vector / 3)
assert tanimoto_similarity(query_vector,
                           title_a_vector / 3) == 1
```

Using simple division, we can manipulate `title_a_vector` to equal `query_vector`. Such manipulation is not possible for `title_b_vector`. Why is this the case? To illustrate the answer, we need to plot all three vectors in 2D space.

How can we visualize our vectors? Well, from a mathematical standpoint, all vectors are geometric objects. Mathematicians treat every vector v as a line stretching from the origin to the numerical coordinates in v . Essentially, our three vectors are merely 2D line segments rising from the origin. We can visualize the segments in a 2D plot where the x-axis represents mentions of *pepperoni* and the y-axis represents mentions of *pizza* (figure 13.6).

Listing 13.26 Plotting TF vectors in 2D space

```
plt.plot([0, query_vector[0]], [0, query_vector[1]], c='k',
         linewidth=3, label='Query Vector')
plt.plot([0, title_a_vector[0]], [0, title_a_vector[1]], c='b',
         linestyle='--', label='Title A Vector')
plt.plot([0, title_b_vector[0]], [0, title_b_vector[1]], c='g',
         linewidth=2, linestyle='-.', label='Title B Vector')
plt.xlabel('Pepperoni')
plt.ylabel('Pizza')
plt.legend()
plt.show()
```

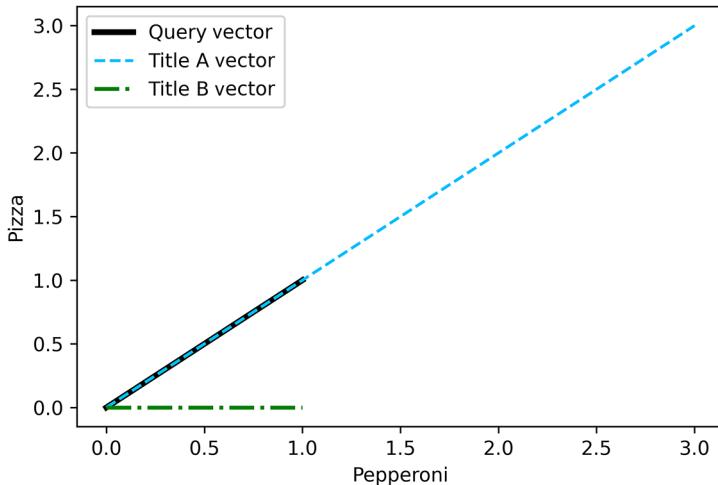


Figure 13.6 Three TF vectors have been plotted as lines in 2D space. Each vector stretches from the origin to its two-dimensional coordinates. The query vector and the title A vector both face in the same direction. The angle between these vectors is zero. However, one of the lines is three times as long as the other. Adjusting the segment lengths will force the two vectors to be identical.

In our plot, `title_a_vector` and `query_vector` point in the same direction. The only difference between the two lines is that `title_a_vector` is three times as long. Shrinking

`title_a_vector` will force the two lines to be identical. Meanwhile, `title_b_vector` and `query_vector` point in different directions. We cannot make these vectors overlap. Shrinking or lengthening `title_b_vector` will not yield alignment with the other two line segments.

We've gained some insight by representing our vectors as line segments. These segments have geometric lengths. Hence, every vector has a geometric length, which is called the *magnitude*. The magnitude is also called the *Euclidean norm* or the *L2 norm*. All vectors have a magnitude, even those that can't be plotted in two dimensions. For instance, in figure 13.7, we illustrate the magnitude of a 3D vector associated with *Pepperoni Pizza Pie*.

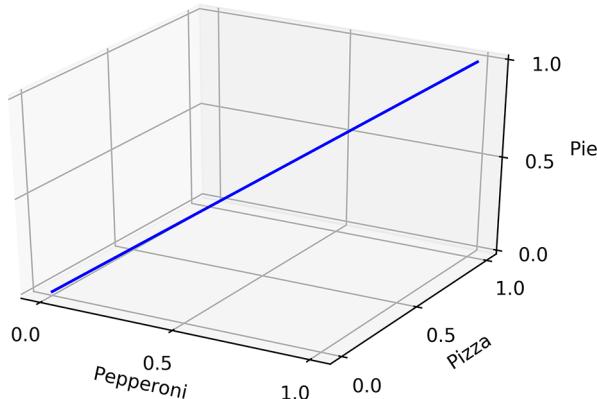


Figure 13.7 The plotted TF vector representation of the three-word title *Pepperoni Pizza Pie*. This 3D vector stretches from the origin to its coordinates of $(1, 1, 1)$. According to the Pythagorean theorem, the length of the plotted 3D segment is equal to $(1 + 1 + 1) ** 0.5$. That length is referred to as the *magnitude* of the vector.

Measuring the magnitude allows us to account for differences in geometric lengths. There are several ways to compute the magnitude in Python. Given vector v , we can measure the magnitude naively by measuring the Euclidean distance between v and the origin. We can also find the magnitude using NumPy by running `np.linalg.norm(v)`. Finally, we can compute the magnitude using the Pythagorean theorem (figure 13.8).

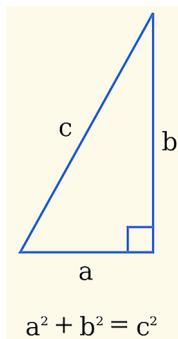


Figure 13.8 Using the Pythagorean theorem to compute the magnitude of a vector. Generally, a two-dimensional vector $[a, b]$ can be represented by a right triangle. The perpendicular segments of the triangle have lengths a and b . Meanwhile, the length of the triangle's hypotenuse is equal to c . According to the Pythagorean theorem, $c * c == a * a + b * b$. Hence, the vector's magnitude is equal to $\sum([value * value for value in vector]) ** 0.5$. This formula extends beyond 2D to any arbitrary number of dimensions.

According to the Pythagorean theorem, the squared distance of coordinate v to the origin is equal to `sum([value * value for value in v])`. This dovetails nicely with our earlier definition of the dot product. As a reminder, the dot product of two vectors v1 and v2 equals `sum([value1 * value2 for value1, value2 in zip(v1, v2)])`. Consequently, the dot product of v with itself equals `sum([value * value for value in v])`. Hence, the magnitude of v equals `(v @ v) ** 0.5`.

Let's output the magnitudes of our search engine vectors. Based on our observations, the magnitude of `title_a_vector` should equal three times the magnitude of `query_vector`.

Listing 13.27 Computing vector magnitude

```
from scipy.spatial.distance import euclidean
from numpy.linalg import norm

vector_names = ['Query Vector', 'Title A Vector', 'Title B Vector']
tf_search_vectors = [query_vector, title_a_vector, title_b_vector]
origin = np.array([0, 0])
for name, tf_vector in zip(vector_names, tf_search_vectors):
    magnitude = euclidean(tf_vector, origin)           ←
    assert magnitude == norm(tf_vector)                ←
    assert magnitude == (tf_vector @ tf_vector) ** 0.5   ←
    print(f"{name}'s magnitude is approximately {magnitude:.4f}")

magnitude_ratio = norm(title_a_vector) / norm(query_vector)
print(f"\nVector A is {magnitude_ratio:.0f}x as long as Query Vector")

Query Vector's magnitude is approximately 1.4142
Title A Vector's magnitude is approximately 4.2426
Title B Vector's magnitude is approximately 1.0000
Vector A is 3x as long as Query Vector
```

The magnitude equals the Euclidean distance between the vector and the origin.

We can also compute the magnitude using the dot product.

NumPy's norm function returns the magnitude.

As expected, there is a threefold difference between the magnitudes of `query_vector` and `title_a_vector`. Furthermore, the magnitudes of both vectors are greater than 1. Meanwhile, the magnitude of `title_vector_b` is equal to exactly 1. A vector with a magnitude of 1 is referred to as a *unit vector*. Unit vectors have many useful properties, which we discuss shortly. One benefit of unit vectors is that they are easy to compare: since unit vectors share an equal magnitude, it doesn't play a role in their similarity. Fundamentally, the difference between unit vectors is determined solely by direction.

Imagine if `title_a_vector` and `query_vector` both had a magnitude of 1. As a consequence, they'd share an equal length while also pointing in the same direction. In essence, the two vectors would be identical. The word count differences between our query and title A would no longer matter.

To illustrate this point, let's convert our TF vectors into unit vectors. Dividing any vector by its magnitude transforms that magnitude to 1. That division by the magnitude is called *normalization*, since the magnitude is also referenced as the L2 norm. Running `v / norm(v)` returns a *normalized vector* with a magnitude of 1.

We now normalize our vectors and generate a unit vector plot (figure 13.9). In the plot, two of the vectors should be identical.

Listing 13.28 Plotting normalized vectors

The two normalized unit vectors are now identical. We use `np.allclose` to confirm, rather than `np.array_equal`, to compensate for minuscule floating-point errors that may occur during normalization.

```
unit_query_vector = query_vector / norm(query_vector)
unit_title_a_vector = title_a_vector / norm(title_a_vector)
assert np.allclose(unit_query_vector, unit_title_a_vector) ←
unit_title_b_vector = title_b_vector ←

plt.plot([0, unit_query_vector[0]], [0, unit_query_vector[1]], c='k',
         linewidth=3, label='Normalized Query Vector')
plt.plot([0, unit_title_a_vector[0]], [0, unit_title_a_vector[1]], c='b',
         linestyle='--', label='Normalized Title A Vector')
plt.plot([0, unit_title_b_vector[0]], [0, unit_title_b_vector[1]], c='g',
         linewidth=2, linestyle='-.', label='Title B Vector')

plt.axis('equal')
plt.legend()
plt.show()
```

This vector is already a unit vector. There is no need to normalize.

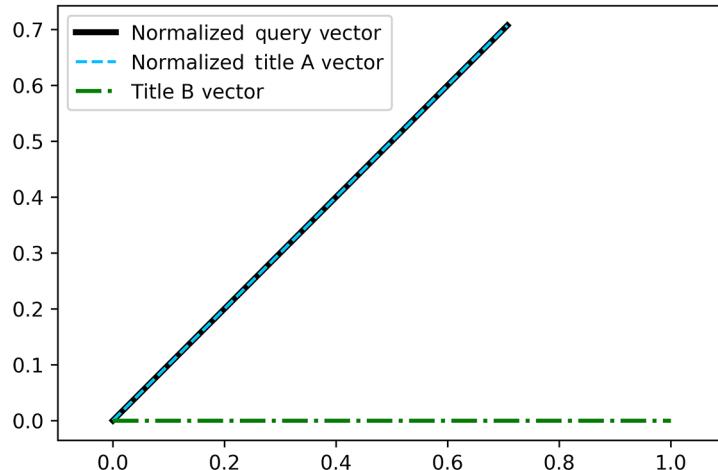


Figure 13.9 Our vectors have been normalized. All plotted vectors now have a magnitude of 1. The normalized query vector and normalized title A vector are identical in the plot.

The normalized query vector and the normalized title A vector are now indistinguishable. All differences arising from text size have been eliminated. Meanwhile, the location of the title B vector diverges from the query vector because the two segments point in different directions. If we rank our unit vectors based on their similarity to

`unit_query_vector`, then `unit_title_a_vector` outranks `unit_title_b_vector`. As a consequence, title A outranks title B relative to the query.

Listing 13.29 Ranking titles by unit vector similarity

```
unit_title_vectors = [unit_title_a_vector, unit_title_b_vector]
similarities = [tanimoto_similarity(unit_query_vector, unit_title_vector)
               for unit_title_vector in unit_title_vectors]

for index in sorted(range(len(titles)), key=lambda i: similarities[i],
                    reverse=True):
    title = titles[index]
    similarity = similarities[index]
    print(f'{title} has a normalized query similarity of {similarity:.4f}')

'A: Pepperoni Pizza! Pepperoni Pizza! Pepperoni Pizza!' has a normalized
query similarity of 1.0000
'B: Pepperoni' has a normalized query similarity of 0.5469
```

Common vector magnitude operations

- `euclidean(vector, vector.size * [0])`—Returns the vector's magnitude, which equals the Euclidean distance between `vector` and the origin
- `norm(vector)`—Returns the vector's magnitude using NumPy's `norm` function
- `(vector @ vector) ** 0.5`—Computes the vector's magnitude using the Pythagorean theorem
- `vector / norm(vector)`—Normalizes the vector so that its magnitude equals 1.0

Vector normalization has fixed a flaw in our search engine: the search engine is no longer overly sensitive to title length. In the process, we have inadvertently made our Tanimoto computation more efficient. Let's discuss why.

Suppose we measure the Tanimoto similarity of two unit vectors, u_1 and u_2 . Logically, we can infer the following:

- The Tanimoto similarity equals $u_1 @ u_2 / (u_1 @ u_1 + u_2 @ u_2 - u_1 @ u_2)$.
- $u_1 @ u_1$ equals $\text{norm}(u_1) ** 2$. Based on our previous discussions, we know that $u_1 @ u_1$ equals the squared magnitude of u .
- u_1 is a unit vector, so $\text{norm}(u_1)$ equals 1. Therefore, $\text{norm}(u_1) ** 2$ equals 1. Thus $u_1 @ u_1$ equals 1.
- By that same logic, $u_2 @ u_2$ also equals 1.
- Hence, the Tanimoto similarity reduces to $u_1 @ u_2 / (2 - u_1 @ u_2)$.

Taking the dot product of each vector with itself is no longer necessary. The only required vector computation is $u_1 @ u_2$.

Let's define a `normalized_tanimoto` function. The function takes as input two normalized vectors, u_1 and u_2 , and computes their Tanimoto similarity directly from $u_1 @ u_2$. That result equals `tanimoto_similarity(u1, u2)`.

Listing 13.30 Computing a unit vector Tanimoto similarity

```
def normalized_tanimoto(u1, u2):
    dot_product = u1 @ u2
    return dot_product / (2 - dot_product)

for unit_title_vector in unit_title_vectors[1:]:
    similarity = normalized_tanimoto(unit_query_vector, unit_title_vector)
    assert similarity == tanimoto_similarity(unit_query_vector,
                                              unit_title_vector)
```

The dot product of two unit vectors is a very special value. It can easily be converted into the angle between the vectors and also into the spatial distance between them. Why is this important? Well, common geometric metrics like vector angle and distance appear in all vector analysis libraries. Meanwhile, the Tanimoto similarity is used less frequently outside of NLP. It usually needs to be implemented from scratch, which can have serious real-world consequences. Imagine the following scenario. You're hired by a search engine company to improve all its pizza-related queries. You propose to use the normalized Tanimoto similarity as a metric of query relevance. However, your manager objects: they insist that, based on company policy, employees can only use relevance metrics that are already included in scikit-learn.

NOTE Sadly, this scenario is entirely realistic. Most organizations tend to validate their core metrics for both speed and quality. In large organizations, the validation process can take months. Thus, it's usually easier to rely on a prevalidated library than validate a brand-new metric.

The manager points you to the scikit-learn documentation that outlines the acceptable metric functions (<http://mng.bz/9aM1>). You see the scikit-learn metric names and functions displayed in the two-column table shown in figure 13.10. Multiple versions of a name can map to the same function. Four of the eight metrics refer to the Euclidean distance, and three refer to the Manhattan and Haversine (aka great circle) distances, which we introduced in section 11. Also, there's a reference to a metric called 'cosine', which we haven't yet discussed. There is no mention of a Tanimoto metric, so you can't use it to evaluate query relevance. What should you do?

The valid distance metrics, and the function they map to, are:

metric	Function
'cityblock'	metrics.pairwise.manhattan_distances
'cosine'	metrics.pairwise.cosine_distances
'euclidean'	metrics.pairwise.euclidean_distances
'haversine'	metrics.pairwise.haversine_distances
'l1'	metrics.pairwise.manhattan_distances
'l2'	metrics.pairwise.euclidean_distances
'manhattan'	metrics.pairwise.manhattan_distances
'nan_euclidean'	metrics.pairwise.nan_euclidean_distances

Figure 13.10 A screenshot of the scikit-learn document for valid distance metric implementations

Fortunately, math gives you a way out. If your vectors are normalized, their Tanimoto similarity can be substituted with the Euclidean and cosine metrics. This is because all three measures are very closely related to the normalized dot product. Let's examine why this is the case.

13.2.2 Using unit vector dot products to convert between relevance metrics

The unit vector dot product unites multiple types of comparison metrics. We've just seen how `tanimoto_similarity(u1, u2)` is a direct function of $u1 @ u2$. As it turns out, the Euclidean distance between unit vectors is also a function of $u1 @ u2$. It's not difficult to prove that `euclidean(u1, u2)` equals $(2 - 2 * u1 @ u2)^{0.5}$. Additionally, the angle between linear unit vectors is likewise dependent on $u1 @ u2$. These relationships are illustrated in figure 13.11.

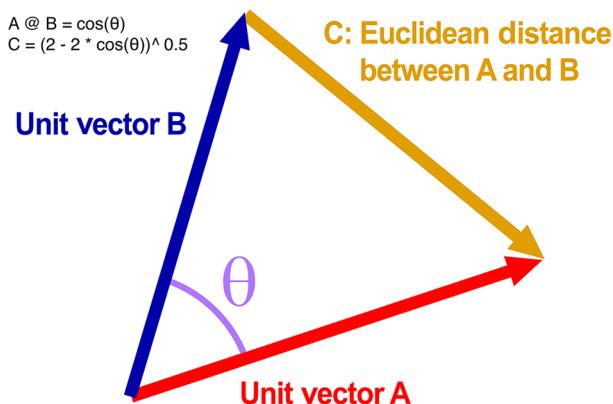


Figure 13.11 Two unit vectors, A and B. The angle between the vectors equals θ . The dot product of the vectors equals $\cos(\theta)$. C represents the Euclidean distance between the vectors, which equals $(2 - 2 * \cos(\theta))^{0.5}$.

Geometrically, the dot product of two unit vectors equals the cosine of the angle between them. Due to its equivalence with the cosine, the dot product of two unit vectors is commonly referred to as the *cosine similarity*. Given the cosine similarity `cs`, we can convert it to either Euclidean distance or the Tanimoto similarity by running $(2 - 2 * cs)^{0.5}$ or `cs / (2 - cs)`, respectively.

NOTE The cosine is a very important function in trigonometry. It maps an angle between lines to a value ranging from -1 to 1 . If two lines point in an identical direction, then the angle between them is 0 , and the cosine of the angle equals 1 . If two lines point in opposite directions, then the angle between them is 180 degrees, and the cosine of that angle equals -1 . Given a pair of vectors `v1` and `v2`, we can compute their cosine similarity by running

$(v1 / \text{norm}(v1)) @ (v2 / \text{norm}(v2))$. Then we can input that result into the inverse cosine function `np.arccos` to measure the angle between the vectors.

Listing 13.31 illustrates how easy it is to convert between the Tanimoto similarity, the cosine similarity, and the Euclidean distance. We compute the Tanimoto similarity between the query vector and each of our unit title vectors. The Tanimoto similarity is subsequently converted into the cosine similarity, and then the cosine similarity is converted into the Euclidean distance.

NOTE Additionally, we utilize the cosine similarity to compute the angle between the vectors. We do this to emphasize how the cosine metric reflects the angle between line segments.

Listing 13.31 Converting between unit vector metrics

```
unit_vector_names = ['Normalized Title A vector', 'Title B Vector']
u1 = unit_query_vector

for unit_vector_name, u2 in zip(unit_vector_names, unit_title_vectors):
    similarity = normalized_tanimoto(u1, u2)
    cosine_similarity = 2 * similarity / (1 + similarity) ←
    assert cosine_similarity == u1 @ u2
    angle = np.arccos(cosine_similarity)
    euclidean_distance = (2 - 2 * cosine_similarity) ** 0.5
    assert round(euclidean_distance, 10) == round(euclidean(u1, u2), 10)
    measurements = {'Tanimoto similarity': similarity,
                    'cosine similarity': cosine_similarity,
                    'Euclidean distance': euclidean_distance,
                    'angle': np.degrees(angle)} ←
    normalized_tanimoto
    is a function of
    cosine_similarity.
    Using basic algebra,
    we can invert the
    function to solve for
    cosine_similarity.

    print("We are comparing Normalized Query Vector and "
          f"{unit_vector_name}")
    for measurement_type, value in measurements.items():
        output = f"The {measurement_type} between vectors is {value:.4f}"
        if measurement_type == 'angle':
            output += ' degrees\n'

    print(output)
```

We are comparing Normalized Query Vector and Normalized Title A vector
The Tanimoto similarity between vectors is 1.0000
The cosine similarity between vectors is 1.0000
The Euclidean distance between vectors is 0.0000
The angle between vectors is 0.0000 degrees

We are comparing Normalized Query Vector and Title B Vector
The Tanimoto similarity between vectors is 0.5469
The cosine similarity between vectors is 0.7071
The Euclidean distance between vectors is 0.7654
The angle between vectors is 45.0000 degrees

The Tanimoto similarity between normalized vectors can be transformed into other metrics of similarity or distance. This is useful for the following reasons:

- Swapping the Tanimoto similarity for Euclidean distance allows us to carry out K-means clustering on text data. We discuss K-means clustering of texts in section 15.
- Swapping the Tanimoto similarity for cosine similarity simplifies our computational requirements. All our computations are reduced to basic dot product operations.

NOTE NLP practitioners commonly use the cosine similarity instead of the Tanimoto similarity. Research shows that in the long term, the Tanimoto similarity is more accurate than the cosine similarity. However, in many practical applications, the two similarities are interchangeable.

Common unit vector comparison metrics

- $u_1 @ u_2$ —The cosine of the angle between unit vectors u_1 and u_2
- $(u_1 @ u_2) / (2 - u_1 @ u_2)$ —The Tanimoto similarity between unit vectors u_1 and u_2
- $(2 - 2 * u_1 @ u_2) ** 0.5$ —The Euclidean distance between unit vectors u_1 and u_2

Vector normalization allows us to swap between multiple comparison metrics. Other benefits of normalization include these:

- *Elimination of text length as a differentiating signal*—This lets us compare long and short texts with similar contents.
- *More efficient Tanimoto similarity computation*—Only a single dot product operation is required.
- *More efficient computation of the similarity between every pair of vectors*—This is called the *all-by-all* similarity.

The last benefit has not yet been discussed. However, we will shortly learn that a table of cross-text similarities can be elegantly computed using *matrix multiplication*. In mathematics, matrix multiplication generalizes the dot product from one-dimensional vectors to two-dimensional arrays. The generalized dot product leads to the efficient computation of similarities across all pairs of texts.

13.3 Matrix multiplication for efficient similarity calculation

When analyzing our *seashell*-centric texts, we compared each text pair individually. What if, instead, we visualized all pairwise similarities in a table? The rows and columns would correspond to individual texts, while the elements would correspond to Tanimoto similarities. The table would provide us with a bird's-eye view of all the relationships between texts. We would finally learn whether `text2` is more similar to `text1` or `text3`.

Let's generate a table of normalized Tanimoto similarities, using the process outlined in figure 13.12. We start by normalizing the TF vectors in our previously pre-computed `tf_vectors` list. Then we iterate over every pair of vectors and compute their Tanimoto similarity. We store the similarities in a 2D `similarities` array, where `similarities[i][j]` equals the similarities between the *i*th text and the *j*th text. Finally, we visualize the `similarities` array using a heatmap (figure 13.13).

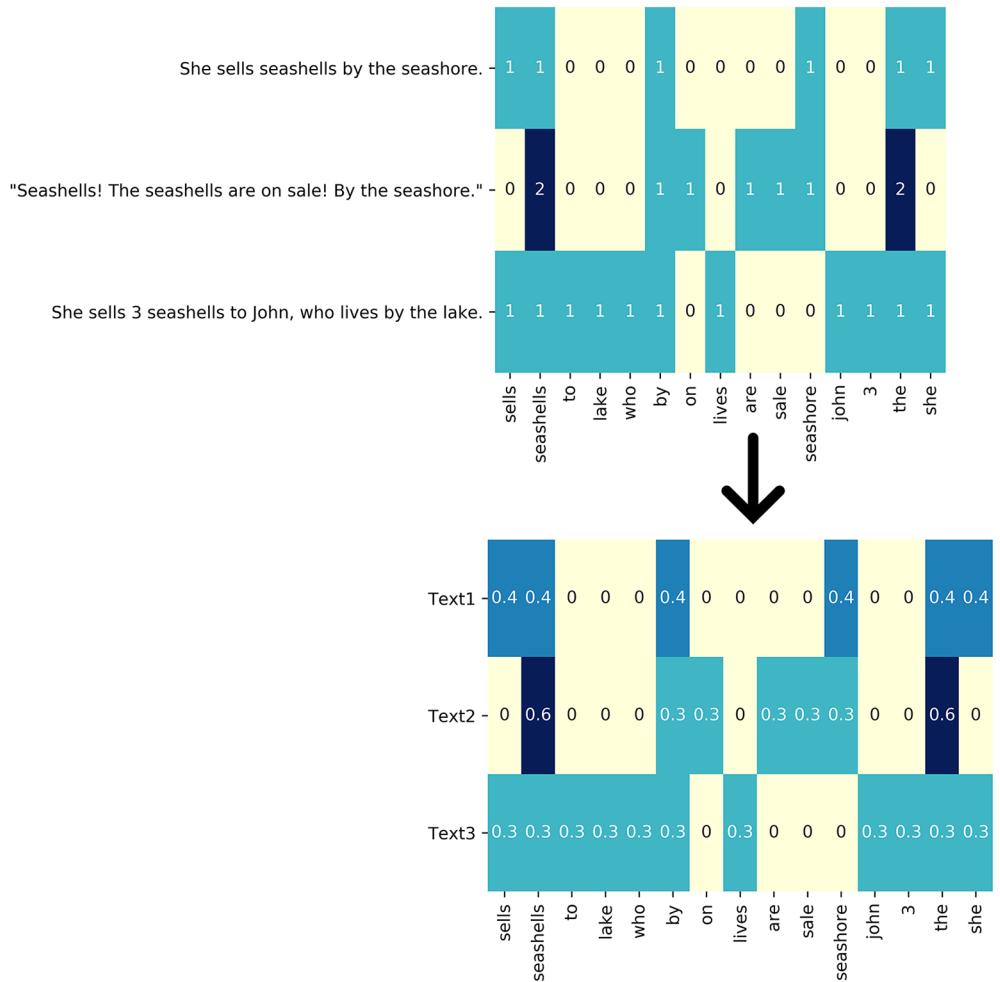


Figure 13.12 Transforming our three texts into a normalized matrix. The initial texts appear in the upper-left corner. These texts share a vocabulary of 15 unique words. We use the vocabulary to transform the texts into a matrix of word counts, which is in the upper-right corner. Its three rows correspond to the three texts, and its 15 columns track the occurrence count of every word in each text. We normalize these counts by dividing each row by its magnitude. The normalization produces the matrix in the lower-right corner. The dot product between any two rows in the normalized matrix equals the cosine similarity between the corresponding texts. Subsequently, running $\cos / (2 - \cos)$ transforms the cosine similarity into the Tanimoto similarity.

Listing 13.32 Computing a table of normalized Tanimoto similarities

```

num_texts = len(tf_vectors)
similarities = np.array([[0.0] * num_texts for _ in range(num_texts)])
unit_vectors = np.array([vector / norm(vector) for vector in tf_vectors])
for i, vector_a in enumerate(unit_vectors):
    for j, vector_b in enumerate(unit_vectors):
        similarities[i][j] = normalized_tanimoto(vector_a, vector_b)

labels = ['Text 1', 'Text 2', 'Text 3']
sns.heatmap(similarities, cmap='YlGnBu', annot=True,
            xticklabels=labels, yticklabels=labels)
plt.yticks(rotation=0)
plt.show()

```

Creates a 2D array containing just zeros. We can more efficiently create this array by running `np.zeros((num_texts, num_texts))`. We fill this empty array with similarities between texts.

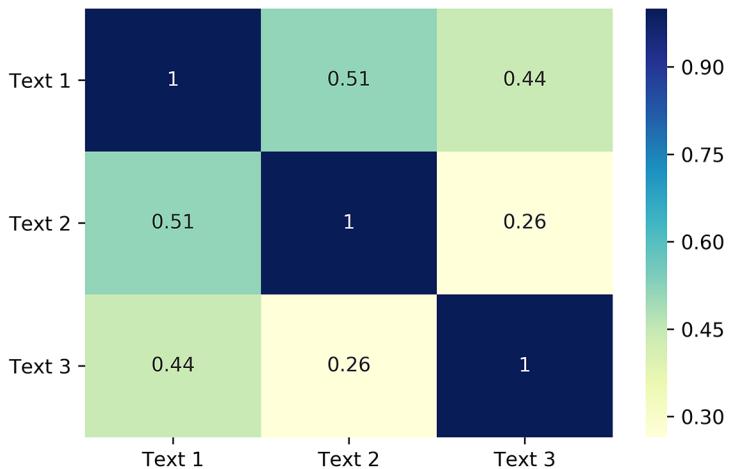


Figure 13.13 A table of normalized Tanimoto similarities across text pairs. The table's diagonal represents the similarity between each text and itself. Not surprisingly, that similarity is 1. Ignoring the diagonal, we see that texts 1 and 2 share the highest similarity. Meanwhile, texts 2 and 3 share the lowest similarity.

Looking at the table is informative. We can immediately tell which text pairs share the highest similarity. However, our table computation relied on inefficient code. The following computations are redundant and can be eliminated.

- The creation of an empty three-by-three array
- The nested for loop iteration across all pairwise vector combinations
- The individual computation of each pairwise vector similarity

We can purge our code of these operations using matrix multiplication. However, first we need to introduce basic matrix operations.

13.3.1 Basic matrix operations

Matrix operations power many subfields of data science, including NLP, network analysis, and machine learning. Therefore, knowing the basics of matrix manipulation is crucial for our data science careers. A *matrix* is the extension of a one-dimensional vector to two dimensions. In other words, a matrix is just a table of numbers. By that definition, `similarities` is a matrix, and so is `unit_vectors`. Most numeric tables discussed in this book are also naturally matrices.

NOTE Every matrix is a numeric table, but not every numeric table is a matrix. All matrix rows must share an equal length. The same is true of all matrix columns. Thus, if a table contains both a five-element column and a seven-element column, it is not a matrix.

Since matrices are tables, they can be analyzed using Pandas. Conversely, numeric tables can be handled using 2D NumPy arrays. Both matrix representations are valid. In fact, Pandas DataFrames and NumPy arrays can sometimes be used interchangeably, because they share certain attributes. For instance, `matrix.shape` returns a count of rows and columns, regardless of whether `matrix` is a DataFrame or an array. Likewise, `matrix.T` transposes rows and columns, regardless of `matrix` type. Let's confirm.

Listing 13.33 Comparing Pandas and NumPy matrix attributes

```
import pandas as pd

matrices = [unit_vectors, pd.DataFrame(unit_vectors)]
matrix_types = ['2D NumPy array', 'Pandas DataFrame']

for matrix_type, matrix in zip(matrix_types, matrices):
    row_count, column_count = matrix.shape
    print(f"Our {matrix_type} contains "
          f"{row_count} rows and {column_count} columns")
    assert (column_count, row_count) == matrix.T.shape
```

Transposing
the matrix flips
the rows and
columns.

```
Our 2D NumPy array contains 3 rows and 15 columns
Our Pandas DataFrame contains 3 rows and 15 columns
```

Pandas and NumPy table structures are similar. Nonetheless, there are certain benefits to storing matrices in 2D NumPy arrays. One immediate benefit is NumPy's integration of Python's built-in arithmetic operators: we can run basic arithmetic operations directly on NumPy arrays.

NUMPY MATRIX ARITHMETIC OPERATIONS

Occasionally in NLP we need to modify a matrix using basic arithmetic. Suppose, for instance, that we wish to compare a collection of documents based on both their bodies and titles. We hypothesize that title similarity is twice as important as body similarity since documents with similar titles are very likely to be thematically related. We thus decide to double the title similarity matrix to better weigh it relative to the body.

NOTE This relative importance of title versus body is particularly true in news articles. Two articles that share a similar title are very likely to refer to the same news story, even if their bodies offer different perspectives on that story. A good heuristic for measuring news article similarity is to compute $2 * \text{title_similarity} + \text{body_similarity}$.

Doubling the values of a matrix is very easy to do in NumPy. For example, we can double our `similarities` matrix by running `2 * similarities`. We can also add `similarities` directly to itself by running `similarities + similarities`. Of course, the two arithmetic outputs will be equal. Meanwhile, running `similarities - similarities` will return a matrix of 0s. Furthermore, running `similarities - similarities - 1` will subtract 1 from each of the zeros.

NOTE We are subtracting `similarities + 1` from `similarities` simply to show the arithmetic flexibility of NumPy. Normally, there's no valid reason to return this operation unless we really need a matrix of negative 1s.

Listing 13.34 NumPy array addition and subtraction

```
double_similarities = 2 * similarities
np.array_equal(double_similarities, similarities + similarities)
zero_matrix = similarities - similarities
negative_1_matrix = similarities - similarities - 1

for i in range(similarities.shape[0]):
    for j in range(similarities.shape[1]):
        assert double_similarities[i][j] == 2 * similarities[i][j]
        assert zero_matrix[i][j] == 0
        assert negative_1_matrix[i][j] == -1
```

In the same manner, we can multiply and divide NumPy arrays. Running `similarities / similarities` will divide each similarity by itself, thus returning a matrix of 1s. Meanwhile, running `similarities * similarities` will return a matrix of squared similarity values.

Listing 13.35 NumPy array multiplication and division

```
squared_similarities = similarities * similarities
assert np.array_equal(squared_similarities, similarities ** 2)
ones_matrix = similarities / similarities

for i in range(similarities.shape[0]):
    for j in range(similarities.shape[1]):
        assert squared_similarities[i][j] == similarities[i][j] ** 2
        assert ones_matrix[i][j] == 1
```

Matrix arithmetic lets us conveniently transition between similarity matrix types. For instance, we can convert our Tanimoto matrix into a cosine similarity matrix simply by running $2 * \text{similarities} / (1 + \text{similarities})$. Thus, if we wish to compare the

Tanimoto similarity with the more popular cosine similarity, we can compute the second cosine matrix in just a single line of code.

Listing 13.36 Converting between matrix similarity-types

```
cosine_similarities = 2 * similarities / (1 + similarities)
for i in range(similarities.shape[0]):
    for j in range(similarities.shape[1]):
        cosine_sim = unit_vectors[i] @ unit_vectors[j]
        assert round(cosine_similarities[i][j],
                     15) == round(cosine_sim, 15)
```

Confirms that the cosine similarity equals the actual vector dot product

Rounds the results because of floating-point errors

NumPy 2D arrays confer additional benefits over Pandas. Accessing rows and columns by index is much more straightforward in NumPy.

NUMPY MATRIX ROW AND COLUMN OPERATIONS

Given any 2D matrix array, we can access the row at index *i* by running `matrix[i]`. Likewise, we can access the column at index *j* by running `matrix[:,j]`. Let's use NumPy indexing to print the first row and column of both `unit_vectors` and `similarities`.

Listing 13.37 Accessing NumPy matrix rows and columns

```
for name, matrix in [('Similarities', similarities),
                     ('Unit Vectors', unit_vectors)]:
    print(f"Accessing rows and columns in the {name} Matrix.")
    row, column = matrix[0], matrix[:,0]
    print(f"Row at index 0 is:\n{row}")
    print(f"\nColumn at index 0 is:\n{column}\n")
```

Accessing rows and columns in the Similarities Matrix.
 Row at index 0 is:
`[1. 0.51442439 0.44452044]`

Column at index 0 is:
`[1. 0.51442439 0.44452044]`

Accessing rows and columns in the Unit Vectors Matrix.
 Row at index 0 is:
`[0.40824829 0.40824829 0. 0.40824829 0. 0.40824829 0.]`

Column at index 0 is:
`[0.40824829 0. 0.30151134]`

All printed rows and columns are one-dimensional NumPy arrays. Given two arrays, we can compute their dot product, but only if the array lengths are the same. In our output, both `similarities[0].size` and `unit_vectors[:,0].size` are equal to 3. Hence, we can take the dot product between the first row of `similarities` and the first column of `unit_vectors`. This particular row-to-column dot product is not useful in our text analysis, but it serves to illustrate our ability to easily compute the dot

product between matrix rows and matrix columns. A little later, we will use that ability to compute text vector similarities with great efficiency.

Listing 13.38 Computing the dot product between a row and column

```
row = similarities[0]
column = unit_vectors[:,0]
dot_product = row @ column
print(f"The dot product between the row and column is: {dot_product:.4f}")
```

The dot product between the row and column is: 0.5423

In that same vein, we can take the dot product between every row in `similarities` and every column in `unit_vectors`. Let's print all possible dot product results.

Listing 13.39 Computing dot products between all rows and columns

```
num_rows = similarities.shape[0]
num_columns = unit_vectors.shape[1]
for i in range(num_rows):
    for j in range(num_columns):
        row = similarities[i]
        column = unit_vectors[:,j]
        dot_product = row @ column
        print(f"The dot product between row {i} column {j} is: "
              f"{dot_product:.4f}")
```

The dot product between row 0 column 0 is: 0.5423
The dot product between row 0 column 1 is: 0.8276
The dot product between row 0 column 2 is: 0.1340
The dot product between row 0 column 3 is: 0.6850
The dot product between row 0 column 4 is: 0.1427
The dot product between row 0 column 5 is: 0.1340
The dot product between row 0 column 6 is: 0.1427
The dot product between row 0 column 7 is: 0.5423
The dot product between row 0 column 8 is: 0.1340
The dot product between row 0 column 9 is: 0.1340
The dot product between row 0 column 10 is: 0.8276
The dot product between row 0 column 11 is: 0.1340
The dot product between row 0 column 12 is: 0.1340
The dot product between row 0 column 13 is: 0.1427
The dot product between row 0 column 14 is: 0.5509
The dot product between row 1 column 0 is: 0.2897
The dot product between row 1 column 1 is: 0.8444
The dot product between row 1 column 2 is: 0.0797
The dot product between row 1 column 3 is: 0.5671
The dot product between row 1 column 4 is: 0.2774
The dot product between row 1 column 5 is: 0.0797
The dot product between row 1 column 6 is: 0.2774
The dot product between row 1 column 7 is: 0.2897
The dot product between row 1 column 8 is: 0.0797
The dot product between row 1 column 9 is: 0.0797
The dot product between row 1 column 10 is: 0.8444
The dot product between row 1 column 11 is: 0.0797

```
The dot product between row 1 column 12 is: 0.0797
The dot product between row 1 column 13 is: 0.2774
The dot product between row 1 column 14 is: 0.4874
The dot product between row 2 column 0 is: 0.4830
The dot product between row 2 column 1 is: 0.6296
The dot product between row 2 column 2 is: 0.3015
The dot product between row 2 column 3 is: 0.5563
The dot product between row 2 column 4 is: 0.0733
The dot product between row 2 column 5 is: 0.3015
The dot product between row 2 column 6 is: 0.0733
The dot product between row 2 column 7 is: 0.4830
The dot product between row 2 column 8 is: 0.3015
The dot product between row 2 column 9 is: 0.3015
The dot product between row 2 column 10 is: 0.6296
The dot product between row 2 column 11 is: 0.3015
The dot product between row 2 column 12 is: 0.3015
The dot product between row 2 column 13 is: 0.0733
The dot product between row 2 column 14 is: 0.2548
```

We've generated 45 dot products: one for each row, column combination. Our printed outputs are excessive. These outputs can be stored more concisely in a table called `dot_products`, where `dot_products[i][j]` is equal to `similarities[i] @ unit_vectors[:, j]`. Of course, by definition, that table is a matrix.

Listing 13.40 Storing all-by-all dot products in a matrix

```
dot_products = np.zeros((num_rows, num_columns)) ← Returns an empty
for i in range(num_rows): array of zeros
    for j in range(num_columns):
        dot_products[i][j] = similarities[i] @ unit_vectors[:, j]

print(dot_products)

[[0.54227624 0.82762755 0.13402795 0.6849519 0.14267565 0.13402795
  0.14267565 0.54227624 0.13402795 0.13402795 0.82762755 0.13402795
  0.13402795 0.14267565 0.55092394]
 [0.28970812 0.84440831 0.07969524 0.56705821 0.2773501 0.07969524
  0.2773501 0.28970812 0.07969524 0.07969524 0.84440831 0.07969524
  0.07969524 0.2773501 0.48736297]
 [0.48298605 0.62960397 0.30151134 0.55629501 0.07330896 0.30151134
  0.07330896 0.48298605 0.30151134 0.30151134 0.62960397 0.30151134
  0.30151134 0.07330896 0.25478367]]
```

The operation we've just executed is called a *matrix product*. It's a generalization of the vector dot product to two dimensions. Given two matrices, `matrix_a` and `matrix_b`, we can compute their product by calculating `matrix_c`, where `matrix_c[i][j]` is equal to `matrix_a[i] @ matrix_b[:, j]` (figure 13.14). Matrix products are crucial to many modern technological advancements. They power the ranking algorithms in massive search engines like Google, serve as the foundation for techniques used to train self-driving cars, and underlie much of modern NLP. The usefulness of matrix

products will become apparent shortly, but first we must discuss matrix product operations in more detail.

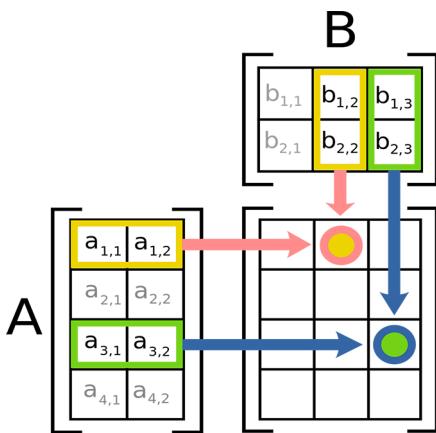


Figure 13.14 Computing the matrix product of matrices A and B. The operation outputs a new matrix. Each element in the i th row and j th column of the output equals the dot product between the i th row of A and the j th column of B. For instance, the element in the first row and second column of the output equals $a_{1,1} * b_{1,2} + a_{1,2} * b_{2,2}$. Meanwhile, the element in the third row and third column of the output equals $a_{3,1} * b_{1,3} + a_{3,2} * b_{2,3}$.

NUMPY MATRIX PRODUCTS

Naively, we can calculate `matrix_c` by running nested `for` loops across `matrix_a` and `matrix_b`. This technique is not efficient. Conveniently, NumPy's product operator `@` can be applied to 2D matrices as well as 1D arrays. If `matrix_a` and `matrix_b` are both NumPy arrays, then `matrix_c` equals `matrix_a @ matrix_b`. Thus, the matrix product of `similarities` and `unit_vectors` equals `similarities @ unit_vectors`. Let's confirm.

Listing 13.41 Computing a matrix product using NumPy

```
matrix_product = similarities @ unit_vectors
assert np.allclose(matrix_product, dot_products) ←
    Asserts that all the elements of matrix_product are nearly
    identical to all the elements of dot_products. There are tiny
    differences in results due to floating-point errors.
```

What will happen if we flip our input matrices and run `unit_vectors @ similarities`? NumPy will throw an error! The computation takes the vector dot product between rows in `unit_vectors` and columns in `similarities`, but these rows and columns have different lengths. Therefore, the computation is not possible (figure 13.15).

Listing 13.42 Computing an erroneous matrix product

```
try:
    matrix_product = unit_vectors @ similarities
except:
    print("We can't compute the matrix product")
We can't compute the matrix product
```

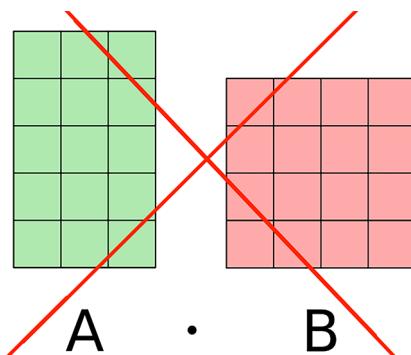


Figure 13.15 An erroneous effort to compute the matrix product of matrices A and B. Matrix A has three columns in each row, and matrix B has four rows in each column. We cannot take the dot product between a three-element row and a four-element column. Thus, running `A @ B` causes an error.

The matrix product is order dependent. The output of `matrix_a @ matrix_b` is not necessarily the same as `matrix_b @ matrix_a`. In words, we can distinguish between `matrix_a @ matrix_b` and `matrix_b @ matrix_a` as follows:

- `matrix_a @ matrix_b` is the product of `matrix_a` and `matrix_b`.
- `matrix_b @ matrix_a` is the product of `matrix_b` and `matrix_a`.

In mathematics, the words *product* and *multiplication* are often interchangeable. Thus, computing the matrix product is commonly called *matrix multiplication*. That name is so ubiquitous that NumPy includes an `np.matmul` function. The output of `np.matmul(matrix_a, matrix_b)` is identical to `matrix_a @ matrix_b`.

Listing 13.43 Running matrix multiplication using `matmul`

```
matrix_product = np.matmul(similarities, unit_vectors)
assert np.array_equal(matrix_product,
                      similarities @ unit_vectors)
```

Common NumPy matrix operations

- `matrix.shape`—Returns a tuple containing the row count and column count in the matrix.
- `matrix.T`—Returns a transposed matrix where rows and columns are swapped.
- `matrix[i]`—Returns the *i*th row in the matrix.
- `matrix[:, j]`—Returns the *j*th column in the matrix.
- `k * matrix`—Multiplies each element of the matrix by a constant `k`.
- `matrix + k`—Adds a constant `k` to each element of the matrix.
- `matrix_a + matrix_b`—Adds each element of `matrix_a` to `matrix_b`. Equivalent to running `matrix_c[i][j] = matrix_a[i][j] + matrix_b[i][j]` for every possible *i* and *j*.
- `matrix_a * matrix_b`—Multiplies each element of `matrix_a` with an element of `matrix_b`. Equivalent to running `matrix_c[i][j] = matrix_a[i][j] * matrix_b[i][j]` for every possible *i* and *j*.

(continued)

- `matrix_a @ matrix_b`—Returns the matrix product of `matrix_a` and `matrix_b`. Equivalent to running `matrix_c[i][j] = matrix_a[i] @ matrix_b[:, j]` for every possible `i` and `j`.
- `np.matmul(matrix_a, matrix_b)`—Returns the matrix product of `matrix_a` and `matrix_b`, without relying on the `@` operator.

NumPy lets us execute matrix multiplication without relying on nested for loops. This improvement is more than just cosmetic. Standard Python for loops are designed to run on generalized data lists; they are not optimized for numbers. Meanwhile, NumPy cleverly optimizes its array iterations. Consequently, matrix multiplication is noticeably faster when run in NumPy.

Let's compare the matrix product speed between NumPy and regular Python (figure 13.16). Listing 13.44 plots the product speed across matrices of variable sizes using both NumPy and Python for loops. Python's built-in `time` module is employed to time the matrix multiplications.

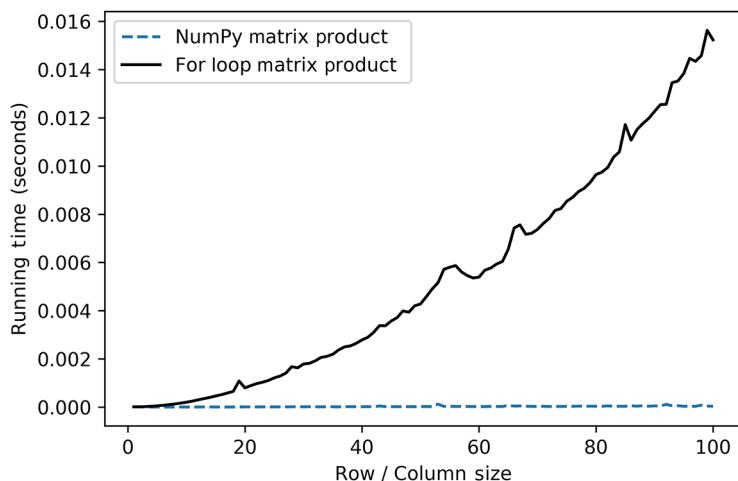


Figure 13.16 Matrix size plotted against product running times for NumPy and regular Python. NumPy is vastly faster than regular Python.

NOTE Running times will fluctuate depending on the local state of the machine that is executing the code.

Listing 13.44 Comparing matrix product running times

```
import time

numpy_run_times = []
for_loop_run_times = []
```

```

matrix_sizes = range(1, 101)
for size in matrix_sizes:
    matrix = np.ones((size, size))           ← Creates a size-by-size matrix
                                              of ones, where size ranges
                                              from 1 to 100

    start_time = time.time()                ← Returns the current
                                              time in seconds
    matrix @ matrix
    numpy_run_times.append(time.time() - start_time) ← Stores the matrix
                                              product speed in
                                              NumPy

    start_time = time.time()
    for i in range(size):
        for j in range(size):
            matrix[i] @ matrix[:,j]

    for_loop_run_times.append(time.time() - start_time) ← Stores the matrix
                                              product speed of
                                              a Python for loop

plt.plot(matrix_sizes, numpy_run_times,
         label='NumPy Matrix Product', linestyle='--')
plt.plot(matrix_sizes, for_loop_run_times,
         label='For-Loop Matrix Product', color='k')
plt.xlabel('Row / Column Size')
plt.ylabel('Running Time (Seconds)')
plt.legend()
plt.show()

```

When it comes to matrix multiplication, NumPy greatly outperforms basic Python. NumPy matrix product code is more efficient to run and also to write. We will now use NumPy to compute our all-by-all text similarities with maximum efficiency.

13.3.2 Computing all-by-all matrix similarities

We've previously computed our text similarities by iterating over a `unit_vectors` matrix. This matrix holds the normalized TF vectors for our three seashell texts. What will happen if we multiply `unit_vectors` by `unit_vectors.T`? Well, `unit_vectors.T` is a transpose of `unit_vectors`. Therefore, each column `i` in the transpose equals row `i` in `unit_vectors`. Taking the dot product of `unit_vectors[i]` and `unit_vectors.T[:, i]` will return the cosine similarity between a unit vector and itself, as shown in figure 13.17. That similarity, of course, will equal 1. By this logic, `unit_vectors[i] @ unit_vectors[j].T` equals the cosine similarity between the `i`th and `j`th vectors. Consequently, `unit_vectors @ unit_vectors.T` returns a matrix of all-by-cosine similarities. The matrix should equal our previously computed `cosine_similarities` array. Let's confirm.

Listing 13.45 Obtaining cosines from a matrix product

```

cosine_matrix = unit_vectors @ unit_vectors.T
assert np.allclose(cosine_matrix, cosine_similarities)

```

Each element in `cosine_matrix` equals the cosine of the angle between two vectorized texts. That cosine can be transformed into a Tanimoto value, which generally

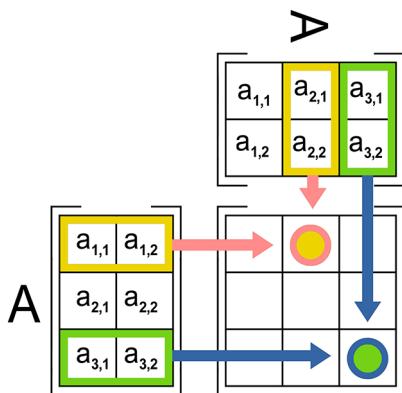


Figure 13.17 Computing the dot product between A and the transpose of A . The operation outputs a new matrix. Each element in the i th row and j th column of the output equals the dot product between the i th row of A and the j th column of A . Thus, the element in the third row and third column of the output equals the dot product of $A[2]$ with itself. If matrix A is normalized, then that dot product will equal 1.0.

reflects word overlap and divergence between texts. Using NumPy arithmetic, we can convert `cosine_matrix` into a Tanimoto similarity matrix by running `cosine_matrix / (2 - cosine_matrix)`.

Listing 13.46 Converting cosines to a Tanimoto matrix

```
tanimoto_matrix = cosine_matrix / (2 - cosine_matrix)
assert np.allclose(tanimoto_matrix, similarities)
```

We've computed all the Tanimoto similarities in just two lines of code. We can also compute these similarities by inputting `unit_vectors` and `unit_vectors.T` directly into our `normalized_tanimoto` function. As a reminder, the function does the following:

- 1 Takes as input two NumPy arrays. Their dimensionality is not constrained.
- 2 Applies the `@` operator to the NumPy arrays. If the arrays are matrices, the operation returns a matrix product.
- 3 Uses arithmetic to modify the product. The arithmetic operations can be equally applied to both numbers and matrices.

Hence it's not surprising that `normalized_tanimoto(unit_vectors, unit_vectors.T)` returns an output equal to `tanimoto_matrix`.

Listing 13.47 Inputting matrices into normalized_tanimoto

```
output = normalized_tanimoto(unit_vectors, unit_vectors.T)
assert np.array_equal(output, tanimoto_matrix)
```

Given a matrix of normalized TF vectors, we can compute their all-by-all similarities in a single, efficient line of code.

Common normalized matrix comparisons

- `norm_matrix @ norm_matrix.T`—Returns a matrix of all-by-all cosine similarities
- `norm_matrix @ norm_matrix.T / (2 - norm_matrix @ norm_matrix.T)`—
Returns a matrix of all-by-all Tanimoto similarities

13.4 Computational limits of matrix multiplication

Matrix multiplication speed is determined by the matrix size. NumPy may optimize for speed, but even NumPy has its limits. These limits become obvious when we compute real-world text matrix products. Issues arise from the matrix column count, which is dependent on vocabulary size. The total words in a vocabulary can spiral out of control when we begin to compare nontrivial texts.

Consider, for instance, the analysis of novels. The average novel contains roughly 5,000 to 10,000 unique words. *The Hobbit*, for example, contains 6,175 unique words. Meanwhile, *A Tale of Two Cities* contains 9,699 unique words. Some of the words overlap between the two novels; others do not. Together, the novels share a vocabulary size of 12,138 words. We can also throw a third novel into the mix. Adding *The Adventures of Tom Sawyer* expands the vocabulary size to 13,935 words. At this rate, adding 27 more novels will expand the vocabulary size to approximately 50,000 words.

Let's assume that 30 novels require a shared vocabulary containing 50,000 words. Furthermore, let's assume we take an all-by-all similarity across the 30 books. How much time is required to compute these similarities? Let's find out! We'll create a 30-book by 50,000-word `book_matrix`. All rows in the matrix will be normalized. Then we'll measure the running time of `normalized_tanimoto(book_matrix, book_matrix.T)`.

NOTE The purpose of our experiment is to test the impact of the matrix column count on running time. Here, the actual matrix contents don't matter. We thus oversimplify the situation by setting all word counts to 1. Consequently, the normalized values in each row equal $1 / 5000$. In a real-world setting, this would not be the case. Also, note that it's possible to optimize running time by tracking all zero-value matrix elements. Here, we don't consider the impact of zero values on matrix multiplication speed.

Listing 13.48 Timing an all-by-all comparison of 30 novels

```
vocabulary_size = 50000
normalized_vector = [1 / vocabulary_size] * vocabulary_size
book_count = 30

def measure_run_time(book_count):
    book_matrix = np.array([normalized_vector] * book_count)
    start_time = time.time()
    normalized_tanimoto(book_matrix, book_matrix.T)
    return time.time() - start_time
```

The function computes the running time on a `book_count`-by-50,000 matrix. The function is reused in the next two code listings.

```
run_time = measure_run_time(book_count)
print(f"It took {run_time:.4f} seconds to compute the similarities across a "
      f"{book_count}-book by {vocabulary_size}-word matrix")

It took 0.0051 seconds to compute the similarities across a 30-book by 50000-
word matrix
```

The similarity matrix took approximately 5 milliseconds to compute. This is a reasonable running time. Will it stay reasonable as the number of analyzed books continues to increase? Let's check. We'll plot the running times across multiple book counts ranging from 30 to nearly 1,000 (figure 13.18). For consistency's sake, we'll keep the vocabulary size at 50,000.

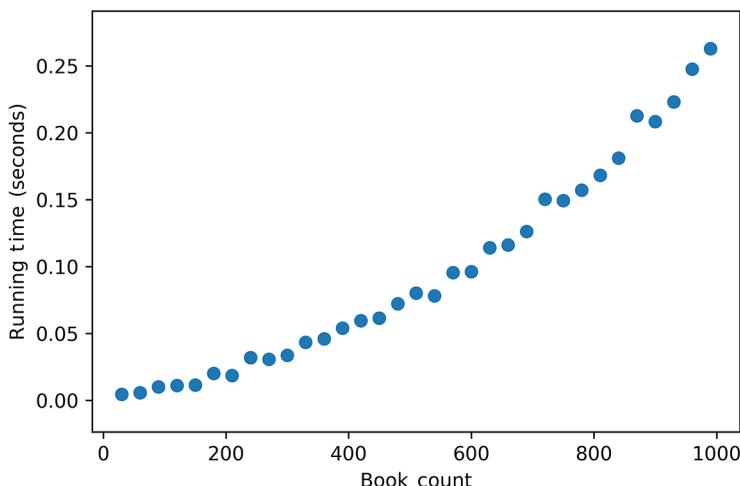


Figure 13.18 Book counts plotted against the running time of text comparisons. The running times increase quadratically.

Listing 13.49 Plotting book counts vs. running times

```
book_counts = range(30, 1000, 30)           ←
run_times = [measure_run_time(book_count)    ←
             for book_count in book_counts]
plt.scatter(book_counts, run_times)          ←
plt.xlabel('Book Count')
plt.ylabel('Running Time (Seconds)')
plt.show()
```

We do not sample every single book count—the accumulated running time would be too slow.

Generates a scatter plot instead of a curve. Figure 13.18 fits the discrete points to a continuous parabolic curve.

The similarity running time rises quadratically with book count. At 1,000 books, the running time increases to approximately 0.27 seconds. This delay is tolerable. However, if we increase the book count even further, the delay is no longer acceptable. We can show this using simple math. Our plotted curve takes on a parabolic shape defined by $y = n * (x ** 2)$. When x is approximately 1,000, y equals approximately

0.27. Thus, we can model our running times using the equation $y = (0.27 / (1000 ** 2)) * (x ** 2)$. Let's confirm by plotting equation outputs together with our precomputed measurements (figure 13.19). The two plots should mostly overlap.

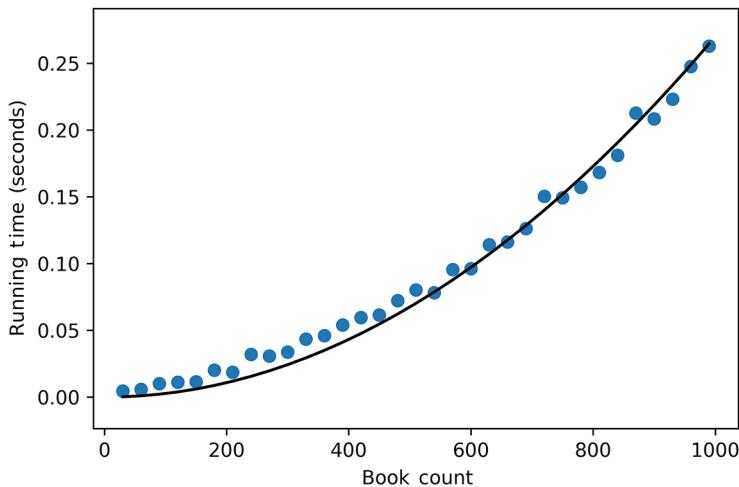


Figure 13.19 A quadratic curve plotted together with running times. The shape of the curve overlaps with the running time scatter plot.

Listing 13.50 Modeling running times using a quadratic curve

```
def y(x): return (0.27 / (1000 ** 2)) * (x ** 2)
plt.scatter(book_counts, run_times)
plt.plot(book_counts, y(np.array(book_counts)), c='k')
plt.xlabel('Book Count')
plt.ylabel('Running Time (Seconds)')
plt.show()
```

Our plotted equation overlaps with the measured times. Thus, we can use the equation to predict the speed of larger book comparisons. Let's see how long it will take to measure the similarity across 300,000 books.

NOTE 300,000 may seem like an unusually large number. However, it reflects the 200,000 to 300,000 English-language novels that are published every year. If we wish to compare all the novels published in a year, we need to multiply matrices containing more than 200,000 rows.

Listing 13.51 Predicting the running time for 300,000 books

```
book_count = 300000
run_time = y(book_count) / 3600      ← Divides by 3600 to convert
print(f"It will take {run_time} hours to compute all-by-all similarities "
     f"from a {book_count}-book by {vocabulary_size}-word matrix")
```

It will take 6.75 hours to compute all-by-all similarities from a 300000-book by 50000-word matrix

It will take nearly 7 hours to compare 300,000 books. This delay in time is not acceptable, especially in industrial NLP systems, which are designed to process millions of texts in mere seconds. We need to somehow reduce the running time. One approach is to reduce the matrix size.

Our matrix is too large, partially because of column size. Each row contains 50,000 columns corresponding to 50,000 words. However, in a real-world setting, not all words are distributed equally. While some words are common across novels, other words may appear only once. For example, take the lengthy novel *Moby Dick*: 44% of the words in it are mentioned once and never used again. Some of the words are rarely mentioned in other novels. Removing them will lower the column size.

On the opposite end of the spectrum are common words that appear in every novel. Words like *the* do not provide a differentiating signal between texts. Removing common words will also reduce the column size.

It's possible to systematically reduce the dimensions of each matrix row from 50,000 to a more reasonable value. In the next section, we introduce a series of dimension-reduction techniques to shrink the shape of any inputted matrix. Reducing the dimensions of text matrices greatly lowers the running times of common NLP computations.

Summary

- We can compare texts using the *Jaccard similarity*. This similarity metric equals the fraction of total unique words that are shared between two texts.
- We can compute the Jaccard similarity by transforming our texts into binary vectors of 1s and 0s. Taking the *dot product* of two binary text vectors returns the shared word count between the texts. Meanwhile, the dot product of a text vector with itself returns the total count of words in the text. These values are sufficient to compute the Jaccard similarity.
- The *Tanimoto similarity* generalizes Jaccard to include non-binary vectors. This allows us to compare vectors of word counts, which are referred to as *TF vectors*.
- *TF vector similarity* is overly dependent on text size. We can eliminate this dependence using *normalization*. A vector can be normalized by first computing its *magnitude*, which is the vector's distance to the origin. Dividing a vector by its magnitude produces a normalized unit vector.
- The magnitude of a *unit vector* is 1. Furthermore, the Tanimoto similarity is partially dependent on vector magnitude. Thus, we can simplify the similarity function if it's run exclusively on unit vectors. Moreover, that unit vector similarity can be converted into other common metrics, such as *cosine similarity* and distance.
- We can efficiently compute all-by-all similarities using *matrix multiplication*. A *matrix* is just a 2D table of numbers. We can multiply two matrices by taking

pairwise dot products between every matrix row and matrix column. If we multiply a normalized matrix by its transpose, we produce a matrix of all-by-all cosine similarities. Using NumPy's matrix arithmetic, we transform these cosine similarities into Tanimoto similarities.

- Matrix multiplication is much faster in NumPy than in pure Python. However, even NumPy has its limits. Once a matrix gets too large, we need to find a way to reduce its size.

Dimension reduction of matrix data

This section covers

- Simplifying matrices with geometric rotations
- What is principal component analysis?
- Advanced matrix operations for reducing matrix size
- What is singular value decomposition?
- Dimension reduction using scikit-learn

Dimension reduction is a series of techniques for shrinking data while retaining its information content. These techniques permeate many of our everyday digital activities. Suppose, for instance, that you've just returned from a vacation in Belize. There are 10 vacation photos on your phone that you wish to message to a friend. Unfortunately, these photos are quite large, and your current wireless connection is slow. Each photo is 1,200 pixels tall and 1,200 pixels wide. It takes up 5.5 MB of memory and requires 15 seconds to transfer. Transferring all 10 photos will take 2.5 minutes. Fortunately, your messaging app offers you a better option: You can shrink each photo from $1,200 \times 1,200$ pixels to 600×480 pixels. This reduces the dimensions of each photo sixfold. By lowering the resolution, you'll sacrifice a little detail. However, the vacation photos will maintain most of their information—the lush jungles, blue seas, and shimmering sands will remain clearly

visible in the images. Therefore, the trade-off is worth it. Reducing the dimensionality by six will increase the transfer speed by six: it will take just 25 seconds to share the 10 photos with your friend.

There are, of course, more benefits to dimension reduction than just transfer speed. Consider, for instance, mapmaking, which can be treated as a dimension-reduction problem. Our Earth is a 3D sphere that can be accurately modeled as a globe. We can turn that globe into a map by projecting its 3D shape onto a 2D piece of paper. The paper map is easier to carry from place to place—unlike a globe, we can fold the map and put it in our pocket. Furthermore, the 2D map offers us additional advantages. Suppose we're asked to locate all the countries that border at least 10 other neighboring territories. Finding these dense border regions on a map is easy: we can glance at the 2D map to hone in on crowded country clusters. If we use a globe, instead, the task becomes more challenging. We need to spin the globe across multiple perspectives because we can't see all the countries at once. In some ways, the globe's curvature acts as a layer of noise that interferes with our given task. Removing that curvature simplifies our effort, but at a price: the continent of Antarctica is essentially deleted from the map. Of course, there are no countries in Antarctica, so from the perspective of our task, that trade-off is worth it.

Our map analogy demonstrates the following advantages of dimensionally reduced data:

- More compact data is easier to transfer and store.
- Algorithmic tasks require less time when our data is smaller.
- Certain complex tasks, like cluster detection, can be simplified by removing unnecessary information.

The last two bullet points are very relevant to this case study. We want to cluster thousands of text documents by topic. The clustering will entail computing a matrix of all-by-all document similarities. As discussed in the previous section, this computation can be slow. Dimension reduction can speed up the process by reducing the number of data matrix columns. As a further bonus, dimensionally reduced text data has been shown to yield higher-quality topic clusters.

Let's dive deeper into the relationship between dimension reduction and data clustering. We start with a simple task in which we cluster 2D data in one dimension.

14.1 Clustering 2D data in one dimension

Dimension reduction has many uses, including more interpretable clustering. Consider a scenario in which we manage an online clothing store. When customers visit our website, they are asked to provide their height and weight. These measurements are added to a customer database. Two database columns are required to store each customer's measurements, and therefore the data is two-dimensional. The 2D measurements are used to offer customers appropriately sized clothing based on whatever

inventory is available. Our inventory comes in three sizes: small, medium, and large. Given the measurement data for 180 customers, we would like to do the following:

- Group our customers into three distinct clusters based on size.
- Build an interpretable model to determine the clothing size category of each new customer using our computed clusters.
- Make our clustering simple enough for our nontechnical investors to comprehend.

The third point in particular limits our decisions. Our clustering can't rely on technical concepts, such as centroids or distances to the mean. Ideally, we'd be able to explain our model in a single figure. We can achieve this level of simplicity using dimension reduction. However, we first need to simulate the 2D measurement data for 180 customers. Let's start by simulating customer heights. Our heights range from 60 inches (5 ft) to 78 inches (6.5 ft). We fabricate these heights by calling `np.arange(60, 78, 0.1)`. This returns an array of heights between 60 and 78 inches, where each consecutive height increases by 0.1 inches.

Listing 14.1 Simulating a range of heights

```
import numpy as np
heights = np.arange(60, 78, 0.1)
```

Meanwhile, our simulated weights depend strongly on the height. A taller person is likely to weigh more than a shorter person. It has been shown that, on average, a person's weight in pounds equals approximately $4 * \text{height} - 130$. Of course, each individual person's weight fluctuates around this average value. We'll model these random fluctuations using a normal distribution with a standard deviation of 10 lb. Thus, given a `height` variable, we can model the weight as `4 * height - 130 + np.random.normal(scale=10)`. Next, we use this relationship to fabricate weights for each of our 180 heights.

Listing 14.2 Simulating weights using heights

```
np.random.seed(0)
random_fluctuations = np.random.normal(scale=10, size=heights.size)
weights = 4 * heights - 130 + random_fluctuations
```

The scale parameter sets the standard deviation.

We can treat the heights and weights as two-dimensional coordinates in a measurements matrix. Let's store and plot these measured coordinates (figure 14.1).

Listing 14.3 Plotting 2D measurements

```
import matplotlib.pyplot as plt
measurements = np.array([heights, weights])
plt.scatter(measurements[0], measurements[1])
plt.xlabel('Height (in)')
plt.ylabel('Weight (lb)')
plt.show()
```

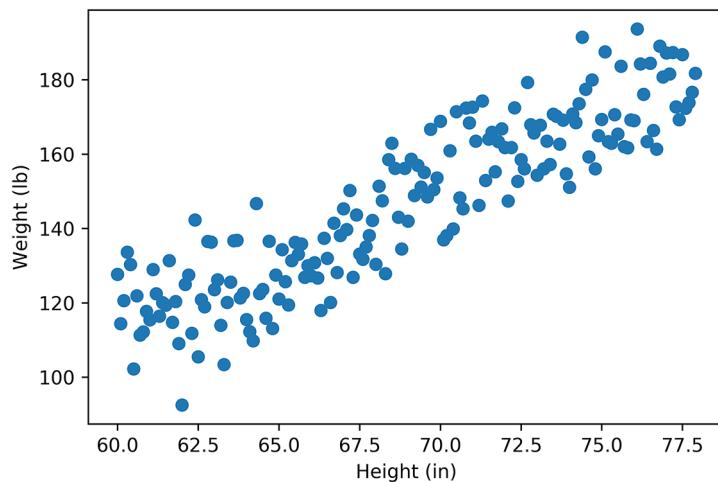


Figure 14.1 A plot of heights vs. weights. Their linear relationship is clearly visible.

The linear relationship between height and weight is clearly visible in the plot. Also, as expected, the height and weight axes are scaled differently. As a reminder, Matplotlib manipulates its 2D axes to make the final plot aesthetically pleasing. Normally, this is a good thing. However, we'll soon be rotating the plot to simplify our data. The rotation will shift the axes scaling, making the rotated data difficult to compare with the original data plot. Consequently, we should equalize our axes to obtain consistent visual output. Let's equalize the axes by calling `plt.axis('equal')` and then regenerate the plot (figure 14.2).

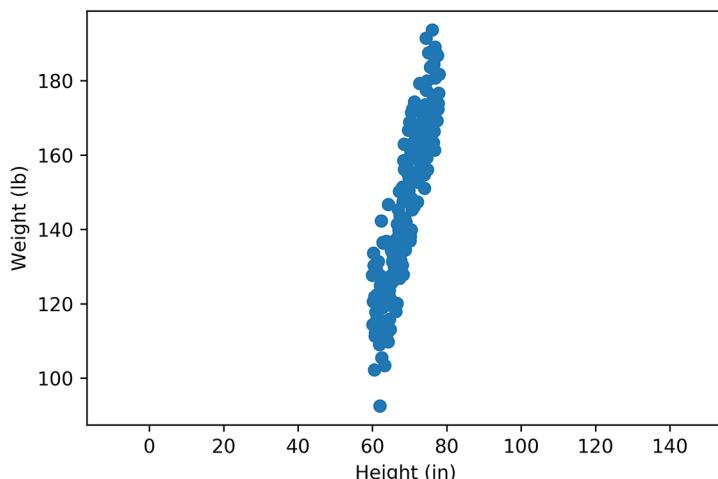


Figure 14.2 A plot of heights vs. weights with both axes scaled equally

Listing 14.4 Plotting 2D measurements using equally scaled axes

```
plt.scatter(measurements[0], measurements[1])
plt.xlabel('Height (in)')
plt.ylabel('Weight (lb)')
plt.axis('equal')
plt.show()
```

Our plot now forms a thin, cigar-like shape. We can cluster the cigar by size if we slice it into three equal parts. One way of obtaining the clusters is to utilize K-means. Of course, interpreting that output will require an understanding of the K-means algorithm. A less technical solution is to just tip the cigar on its side. If the cigar-shaped plot was positioned horizontally, we could separate it into three parts using two vertical slices, as shown in figure 14.3. The first slice would isolate 60 of the leftmost data points, and the second slice would isolate 60 of the rightmost customer points. These operations would segment our customers in a manner that's easy to explain, even to a nontechnical person.

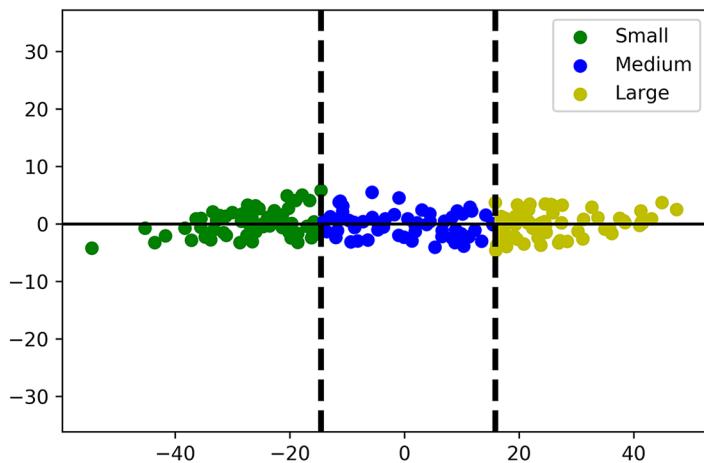


Figure 14.3 Our linear measurements, rotated horizontally so that they lie primarily on the x-axis. Two vertical cutoffs are sufficient to divide the data into three equal clusters: small, medium, and large. In the plot, the x-axis is sufficient for distinguishing between measurements. Thus, we can eliminate the y-axis with minimal information loss. Note that this figure was generated using listing 14.15.

NOTE For the purposes of the exercise, we assume that small, medium, and large sizes are distributed equally. In the real-world clothing industry, this might not be the case.

If we rotate our data toward the x-axis, the horizontal x-values should be sufficient to distinguish between points. We can thus cluster the data without relying on the vertical

y-values. Effectively, we'll be able to delete the y-values with minimal information loss. That deletion will reduce our data from two dimensions to one (figure 14.4).

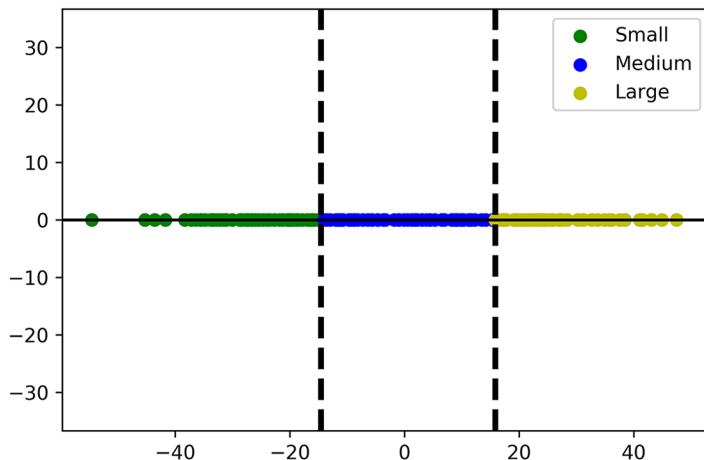


Figure 14.4 Our linear measurements, reduced to one dimension using horizontal rotation. The data points have been rotated toward the x-axis, and their y-value coordinates have been deleted. Nevertheless, the remaining x-values are sufficient to distinguish between points. Thus, our 1D output still allows us to split the data into three equal clusters.

We'll now attempt to cluster our 2D data by flipping the data on its side. This horizontal rotation will allow us to both cluster the data and reduce it to one dimension.

14.1.1 Reducing dimensions using rotation

To flip our data on its side, we must execute two separate steps:

- 1 Shift all of our data points so that they are centered on the origin of the plot, which is located at coordinates (0, 0). This will make it easier to rotate the plot toward the x-axis.
- 2 Rotate the plotted data until the total distance of the data points to the x-axis is minimized.

Centering our data at the origin is trivial. The central point of every dataset is equal to its mean. Thus, we need to adjust our coordinates so that their x-value mean and y-value mean both equal zero. This can be done by subtracting the current mean from every coordinate; in other words, subtracting the mean height for heights and the mean weight from weights will produce a dataset that's centered at (0, 0).

Let's shift our height and weight coordinates and store these changes in a `centered_data` array. Then we plot the shifted coordinates to verify that they are centered on the origin (figure 14.5).

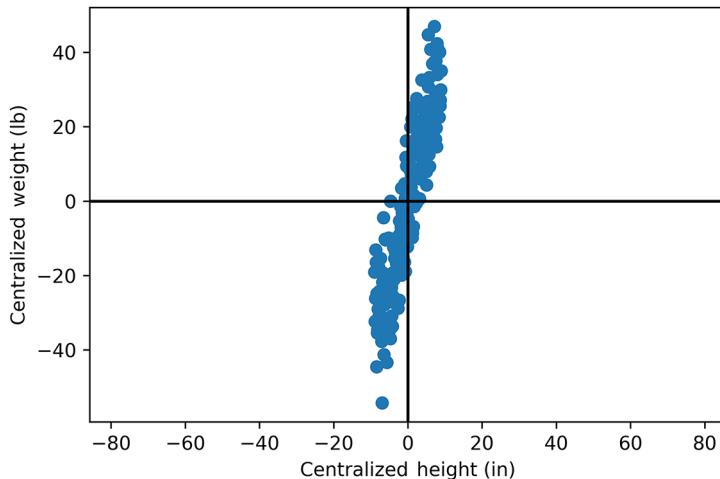


Figure 14.5 A plot of heights vs. weights, centered at the origin. The centered data can be rotated like a propeller.

Listing 14.5 Centering the measurements at the origin

```
centered_data = np.array([heights - heights.mean(),
                        weights - weights.mean()])
plt.scatter(centered_data[0], centered_data[1])
plt.axhline(0, c='black')           ← Visualizes the x-axis
plt.axvline(0, c='black')           and y-axis to mark the
plt.xlabel('Centralized Height (in)') location of the origin
plt.ylabel('Centralized Weight (lb)')
plt.axis('equal')
plt.show()
```

Our data is now perfectly centered at the origin. However, the orientation of the data is closer to the y-axis than the x-axis. Our goal is to adjust this orientation through rotation. We want to spin the plotted points around the origin until they overlap with the x-axis. Rotating a 2D plot around its center requires the use of a *rotation matrix*: a two-by-two array of the form `np.array([[cos(x), -sin(x)], [sin(x), cos(x)]])`, where x is the angle of rotation. The matrix product of this array and `centered_data` rotates the data by x radians. The rotation occurs in the counterclockwise direction. We can also rotate the data in the clockwise direction by inputting `-x` instead of x .

Let's utilize the rotation matrix to rotate `centered_data` clockwise by 90 degrees. Then we plot both the rotated data and the original `centered_data` array (figure 14.6).

Listing 14.6 Rotating centered_data by 90 degrees

```
from math import sin, cos
angle = np.radians(-90)           ← Converts the angle from
rotation_matrix = np.array([[cos(angle), -sin(angle)],
                           [sin(angle), cos(angle)]])
```

```

rotated_data = rotation_matrix @ centered_data
plt.scatter(centered_data[0], centered_data[1], label='Original Data')
plt.scatter(rotated_data[0], rotated_data[1], c='y', label='Rotated Data')
plt.axhline(0, c='black')
plt.axvline(0, c='black')
plt.legend()
plt.axis('equal')
plt.show()

```

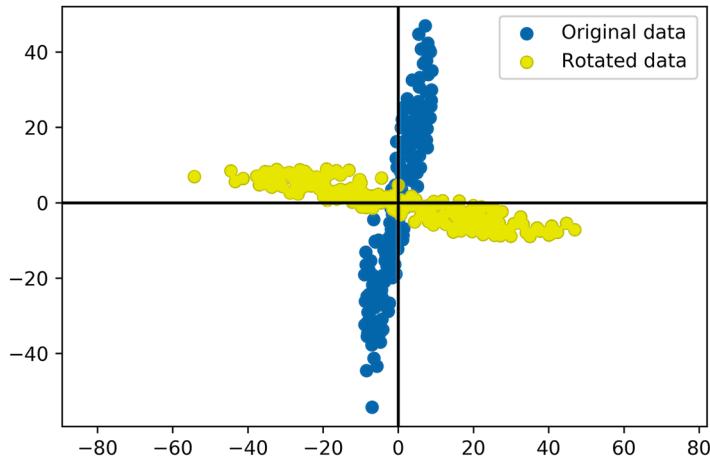


Figure 14.6 A plot of `centered_data` before and after a rotation. The data has been rotated 90 degrees about the origin. It is now positioned closer to the x-axis.

As expected, our `rotated_data` result is perpendicular to the `centered_data` plot. We have successfully rotated the plot by 90 degrees. Furthermore, our rotation has shifted the plot closer to the x-axis. We need a way to quantify this shift. Let's generate a penalty score that decreases as the data is rotated toward the x-axis.

We'll penalize all vertical y-axis values. Our penalty is based on the concept of squared distance, introduced in section 5. The penalty square equals the average squared y-value of `rotated_data`. Since y-values represent the distance to the x-axis, our penalty equals the average squared distance to the x-axis. When a rotated dataset moves closer to the x-axis, its average squared y-value decreases.

Given any `y_values` array, we can compute the penalty by running `sum([y ** 2 for y in y_values]) / y_values.size`. However, we can also compute the penalty by executing `y_values @ y_values / y.size`. The two results are identical, and the dot product computation is more efficient. Let's compare the penalty scores for `rotated_data` and `centered_data`.

Listing 14.7 Penalizing vertical y-values

```
data_labels = ['unrotated', 'rotated']
data_list = [centered_data, rotated_data]
for data_label, data in zip(data_labels, data_list):
    y_values = data[1]
    penalty = y_values @ y_values / y_values.size
    print(f"The penalty score for the {data_label} data is {penalty:.2f}")
```

The penalty score for the unrotated data is 519.82
The penalty score for the rotated data is 27.00

Rotating the data has reduced the penalty score by more than 20-fold. This reduction carries a statistical interpretation. We can link the penalty to the variance if we consider the following:

- Our penalty score equals the average squared y-value distance from 0 across a `y_values` array.
- `y_values.mean()` equals 0.
- Thus our penalty square equals the average squared y-value distance from the mean.
- The average square distance from the mean equals the variance.
- Our penalty score equals `y_values.var()`.

We've inferred that the penalty score equals the y-axis variance. Consequently, our data rotation has reduced the y-axis variance by more than 20-fold. Let's confirm.

Listing 14.8 Equating penalties with y-axis variance

```
for data_label, data in zip(data_labels, data_list):
    y_var = data[1].var()
    penalty = data[1] @ data[1] / data[0].size
    assert round(y_var, 14) == round(penalty, 14) ←
    print(f"The y-axis variance for the {data_label} data is {y_var:.2f}")
```

Rounds to take floating-point errors into account

The y-axis variance for the unrotated data is 519.82
The y-axis variance for the rotated data is 27.00

We can score rotations based on variance. Rotating the data toward the x-axis reduces the variance along the y-axis. How does this rotation influence the variance along the x-axis? Let's find out.

Listing 14.9 Measuring rotational x-axis variance

```
for data_label, data in zip(data_labels, data_list):
    x_var = data[0].var()
    print(f"The x-axis variance for the {data_label} data is {x_var:.2f}")
```

The x-axis variance for the unrotated data is 27.00
The x-axis variance for the rotated data is 519.82

The rotation has completely flipped the x-axis variance and the y-axis variance. However, the total sum of variance values has remained unchanged. Total variance is conserved even after the rotation. Let's verify this fact.

Listing 14.10 Confirming the conservation of total variance

```
total_variance = centered_data[0].var() + centered_data[1].var()
assert total_variance == rotated_data[0].var() + rotated_data[1].var()
```

Conservation of variance allows us to infer the following:

- x-axis variance and y-axis variance can be combined into a single percentage score, where $x_values.var() / total_variance$ is equal to $1 - y_values.var() / total_variance$.
- Rotating the data toward the x-axis leads to an increase in the x-axis variance and an equivalent decrease in the y-axis variance. Decreasing the vertical dispersion by p percent increases the horizontal dispersion by p percent.

The following code confirms these conclusions.

Listing 14.11 Exploring the percent coverage of axis variance

```
for data_label, data in zip(data_labels, data_list):
    percent_x_axis_var = 100 * data[0].var() / total_variance
    percent_y_axis_var = 100 * data[1].var() / total_variance
    print(f"In the {data_label} data, {percent_x_axis_var:.2f}% of the "
          "total variance is distributed across the x-axis")
    print(f"The remaining {percent_y_axis_var:.2f}% of the total "
          "variance is distributed across the y-axis\n")
```

In the unrotated data, 4.94% of the total variance is distributed across the x-axis

The remaining 95.06% of the total variance is distributed across the y-axis

In the rotated data, 95.06% of the total variance is distributed across the x-axis

The remaining 4.94% of the total variance is distributed across the y-axis

Rotating the data toward the x-axis has increased the x-axis variance by 90 percentage points. Simultaneously, the rotation has reduced the y-axis variance by these same 90 percentage points.

Let's rotate `centered_data` even further until its distance to the x-axis is minimized. Minimizing the distance to the x-axis is equivalent to

- Minimizing the percent of total variance covered by the y-axis. This minimizes vertical dispersion.
- Maximizing the percent of total variance covered by the x-axis. This maximizes horizontal dispersion.

We rotate centered_data toward the x-axis by maximizing horizontal dispersion. That dispersion is measured over all angles ranging from 1 degree to 180 degrees. We visualize these measurements in a plot (figure 14.7). Additionally, we extract the rotation angle that maximizes the percent of x-axis coverage. Our code prints out that angle and percentage while also marking the angle in our plot.

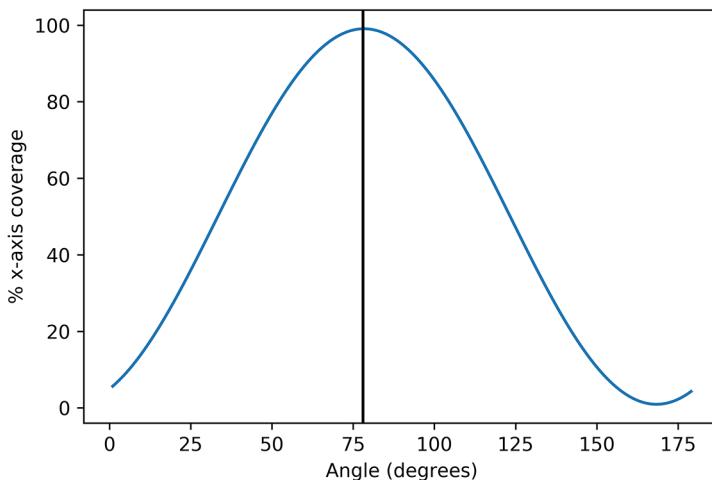


Figure 14.7 A plot of the rotation-angle vs. the percent of total variance covered by the x-axis. A vertical line marks the angle at which the x-axis variance is maximized. A rotation angle of 78.3 degrees shifts over 99% of the total variance to the x-axis. Rotating by that angle will allow us to dimensionally reduce our data.

Listing 14.12 Maximizing horizontal dispersion

Returns an array of angles ranging from 0 to 180, where each consecutive angle increases by 0.1 degrees

```
def rotate(angle, data=centered_data):
    angle = np.radians(-angle)
    rotation_matrix = np.array([[cos(angle), -sin(angle)],
                               [sin(angle), cos(angle)]])
    return rotation_matrix @ data
```

Rotates the data by input degrees. The data variable is preset to centered_data.

→ angles = np.arange(1, 180, 0.1)
x_variances = [(rotate(angle)[0].var()) for angle in angles]

Computes the x-axis variance for each rotation across every angle

Plots a vertical line through optimal_angle.

```
percent_x_variances = 100 * np.array(x_variances) / total_variance
optimal_index = np.argmax(percent_x_variances)
optimal_angle = angles[optimal_index]
plt.plot(angles, percent_x_variances)
plt.axvline(optimal_angle, c='k')
plt.xlabel('Angle (degrees)')
```

Computes the angle of rotation resulting in the maximum variance

```

plt.ylabel('% x-axis coverage')
plt.show()

max_coverage = percent_x_variances[optimal_index]
max_x_var = x_variances[optimal_index]

print("The horizontal variance is maximized to approximately "
      f"{int(max_x_var)} after a {optimal_angle:.1f} degree rotation.")
print(f"That rotation distributes {max_coverage:.2f}% of the total "
      "variance onto the x-axis.")

```

The horizontal variance is maximized to approximately 541 after a 78.3 degree rotation.

That rotation distributes 99.08% of the total variance onto the x-axis.

Rotating `centered_data` by 78.3 degrees will maximize the horizontal dispersion. At that rotation angle, 99.08% of the total variance will be distributed across the x-axis. Thus, we can expect the rotated data to mostly lie along the 1D axis line. Let's confirm by running the rotation and then plotting the results (figure 14.8).

Listing 14.13 Plotting rotated data with high x-axis coverage

```

best_rotated_data = rotate(optimal_angle)
plt.scatter(best_rotated_data[0], best_rotated_data[1])
plt.axhline(0, c='black')
plt.axvline(0, c='black')
plt.axis('equal')
plt.show()

```

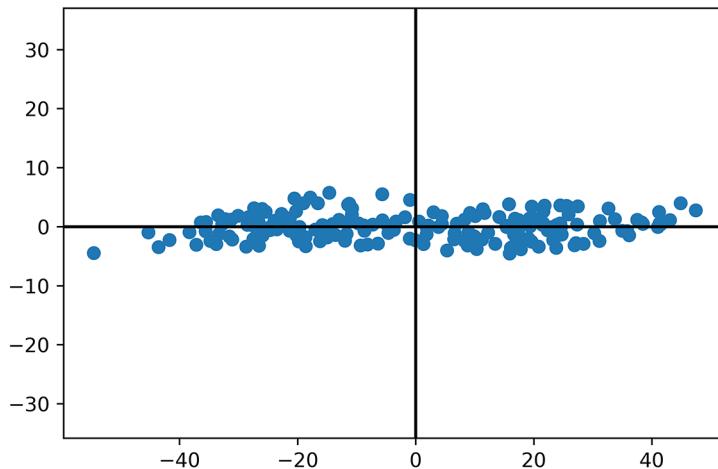


Figure 14.8 A plot of `centered_data` rotated by 78.3 degrees. This rotation maximizes variance along the x-axis and minimizes variance along the y-axis. Less than 1% of total variance lies along the y-axis. Thus, we can delete the y-coordinate with minimal information loss.

Most of the data lies close to the x-axis. The data's dispersion is maximized in that horizontal direction. Highly dispersed points are, by definition, highly separated. Separated points are easier to distinguish from each other. By contrast, the dispersion along our vertical y-axis has been minimized. Vertically, the data points are difficult to distinguish. Consequently, we can delete all y-axis coordinates with minimal information loss. That deletion should account for less than 1% of the total variance, so the remaining x-axis values will be sufficient to cluster our measurements.

Let's reduce `best_rotated_data` to 1D by disposing of the y-axis. Then we'll use the remaining 1D array to extract two clustering thresholds. The first threshold separates the small-sized customers from the medium-sized customers, and the second threshold separates the medium-sized customers from the large-sized customers. Together, the two thresholds separate our 180 customers into three equally sized clusters.

Listing 14.14 Reducing the rotated data to 1D for the purposes of clustering

```
x_values = best_rotated_data[0]
sorted_x_values = sorted(x_values)
cluster_size = int(x_values.size / 3)
small_cutoff = max(sorted_x_values[:cluster_size])
large_cutoff = min(sorted_x_values[-cluster_size:])
print(f"A 1D threshold of {small_cutoff:.2f} separates the small-sized "
      "and medium-sized customers.")
print(f"A 1D threshold of {large_cutoff:.2f} separates the medium-sized "
      "and large-sized customers.")

A 1D threshold of -14.61 separates the small-sized and medium-sized
customers.
A 1D threshold of 15.80 separates the medium-sized and large-sized customers.
```

We can visualize our thresholds by utilizing them to vertically slice our `best_reduced_data` plot. The two slices split the plot into three segments, where each segment corresponds to a customer size. Next, we visualize the thresholds and the segments while coloring each segment (figure 14.9).

Listing 14.15 Plotting horizontal customer data separated into three segments

Takes as input a horizontally positioned customer dataset, segments the data using vertical thresholds, and plots each customer segment separately. This function is reused elsewhere in this section.

```
def plot_customer_segments(horizontal_2d_data):
    small, medium, large = [], [], []
    cluster_labels = ['Small', 'Medium', 'Large']
    for x_value, y_value in horizontal_2d_data.T:
        if x_value <= small_cutoff:
            small.append([x_value, y_value])
        elif small_cutoff < x_value < large_cutoff:
            medium.append([x_value, y_value])
        else:
            large.append([x_value, y_value])
```

1D x-value thresholds are utilized to segment the data.

```

for i, cluster in enumerate([small, medium, large]):           ←
    cluster_x_values, cluster_y_values = np.array(cluster).T
    plt.scatter(cluster_x_values, cluster_y_values,
                color=['g', 'b', 'y'][i],
                label=cluster_labels[i])

plt.axhline(0, c='black')
plt.axvline(large_cutoff, c='black', linewidth=3, linestyle='--')
plt.axvline(small_cutoff, c='black', linewidth=3, linestyle='--')
plt.axis('equal')
plt.legend()
plt.show()

plot_customer_segments(best_rotated_data)

```

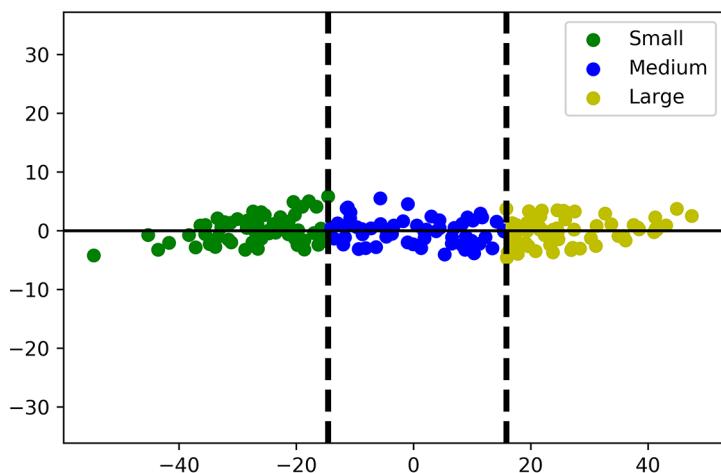


Figure 14.9 A horizontal plot of `centered_data`, segmented using two vertical thresholds. The segmentation splits the plot into three customer clusters: small, medium, and large. The one-dimensional x-axis is sufficient to extract these clusters.

Our 1D `x_values` array can sufficiently segment the customer data because it captures 99.08% of the data's variance. Consequently, we can use the array to reproduce 99.08% of our `centered_data` dataset (figure 14.10). We simply need to reintroduce the y-axis dimension by adding an array of zeros. Then we need to rotate the resulting array back to its original position. These steps are carried out next.

Listing 14.16 Reproducing 2D data from a 1D array

```

zero_y_values = np.zeros(x_values.size)           ← Returns a vector of zeros
reproduced_data = rotate(-optimal_angle, data=[x_values, zero_y_values])

```

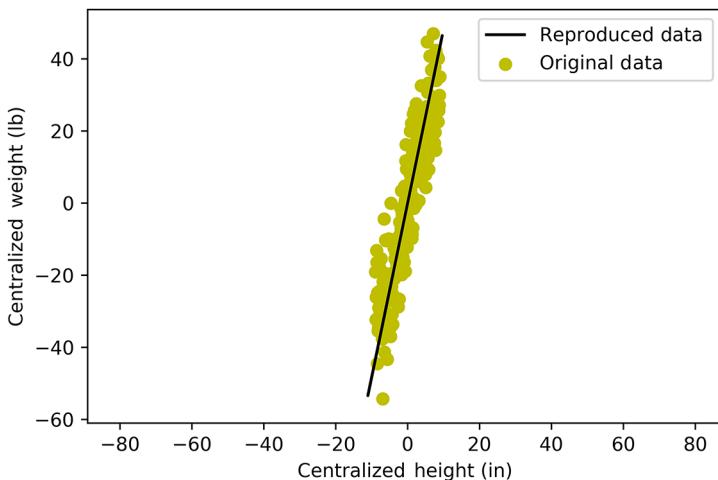


Figure 14.10 A plot of the reproduced data together with the original data points. Our `reproduced_data` array forms a single line that cuts through our `centered_data` scatter plot. That line represents the linear direction in which data variance is maximized. 99.08% of the total variance is covered by the `reproduced_data` line.

Let's plot `reproduced_data` together with our `centered_data` matrix to gauge the quality of the reproduction.

Listing 14.17 Plotting reproduced and original data

```
plt.plot(reproduced_data[0], reproduced_data[1], c='k',
         label='Reproduced Data')
plt.scatter(centered_data[0], centered_data[1], c='y',
            label='Original Data')
plt.axis('equal')
plt.legend()
plt.show()
```

The reproduced data forms a line that cuts directly through the middle of the `centered_data` scatter plot. The line represents the *first principal direction*, which is the linear direction in which the data's variance is maximized. Most 2D datasets contain two principal directions. The *second principal direction* is perpendicular to the first; it represents the remaining variance not covered by the first direction.

We can use the first principal direction to process the heights and weights of future customers. We'll assume that these customers originate from the same distribution that underlies our existing measurements data. If so, then their centralized heights and weights also lie along the first principal direction seen in figure 14.10. That alignment will eventually allow us to segment the new customer data using our existing thresholds.

Let's explore this scenario more concretely by simulating new customer measurements. Then we'll centralize and plot our measurement data (figure 14.11). We also plot a line representing the first principal direction—we expect the plotted measurements to align with that directional line.

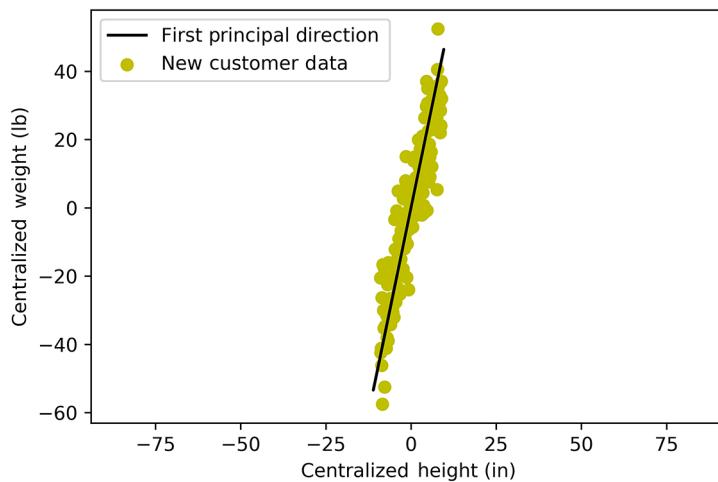


Figure 14.11 A centralized plot containing new customer data together with the principal direction from our original customer dataset. The principal direction cuts directly through a previously unseen data plot. That direction's angle with the x-axis is known. Thus, we can confidently flip that data on its side for the purposes of clustering.

NOTE The first principal direction intersects with the origin. Thus, we need to ensure that our new customer data also intersects with the origin, for alignment purposes. Consequently, we must centralize that data.

Listing 14.18 Simulating and plotting new customer data

```
np.random.seed(1)
new_heights = np.arange(60, 78, .11)           ← Separates all new heights by
random_fluctuations = np.random.normal(scale=10, size=new_heights.size)   0.11 inches to minimize overlap
new_weights = 4 * new_heights - 130 + random_fluctuations                     with previous heights
new_centered_data = np.array([new_heights - heights.mean(),
                             new_weights - weights.mean()])
plt.scatter(new_centered_data[0], new_centered_data[1], c='y',
            label='New Customer Data')
plt.plot(reproduced_data[0], reproduced_data[1], c='k',
            label='First Principal Direction')
plt.xlabel('Centralized Height (in)')
plt.ylabel('Centralized Weight (lb)')
plt.axis('equal')
plt.legend()
plt.show()
```

We assume that the new customer distribution is the same as the previously seen distribution. This allows us to utilize existing means for data centralization.

Our new customer data continues to lie along the first principal direction. That direction covers more than 99% of the data variance while also forming a 78.3-degree angle with the x-axis. Consequently, we can flip our new data on its side by rotating it 78.3 degrees. The resulting x-values cover more than 99% of the total variance. That high horizontal dispersion permits us to segment our customers without relying on y-value information. Our existing 1D segmentation thresholds should prove sufficient for that purpose.

Next, we position our new customer data horizontally and segment that data using our `plot_customer_segments` function (figure 14.12).

Listing 14.19 Rotating and segmenting our new customer data

```
new_horizontal_data = rotate(optimal_angle, data=new_centered_data)
plot_customer_segments(new_horizontal_data)
```

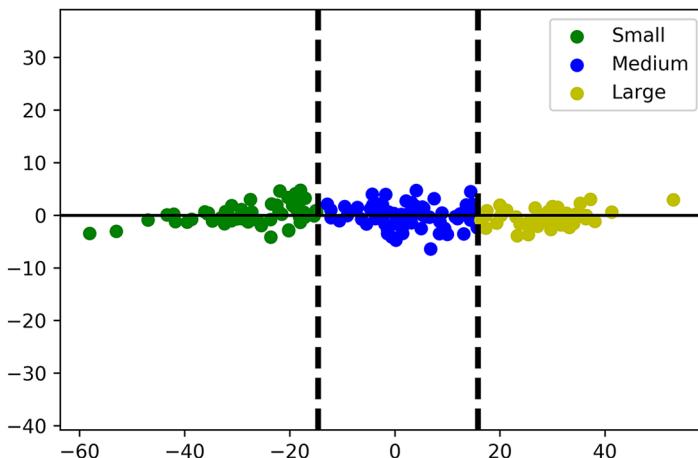


Figure 14.12 A horizontal plot of our new customer data. The data is segmented using two previously computed vertical thresholds. The segmentation splits the plot into three customer clusters: small, medium, and large. The one-dimensional x-axis is sufficient to extract these clusters.

We'll now briefly recap our observations. We can reduce any 2D array of customer measurements to one dimension by flipping the data on its side. The 1D horizontal x-values should be sufficient to cluster customers by size. Also, it's easier to flip the data when we know the principal direction along which the variance is maximized. Given the first principal direction, we dimensionally reduce customer data for easier clustering.

NOTE As an added bonus, dimension reduction allows us to simplify our customer database. Rather than storing both height and weight, we can just store the horizontal x-value. Reducing database storage from 2D to one dimension will speed up customer lookups and lower our storage costs.

Thus far, we have extracted the first principal direction by rotating our data to maximize the variance. Unfortunately, this technique does not scale to higher dimensions. Imagine if we analyze a 1,000-dimensional dataset; checking every angle across 1,000 different axes is not computationally feasible. Fortunately, there's an easier way to extract all principal directions. We just need to apply a scalable algorithm known as *principal component analysis* (PCA).

We explore PCA in the next few subsections. It is simple to implement, but it can be tricky to understand. We thus explore the algorithm in parts. We begin by running scikit-learn's PCA implementation. We apply PCA to several datasets to achieve better clustering and visualization. Then we probe the weaknesses of the algorithm by deriving PCA from scratch. Finally, we eliminate these weaknesses.

14.2 Dimension reduction using PCA and scikit-learn

The PCA algorithm adjusts a dataset's axes so that most of the variance is spread across a small number of dimensions. Consequently, not every dimension is required to distinguish between the data points. Simplified data distinction leads to simplified clustering. Hence, it is fortunate that scikit-learn provides a principal component analysis class called PCA. Let's import PCA from `sklearn.decomposition`.

Listing 14.20 Importing PCA from scikit-learn

```
from sklearn.decomposition import PCA
```

Running `PCA()` initializes a `pca_model` object, which is structured similarly to the scikit-learn `cluster_model` objects utilized in section 10. In that section, we created models capable of clustering inputted arrays. Now, we create a PCA model capable of flipping our measurements array onto its side.

Listing 14.21 Initializing a pca_model object

```
pca_object = PCA()
```

Using `pca_model`, we can horizontally flip a 2D data matrix by running `pca_model.fit_transform(data)`. That method call assigns axes to the matrix columns and subsequently reorients these axes to maximize the variance. However, in our `measurements` array, the axes are stored in the matrix rows. Thus, we need to swap the rows and columns by taking the transpose of the matrix. Running `pca_model.fit_transform(measurements.T)` returns a `pca_transformed_data` matrix. The first matrix column represents the x-axis across which the variance is maximized, and the second column represents the y-axis across which the variance is minimized. The plot of these two columns should resemble a cigar lying on its side. Let's verify: we run the `fit_transform` method on `measurements.T` and then plot the columns of the result (figure 14.13). As a reminder, the *i*th column of a NumPy matrix `M` can be accessed by running `M[:, i]`.

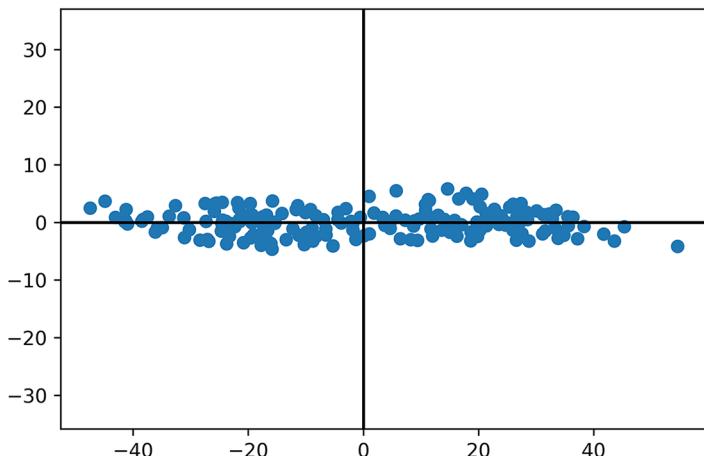


Figure 14.13 Plotted output from scikit-learn's PCA implementation. The plot is a mirror image of the horizontally positioned customer data from figure 14.8. PCA has reoriented that data so its variance lies primarily along the x-axis. Thus, the y-axis can be deleted with minimal information loss.

Listing 14.22 Running PCA using scikit-learn

```
pca_transformed_data = pca_object.fit_transform(measurements.T)
plt.scatter(pca_transformed_data[:,0], pca_transformed_data[:,1])
plt.axhline(0, c='black')
plt.axvline(0, c='black')
plt.axis('equal')
plt.show()
```

Our plot is a mirror of figure 14.8, with the y-values reflected across the y-axis. Running PCA on a 2D dataset is guaranteed to tip over that data so it lies horizontally on the x-axis. However, the actual reflection of data is not restricted to one particular orientation.

NOTE We can re-create our original horizontal plot by multiplying all y-values by -1 . Consequently, running `plot_customer_segments((pca_transformed_data * np.array([1, -1])).T)` generates a plot of segmented customer sizes that's equivalent to figure 14.9.

Even though our plotted data is oriented differently, its x-axis variance coverage should remain consistent with our previous observations. We can confirm using the `explained_variance_ratio_` attribute of `pca_object`. This attribute holds an array of fractional variances covered by each axis. Thus, `100 * pca_object.explained_variance_ratio_[0]` should equal the previously observed x-axis coverage of approximately 99.08%. Let's verify.

Listing 14.23 Extracting variance from scikit-learn's PCA output

The attribute is a NumPy array containing fractional coverage for each axis. Multiplying by 100 converts these fractions into percentages.

```
percent_variance_coverages = 100 * pca_object.explained_variance_ratio_ ←
x_axis_coverage, y_axis_coverage = percent_variance_coverages ←
print(f"The x-axis of our PCA output covers {x_axis_coverage:.2f}% of "
     "the total variance")
```

The x-axis of our PCA output covers 99.08% of the total variance

Each *i*th element of the array corresponds to
the variance coverage of the *i*th axis.

Our `pca_object` has maximized the x-axis variance by uncovering the dataset's two principal directions. These directions are stored as vectors in the `pca.components` attribute (which is itself a matrix). As a reminder, vectors are linear segments that point in a certain direction from the origin. Also, the first principal direction is a line that rises from the origin. Consequently, we can represent the first principal direction as a vector called the *first principal component*. We can access the first principal component of our data by printing `pca_object.components[0]`. Next, we output that vector, as well as its magnitude.

Listing 14.24 Outputting the first principal component

```
first_pc = pca_object.components_[0]
magnitude = norm(first_pc)
print(f"Vector {first_pc} points in a direction that covers "
      f"{x_axis_coverage:.2f}% of the total variance.")
print(f"The vector has a magnitude of {magnitude}")

Vector [-0.20223994 -0.979336] points in a direction that covers 99.08%
of the total variance.
The vector has a magnitude of 1.0
```

The first principal component is a unit vector with a magnitude of 1.0. It stretches one whole unit length from the origin. Multiplying the vector by a number stretches the magnitude out further. If we stretch it far enough, we can capture the entire principal direction of our data. In other words, we can stretch the vector until it completely skewers the interior of our cigar-shaped plot, like a corn dog on a stick. The visualized result should be identical to figure 14.10.

NOTE If vector `pc` is a principal component, then `-pc` is also a principal component. The `-pc` vector represents the mirror image of `pc`. Both vectors lie along the first principal direction, even though they point away from each other. Hence, `pc` and `-pc` can be used interchangeably during the dimension reduction process.

Next, we generate that plot (figure 14.14). First, we stretch our `first_pc` vector until it extends 50 units in both the positive and negative directions from the origin. We

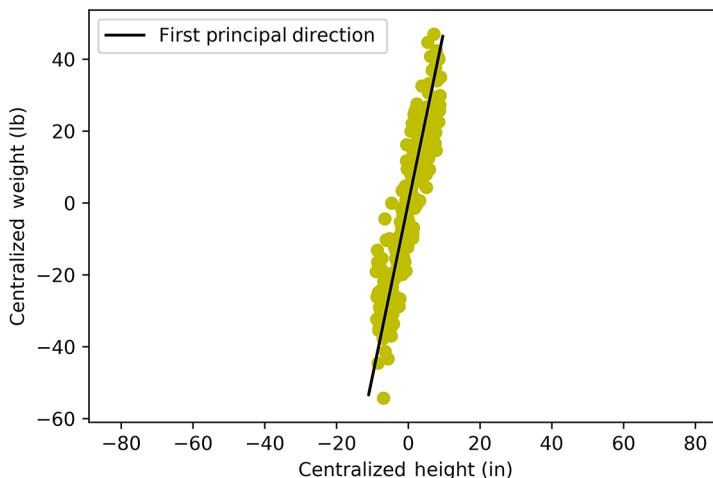


Figure 14.14 A centralized plot containing customer data together with the first principal direction. The direction has been plotted by stretching out the first principal component.

then plot the stretched segment along with our previously computed `centered_data` matrix. Later, we'll use the plotted segment to gain deeper insights into the workings of the PCA algorithm.

NOTE We plot `centered_data` and not `measurements` because `centered_data` is centered at the origin. Our stretched-out vector is also centered at the origin. This makes the centered matrix and the vector visually comparable.

Listing 14.25 Stretching a unit vector to cover the first principal direction

```
def plot_stretched_vector(v, **kwargs):
    plt.plot([-50 * v[0], 50 * v[0]], [-50 * v[1], 50 * v[1]], **kwargs)
    ←
    plt.plot(reproduced_data[0], reproduced_data[1], c='k',
              label='First Principal Direction')
    plt.scatter(centered_data[0], centered_data[1], c='y')
    plt.xlabel('Centralized Height (in)')
    plt.ylabel('Centralized Weight (lb)')
    plt.axis('equal')
    plt.legend()
    plt.show()
```

The function stretches out an inputted unit vector `v`. The stretched segment extends 50 units in both the positive and negative directions from the origin. Then the stretched segment is plotted. We reuse this function shortly.

We've used the first principal component to skewer our dataset along its first principal direction. In that same manner, we can stretch the other directional unit vector returned by the PCA algorithm. As stated earlier, most 2D datasets contain two principal directions. The second principal direction is perpendicular to the first, and its

vectorized representation is called the *second principal component*. That component is stored in the second row of our computed components matrix.

Why should we care about the second principal component? After all, it points in a direction that covers less than 1% of data variance. Nonetheless, that component has its uses. Both the first and second principal components share a special relationship with our data's x- and y-axes. Visually uncovering that relationship will make PCA easier to understand. Hence, we now stretch and plot both of the components in the components matrix. Additionally, we plot centered_data, as well as both our axes. The final visualization will provide us with valuable insights (figure 14.15).

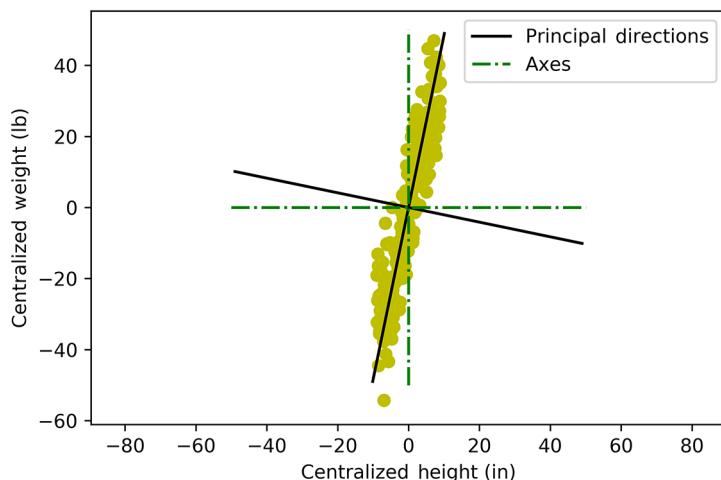


Figure 14.15 The first and second principal directions, plotted together with the customer data. These directions are perpendicular to each other. If we were to rotate the x- and y-axes by 78.3 degrees, they would align perfectly with the principal directions. Thus, swapping the axes with the principal directions reproduces the horizontal plot shown in figure 14.8.

Listing 14.26 Plotting principal directions, axes, and data information

```
principal_components = pca_object.components_
for i, pc in enumerate(principal_components):
    plot_stretched_vector(pc, c='k',
                          label='Principal Directions' if i == 0 else None)

for i, axis_vector in enumerate([np.array([0, 1]), np.array([1, 0])]):
    plot_stretched_vector(axis_vector, c='g', linestyle='-.',
                          label='Axes' if i == 0 else None)
```

Plots the x-axis and
y-axis by stretching out
two unit vectors (one vertical
and one horizontal) so that their
magnitudes align with the stretched principal
components. Consequently, the stretched axes and
stretched principal components are rendered visually comparable.

According to the plot, the two principal directions are essentially rotated versions of the x- and y-axes. Imagine if we rotated our two axes counterclockwise by 78.3 degrees. After the rotation, the x-axis and y-axis would align with the two principal components. The variances covered by these axes would equal 99.02 and 0.08%, respectively. Hence, this axis swap would reproduce the horizontal plot in figure 14.8.

NOTE Tilting your head to the left while staring at figure 14.15 will help you picture this outcome.

The aforementioned axis swap is known as a *projection*. Swapping our two axes for the principal directions is referred to as a *projection onto the principal directions*. Using trigonometry, we can show that the projection of `centered_data` onto the principal directions is equal to the matrix product of `centered_data` and the two principal components. In other words, `principal_components @ centered_data` repositions the dataset's x and y coordinates relative to the principal directions. The final output should equal `pca_transformed_data.T`. Let's confirm with the code in listing 14.27.

NOTE More generally, the dot product between the i th principal component and a centered data point projects that data point onto the i th principal direction. Thus, running `first_pc @ centered_data[i]` projects the i th data point onto the first principal direction. The result equals the x-value obtained when the x-axis is swapped with the first principal direction (`pca_transformed_data[i][0]`). In this manner, we can project multiple data points onto multiple principal directions using matrix multiplication.

Listing 14.27 Swapping standard axes for principal directions using projection

```
projections = principal_components @ centered_data
assert np.allclose(pca_transformed_data.T, projections)
```

The reoriented output of PCA is dependent on projection. In general terms, the PCA algorithm works as follows:

- 1 Centralize the input data by subtracting the mean from each data point.
- 2 Compute the dataset's principal components. The computation details are discussed later in this section.
- 3 Take the matrix product between the centralized data and the principal components. This swaps the data's standard axes for its principal directions.

Generally, an N -dimensional dataset has N principal directions (one for each axis). The k th principal direction maximizes the variance not covered by the first $k - 1$ directions. Thus, a 4D dataset has four principal directions: the first principal direction maximizes unidirectional dispersion, the second maximizes all unidirectional dispersion not covered by the first, and the final two cover all the remaining variance.

Here's where it gets interesting. Suppose we project a 4D dataset onto its four principal directions. The dataset's standard axes are thus swapped with its principal

directions. Under the right circumstances, two of the new axes will cover a good chunk of the variance. Consequently, the remaining axes could be discarded, with minimal information loss. Disposing of the two axes would reduce the 4D dataset to two dimensions. We would then be able to visualize that data in a 2D scatter plot. Ideally, the 2D plot would maintain enough dispersion for us to correctly identify data clusters. Let's explore an actual scenario where we visualize 4D data in two dimensions.

Key terminology

- *First principal direction*—The linear direction in which data dispersion is maximized. Swapping the x-axis with the first principal direction reorients the dataset to maximize its spread horizontally. The reorientation can allow for more straightforward 1D clustering.
- *Kth principal direction*—The linear direction that maximizes the variance not covered by the first $K - 1$ principal directions.
- *Kth principal component*—A unit vector representation of the K th principal direction. This vector can be utilized for directional projection.
- *Projection*—Projecting data onto a principal direction is analogous to swapping a standard axis with that direction. We can project a dataset onto its top K principal directions by taking the matrix product of the centralized dataset with its top K principal components.

14.3 Clustering 4D data in two dimensions

Imagine we are botanists studying flowers in a blooming meadow. We randomly select 150 flowers. For every flower, we record the following measurements:

- The length of a colorful petal
- The width of the colorful petal
- The length of a green leaf supporting the petal
- The width of the green leaf supporting the petal

These 4D flower measurements already exist and can be accessed using scikit-learn. We can obtain the measurements by importing `load_iris` from `sklearn.datasets`. Calling `load_iris() ['data']` returns a matrix containing 150 rows and 4 columns: each row corresponds to a flower, and each column corresponds to the leaf and petal measurements. Next, we load the data and print the measurements for a single flower. All recorded measurements are in centimeters.

Listing 14.28 Loading flower measurements from scikit-learn

```
from sklearn.datasets import load_iris
flower_data = load_iris()
flower_measurements = flower_data['data']
num_flowers, num_measurements = flower_measurements.shape
print(f"{num_flowers} flowers have been measured.")
```

```

print(f"{num_measurements} measurements were recorded for every flower.")
print("The first flower has the following measurements (in cm): "
      f"[{flower_measurements[0]}]")
print()
print(150, "flowers have been measured.")
print(4, "measurements were recorded for every flower.")
print("The first flower has the following measurements (in cm): [5.1 3.5 1.4 0.2]")

```

Given this matrix of flower measurements, our goals are as follows:

- We want to visualize our flower data in 2D space.
- We want to determine whether any clusters are present in the 2D visualization.
- We want to build a very simple model to distinguish between flower cluster types (assuming any clusters are found).

We start by visualizing the data. It is four-dimensional, but we want to plot it in 2D. Reducing the data to two dimensions requires that we project it onto its first and second principal directions. The remaining two directions can be discarded. Thus, our analysis only requires the first two principal components.

Using scikit-learn, we can limit PCA analysis to the top two principal components. We just need to run `PCA(n_components=2)` during the initialization of the `PCA` object. The initialized object will be capable of reducing input data to a two-dimensional projection. Next, we initialize a two-component `PCA` object and use `fit_transform` to reduce our flower measurements to 2D.

Listing 14.29 Reducing flower measurements to two dimensions

```

pca_object_2D = PCA(n_components=2)
transformed_data_2D = pca_object_2D.fit_transform(flower_measurements)

```

The computed `transformed_data_2D` matrix should be two-dimensional, containing just two columns. Let's confirm.

Listing 14.30 Checking the shape of a dimensionally reduced matrix

```

row_count, column_count = transformed_data_2D.shape
print(f"The matrix contains {row_count} rows, corresponding to "
      f"{row_count} recorded flowers.")
print(f"It also contains {column_count} columns, corresponding to "
      f"{column_count} dimensions.")

```

The matrix contains 150 rows, corresponding to 150 recorded flowers.
It also contains 2 columns, corresponding to 2 dimensions.

How much of the total data variance is covered by our outputted data matrix? We can find out using the `explained_variance_ratio_` attribute of `pca_object_2D`.

Listing 14.31 Measuring the variance coverage of a dimensionally reduced matrix

```
def print_2D_variance_coverage(pca_object):           ←
    percent_var_covers = 100 * pca_object.explained_variance_ratio_
    x_axis_coverage, y_axis_coverage = percent_var_covers
    total_coverage = x_axis_coverage + y_axis_coverage
    print(f"The x-axis covers {x_axis_coverage:.2f}% "
          "of the total variance")
    print(f"The y-axis covers {y_axis_coverage:.2f}% "
          "of the total variance")
    print(f"Together, the 2 axes cover {total_coverage:.2f}% "
          "of the total variance")

print_2D_variance_coverage(pca_object_2D)
```

Computes the variance coverage of the dimensionally reduced 2D dataset associated with `pca_object`. This function is reused elsewhere in the section.

```
The x-axis covers 92.46% of the total variance
The y-axis covers 5.31% of the total variance
Together, the 2 axes cover 97.77% of the total variance
```

Our dimensionally reduced matrix covers more than 97% of the total data variance. Thus, a scatter plot of `transformed_data_2D` should display most of the clustering patterns present in the dataset (figure 14.16).

Listing 14.32 Plotting flower data in 2D

```
plt.scatter(transformed_data_2D[:,0], transformed_data_2D[:,1])
plt.show()
```

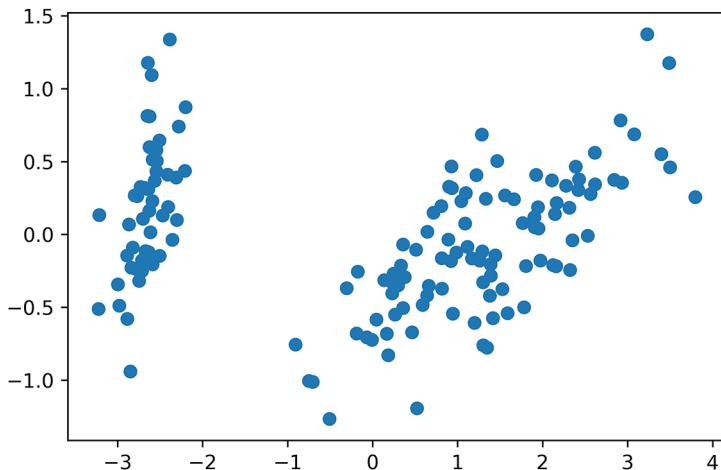


Figure 14.16 4D flower measurements plotted in 2D. The measurements were reduced to two dimensions using PCA. The reduced data covers more than 97% of the total variance. Our 2D plot is informative: two or three flower clusters are clearly visible.

Our flower data forms clusters when plotted in 2D. Based on the clustering, we can assume that two or three flower types are present. In fact, our measured data represents three unique species of flowers. This species information is stored in the `flower_data` dictionary. Next, we color our flower plot by species and verify that the colors fall in three distinct clusters (figure 14.17).

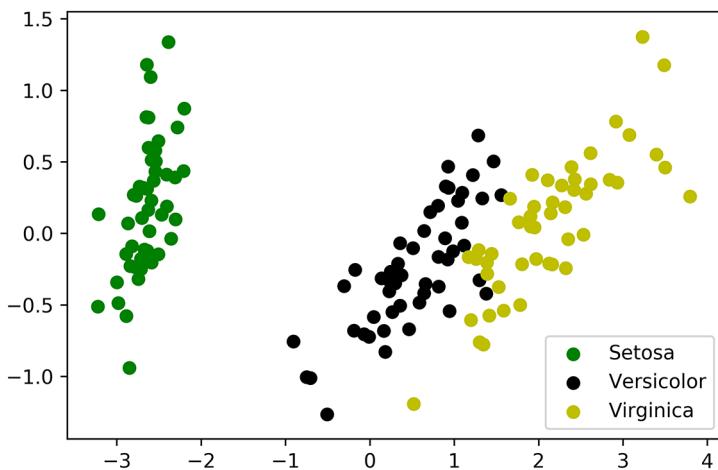


Figure 14.17 4D flower measurements plotted in 2D. Each plotted flower point is colored based on its species. The three species fall into three clusters. Thus, our 2D reduction correctly captures the signal required to distinguish between species.

Listing 14.33 Coloring plotted data by flower species

```

Plots dimensionally reduced flower data while coloring it by
species. This function is reused elsewhere in the section.           ↗ Returns the names
                                                               ↗ of the three flower
                                                               species in our dataset

def visualize_flower_data(dim_reduced_data):
    species_names = flower_data['target_names']
    for i, species in enumerate(species_names):
        species_data = np.array([dim_reduced_data[j]
                                for j in range(dim_reduced_data.shape[0])
                                if flower_data['target'][j] == i]).T
        plt.scatter(species_data[0], species_data[1], label=species.title(),
                    color=['g', 'k', 'y'][i])
    plt.legend()
    plt.show()

visualize_flower_data(transformed_data_2D)

```

Plots each species using a unique color ↗ Extracts just those coordinates associated with a particular species. For filtering purposes, we use `flower_data[target]`, which maps to a list of species IDs. The IDs correspond to the three species names. If the j th flower corresponds to `species_name[i]`, then its species ID equals j .

For the most part, the three species are spatially distinct. *Versicolor* and *Virgincia* share a bit of overlap, implying that they have similar petal properties. On the other hand, *Setosa* forms an entirely separate cluster. A vertical x-value threshold of -2 is sufficient to isolate *Setosa* from all other species. Hence, we can define a very simple *Setosa* detection function. The function takes as input a four-element array called `flower_sample`, which holds four petal measurements. The function will do the following:

- 1 Centralize the sample by subtracting the mean of `flower_measurements` from `flower_sample`. This mean is stored as an attribute in `pca_object_2D`. It is equal to `pca_object_2D.mean_`.
- 2 Project the centralized sample onto the first principal direction by taking its dot product with the first principal component. As a reminder, the first principal component is stored in `pca_object_2D.components_[0]`.
- 3 Check if the projected value is less than -2 . If so, then the flower sample will be treated as a possible *Setosa* species.

NOTE Our *Setosa* detection function doesn't take into account any flower species beyond the three that we've recorded. However, the function should still sufficiently analyze new flowers in our meadow, where no additional species have been observed.

Next, we define the `detect_setosa` function and then analyze a flower sample with measurements (in cm) of $[4.8, 3.7, 1.2, 0.24]$.

Listing 14.34 Defining a Setosa detector based on dimensionally reduced data

```
def detect_setosa(flower_sample):
    centered_sample = flower_sample - pca_object_2D.mean_
    projection = pca_object_2D.components_[0] @ centered_sample
    if projection < -2:
        print("The sample could be a Setosa")
    else:
        print("The sample is not a Setosa")

new_flower_sample = np.array([4.8, 3.7, 1.2, 0.24])
detect_setosa(new_flower_sample)

The sample could be a Setosa
```

The flower sample could be a *Setosa* according to our simple threshold analysis, which was made possible by PCA. Various benefits of PCA include the following:

- Visualization of complex data.
- Simplified data classification and clustering.
- Simplified classification.
- Decreased memory usage. Reducing the data from four to two dimensions halves the number of bytes required to store the data.

- Faster computations. Reducing the data from four to two dimensions speeds up the computation time required to compute a similarity matrix fourfold.

So are we ready to use PCA to cluster our text data? Unfortunately, the answer is no. We must first discuss and address certain flaws that are inherent to the algorithm.

Common scikit-learn PCA methods

- `pca_object = PCA()`—Creates a PCA object capable of reorienting input data so that its axes align with its principal directions.
- `pca_object = PCA(n_components=K)`—Creates a PCA object capable of reorienting input data so that K of its axes align with the top K principal directions. All other axes are ignored. This reduces data to K dimensions.
- `pca_transformed_data = pca_object.fit_transform(data)`—Executes PCA on inputted data using an initialized PCA object. The `fit_transform` method assumes that the columns of the data matrix correspond to spatial axes. The axes are subsequently aligned with the data's principal direction. This result is stored in the `pca_transformed_data` matrix.
- `pca_object.explained_variance_ratio_`—Returns the fractional variance coverage associated with each principal direction of a fitted PCA object. Each i th element corresponds to fractional variance coverage along the i th principal direction.
- `pca_object.mean_`—Returns the mean of the input data, which has been fitted to the PCA object.
- `pca_object.components_`—Returns the principal components of the input data, which have been fitted to the PCA object. Each i th row of the `components_` matrix corresponds to the i th principal component. Running `pca_object.components_[i] @ (data[j] - pca_object.mean_)` projects the j th data point onto the i th principal component. The projected output equals `pca_transformed_data[j][i]`.

14.3.1 Limitations of PCA

PCA does have some serious limitations. It is overly sensitive to units of measurement. For example, our flower measurements are all in centimeters, but we can imagine converting the first axis into millimeters by running `10 * flower_measurements[0]`. The information content of that axis should not change; however, its variance will shift. Let's convert the axis units to evaluate how the variance is affected.

Listing 14.35 Measuring the effect of unit change on axis variance

```
first_axis_var = flower_measurements[:,0].var()
print(f"The variance of the first axis is: {first_axis_var:.2f}")

flower_measurements[:,0] *= 10
first_axis_var = flower_measurements[:,0].var()
print("We've converted the measurements from cm to mm.\nThat variance "
      "now equals {first_axis_var:.2f}")
```

The variance of the first axis is: 0.68
 We've converted the measurements from cm to mm.
 That variance now equals 68.11

Our variance has increased 100-fold. Now the first axis variance dominates our dataset. Consider the consequences of running PCA on these modified flower measurements: PCA will attempt to find the axis where variance is maximized. This, of course, will yield the first axis, where variance has increased from 0.68 to 68. Consequently, PCA will project all the data onto the first axis. Our reduced data will collapse to one dimension! We can prove this by refitting `pca_object_2D` to `flower_measurements` and then printing the variance coverage.

Listing 14.36 Measuring the effect of unit change on PCA

```
pca_object_2D.fit_transform(flower_measurements)
print_2D_variance_coverage(pca_object_2D)
```

The x-axis covers 98.49% of the total variance
 The y-axis covers 1.32% of the total variance
 Together, the 2 axes cover 99.82% of the total variance

← Refits our PCA object to the updated flower_measurements data

More than 98% of the variance now lies along a single axis. Previously, two dimensions were required to capture 97% of the data variance. Clearly, we've introduced an error into our data. How do we resolve it? One obvious solution is to ensure that all the axes share the same units of measurements. However, such a practical approach is not always possible. Sometimes the units of measurement are just not available. Other times, the axes correspond to different measurement types (such as length and weight), so the units are incompatible. What should we do?

Let's consider the root cause of our variance shift. We've made the values in `flower_measurements[:, 0]` larger, so their variance also grew larger. Differences in axis variance are caused by differences in value size. In the previous section, we were able to eliminate such differences in size using normalization. As a reminder, during normalization, a vector is divided by its magnitude. This produces a unit vector whose magnitude equals 1.0. Thus, if we normalize our axes, all the axes values will lie between 0 and 1. The dominance of the first axis will thus be eliminated. Let's normalize `flower_measurements` and then reduce the normalized data to two dimensions. The resulting 2D variance coverage should once again approximate 97%.

Listing 14.37 Normalizing data to eliminate measurement-unit differences

```
for i in range(flower_measurements.shape[1]):
    flower_measurements[:, i] /= norm(flower_measurements[:, i])

transformed_data_2D = pca_object_2D.fit_transform(flower_measurements)
print_2D_variance_coverage(pca_object_2D)
```

The x-axis covers 94.00% of the total variance
 The y-axis covers 3.67% of the total variance
 Together, the 2 axes cover 97.67% of the total variance

Normalization has slightly modified our data. Now the first principal direction covers 94% of the total variance rather than 92.46%. Meanwhile, the second principal component covers 3.67% of total variance rather than 5.31%. Despite these changes, the total 2D variance coverage still sums to approximately 97%. We replot the PCA output to confirm that the 2D clustering patterns remain unchanged (figure 14.18).

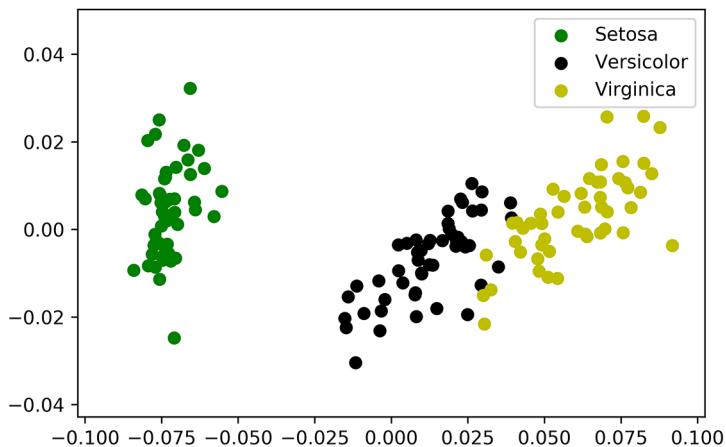


Figure 14.18 4D normalized flower measurements plotted in 2D. Each plotted flower point is colored based on its species. The three species fall into three clusters. Thus, our 2D reduction correctly captures the signal required to distinguish between species.

Listing 14.38 Plotting 2D PCA output after normalization

```
visualize_flower_data(transformed_data_2D)
```

Our plot is slightly different from our previous observations. However, the three species of flowers continue to separate into three clusters, and *Setosa* remains spatially distinct from other species. Normalization has retained existing cluster separations while eliminating the error caused by unit differences.

Unfortunately, normalization does lead to unintended consequences. Our normalized axis values now lie between 0 and 1, so each axis mean is likewise between 0 and 1. All values lie less than 1 unit from their mean. This is a problem: PCA requires us to subtract the mean from each axis value to centralize our data. Then the centralized matrix is multiplied by the principal components to realign the axes. Regrettably, data centralization is not always achievable due to floating-point errors. It's computationally difficult to subtract similar values with 100% precision, so it is difficult to subtract the mean from a value that is very close to that mean. For example, suppose we analyze an array containing two data points, $1 + 1e-3$ and $1 - 1e-3$. The mean of the array

is equal to 1. Subtracting 1 from the array should lead to a centralized mean of 0, but the actual mean will not equal 0 due to an error, illustrated next.

Listing 14.39 Illustrating errors caused by values proximate to their mean

```
data = np.array([1 + 1e-3, 1 - 1e-3])
mean = data.mean()
assert mean == 1
centralized_data = data - 2 * [mean]
assert centralized_data.mean() != 0
print(f"Actual mean is equal to {centralized_data.mean()}")
```

Actual mean is equal to -5.551115123125783e-17

The mean of the centralized data does not equal 0 as intended.

We cannot reliably centralize data that lies close to the mean. Hence, we can't reliably execute PCA on normalized data. What should we do?

A solution to our problem does exist. However, to derive it, we must dive deep into the guts of the PCA algorithm. We must learn how to compute the principal components from scratch without rotation. That computation process is a bit abstract, but it can be understood without studying advanced mathematics. Once we derive the PCA algorithm, we'll be able to modify it slightly. That minor modification will completely bypass data centralization. The modified algorithm, known as *singular value decomposition* (SVD), will allow us to efficiently cluster text data.

NOTE If you're not interested in the SVD derivation, you can skip ahead to the final subsection. It describes SVD usage in scikit-learn.

14.4 Computing principal components without rotation

In this subsection, we learn how to extract the principal components from scratch. To better illustrate the extraction process, we'll visualize our component vectors. Of course, vectors are easier to plot when they are two-dimensional. Thus, we start by revisiting our customer measurements dataset whose principal components are 2D. As a reminder, we've computed the following outputs for this data:

- `centralized_data`—A centralized version of the `measurements` dataset. The mean of `centralized_data` is `[0 0]`.
- `first_pc`—The first principal component of the `measurements` dataset. It is a two-element array.

As we've discussed, `first_pc` is a unit vector that points in the first principal direction. That direction maximizes the dispersion of the data. Earlier, we discovered the first principal direction by rotating our 2D dataset. The goal of that rotation was to either maximize the x-axis variance or minimize the y-axis variance. Previously, we computed axis variances using vector dot-product operations. However, we can measure axis variance more efficiently using matrix multiplication. More importantly, by storing all our variances in a matrix, we can extract our components without rotation. Let's consider the following:

- We've already shown that the variance of an axis array equals `axis @ axis / axis.size` (see listing 14.8).
- Thus, the variance of axis i in `centered_data` equals `centered_data[i] @ centered_data[i] / centered_data.shape[1]`.
- Consequently, running `centered_data @ centered_data.T / centered_data.shape[1]` will produce a matrix m , where $m[i][i]$ equals the variance of axis i .

Essentially, we can compute all axis variances in a single matrix operation. We just need to multiply our matrix with its transpose while also dividing by data size. This produces a new matrix, called the *covariance matrix*. The diagonal of a covariance matrix stores the variance along each axis.

NOTE The non-diagonal elements of the covariance matrix have informative properties as well: they determine the direction of the linear slope between two axes. In `centered_data`, the slope between x and y is positive. Therefore, the non-diagonal elements in `centered_data @ centered_data.T` are also positive.

Next we compute the covariance matrix of `centered_data` and assign it to variable `cov_matrix`. Then we confirm that `cov_matrix[i][i]` equals the variance of the i th axis for every i .

Listing 14.40 Computing a covariance matrix

```
cov_matrix = centered_data @ centered_data.T / centered_data.shape[1]
print(f"Covariance matrix:\n {cov_matrix}")
for i in range(centered_data.shape[0]):
    variance = cov_matrix[i][i]
    assert round(variance, 10) == round(centered_data[i].var(), 10)      ←
    Covariance matrix:
    [[ 26.99916667 106.30456732]
     [106.30456732 519.8206294 ]]
```

Rounds because of
floating-point errors

The covariance matrix and the principal components share a very special (and useful) relationship: the normalized product of a covariance matrix and a principal component equals that principal component! Thus, normalizing `cov_matrix @ first_pc` produces a vector that's identical to `first_pc`. Let's illustrate this relationship by plotting `first_pc` and the normalized product of `cov_matrix` and `first_pc` (figure 14.19).

NOTE When taking the product of a matrix and a vector, we treat the vector as a single-column table. Thus, a vector with x elements is treated as a matrix with x rows and one column. Once the vector is reconfigured as a matrix, standard matrix multiplication is carried out. That multiplication produces another single-column matrix, which is equivalent to a vector. Consequently, the product of matrix M and vector v is equal to `np.array([row @ v for row in M])`.

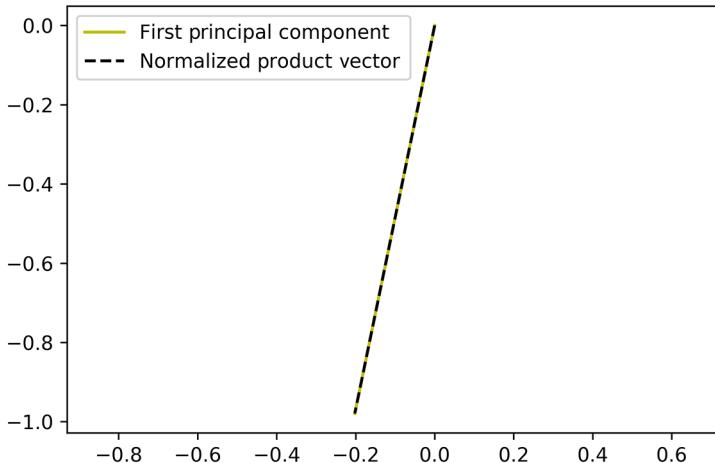


Figure 14.19 A plot of `first_pc` together with the normalized product of `cov_matrix` and `first_pc`. The two plotted vectors are identical. The product of the covariance matrix and the principal component points in the same direction as that principal component.

Listing 14.41 Exposing the relationship between `cov_matrix` and `first_pc`

```
def plot_vector(vector, **kwargs):
    plt.plot([0, vector[0]], [0, vector[1]], **kwargs)

plot_vector(first_pc, c='y', label='First Principal Component')
product_vector = cov_matrix @ first_pc
product_vector /= norm(product_vector)
plot_vector(product_vector, c='k', linestyle='--',
            label='Normalized Product Vector')

plt.legend()
plt.axis('equal')
plt.show()
```

This helper function plots a 2D vector as a line segment stretching from the origin.

The two plotted vectors are identical! The matrix-vector product of `cov_matrix` and `first_pc` points in the same direction as `first_pc`. Thus, by definition, `first_pc` is an *eigenvector* of `cov_matrix`. An eigenvector of a matrix satisfies the following special property: the product of the matrix and the eigenvector points in the same direction as the eigenvector. The direction will not shift, no matter how many times we take the product. So, `cov_matrix @ product_vector` points in the same direction as `product_vector`, and the angle between the vectors equals zero. Let's confirm.

Listing 14.42 Computing the angle between eigenvector products

```
product_vector2 = cov_matrix @ product_vector
product_vector2 /= norm(product_vector2)
```

```

cosine_similarity = product_vector @ product_vector2 ←
→ angle = np.degrees(np.arccos(cosine_similarity))
print(f"The angle between vectors equals {angle:.2f} degrees")

```

The angle between vectors equals 0.00 degrees

Taking the arccosine of the cosine similarity returns the angle between vectors.

Both vectors are unit vectors. As discussed in section 13, the dot product of two unit vectors equals the cosine of their angle.

The product of a matrix and its eigenvector maintains the eigenvector's direction. However, in most cases, it alters the eigenvector's magnitude. For example, `first_pc` is an eigenvector with a magnitude of 1. Multiplying `first_pc` by the covariance matrix will increase that magnitude x-fold. Let's print the actual shift in magnitude by running `norm(cov_matrix @ first_pc)`.

Listing 14.43 Measuring the shift in magnitude

```

new_magnitude = norm(cov_matrix @ first_pc)
print("Multiplication has stretched the first principal component by "
      f"approximately {new_magnitude:.1f} units.")

```

Multiplication has stretched the first principal component by approximately 541.8 units

Multiplication has stretched `first_pc` by 541.8 units along the first principal direction. Thus, `cov_matrix @ first_pc` equals $541.8 * \text{first_pc}$. Given any matrix `m` and its eigenvector `eigen_vec`, the product of `m` and `eigen_vec` always equals `v * eigen_vec`, where `v` is a numeric value formally called the *eigenvalue*. Our `first_pc` eigenvector has an eigenvalue of approximately 541. This value may seem familiar because we have seen it before: early in this section, we printed the maximized x-axis variance, which was approximately 541. Thus, our eigenvalue equals the variance along the first principal direction. We can confirm by calling `(centered_data @ first_pc).var()`.

Listing 14.44 Comparing an eigenvalue to the variance

```

variance = (centered_data.T @ first_pc).var()
direction1_var = projections[0].var()
assert round(variance, 10) == round(direction1_var, 10)
print("The variance along the first principal direction is approximately"
      f" {variance:.1f}")

```

The variance along the first principal direction is approximately 541.8

Let's recap our observations:

- The first principal component is an eigenvector of the covariance matrix.
- The associated eigenvalue equals the variance along the first principal direction.