DATA SCIENCE SERIES

DATA SCIENCE

A First Introduction with Python

TIFFANY TIMBERS
TREVOR CAMPBELL
MELISSA LEE
JOEL OSTBLOM
LINDSEY HEAGY

A **Chapman & Hall** Book

CRC Press
Taylor & Francis Group

Data Science

Data Science: A First Introduction with Python focuses on using the Python programming language in Jupyter notebooks to perform data manipulation and cleaning, create effective visualizations, and extract insights from data using classification, regression, clustering, and inference. It emphasizes workflows that are clear, reproducible, and shareable, and includes coverage of the basics of version control. Based on educational research and active learning principles, the book uses a modern approach to Python and includes accompanying autograded Jupyter worksheets for interactive, self-directed learning. The text will leave readers well-prepared for data science projects. It is designed for learners from all disciplines with minimal prior knowledge of mathematics and programming. The authors have honed the material through years of experience teaching thousands of undergraduates at the University of British Columbia.

Key Features:

- Includes autograded worksheets for interactive, self-directed learning.
- Introduces readers to modern data analysis and workflow tools such as Jupyter notebooks and GitHub, and covers cutting-edge data analysis and manipulation Python libraries such as pandas, scikit-learn, and altair.
- Is designed for a broad audience of learners from all backgrounds and disciplines.

CHAPMAN & HALL/CRC DATA SCIENCE SERIES

Reflecting the interdisciplinary nature of the field, this book series brings together researchers, practitioners, and instructors from statistics, computer science, machine learning, and analytics. The series will publish cutting-edge research, industry applications, and textbooks in data science.

The inclusion of concrete examples, applications, and methods is highly encouraged. The scope of the series includes titles in the areas of machine learning, pattern recognition, predictive analytics, business analytics, Big Data, visualization, programming, software, learning analytics, data wrangling, interactive graphics, and reproducible research.

Recently Published Titles

Big Data Analytics
A Guide to Data Science Practitioners Making the Transition to Big Data
Ulrich Matter

Data Science for Sensory and Consumer Scientists
Thierry Worch, Julien Delarue, Vanessa Rios De Souza and John Ennis

Data Science in Practice
Tom Alby

Introduction to NFL Analytics with R
Bradley J. Congelio

Soccer Analytics
An Introduction Using R
Clive Beggs

Spatial Statistics for Data Science
Theory and Practice with R
Paula Moraga

Research Software Engineering
A Guide to the Open Source Ecosystem
Matthias Bannert

The Data Preparation Journey
Finding Your Way With R
Martin Hugh Monkman

Getting (more out of) Graphics
Practice and Principles of Data Visualisation
Antony Unwin

Introduction to Data Science
Data Wrangling and Visualization with R Second Edition
Rafael A. Irizarry

Data Science
A First Introduction with Python
Tiffany Timbers, Trevor Campbell, Melissa Lee, Joel Ostblom and Lindsey Heagy

For more information about this series, please visit: https://www.routledge.com/Chapman--HallCRC-Data-Science-Series/book-series/CHDSS

Data Science

A First Introduction with Python

Tiffany Timbers, Trevor Campbell,
Melissa Lee, Joel Ostblom and Lindsey Heagy

CRC Press
Taylor & Francis Group
Boca Raton London New York

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

Designed cover image: © Jaki King

First edition published 2025
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton FL 33431

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

ISBN: 978-1-032-57219-2 (hbk)
ISBN: 978-1-032-57223-9 (pbk)
ISBN: 978-1-003-43839-7 (ebk)

DOI: 10.1201/9781003438397

Typeset in LM Roman
by KnowledgeWorks Global Ltd.

Publisher's note: This book has been prepared from camera-ready copy provided by the authors.

Contents

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

Preface

This textbook aims to be an approachable introduction to the world of data science. In this book, we define **data science** as the process of generating insight from data through **reproducible** and **auditable** processes. If you analyze some data and give your analysis to a friend or colleague, they should be able to rerun the analysis from start to finish and get the same result you did (*reproducibility*). They should also be able to see and understand all the steps in the analysis, as well as the history of how the analysis developed (*auditability*). Creating reproducible and auditable analyses allows both you and others to easily double-check and validate your work.

At a high level, in this book, you will learn how to

1. identify common problems in data science, and
2. solve those problems with reproducible and auditable workflows.

Fig. 1 summarizes what you will learn in each chapter of this book. Throughout, you will learn how to use the Python programming language[1] to perform all the tasks associated with data analysis. You will spend the first four chapters learning how to use Python to load, clean, wrangle (i.e., restructure the data into a usable format), and visualize data while answering descriptive and exploratory data analysis questions. In the next six chapters, you will learn how to answer predictive, exploratory, and inferential data analysis questions with common methods in data science, including classification, regression, clustering, and estimation. In the final chapters you will learn how to combine Python code, formatted text, and images in a single coherent document with Jupyter, use version control for collaboration, and install and configure the software needed for data science on your own computer. If you are reading this book as part of a course that you are taking, the instructor may have set up all of these tools already for you; in this case, you can continue on through the book reading the chapters in order. But if you are reading this independently, you may want to jump to these last three chapters early before going on to make sure your computer is set up in such a way that you can try out the example code that we include throughout the book.

[1] https://www.python.org/

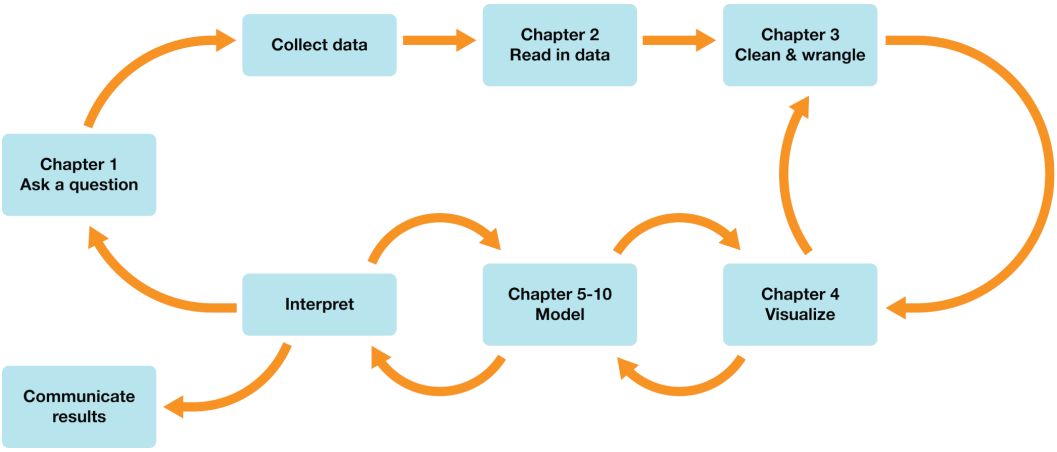


FIGURE 1 Where are we going?

Each chapter in the book has an accompanying worksheet that provides exercises to help you practice the concepts you will learn. We strongly recommend that you work through the worksheet when you finish reading each chapter before moving on to the next chapter. All of the worksheets are available at https://worksheets.python.datasciencebook.ca; the "Exercises" section at the end of each chapter points you to the right worksheet for that chapter. For each worksheet, you can either launch an interactive version of the worksheet in your browser by clicking the "launch binder" button or preview a non-interactive version of the worksheet by clicking "view worksheet". If you instead decide to download the worksheet and run it on your own machine, make sure to follow the instructions for computer setup found in Chapter 13. This will ensure that the automated feedback and guidance that the worksheets provide will function as intended.

Foreword

Roger D. Peng

Johns Hopkins Bloomberg School of Public Health

2023-11-30

The field of data science has expanded and grown significantly in recent years, attracting excitement and interest from many different directions. The demand for introductory educational materials has grown concurrently with the growth of the field itself, leading to a proliferation of textbooks, courses, blog posts, and tutorials. This book is an important contribution to this fast-growing literature, but given the wide availability of materials, a reader should be inclined to ask, "What is the unique contribution of *this* book?" In order to answer that question, it is useful to step back for a moment and consider the development of the field of data science over the past few years.

When thinking about data science, it is important to consider two questions: "What is data science?" and "How should one do data science?" The former question is under active discussion among a broad community of researchers and practitioners and there does not appear to be much consensus to date. However, there seems a general understanding that data science focuses on the more "active" elements—data wrangling, cleaning, and analysis—of answering questions with data. These elements are often highly problem-specific and may seem difficult to generalize across applications. Nevertheless, over time we have seen some core elements emerge that appear to repeat themselves as useful concepts across different problems. Given the lack of clear agreement over the definition of data science, there is a strong need for a book like this one to propose a vision for what the field is and what the implications are for the activities in which members of the field engage.

The first important concept addressed by this book is tidy data, which is a format for tabular data formally introduced to the statistical community in a 2014 paper by Hadley Wickham. Although originally popularized within the R programming language community via the Tidyverse package collection, the tidy data format is a language-independent concept that facilitates the application of powerful generalized data cleaning and wrangling tools. The second key concept is the development of workflows for reproducible and auditable

data analyses. Modern data analyses have only grown in complexity due to the availability of data and the ease with which we can implement complex data analysis procedures. Furthermore, these data analyses are often part of decision-making processes that may have significant impacts on people and communities. Therefore, there is a critical need to build reproducible analyses that can be studied and repeated by others in a reliable manner. Statistical methods clearly represent an important element of data science for building prediction and classification models and for making inferences about unobserved populations. Finally, because a field can succeed only if it fosters an active and collaborative community, it has become clear that being fluent in the tools of collaboration is a core element of data science.

This book takes these core concepts and focuses on how one can apply them to *do* data science in a rigorous manner. Students who learn from this book will be well-versed in the techniques and principles behind producing reliable evidence from data. This book is centered around the implementation of the tidy data framework within the Python programming language, and as such employs the most recent advances in data analysis coding. The use of Jupyter notebooks for exercises immediately places the student in an environment that encourages auditability and reproducibility of analyses. The integration of git and GitHub into the course is a key tool for teaching about collaboration and community, key concepts that are critical to data science.

The demand for training in data science continues to increase. The availability of large quantities of data to answer a variety of questions, the computational power available to many more people than ever before, and the public awareness of the importance of data for decision-making have all contributed to the need for high-quality data science work. This book provides a sophisticated first introduction to the field of data science and provides a balanced mix of practical skills along with generalizable principles. As we continue to introduce students to data science and train them to confront an expanding array of data science problems, they will be well-served by the ideas presented here.

Acknowledgments

Acknowledgments for the R Edition

We'd like to thank everyone who has contributed to the development of *Data Science: A First Introduction*[2]. This is an open-source textbook that began as a collection of course readings for DSCI 100, a new introductory data science course at the University of British Columbia (UBC). Several faculty members in the UBC Department of Statistics were pivotal in shaping the direction of that course, and as such, contributed greatly to the broad structure and list of topics in this book. We would especially like to thank Matías Salibían-Barrera for his mentorship during the initial development and roll-out of both DSCI 100 and this book. His door was always open when we needed to chat about how to best introduce and teach data science to our first-year students. We would also like to thank Gabriela Cohen Freue for her DSCI 561 (Regression I) teaching materials from the UBC Master of Data Science program, as some of our linear regression figures were inspired from these.

We would also like to thank all those who contributed to the process of publishing this book. In particular, we would like to thank all of our reviewers for their feedback and suggestions: Rohan Alexander, Isabella Ghement, Virgilio Gómez Rubio, Albert Kim, Adam Loy, Maria Prokofieva, Emily Riederer, and Greg Wilson. The book was improved substantially by their insights. We would like to give special thanks to Jim Zidek for his support and encouragement throughout the process, and to Roger Peng for graciously offering to write the Foreword.

Finally, we owe a debt of gratitude to all of the students of DSCI 100 over the past few years. They provided invaluable feedback on the book and worksheets; they found bugs for us (and stood by very patiently in class while we frantically fixed those bugs); and they brought a level of enthusiasm to the class that sustained us during the hard work of creating a new course and writing a textbook. Our interactions with them taught us how to teach data science, and that learning is reflected in the content of this book.

[2] https://datasciencebook.ca

Acknowledgments for the Python Edition

We'd like to thank everyone who has contributed to the development of *Data Science: A First Introduction (Python Edition)*[3]. This is an open-source Python translation of the original book, which focused on the R programming language. Both of these books are used to teach DSCI 100 at the University of British Columbia (UBC). We would like to give special thanks to Navya Dahiya and Gloria Ye for completing the first round of translation of the R material to Python, and to Philip Austin for his leadership and guidance throughout the translation process. We also gratefully acknowledge the UBC Open Educational Resources Fund, the UBC Department of Statistics, and the UBC Department of Earth, Ocean, and Atmospheric Sciences for supporting the translation of the original R textbook and exercises to the Python programming language.

[3] https://python.datasciencebook.ca

About the authors

The original version of this textbook was developed by Tiffany Timbers, Trevor Campbell, and Melissa Lee for the R programming language. The content of the R textbook was adapted to Python by Trevor Campbell, Joel Ostblom, and Lindsey Heagy.

Tiffany Timbers[4] is an Associate Professor of Teaching in the Department of Statistics and Co-Director for the Master of Data Science program (Vancouver Option) at the University of British Columbia. In these roles she teaches and develops curriculum around the responsible application of Data Science to solve real-world problems. One of her favorite courses she teaches is a graduate course on collaborative software development, which focuses on teaching how to create R and Python packages using modern tools and workflows.

Trevor Campbell[5] is an Associate Professor in the Department of Statistics at the University of British Columbia. His research focuses on automated, scalable Bayesian inference algorithms, Bayesian nonparametrics, streaming data, and Bayesian theory. He was previously a postdoctoral associate advised by Tamara Broderick in the Computer Science and Artificial Intelligence Laboratory (CSAIL) and Institute for Data, Systems, and Society (IDSS) at MIT, a Ph.D. candidate under Jonathan How in the Laboratory for Information and Decision Systems (LIDS) at MIT, and before that he was in the Engineering Science program at the University of Toronto.

Melissa Lee[6] is an Assistant Professor of Teaching in the Department of Statistics at the University of British Columbia. She teaches and develops curriculum for undergraduate statistics and data science courses. Her work focuses on student-centered approaches to teaching, developing and assessing open educational resources, and promoting equity, diversity, and inclusion initiatives.

Joel Ostblom[7] is an Assistant Professor of Teaching in the Department of Statistics at the University of British Columbia. During his PhD, Joel developed a passion for data science and reproducibility through the development

[4] https://www.tiffanytimbers.com/
[5] https://trevorcampbell.me/
[6] https://www.stat.ubc.ca/users/melissa-lee
[7] https://joelostblom.com/

of quantitative image analysis pipelines for studying stem cell and developmental biology. He has since co-created or lead the development of several courses and workshops at the University of Toronto and is now an assistant professor of teaching in the statistics department at the University of British Columbia. Joel cares deeply about spreading data literacy and excitement over programmatic data analysis, which is reflected in his contributions to open-source projects and data science learning resources.

Lindsey Heagy[8] is an Assistant Professor in the Department of Earth, Ocean, and Atmospheric Sciences and director of the Geophysical Inversion Facility at the University of British Columbia. Her research combines computational methods in numerical simulations, inversions, and machine learning to answer questions about the subsurface of the Earth. Primary applications include mineral exploration, carbon sequestration, groundwater, and environmental studies. She completed her BSc at the University of Alberta, her PhD at the University of British Columbia, and held a Postdoctoral research position at the University of California Berkeley prior to starting her current position at UBC.

[8] https://lindseyjh.ca/

1

Python and Pandas

1.1 Overview

This chapter provides an introduction to data science and the Python programming language. The goal here is to get your hands dirty right from the start. We will walk through an entire data analysis, and along the way introduce different types of data analysis question, some fundamental programming concepts in Python, and the basics of loading, cleaning, and visualizing data. In the following chapters, we will dig into each of these steps in much more detail, but for now, let's jump in to see how much we can do with data science.

1.2 Chapter learning objectives

By the end of the chapter, readers will be able to do the following:

- Identify the different types of data analysis question and categorize a question into the correct type.
- Load the `pandas` package into Python.
- Read tabular data with `read_csv`.
- Create new variables and objects in Python using the assignment symbol.
- Create and organize subsets of tabular data using `[]`, `loc[]`, `sort_values`, and `head`.
- Add and modify columns in tabular data using column assignment.
- Chain multiple operations in sequence.
- Visualize data with an `altair` bar plot.
- Use `help()` and `?` to access help and documentation tools in Python.

DOI: 10.1201/9781003438397-1

1.3 Canadian languages data set

In this chapter, we will walk through a full analysis of a data set relating to languages spoken at home by Canadian residents (Fig. 1.1). Many Indigenous peoples exist in Canada with their own cultures and languages; these languages are often unique to Canada and not spoken anywhere else in the world [Statistics Canada, 2018]. Sadly, colonization has led to the loss of many of these languages. For instance, generations of children were not allowed to speak their mother tongue (the first language an individual learns in childhood) in Canadian residential schools. Colonizers also renamed places they had "discovered" [Wilson, 2018]. Acts such as these have significantly harmed the continuity of Indigenous languages in Canada, and some languages are considered "endangered" as few people report speaking them. To learn more, see *Canadian Geographic*'s article, "Mapping Indigenous Languages in Canada" [Walker, 2017], *They Came for the Children: Canada, Aboriginal peoples, and Residential Schools* [Truth and Reconciliation Commission of Canada, 2012], and the *Truth and Reconciliation Commission of Canada's Calls to Action* [Truth and Reconciliation Commission of Canada, 2015].

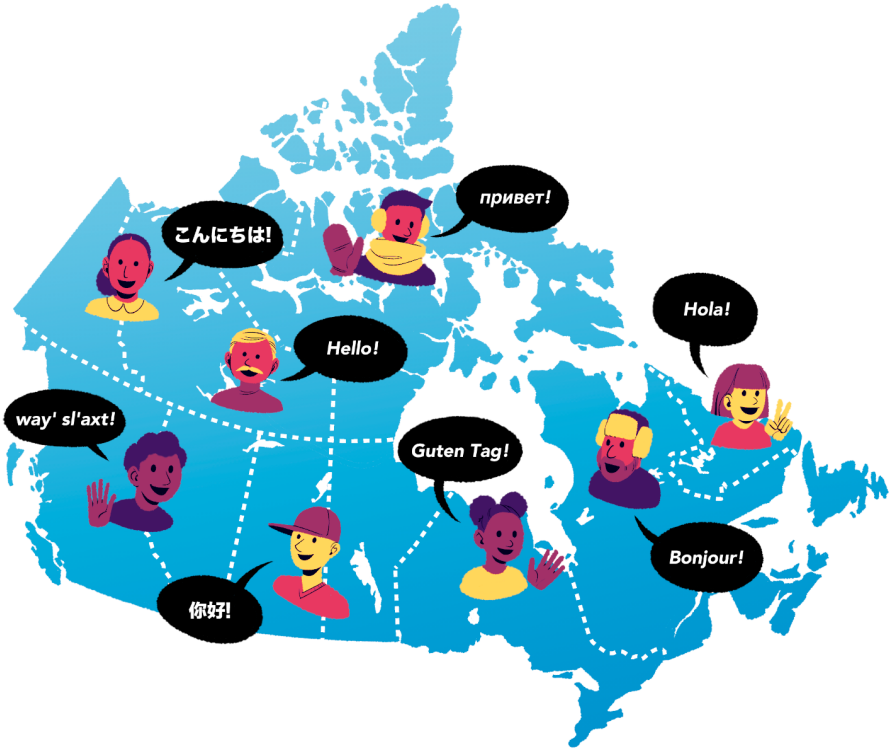


FIGURE 1.1 Map of Canada.

The data set we will study in this chapter is taken from the `canlang` R data package[1] [Timbers, 2020], which has population language data collected during the 2016 Canadian census [Statistics Canada, 2016]. In this data, there are 214 languages recorded, each having six different properties:

1. `category`: Higher-level language category, describing whether the language is an Official Canadian language, an Aboriginal (i.e., Indigenous) language, or a Non-Official and Non-Aboriginal language.
2. `language`: The name of the language.
3. `mother_tongue`: Number of Canadian residents who reported the language as their mother tongue. Mother tongue is generally defined as the language someone was exposed to since birth.
4. `most_at_home`: Number of Canadian residents who reported the language as being spoken most often at home.
5. `most_at_work`: Number of Canadian residents who reported the language as being used most often at work.
6. `lang_known`: Number of Canadian residents who reported knowledge of the language.

According to the census, more than 60 Aboriginal languages were reported as being spoken in Canada. Suppose we want to know which are the most common; then we might ask the following question, which we wish to answer using our data:

Which ten Aboriginal languages were most often reported in 2016 as mother tongues in Canada, and how many people speak each of them?

Note: Data science cannot be done without a deep understanding of the data and problem domain. In this book, we have simplified the data sets used in our examples to concentrate on methods and fundamental concepts. But in real life, you cannot and should not practice data science without a domain expert. Alternatively, it is common to practice data science in your own domain of expertise. Remember that when you work with data, it is essential to think about *how* the data were collected, which affects the conclusions you can draw. If your data are biased, then your results will be biased.

---

[1] https://ttimbers.github.io/canlang/

1.4 Asking a question

Every good data analysis begins with a *question*—like the above—that you aim to answer using data. As it turns out, there are actually a number of different *types* of question regarding data: descriptive, exploratory, predictive, inferential, causal, and mechanistic, all of which are defined in Table 1.1. [Leek and Peng, 2015; Peng and Matsui, 2015] Carefully formulating a question as early as possible in your analysis—and correctly identifying which type of question it is—will guide your overall approach to the analysis as well as the selection of appropriate tools.

TABLE 1.1 Types of data analysis question.

Question type	Description	Example
Descriptive	A question that asks about summarized characteristics of a data set without interpretation (i.e., report a fact).	How many people live in each province and territory in Canada?
Exploratory	A question that asks if there are patterns, trends, or relationships within a single data set. Often used to propose hypotheses for future study.	Does political party voting change with indicators of wealth in a set of data collected on 2,000 people living in Canada?
Predictive	A question that asks about predicting measurements or labels for individuals (people or things). The focus is on what things predict some outcome, but not what causes the outcome.	What political party will someone vote for in the next Canadian election?
Inferential	A question that looks for patterns, trends, or relationships in a single data set **and** also asks for quantification of how applicable these findings are to the wider population.	Does political party voting change with indicators of wealth for all people living in Canada?
Causal	A question that asks about whether changing one factor will lead to a change in another factor, on average, in the wider population.	Does wealth lead to voting for a certain political party in Canadian elections?
Mechanistic	A question that asks about the underlying mechanism of the observed patterns, trends, or relationships (i.e., how does it happen?)	How does wealth lead to voting for a certain political party in Canadian elections?

In this book, you will learn techniques to answer the first four types of question: descriptive, exploratory, predictive, and inferential; causal and mechanistic questions are beyond the scope of this book. In particular, you will learn how to apply the following analysis tools:

1. **Summarization:** computing and reporting aggregated values pertaining to a data set. Summarization is most often used to answer descriptive questions, and can occasionally help with answering exploratory questions. For example, you might use summarization to answer the following question: *What is the average race time for runners in this data set?* Tools for summarization are covered in detail in Chapters 2 and 3, but appear regularly throughout the text.

2. **Visualization:** plotting data graphically. Visualization is typically used to answer descriptive and exploratory questions, but plays a critical supporting role in answering all of the types of question in Table 1.1. For example, you might use visualization to answer the following question: *Is there any relationship between race time and age for runners in this data set?* This is covered in detail in Chapter 4 but again appears regularly throughout the book.

3. **Classification:** predicting a class or category for a new observation. Classification is used to answer predictive questions. For example, you might use classification to answer the following question: *Given measurements of a tumor's average cell area and perimeter, is the tumor benign or malignant?* Classification is covered in Chapters 5 and 6.

4. **Regression:** predicting a quantitative value for a new observation. Regression is also used to answer predictive questions. For example, you might use regression to answer the following question: *What will be the race time for a 20-year-old runner who weighs 50 kg?* Regression is covered in Chapters 7 and 8.

5. **Clustering:** finding previously unknown/unlabeled subgroups in a data set. Clustering is often used to answer exploratory questions. For example, you might use clustering to answer the following question: *What products are commonly bought together on Amazon?* Clustering is covered in Chapter 9.

6. **Estimation:** taking measurements for a small number of items from a large group and making a good guess for the average or proportion for the large group. Estimation is used to answer inferential questions. For example, you might use estimation to answer the following

question: *Given a survey of cellphone ownership of 100 Canadians, what proportion of the entire Canadian population own Android phones?* Estimation is covered in Chapter 10.

Referring to Table 1.1, our question about Aboriginal languages is an example of a *descriptive question*: we are summarizing the characteristics of a data set without further interpretation. And referring to the list above, it looks like we should use visualization and perhaps some summarization to answer the question. So in the remainder of this chapter, we will work toward making a visualization that shows us the ten most common Aboriginal languages in Canada and their associated counts, according to the 2016 census.

1.5 Loading a tabular data set

A data set is, at its core essence, a structured collection of numbers and characters. Aside from that, there are really no strict rules; data sets can come in many different forms. Perhaps the most common form of data set that you will find in the wild, however, is *tabular data*. Think spreadsheets in Microsoft Excel: tabular data are rectangular-shaped and spreadsheet-like, as shown in Fig. 1.2. In this book, we will focus primarily on tabular data.

Since we are using Python for data analysis in this book, the first step for us is to load the data into Python. When we load tabular data into Python, it is represented as a *data frame* object. Fig. 1.2 shows that a Python data frame is very similar to a spreadsheet. We refer to the rows as **observations**; these are the individual objects for which we collect data. In Fig. 1.2, the observations are languages. We refer to the columns as **variables**; these are the characteristics of each observation. In Fig. 1.2, the variables are the language's category, its name, the number of mother tongue speakers, etc.

The first kind of data file that we will learn how to load into Python as a data frame is the *comma-separated values* format (`.csv` for short). These files have names ending in `.csv`, and can be opened and saved using common spreadsheet programs like Microsoft Excel and Google Sheets. For example, the `.csv` file named `can_lang.csv` is included with the code for this book[2]. If we were to open this data in a plain text editor (a program like Notepad that just shows text with no formatting), we would see each row on its own line, and each entry in the table separated by a comma:

[2] https://github.com/UBC-DSCI/introduction-to-datascience-python/tree/main/source/data

Spreadsheet

	A	B	C	D	E	F
1	category	language	mother_tongue	most_at_home	most_at_work	lang_known
2	Aboriginal languages	Aboriginal languages, n.o.s.	590	235	30	665
3	Non-Official & Non-Aboriginal languages	Afrikaans	10260	4785	85	23415
4	Non-Official & Non-Aboriginal languages	Afro-Asiatic languages, n.i.e.	1150	445	10	2775
5	Non-Official & Non-Aboriginal languages	Akan (Twi)	13460	5985	25	22150
6	Non-Official & Non-Aboriginal languages	Albanian	26895	13135	345	31930
7	Aboriginal languages	Algonquian languages, n.i.e.	45	10	0	120
8	Aboriginal languages	Algonquin	1260	370	40	2480
9	Non-Official & Non-Aboriginal languages	American Sign Language	2685	3020	1145	21930
10	Non-Official & Non-Aboriginal languages	Amharic	22465	12785	200	33670
11	Non-Official & Non-Aboriginal languages	Arabic	419890	223535	5585	629055

DataFrame in Python

	category	language	mother_tongue	most_at_home	most_at_work	lang_known
0	Aboriginal languages	Aboriginal languages, n.o.s.	590	235	30	665
1	Non-Official & Non-Aboriginal languages	Afrikaans	10260	4785	85	23415
2	Non-Official & Non-Aboriginal languages	Afro-Asiatic languages, n.i.e.	1150	445	10	2775
3	Non-Official & Non-Aboriginal languages	Akan (Twi)	13460	5985	25	22150
4	Non-Official & Non-Aboriginal languages	Albanian	26895	13135	345	31930
...	...	...	...	...	...	...
209	Non-Official & Non-Aboriginal languages	Wolof	3990	1385	10	8240
210	Aboriginal languages	Woods Cree	1840	800	75	2665
211	Non-Official & Non-Aboriginal languages	Wu (Shanghainese)	12915	7650	105	16530
212	Non-Official & Non-Aboriginal languages	Yiddish	13555	7085	895	20985
213	Non-Official & Non-Aboriginal languages	Yoruba	9080	2615	15	22415

214 rows × 6 columns

FIGURE 1.2 A spreadsheet versus a data frame in Python.

```
category,language,mother_tongue,most_at_home,most_at_work,lang_known
Aboriginal languages,"Aboriginal languages, n.o.s.",590,235,30,665
Non-Official & Non-Aboriginal languages,Afrikaans,10260,4785,85,23415
Non-Official & Non-Aboriginal languages,"Afro-Asiatic languages, n.i.e.",1150,44
Non-Official & Non-Aboriginal languages,Akan (Twi),13460,5985,25,22150
Non-Official & Non-Aboriginal languages,Albanian,26895,13135,345,31930
Aboriginal languages,"Algonquian languages, n.i.e.",45,10,0,120
Aboriginal languages,Algonquin,1260,370,40,2480
Non-Official & Non-Aboriginal languages,American Sign Language,2685,3020,1145,21
Non-Official & Non-Aboriginal languages,Amharic,22465,12785,200,33670
```

To load this data into Python so that we can do things with it (e.g., perform analyses or create data visualizations), we will need to use a *function*. A function is a special word in Python that takes instructions (we call these *arguments*) and does something. The function we will use to load a `.csv` file into Python is called `read_csv`. In its most basic use-case, `read_csv` expects that the data file:

- has column names (or *headers*),
- uses a comma (`,`) to separate the columns, and

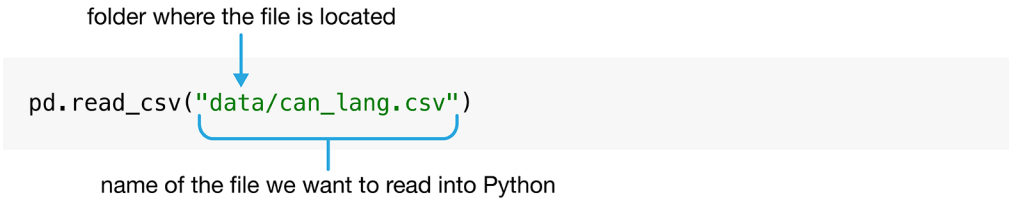


FIGURE 1.3 Syntax for the `read_csv` function.

- does not have row names.

Below you'll see the code used to load the data into Python using the `read_csv` function. Note that the `read_csv` function is not included in the base installation of Python, meaning that it is not one of the primary functions ready to use when you install Python. Therefore, you need to load it from somewhere else before you can use it. The place from which we will load it is called a Python *package*. A Python package is a collection of functions that can be used in addition to the built-in Python package functions once loaded. The `read_csv` function, in particular, can be made accessible by loading the `pandas` Python package[3] [The Pandas Development Team, 2020, Wes McKinney, 2010] using the `import` command. The `pandas` package contains many functions that we will use throughout this book to load, clean, wrangle, and visualize data.

```
import pandas as pd
```

This command has two parts. The first is `import pandas`, which loads the `pandas` package. The second is `as pd`, which give the `pandas` package the much shorter *alias* (another name) `pd`. We can now use the `read_csv` function by writing `pd.read_csv`, i.e., the package name, then a dot, then the function name. You can see why we gave `pandas` a shorter alias; if we had to type `pandas` before every function we wanted to use, our code would become much longer and harder to read.

Now that the `pandas` package is loaded, we can use the `read_csv` function by passing it a single argument: the name of the file, `"can_lang.csv"`. We have to put quotes around file names and other letters and words that we use in our code to distinguish it from the special words (like functions!) that make up the Python programming language. The file's name is the only argument we need to provide because our file satisfies everything else that the `read_csv` function expects in the default use case. Fig. 1.3 describes how we use the `read_csv` to read data into Python.

[3] https://pypi.org/project/pandas/

```
pd.read_csv("data/can_lang.csv")
```

```
                                    category                      language ␣
↪\
0                      Aboriginal languages    Aboriginal languages, n.o.s.
1    Non-Official & Non-Aboriginal languages                       Afrikaans
2    Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3    Non-Official & Non-Aboriginal languages                     Akan (Twi)
4    Non-Official & Non-Aboriginal languages                       Albanian
..                                      ...                             ...
209  Non-Official & Non-Aboriginal languages                         Wolof
210                    Aboriginal languages                   Woods Cree
211  Non-Official & Non-Aboriginal languages              Wu (Shanghainese)
212  Non-Official & Non-Aboriginal languages                       Yiddish
213  Non-Official & Non-Aboriginal languages                        Yoruba

     mother_tongue  most_at_home  most_at_work  lang_known
0              590           235            30         665
1            10260          4785            85       23415
2             1150           445            10        2775
3            13460          5985            25       22150
4            26895         13135           345       31930
..             ...           ...           ...         ...
209           3990          1385            10        8240
210           1840           800            75        2665
211          12915          7650           105       16530
212          13555          7085           895       20985
213           9080          2615            15       22415

[214 rows x 6 columns]
```

1.6 Naming things in Python

When we loaded the 2016 Canadian census language data using `read_csv`, we did not give this data frame a name. Therefore the data was just printed on the screen, and we cannot do anything else with it. That isn't very useful. What would be more useful would be to give a name to the data frame that `read_csv` outputs so that we can refer to it later for analysis and visualization.

The way to assign a name to a value in Python is via the *assignment symbol* `=`. On the left side of the assignment symbol you put the name that you want to use, and on the right side of the assignment symbol you put the value that you want the name to refer to. Names can be used to refer to almost anything in Python, such as numbers, words (also known as *strings* of characters), and data frames. Below, we set `my_number` to `3` (the result of `1+2`) and we set `name` to the string `"Alice"`.

```
my_number = 1 + 2
name = "Alice"
```

Note that when we name something in Python using the assignment symbol, =, we do not need to surround the name we are creating with quotes. This is because we are formally telling Python that this special word denotes the value of whatever is on the right-hand side. Only characters and words that act as *values* on the right-hand side of the assignment symbol—e.g., the file name `"data/can_lang.csv"` that we specified before, or `"Alice"` above—need to be surrounded by quotes.

After making the assignment, we can use the special name words we have created in place of their values. For example, if we want to do something with the value `3` later on, we can just use `my_number` instead. Let's try adding 2 to `my_number`; you will see that Python just interprets this as adding 2 and 3:

```
my_number + 2
```

```
5
```

Object names can consist of letters, numbers, and underscores (_). Other symbols won't work since they have their own meanings in Python. For example, - is the subtraction symbol; if we try to assign a name with the - symbol, Python will complain and we will get an error.

```
my-number = 1
```

```
SyntaxError: cannot assign to expression here. Maybe you meant '==' instead␣
↪of '='?
```

There are certain conventions for naming objects in Python. When naming an object we suggest using only lowercase letters, numbers, and underscores _ to separate the words in a name. Python is case sensitive, which means that `Letter` and `letter` would be two different objects in Python. You should also try to give your objects meaningful names. For instance, you *can* name a data frame `x`. However, using more meaningful terms, such as `language_data`, will help you remember what each name in your code represents. We recommend following the **PEP 8** naming conventions outlined in the *PEP 8*[4] [Guido van Rossum, 2001]. Let's now use the assignment symbol to give the name `can_lang` to the 2016 Canadian census language data frame that we get from `read_csv`.

```
can_lang = pd.read_csv("data/can_lang.csv")
```

[4] https://peps.python.org/pep-0008/

Wait a minute, nothing happened this time. Where's our data? Actually, something did happen: the data was loaded in and now has the name `can_lang` associated with it. And we can use that name to access the data frame and do things with it. For example, we can type the name of the data frame to print both the first few rows and the last few rows. The three dots (`...`) indicate that there are additional rows that are not printed. You will also see that the number of observations (i.e., rows) and variables (i.e., columns) are printed just underneath the data frame (214 rows and 6 columns in this case). Printing a few rows from data frame like this is a handy way to get a quick sense for what is contained in it.

```
can_lang
```

```
                                    category                         language ␣
↪\
0                        Aboriginal languages    Aboriginal languages, n.o.s.
1    Non-Official & Non-Aboriginal languages                       Afrikaans
2    Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3    Non-Official & Non-Aboriginal languages                     Akan (Twi)
4    Non-Official & Non-Aboriginal languages                       Albanian
..                                      ...                            ...
209  Non-Official & Non-Aboriginal languages                         Wolof
210                      Aboriginal languages                   Woods Cree
211  Non-Official & Non-Aboriginal languages              Wu (Shanghainese)
212  Non-Official & Non-Aboriginal languages                        Yiddish
213  Non-Official & Non-Aboriginal languages                         Yoruba

     mother_tongue  most_at_home  most_at_work  lang_known
0              590           235            30         665
1            10260          4785            85       23415
2             1150           445            10        2775
3            13460          5985            25       22150
4            26895         13135           345       31930
..             ...           ...           ...         ...
209           3990          1385            10        8240
210           1840           800            75        2665
211          12915          7650           105       16530
212          13555          7085           895       20985
213           9080          2615            15       22415

[214 rows x 6 columns]
```

1.7 Creating subsets of data frames with `[]` & `loc[]`

Now that we've loaded our data into Python, we can start wrangling the data to find the ten Aboriginal languages that were most often reported in 2016 as mother tongues in Canada. In particular, we want to construct a table with the ten Aboriginal languages that have the largest counts in the `mother_tongue` column. The first step is to extract from our `can_lang` data

only those rows that correspond to Aboriginal languages, and then the second step is to keep only the `language` and `mother_tongue` columns. The `[]` and `loc[]` operations on the `pandas` data frame will help us here. The `[]` allows you to obtain a subset of (i.e., *filter*) the rows of a data frame, or to obtain a subset of (i.e., *select*) the columns of a data frame. The `loc[]` operation allows you to both filter rows *and* select columns at the same time. We will first investigate filtering rows and selecting columns with the `[]` operation, and then use `loc[]` to do both in our analysis of the Aboriginal languages data.

Note: The `[]` and `loc[]` operations, and related operations, in `pandas` are much more powerful than we describe in this chapter. You will learn more sophisticated ways to index data frames later on in Chapter 3.

1.7.1 Using `[]` to filter rows

Looking at the `can_lang` data above, we see the column `category` contains different high-level categories of languages, which include "Aboriginal languages", "Non-Official & Non-Aboriginal languages", and "Official languages". To answer our question we want to filter our data set so we restrict our attention to only those languages in the "Aboriginal languages" category.

We can use the `[]` operation to obtain the subset of rows with desired values from a data frame. Fig. 1.4 shows the syntax we need to use to filter rows with the `[]` operation. First, we type the name of the data frame—here, `can_lang`—followed by square brackets. Inside the square brackets, we write a *logical statement* to use when filtering the rows. A logical statement evaluates to either `True` or `False` for each row in the data frame; the `[]` operation keeps only those rows for which the logical statement evaluates to `True`. For example, in our analysis, we are interested in keeping only languages in the `"Aboriginal languages"` higher-level category. We can use the *equivalency operator* == to compare the values of the `category` column—denoted by `can_lang["category"]`—with the value `"Aboriginal languages"`. You will learn about many other kinds of logical statement in Chapter 3. Similar to when we loaded the data file and put quotes around the file name, here we need to put quotes around both `"Aboriginal languages"` and `"category"`. Using quotes tells Python that this is a *string value* (e.g., a column name, or word data) and not one of the special words that make up the Python programming language, or one of the names we have given to objects in the code we have already written.

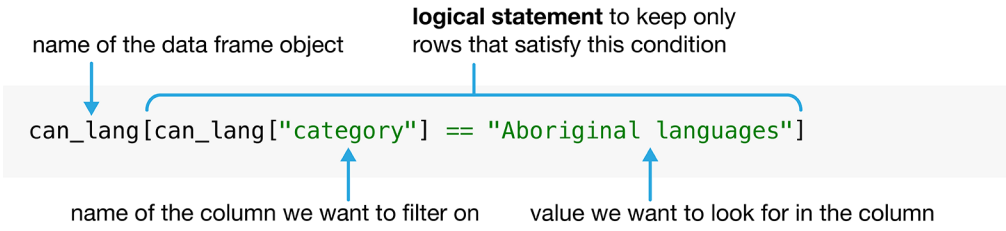


FIGURE 1.4 Syntax for using the `[]` operation to filter rows.

Note: In Python, single quotes (`'`) and double quotes (`"`) are generally treated the same. So we could have written `'Aboriginal languages'` instead of `"Aboriginal languages"` above, or `'category'` instead of `"category"`. Try both out for yourself.

This operation returns a data frame that has all the columns of the input data frame, but only those rows corresponding to Aboriginal languages that we asked for in the logical statement.

```
can_lang[can_lang["category"] == "Aboriginal languages"]
```

```
                 category                      language  mother_tongue  \
0    Aboriginal languages  Aboriginal languages, n.o.s.            590
5    Aboriginal languages  Algonquian languages, n.i.e.             45
6    Aboriginal languages                    Algonquin           1260
12   Aboriginal languages  Athabaskan languages, n.i.e.             50
13   Aboriginal languages                     Atikamekw           6150
..                    ...                           ...            ...
191  Aboriginal languages        Thompson (Ntlakapamux)            335
195  Aboriginal languages                      Tlingit             95
196  Aboriginal languages                     Tsimshian            200
206  Aboriginal languages    Wakashan languages, n.i.e.             10
210  Aboriginal languages                   Woods Cree           1840

     most_at_home  most_at_work  lang_known
0             235            30         665
5              10             0         120
6             370            40        2480
12             10             0          85
13           5465          1100        6645
..            ...           ...         ...
191            20             0         450
195             0            10         260
196            30            10         410
206             0             0          25
210           800            75        2665

[67 rows x 6 columns]
```

FIGURE 1.5 Syntax for using the `[]` operation to select columns.

1.7.2 Using `[]` to select columns

We can also use the `[]` operation to select columns from a data frame. Fig. 1.5 displays the syntax needed to select columns. We again first type the name of the data frame—here, `can_lang`—followed by square brackets. Inside the square brackets, we provide a *list* of column names. In Python, we denote a *list* using square brackets, where each item is separated by a comma (`,`). So if we are interested in selecting only the `language` and `mother_tongue` columns from our original `can_lang` data frame, we put the list `["language", "mother_tongue"]` containing those two column names inside the square brackets of the `[]` operation.

This operation returns a data frame that has all the rows of the input data frame, but only those columns that we named in the selection list.

```
can_lang[["language", "mother_tongue"]]
```

```
                            language  mother_tongue
0       Aboriginal languages, n.o.s.           590
1                           Afrikaans         10260
2     Afro-Asiatic languages, n.i.e.           1150
3                          Akan (Twi)         13460
4                            Albanian         26895
..                                ...           ...
209                            Wolof           3990
210                      Woods Cree            1840
211               Wu (Shanghainese)           12915
212                          Yiddish         13555
213                           Yoruba           9080

[214 rows x 2 columns]
```

1.7.3 Using `loc[]` to filter rows and select columns

The `[]` operation is only used when you want to filter rows *or* select columns; it cannot be used to do both operations at the same time. But in order to answer our original data analysis question in this chapter, we need to *both* filter the rows for Aboriginal languages, *and* select the `language` and `mother_tongue` columns. Fortunately, `pandas` provides the `loc[]` operation, which lets us do just that. The syntax is very similar to the `[]` operation we have already

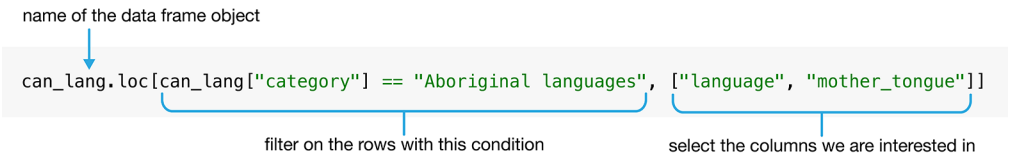


FIGURE 1.6 Syntax for using the `loc[]` operation to filter rows and select columns.

covered: we will essentially combine both our row filtering and column selection steps from before. In particular, we first write the name of the data frame—`can_lang` again—then follow that with the `.loc[]` operation. Inside the square brackets, we write our row filtering logical statement, then a comma, then our list of columns to select (Fig. 1.6).

```
aboriginal_lang = can_lang.loc[can_lang["category"] == "Aboriginal languages", [
↪"language", "mother_tongue"]]
```

There is one very important thing to notice in this code example. The first is that we used the `loc[]` operation on the `can_lang` data frame by writing `can_lang.loc[]`—first the data frame name, then a dot, then `loc[]`. There's that dot again. If you recall, earlier in this chapter we used the `read_csv` function from `pandas` (aliased as `pd`), and wrote `pd.read_csv`. The dot means that the thing on the left (`pd`, i.e., the `pandas` package) *provides* the thing on the right (the `read_csv` function). In the case of `can_lang.loc[]`, the thing on the left (the `can_lang` data frame) *provides* the thing on the right (the `loc[]` operation). In Python, both packages (like `pandas`) *and* objects (like our `can_lang` data frame) can provide functions and other objects that we access using the dot syntax.

Note: A note on terminology: when an object `obj` provides a function `f` with the dot syntax (as in `obj.f()`), sometimes we call that function `f` a *method* of `obj` or an *operation* on `obj`. Similarly, when an object `obj` provides another object `x` with the dot syntax (as in `obj.x`), sometimes we call the object `x` an *attribute* of `obj`. We will use all of these terms throughout the book, as you will see them used commonly in the community. And just because we programmers like to be confusing for no apparent reason: we *don't* use the "method", "operation", or "attribute" terminology when referring to functions and objects from packages, like `pandas`. So, for example, `pd.read_csv` would typically just be referred to as a function, but not as a method or operation, even though it uses the dot syntax.

At this point, if we have done everything correctly, `aboriginal_lang` should be a data frame containing *only* rows where the `category` is `"Aboriginal languages"`, and containing *only* the `language` and `mother_tongue` columns. Any time you take a step in a data analysis, it's good practice to check the output by printing the result.

```
aboriginal_lang
```

```
                          language  mother_tongue
0    Aboriginal languages, n.o.s.            590
5    Algonquian languages, n.i.e.             45
6                        Algonquin           1260
12   Athabaskan languages, n.i.e.             50
13                       Atikamekw           6150
..                             ...            ...
191        Thompson (Ntlakapamux)            335
195                       Tlingit             95
196                      Tsimshian            200
206    Wakashan languages, n.i.e.             10
210                     Woods Cree           1840

[67 rows x 2 columns]
```

We can see the original `can_lang` data set contained 214 rows with multiple kinds of `category`. The data frame `aboriginal_lang` contains only 67 rows, and looks like it only contains Aboriginal languages. So it looks like the `loc[]` operation gave us the result we wanted.

1.8 Using `sort_values` and `head` to select rows by ordered values

We have used the `[]` and `loc[]` operations on a data frame to obtain a table with only the Aboriginal languages in the data set and their associated counts. However, we want to know the **ten** languages that are spoken most often. As a next step, we will order the `mother_tongue` column from largest to smallest value and then extract only the top ten rows. This is where the `sort_values` and `head` functions come to the rescue.

The `sort_values` function allows us to order the rows of a data frame by the values of a particular column. We need to specify the column name by which we want to sort the data frame by passing it to the argument `by`. Since we want to choose the ten Aboriginal languages most often reported as a mother tongue language, we will use the `sort_values` function to order the rows in our `selected_lang` data frame by the `mother_tongue` column. We want to

arrange the rows in descending order (from largest to smallest), so we specify the argument `ascending` as `False` (Fig. 1.7).

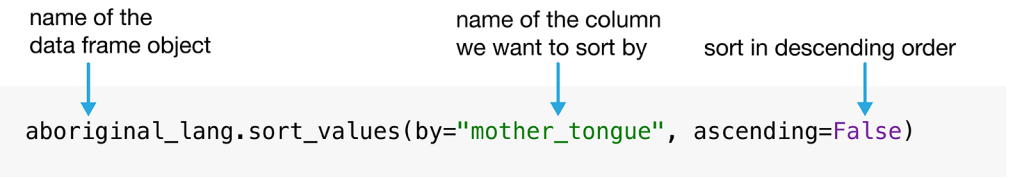


FIGURE 1.7 Syntax for using `sort_values` to arrange rows in descending order.

```
arranged_lang = aboriginal_lang.sort_values(by="mother_tongue", ascending=False)
arranged_lang
```

```
                          language  mother_tongue
40                   Cree, n.o.s.          64050
89                      Inuktitut          35210
138                       Ojibway          17885
137                      Oji-Cree          12855
48                           Dene          10700
..                            ...            ...
5    Algonquian languages, n.i.e.             45
32                         Cayuga             45
179                       Squamish             40
90    Iroquoian languages, n.i.e.             35
206    Wakashan languages, n.i.e.             10

[67 rows x 2 columns]
```

Next, we will obtain the ten most common Aboriginal languages by selecting only the first ten rows of the `arranged_lang` data frame. We do this using the `head` function, and specifying the argument `10`.

```
ten_lang = arranged_lang.head(10)
ten_lang
```

```
              language  mother_tongue
40         Cree, n.o.s.          64050
89            Inuktitut          35210
138             Ojibway          17885
137            Oji-Cree          12855
48                 Dene          10700
125  Montagnais (Innu)          10235
119           Mi'kmaq           6690
13          Atikamekw           6150
149        Plains Cree           3065
180             Stoney           3025
```

1.9 Adding and modifying columns

Recall that our data analysis question referred to the *count* of Canadians that speak each of the top ten most commonly reported Aboriginal languages as their mother tongue, and the `ten_lang` data frame indeed contains those counts … But perhaps, seeing these numbers, we became curious about the *percentage* of the population of Canada associated with each count. It is common to come up with new data analysis questions in the process of answering a first one—so fear not and explore. To answer this small question along the way, we need to divide each count in the `mother_tongue` column by the total Canadian population according to the 2016 census—i.e., 35,151,728—and multiply it by 100. We can perform this computation using the code `100 * ten_lang["mother_tongue"] / canadian_population`. Then to store the result in a new column (or overwrite an existing column), we specify the name of the new column to create (or old column to modify), then the assignment symbol =, and then the computation to store in that column. In this case, we will opt to create a new column called `mother_tongue_percent`.

Note: You will see below that we write the Canadian population in Python as `35_151_728`. The underscores (_) are just there for readability, and do not affect how Python interprets the number. In other words, `35151728` and `35_151_728` are treated identically in Python, although the latter is much clearer.

```
canadian_population = 35_151_728
ten_lang["mother_tongue_percent"] = 100 * ten_lang["mother_tongue"] / canadian_
↪population
ten_lang
```

	language	mother_tongue	mother_tongue_percent
40	Cree, n.o.s.	64050	0.182210
89	Inuktitut	35210	0.100166
138	Ojibway	17885	0.050879
137	Oji-Cree	12855	0.036570
48	Dene	10700	0.030439
125	Montagnais (Innu)	10235	0.029117
119	Mi'kmaq	6690	0.019032
13	Atikamekw	6150	0.017496
149	Plains Cree	3065	0.008719
180	Stoney	3025	0.008606

The `ten_lang_percent` data frame shows that the ten Aboriginal languages in the `ten_lang` data frame were spoken as a mother tongue by between 0.008% and 0.18% of the Canadian population.

1.10 Combining steps with chaining and multiline expressions

It took us 3 steps to find the ten Aboriginal languages most often reported in 2016 as mother tongues in Canada. Starting from the `can_lang` data frame, we:

1) used `loc` to filter the rows so that only the `Aboriginal languages` category remained, and selected the `language` and `mother_tongue` columns,
2) used `sort_values` to sort the rows by `mother_tongue` in descending order, and
3) obtained only the top 10 values using `head`.

One way of performing these steps is to just write multiple lines of code, storing temporary, intermediate objects as you go.

```
aboriginal_lang = can_lang.loc[can_lang["category"] == "Aboriginal languages", [
↪"language", "mother_tongue"]]
arranged_lang_sorted = aboriginal_lang.sort_values(by="mother_tongue",␣
↪ascending=False)
ten_lang = arranged_lang_sorted.head(10)
```

You might find that code hard to read. You're not wrong; it is. There are two main issues with readability here. First, each line of code is quite long. It is hard to keep track of what methods are being called, and what arguments were used. Second, each line introduces a new temporary object. In this case, both `aboriginal_lang` and `arranged_lang_sorted` are just temporary results on the way to producing the `ten_lang` data frame. This makes the code hard to read, as one has to trace where each temporary object goes, and hard to understand, since introducing many named objects also suggests that they are of some importance, when really they are just intermediates. The need to call multiple methods in a sequence to process a data frame is quite common, so this is an important issue to address.

To solve the first problem, we can actually split the long expressions above across multiple lines. Although in most cases, a single expression in Python must be contained in a single line of code, there are a small number of situations where lets us do this. Let's rewrite this code in a more readable format using multiline expressions.

```
aboriginal_lang = can_lang.loc[
    can_lang["category"] == "Aboriginal languages",
```

(continues on next page)

(continued from previous page)

```
    ["language", "mother_tongue"]
]
arranged_lang_sorted = aboriginal_lang.sort_values(
    by="mother_tongue",
    ascending=False
)
ten_lang = arranged_lang_sorted.head(10)
```

This code is the same as the code we showed earlier; you can see the same sequence of methods and arguments is used. But long expressions are split across multiple lines when they would otherwise get long and unwieldy, improving the readability of the code. How does Python know when to keep reading on the next line for a single expression? For the line starting with `aboriginal_lang = ...`, Python sees that the line ends with a left bracket symbol `[`, and knows that our expression cannot end until we close it with an appropriate corresponding right bracket symbol `]`. We put the same two arguments as we did before, and then the corresponding right bracket appears after `["language", "mother_tongue"]`). For the line starting with `arranged_lang_sorted = ...`, Python sees that the line ends with a left parenthesis symbol `(`, and knows the expression cannot end until we close it with the corresponding right parenthesis symbol `)`. Again we use the same two arguments as before, and then the corresponding right parenthesis appears right after `ascending=False`. In both cases, Python keeps reading the next line to figure out what the rest of the expression is. We could, of course, put all of the code on one line of code, but splitting it across multiple lines helps a lot with code readability.

We still have to handle the issue that each line of code—i.e., each step in the analysis—introduces a new temporary object. To address this issue, we can *chain* multiple operations together without assigning intermediate objects. The key idea of chaining is that the *output* of each step in the analysis is a data frame, which means that you can just directly keep calling methods that operate on the output of each step in a sequence. This simplifies the code and makes it easier to read. The code below demonstrates the use of both multiline expressions and chaining together. The code is now much cleaner, and the `ten_lang` data frame that we get is equivalent to the one from the messy code above.

```
# obtain the 10 most common Aboriginal languages
ten_lang = (
    can_lang.loc[
       can_lang["category"] == "Aboriginal languages",
       ["language", "mother_tongue"]
    ]
    .sort_values(by="mother_tongue", ascending=False)
    .head(10)
```

(continues on next page)

(continued from previous page)

```
)
ten_lang
```

	language	mother_tongue
40	Cree, n.o.s.	64050
89	Inuktitut	35210
138	Ojibway	17885
137	Oji-Cree	12855
48	Dene	10700
125	Montagnais (Innu)	10235
119	Mi'kmaq	6690
13	Atikamekw	6150
149	Plains Cree	3065
180	Stoney	3025

Let's parse this new block of code piece by piece. The code above starts with a left parenthesis, `(`, and so Python knows to keep reading to subsequent lines until it finds the corresponding right parenthesis symbol `)`. The `loc` method performs the filtering and selecting steps as before. The line after this starts with a period (`.`) that "chains" the output of the `loc` step with the next operation, `sort_values`. Since the output of `loc` is a data frame, we can use the `sort_values` method on it without first giving it a name. That is what the `.sort_values` does on the next line. Finally, we once again "chain" together the output of `sort_values` with `head` to ask for the 10 most common languages. Finally, the right parenthesis `)` corresponding to the very first left parenthesis appears on the second last line, completing the multiline expression. Instead of creating intermediate objects, with chaining, we take the output of one operation and use that to perform the next operation. In doing so, we remove the need to create and store intermediates. This can help with readability by simplifying the code.

Now that we've shown you chaining as an alternative to storing temporary objects and composing code, does this mean you should *never* store temporary objects or compose code? Not necessarily. There are times when temporary objects are handy to keep around. For example, you might store a temporary object before feeding it into a plot function so you can iteratively change the plot without having to redo all of your data transformations. Chaining many functions can be overwhelming and difficult to debug; you may want to store a temporary object midway through to inspect your result before moving on with further steps.

1.11 Exploring data with visualizations

The `ten_lang` table answers our initial data analysis question. Are we done? Well, not quite; tables are almost never the best way to present the result of your analysis to your audience. Even the `ten_lang` table with only two columns presents some difficulty: for example, you have to scrutinize the table quite closely to get a sense for the relative numbers of speakers of each language. When you move on to more complicated analyses, this issue only gets worse. In contrast, a *visualization* would convey this information in a much more easily understood format. Visualizations are a great tool for summarizing information to help you effectively communicate with your audience, and creating effective data visualizations is an essential component of any data analysis. In this section we will develop a visualization of the ten Aboriginal languages that were most often reported in 2016 as mother tongues in Canada, as well as the number of people that speak each of them.

1.11.1 Using `altair` to create a bar plot

In our data set, we can see that `language` and `mother_tongue` are in separate columns (or variables). In addition, there is a single row (or observation) for each language. The data is, therefore, in what we call a *tidy data* format. Tidy data is a fundamental concept and will be a significant focus in the remainder of this book: many of the functions from `pandas` require tidy data, as does the `altair` package that we will use shortly for our visualization. We will formally introduce tidy data in Chapter 3.

We will make a bar plot to visualize our data. A bar plot is a chart where the lengths of the bars represent certain values, like counts or proportions. We will make a bar plot using the `mother_tongue` and `language` columns from our `ten_lang` data frame. To create a bar plot of these two variables using the `altair` package, we must specify the data frame, which variables to put on the x and y axes, and what kind of plot to create. First, we need to import the `altair` package.

```
import altair as alt
```

The fundamental object in `altair` is the `Chart`, which takes a data frame as an argument: `alt.Chart(ten_lang)`. With a chart object in hand, we can now specify how we would like the data to be visualized. We first indicate what kind of graphical *mark* we want to use to represent the data. Here we set the mark attribute of the chart object using the `Chart.mark_bar` function, because we want to create a bar chart. Next, we need to *encode* the variables

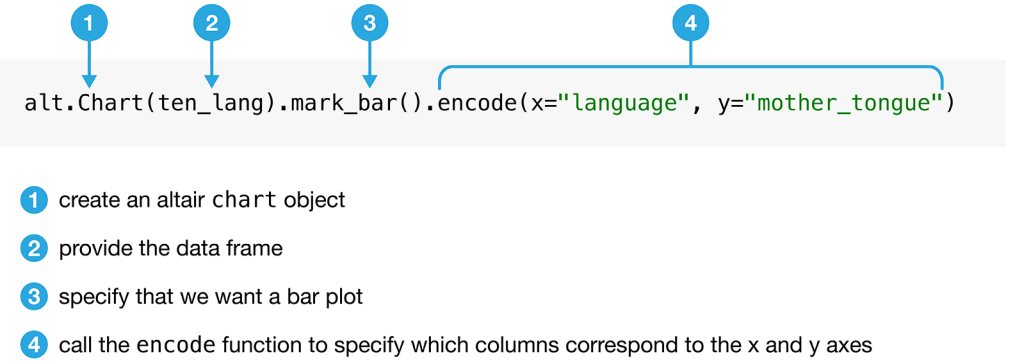


FIGURE 1.8 Syntax for using `altair` to make a bar chart.

of the data frame using the `x` and `y` *channels* (which represent the x-axis and y-axis position of the points). We use the `encode()` function to handle this: we specify that the `language` column should correspond to the x-axis, and that the `mother_tongue` column should correspond to the y-axis (Figs. 1.8–1.9).

```
barplot_mother_tongue = (
  alt.Chart(ten_lang).mark_bar().encode(x="language", y="mother_tongue")
)
```

1.11.2 Formatting `altair` charts

It is exciting that we can already visualize our data to help answer our question, but we are not done yet. We can (and should) do more to improve the interpretability of the data visualization that we created. For example, by default, Python uses the column names as the axis labels. Usually these column names do not have enough information about the variable in the column. We really should replace this default with a more informative label. For the example above, Python uses the column name `mother_tongue` as the label for the y-axis, but most people will not know what that is. And even if they did, they will not know how we measured this variable, or the group of people on which the measurements were taken. An axis label that reads "Mother Tongue (Number of Canadian Residents)" would be much more informative. To make the code easier to read, we're spreading it out over multiple lines just as we did in the previous section with pandas.

Adding additional labels to our visualizations that we create in `altair` is one common and easy way to improve and refine our data visualizations. We can add titles for the axes in the `altair` objects using `alt.X` and `alt.Y` with the `title` method to make the axes titles more informative (you will learn more about `alt.X` and `alt.Y` in Chapter 4). Again, since we are specifying

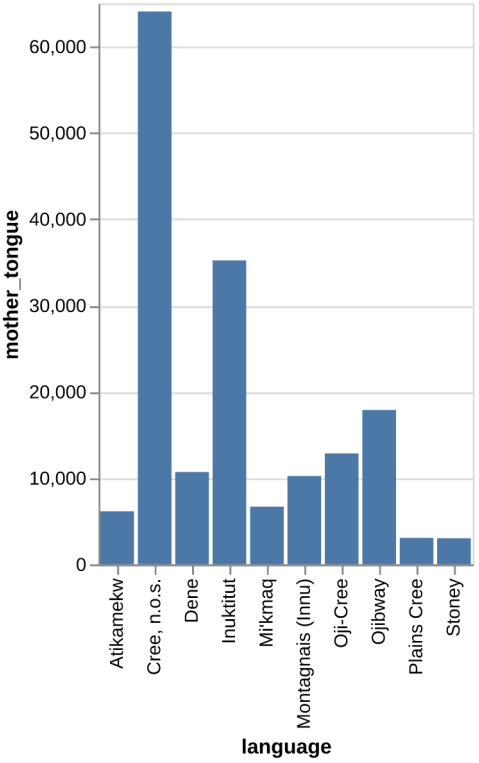


FIGURE 1.9 Bar plot of the ten Aboriginal languages most often reported by Canadian residents as their mother tongue.

words (e.g., `"Mother Tongue (Number of Canadian Residents)"`) as arguments to the `title` method, we surround them with quotation marks. We can do many other modifications to format the plot further, and we will explore these in Chapter 4.

```
barplot_mother_tongue = alt.Chart(ten_lang).mark_bar().encode(
    x=alt.X("language").title("Language"),
    y=alt.Y("mother_tongue").title("Mother Tongue (Number of Canadian Residents)
↪")
)
```

The result is shown in Fig. 1.10. This is already quite an improvement. Let's tackle the next major issue with the visualization in Fig. 1.10: the vertical x axis labels, which are currently making it difficult to read the different language names. One solution is to rotate the plot such that the bars are horizontal rather than vertical. To accomplish this, we will swap the x and y coordinate axes:

```
barplot_mother_tongue_axis = alt.Chart(ten_lang).mark_bar().encode(
    x=alt.X("mother_tongue").title("Mother Tongue (Number of Canadian Residents)
↪"),
```

(continues on next page)

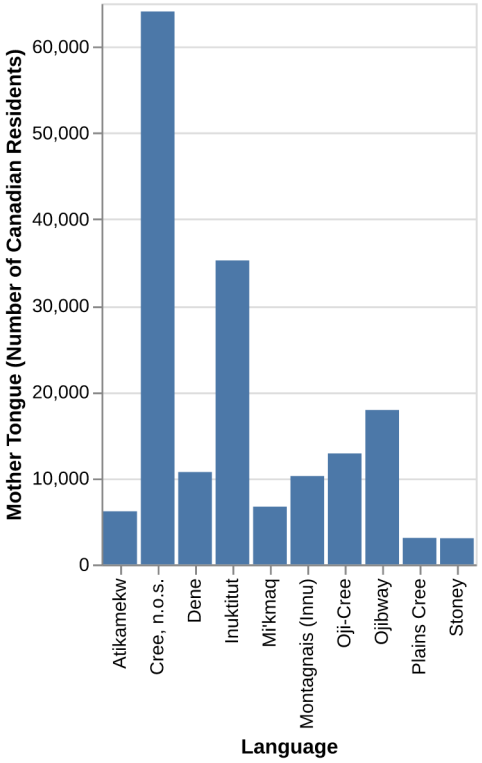


FIGURE 1.10 Bar plot of the ten Aboriginal languages most often reported by Canadian residents as their mother tongue with x and y labels. Note that this visualization is not done yet; there are still improvements to be made.

(continued from previous page)

```
    y=alt.Y("language").title("Language")
)
```

Another big step forward, as shown in Fig. 1.11. There are no more serious issues with the visualization. Now comes time to refine the visualization to make it even more well-suited to answering the question we asked earlier in this chapter. For example, the visualization could be made more transparent by organizing the bars according to the number of Canadian residents reporting each language, rather than in alphabetical order. We can reorder the bars using the `sort` method, which orders a variable (here `language`) based on the values of the variable (`mother_tongue`) on the `x-axis`.

```
ordered_barplot_mother_tongue = alt.Chart(ten_lang).mark_bar().encode(
    x=alt.X("mother_tongue").title("Mother Tongue (Number of Canadian Residents)
↪"),
    y=alt.Y("language").sort("x").title("Language")
)
```

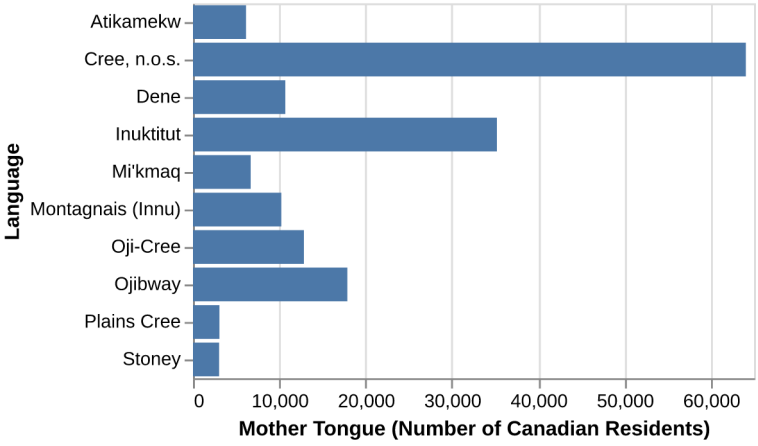


FIGURE 1.11 Horizontal bar plot of the ten Aboriginal languages most often reported by Canadian residents as their mother tongue. There are no more serious issues with this visualization, but it could be refined further.

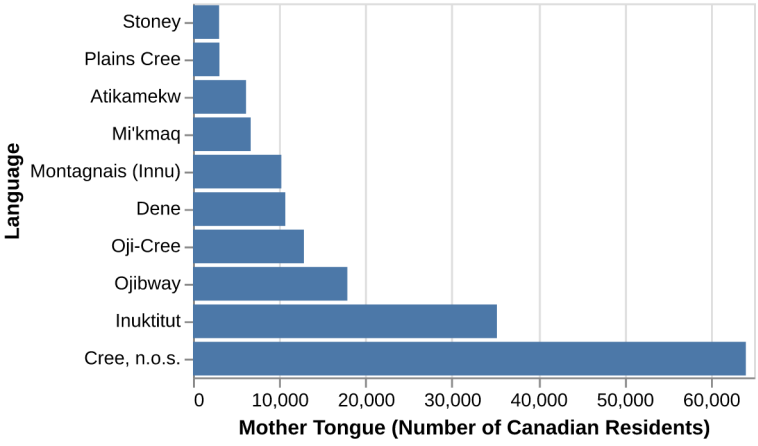


FIGURE 1.12 Bar plot of the ten Aboriginal languages most often reported by Canadian residents as their mother tongue with bars reordered.

Fig. 1.12 provides a very clear and well-organized answer to our original question; we can see what the ten most often reported Aboriginal languages were, according to the 2016 Canadian census, and how many people speak each of them. For instance, we can see that the Aboriginal language most often reported was Cree n.o.s. with over 60,000 Canadian residents reporting it as their mother tongue.

Note: "n.o.s." means "not otherwise specified", so Cree n.o.s. refers to individuals who reported Cree as their mother tongue. In this data set, the Cree languages include the following categories: Cree n.o.s., Swampy Cree, Plains Cree, Woods Cree, and a "Cree not included elsewhere" category (which includes Moose Cree, Northern East Cree and Southern East Cree) [Statistics Canada, 2016].

1.11.3 Putting it all together

In the block of code below, we put everything from this chapter together, with a few modifications. In particular, we have combined all of our steps into one expression split across multiple lines using the left and right parenthesis symbols `(` and `)`. We have also provided *comments* next to many of the lines of code below using the hash symbol `#`. When Python sees a `#` sign, it will ignore all of the text that comes after the symbol on that line. So you can use comments to explain lines of code for others, and perhaps more importantly, your future self. It's good practice to get in the habit of commenting your code to improve its readability.

This exercise demonstrates the power of Python. In relatively few lines of code, we performed an entire data science workflow with a highly effective data visualization. We asked a question, loaded the data into Python, wrangled the data (using `[]`, `loc[]`, `sort_values`, and `head`) and created a data visualization to help answer our question (Fig. 1.13). In this chapter, you got a quick taste of the data science workflow; continue on with the next few chapters to learn each of these steps in much more detail!

```
# load the data set
can_lang = pd.read_csv("data/can_lang.csv")

# obtain the 10 most common Aboriginal languages
ten_lang = (
    can_lang.loc[can_lang["category"] == "Aboriginal languages", ["language",
↪"mother_tongue"]]
    .sort_values(by="mother_tongue", ascending=False)
    .head(10)
)

# create the visualization
ten_lang_plot = alt.Chart(ten_lang).mark_bar().encode(
    x=alt.X("mother_tongue").title("Mother Tongue (Number of Canadian Residents)
↪"),
    y=alt.Y("language").sort("x").title("Language")
)
```

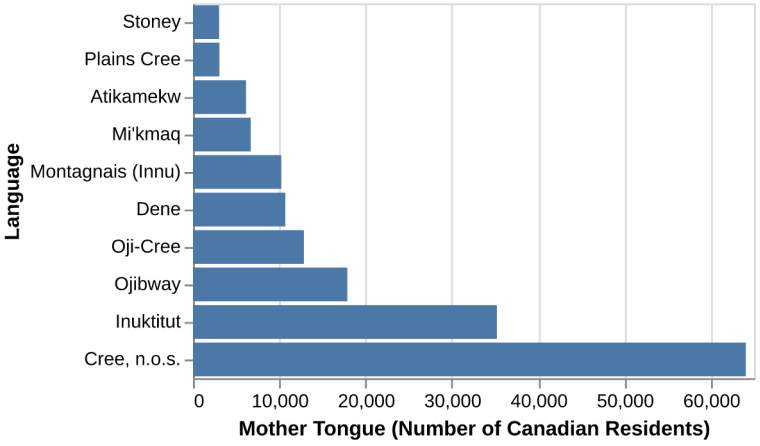


FIGURE 1.13 Bar plot of the ten Aboriginal languages most often reported by Canadian residents as their mother tongue.

1.12 Accessing documentation

There are many Python functions in the `pandas` package (and beyond!), and nobody can be expected to remember what every one of them does or all of the arguments we have to give them. Fortunately, Python provides the `help` function, which provides an easy way to pull up the documentation for most functions quickly. To use the `help` function to access the documentation, you just put the name of the function you are curious about as an argument inside the `help` function. For example, if you had forgotten what the `pd.read_csv` function did or exactly what arguments to pass in, you could run the following code:

```
help(pd.read_csv)
```

Fig. 1.14 shows the documentation that will pop up, including a high-level description of the function, its arguments, a description of each, and more. Note that you may find some of the text in the documentation a bit too technical right now. Fear not: as you work through this book, many of these terms will be introduced to you, and slowly but surely you will become more adept at understanding and navigating documentation like that shown in Fig. 1.14. But do keep in mind that the documentation is not written to *teach* you about a function; it is just there as a reference to *remind* you about the different arguments and usage of functions that you have already learned about elsewhere.

```
Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file
into chunks.

Additional help can be found in the online docs for
`IO Tools <https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html>`_.

Parameters
----------
filepath_or_buffer : str, path object or file-like object
    Any valid string path is acceptable. The string could be a URL. Valid
    URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is
    expected. A local file could be: file://localhost/path/to/table.csv.

    If you want to pass in a path object, pandas accepts any ``os.PathLike``.

    By file-like object, we refer to objects with a ``read()`` method, such as
    a file handle (e.g. via builtin ``open`` function) or ``StringIO``.
sep : str, default ','
    Delimiter to use. If sep is None, the C engine cannot automatically detect
    the separator, but the Python parsing engine can, meaning the latter will
    be used and automatically detect the separator by Python's builtin sniffer
    tool, ``csv.Sniffer``. In addition, separators longer than 1 character and
    different from ``'\s+'`` will be interpreted as regular expressions and
    will also force the use of the Python parsing engine. Note that regex
    delimiters are prone to ignoring quoted data. Regex example: ``'\r\t'``.
delimiter : str, default ``None``
    Alias for sep.
header : int, list of int, None, default 'infer'
    Row number(s) to use as the column names, and the start of the
    data.  Default behavior is to infer the column names: if no names
    are passed the behavior is identical to ``header=0`` and column
    names are inferred from the first line of the file, if column
    names are passed explicitly then the behavior is identical to
    ``header=None``. Explicitly pass ``header=0`` to be able to
    replace existing names. The header can be a list of integers that
    specify row locations for a multi-index on the columns
    e.g. [0,1,3]. Intervening rows that are not specified will be
    skipped (e.g. 2 in this example is skipped). Note that this
    parameter ignores commented lines and empty lines if
```

FIGURE 1.14 The documentation for the read_csv function including a high-level description, a list of arguments and their meanings, and more.

If you are working in a Jupyter Lab environment, there are some conveniences that will help you lookup function names and access the documentation. First, rather than `help`, you can use the more concise `?` character. So, for example, to read the documentation for the `pd.read_csv` function, you can run the following code:

```
?pd.read_csv
```

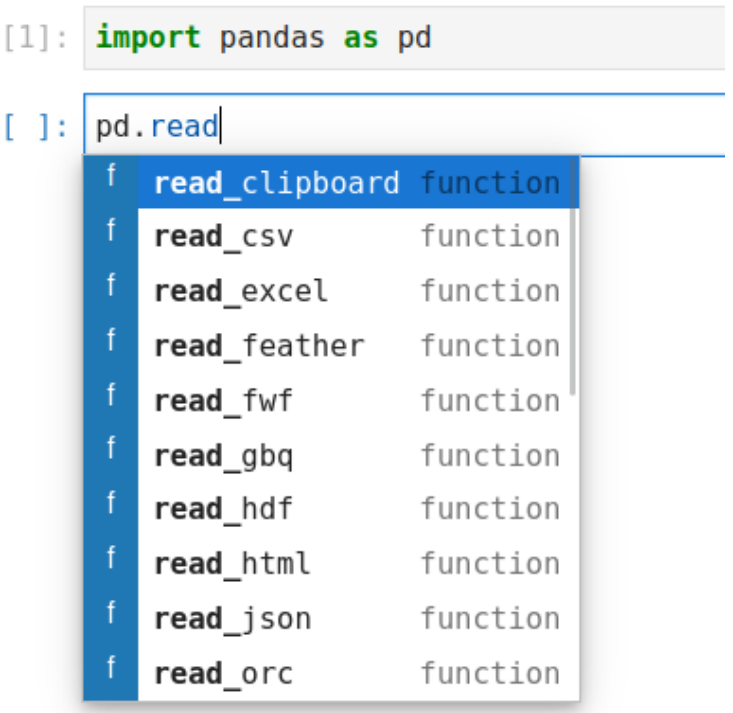


FIGURE 1.15 The suggestions that are shown after typing `pd.read` and pressing Tab.

You can also type the first characters of the function you want to use, and then press Tab to bring up small menu that shows you all the available functions that starts with those characters. This is helpful both for remembering function names and to prevent typos (Fig. 1.15).

To get more info on the function you want to use, you can type out the full name and then hold Shift while pressing Tab to bring up a help dialogue including the same information as when using `help()` (Fig. 1.16).

Finally, it can be helpful to have this help dialog open at all times, especially when you start out learning about programming and data science. You can achieve this by clicking on the `Help` text in the menu bar at the top and then selecting `Show Contextual Help`.

```
[1]: import pandas as pd

[ ]: pd.read_csv
```

```
Docstring:
Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file
into chunks.

Additional help can be found in the online docs for
`IO Tools <https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html>`_.

Parameters
----------
filepath_or_buffer : str, path object or file-like object
    Any valid string path is acceptable. The string could be a URL. Valid
    URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is
    expected. A local file could be: file://localhost/path/to/table.csv.

    If you want to pass in a path object, pandas accepts any ``os.PathLike``.

    By file-like object, we refer to objects with a ``read()`` method, such as
    a file handle (e.g. via builtin ``open`` function) or ``StringIO``.
```

FIGURE 1.16 The help dialog that is shown after typing `pd.read_csv` and then pressing Shift + Tab.

1.13 Exercises

Practice exercises for the material covered in this chapter can be found in the accompanying worksheets repository[5] in the "Python and Pandas" row. You can launch an interactive version of the worksheet in your browser by clicking the "launch binder" button. You can also preview a non-interactive version of the worksheet by clicking "view worksheet". If you instead decide to download the worksheet and run it on your own machine, make sure to follow the instructions for computer setup found in Chapter 13. This will ensure that the automated feedback and guidance that the worksheets provide will function as intended.

[5] https://worksheets.python.datasciencebook.ca

2

Reading in data locally and from the web

2.1 Overview

In this chapter, you'll learn to read tabular data of various formats into Python from your local device (e.g., your laptop) and the web. "Reading" (or "loading") is the process of converting data (stored as plain text, a database, HTML, etc.) into an object (e.g., a data frame) that Python can easily access and manipulate. Thus reading data is the gateway to any data analysis; you won't be able to analyze data unless you've loaded it first. And because there are many ways to store data, there are similarly many ways to read data into Python. The more time you spend upfront matching the data reading method to the type of data you have, the less time you will have to devote to re-formatting, cleaning and wrangling your data (the second step to all data analyses). It's like making sure your shoelaces are tied well before going for a run so that you don't trip later on.

2.2 Chapter learning objectives

By the end of the chapter, readers will be able to do the following:

- Define the types of path and use them to locate files:
  - absolute file path
  - relative file path
  - Uniform Resource Locator (URL)
- Read data into Python from various types of path using:
  - `read_csv`
  - `read_excel`
- Compare and contrast `read_csv` and `read_excel`.

DOI: 10.1201/9781003438397-2

- Describe when to use the following `read_csv` function arguments:
  - `skiprows`
  - `sep`
  - `header`
  - `names`
- Choose the appropriate `read_csv` function arguments to load a given plain text tabular data set into Python.
- Use the `rename` function to rename columns in a data frame.
- Use `pandas` package's `read_excel` function and arguments to load a sheet from an excel file into Python.
- Work with databases using functions from the `ibis` package:
  - Connect to a database with `connect`.
  - List tables in the database with `list_tables`.
  - Create a reference to a database table with `table`.
  - Bring data from a database into Python with `execute`.
- Use `to_csv` to save a data frame to a `.csv` file.
- (*Optional*) Obtain data from the web using scraping and application programming interfaces (APIs):
  - Read HTML source code from a URL using the `BeautifulSoup` package.
  - Read data from the NASA "Astronomy Picture of the Day" using the `requests` package.
  - Compare downloading tabular data from a plain text file (e.g., `.csv`), accessing data from an API, and scraping the HTML source code from a website.

2.3 Absolute and relative file paths

This chapter will discuss the different functions we can use to import data into Python, but before we can talk about *how* we read the data into Python with these functions, we first need to talk about *where* the data lives. When

you load a data set into Python, you first need to tell Python where those files live. The file could live on your computer (*local*) or somewhere on the internet (*remote*).

The place where the file lives on your computer is referred to as its "path". You can think of the path as directions to the file. There are two kinds of paths: *relative* paths and *absolute* paths. A relative path indicates where the file is with respect to your *working directory* (i.e., "where you are currently") on the computer. On the other hand, an absolute path indicates where the file is with respect to the computer's filesystem base (or *root*) folder, regardless of where you are working.

Suppose our computer's filesystem looks like the picture in Fig. 2.1. We are working in a file titled `project3.ipynb`, and our current working directory is `project3`; typically, as is the case here, the working directory is the directory containing the file you are currently working on.

Let's say we wanted to open the `happiness_report.csv` file. We have two options to indicate where the file is: using a relative path, or using an absolute path. The absolute path of the file always starts with a slash `/`—representing the root folder on the computer—and proceeds by listing out the sequence of folders you would have to enter to reach the file, each separated by another slash `/`. So in this case, `happiness_report.csv` would be reached by starting at the root, and entering the `home` folder, then the `dsci-100` folder, then the `project3` folder, and then finally the `data` folder. So its absolute path would be `/home/dsci-100/project3/data/happiness_report.csv`. We can load the file using its absolute path as a string passed to the `read_csv` function from `pandas`.

```
happy_data = pd.read_csv("/home/dsci-100/project3/data/happiness_report.csv")
```

If we instead wanted to use a relative path, we would need to list out the sequence of steps needed to get from our current working directory to the file, with slashes `/` separating each step. Since we are currently in the `project3` folder, we just need to enter the `data` folder to reach our desired file. Hence the relative path is `data/happiness_report.csv`, and we can load the file using its relative path as a string passed to `read_csv`.

```
happy_data = pd.read_csv("data/happiness_report.csv")
```

Note that there is no forward slash at the beginning of a relative path; if we accidentally typed `"/data/happiness_report.csv"`, Python would look for a folder named `data` in the root folder of the computer—but that doesn't exist.

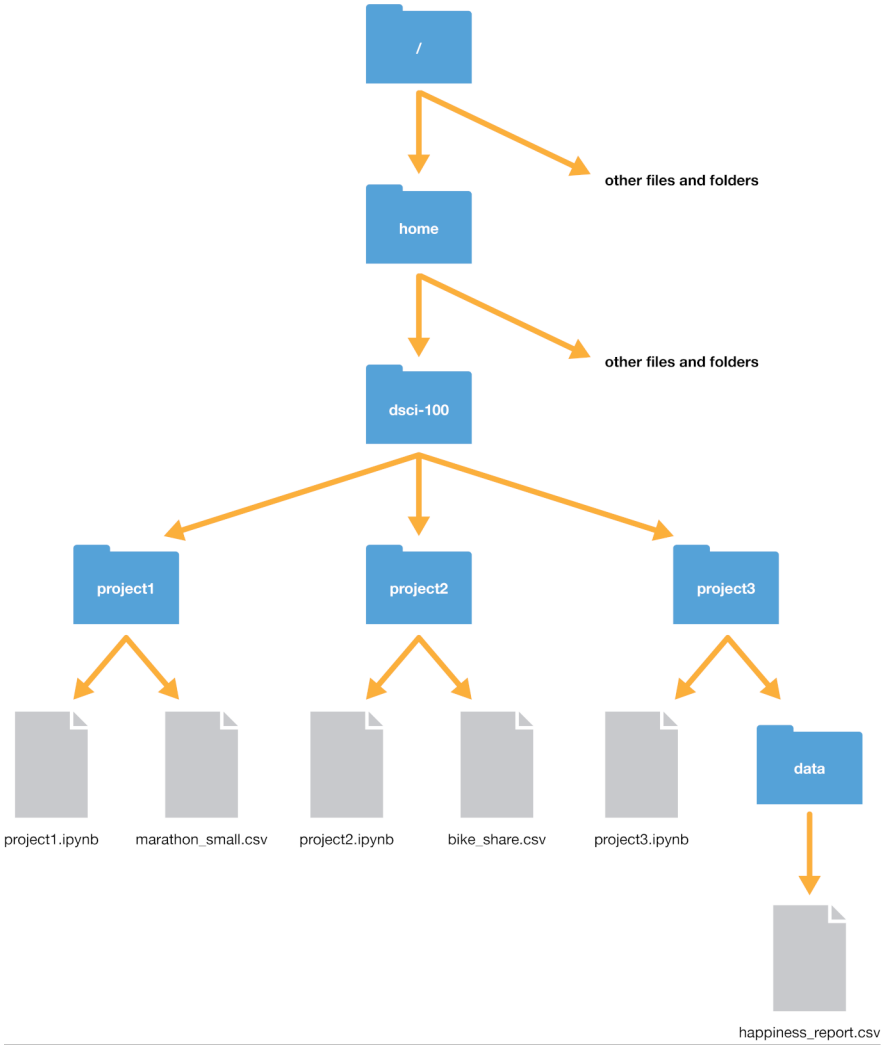


FIGURE 2.1 Example file system.

Aside from specifying places to go in a path using folder names (like `data` and `project3`), we can also specify two additional special places: the *current directory* and the *previous directory*. We indicate the current working directory with a single dot `.`, and the previous directory with two dots `..`. So for instance, if we wanted to reach the `bike_share.csv` file from the `project3` folder, we could use the relative path `../project2/bike_share.csv`. We can even combine these two; for example, we could reach the `bike_share.csv` file using the (very silly) path `../project2/../project2/./bike_share.csv` with quite a few redundant directions: it says to go back a folder, then open `project2`, then go back a folder again, then open `project2` again, then stay in the current directory, then finally get to `bike_share.csv`. Whew, what a long trip.

So which kind of path should you use: relative, or absolute? Generally speaking, you should use relative paths. Using a relative path helps ensure that your code can be run on a different computer (and as an added bonus, relative paths are often shorter—easier to type!). This is because a file's relative path is often the same across different computers, while a file's absolute path (the names of all of the folders between the computer's root, represented by `/`, and the file) isn't usually the same across different computers. For example, suppose Fatima and Jayden are working on a project together on the `happiness_report.csv` data. Fatima's file is stored at

```
/home/Fatima/project3/data/happiness_report.csv
```

while Jayden's is stored at

```
/home/Jayden/project3/data/happiness_report.csv
```

Even though Fatima and Jayden stored their files in the same place on their computers (in their home folders), the absolute paths are different due to their different usernames. If Jayden has code that loads the `happiness_report.csv` data using an absolute path, the code won't work on Fatima's computer. But the relative path from inside the `project3` folder (`data/happiness_report.csv`) is the same on both computers; any code that uses relative paths will work on both. In the additional resources section, we include a link to a short video on the difference between absolute and relative paths.

Beyond files stored on your computer (i.e., locally), we also need a way to locate resources stored elsewhere on the internet (i.e., remotely). For this purpose we use a *Uniform Resource Locator (URL)*, i.e., a web address that looks something like https://python.datasciencebook.ca/. URLs indicate the location of a resource on the internet, and start with a web domain, followed by a forward slash `/`, and then a path to where the resource is located on the remote machine.

2.4 Reading tabular data from a plain text file into Python

2.4.1 `read_csv` to read in comma-separated values files

Now that we have learned about *where* data could be, we will learn about *how* to import data into Python using various functions. Specifically, we will learn how to *read* tabular data from a plain text file (a document containing only text) *into* Python and *write* tabular data to a file *out of* Python. The

function we use to do this depends on the file's format. For example, in the last chapter, we learned about using the `read_csv` function from `pandas` when reading `.csv` (**c**omma-**s**eparated **v**alues) files. In that case, the *separator* that divided our columns was a comma (`,`). We only learned the case where the data matched the expected defaults of the `read_csv` function (column names are present, and commas are used as the separator between columns). In this section, we will learn how to read files that do not satisfy the default expectations of `read_csv`.

Before we jump into the cases where the data aren't in the expected default format for `pandas` and `read_csv`, let's revisit the more straightforward case where the defaults hold, and the only argument we need to give to the function is the path to the file, `data/can_lang.csv`. The `can_lang` data set contains language data from the 2016 Canadian census. We put `data/` before the file's name when we are loading the data set because this data set is located in a sub-folder, named `data`, relative to where we are running our Python code. Here is what the text in the file `data/can_lang.csv` looks like.

```
category,language,mother_tongue,most_at_home,most_at_work,lang_known
Aboriginal languages,"Aboriginal languages, n.o.s.",590,235,30,665
Non-Official & Non-Aboriginal languages,Afrikaans,10260,4785,85,23415
Non-Official & Non-Aboriginal languages,"Afro-Asiatic languages, n.i.e.",1150,44
Non-Official & Non-Aboriginal languages,Akan (Twi),13460,5985,25,22150
Non-Official & Non-Aboriginal languages,Albanian,26895,13135,345,31930
Aboriginal languages,"Algonquian languages, n.i.e.",45,10,0,120
Aboriginal languages,Algonquin,1260,370,40,2480
Non-Official & Non-Aboriginal languages,American Sign Language,2685,3020,1145,21
Non-Official & Non-Aboriginal languages,Amharic,22465,12785,200,33670
```

And here is a review of how we can use `read_csv` to load it into Python. First, we load the `pandas` package to gain access to useful functions for reading the data.

```
import pandas as pd
```

Next, we use `read_csv` to load the data into Python, and in that call we specify the relative path to the file.

```
canlang_data = pd.read_csv("data/can_lang.csv")
canlang_data
```

```
                                    category                          language ␣
↪\
0                       Aboriginal languages    Aboriginal languages, n.o.s.
1    Non-Official & Non-Aboriginal languages                       Afrikaans
2    Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3    Non-Official & Non-Aboriginal languages                     Akan (Twi)
4    Non-Official & Non-Aboriginal languages                       Albanian
..                                      ...                             ...
209  Non-Official & Non-Aboriginal languages                         Wolof
```

(continues on next page)

(continued from previous page)

```
210                     Aboriginal languages                      Woods Cree
211  Non-Official & Non-Aboriginal languages                 Wu (Shanghainese)
212  Non-Official & Non-Aboriginal languages                           Yiddish
213  Non-Official & Non-Aboriginal languages                            Yoruba

     mother_tongue  most_at_home  most_at_work  lang_known
0              590           235            30         665
1            10260          4785            85       23415
2             1150           445            10        2775
3            13460          5985            25       22150
4            26895         13135           345       31930
..             ...           ...           ...         ...
209           3990          1385            10        8240
210           1840           800            75        2665
211          12915          7650           105       16530
212          13555          7085           895       20985
213           9080          2615            15       22415

[214 rows x 6 columns]
```

2.4.2 Skipping rows when reading in data

Oftentimes, information about how data was collected, or other relevant information, is included at the top of the data file. This information is usually written in sentence and paragraph form, with no separator because it is not organized into columns. An example of this is shown below. This information gives the data scientist useful context and information about the data, however, it is not well formatted or intended to be read into a data frame cell along with the tabular data that follows later in the file.

```
Data source: https://ttimbers.github.io/canlang/
Data originally published in: Statistics Canada Census of Population 2016.
Reproduced and distributed on an as-is basis with their permission.
category,language,mother_tongue,most_at_home,most_at_work,lang_known
Aboriginal languages,"Aboriginal languages, n.o.s.",590,235,30,665
Non-Official & Non-Aboriginal languages,Afrikaans,10260,4785,85,23415
Non-Official & Non-Aboriginal languages,"Afro-Asiatic languages, n.i.e.",1150,445,
↪10,2775
Non-Official & Non-Aboriginal languages,Akan (Twi),13460,5985,25,22150
Non-Official & Non-Aboriginal languages,Albanian,26895,13135,345,31930
Aboriginal languages,"Algonquian languages, n.i.e.",45,10,0,120
Aboriginal languages,Algonquin,1260,370,40,2480
Non-Official & Non-Aboriginal languages,American Sign Language,2685,3020,1145,
↪21930
Non-Official & Non-Aboriginal languages,Amharic,22465,12785,200,33670
```

With this extra information being present at the top of the file, using `read_csv` as we did previously does not allow us to correctly load the data into Python. In the case of this file, Python just prints a `ParserError` message, indicating that it wasn't able to read the file.

```
canlang_data = pd.read_csv("data/can_lang_meta-data.csv")
```

```
ParserError: Error tokenizing data. C error: Expected 1 fields in line 4, saw␣
↪6
```

To successfully read data like this into Python, the `skiprows` argument can be useful to tell Python how many rows to skip before it should start reading in the data. In the example above, we would set this value to 3 to read and load the data correctly.

```
canlang_data = pd.read_csv("data/can_lang_meta-data.csv", skiprows=3)
canlang_data
```

```
                                    category                         language ␣
↪\
0                       Aboriginal languages      Aboriginal languages, n.o.s.
1    Non-Official & Non-Aboriginal languages                         Afrikaans
2    Non-Official & Non-Aboriginal languages    Afro-Asiatic languages, n.i.e.
3    Non-Official & Non-Aboriginal languages                       Akan (Twi)
4    Non-Official & Non-Aboriginal languages                         Albanian
..                                      ...                              ...
209  Non-Official & Non-Aboriginal languages                           Wolof
210                     Aboriginal languages                     Woods Cree
211  Non-Official & Non-Aboriginal languages                 Wu (Shanghainese)
212  Non-Official & Non-Aboriginal languages                         Yiddish
213  Non-Official & Non-Aboriginal languages                          Yoruba

     mother_tongue  most_at_home  most_at_work  lang_known
0              590           235            30         665
1            10260          4785            85       23415
2             1150           445            10        2775
3            13460          5985            25       22150
4            26895         13135           345       31930
..             ...           ...           ...         ...
209           3990          1385            10        8240
210           1840           800            75        2665
211          12915          7650           105       16530
212          13555          7085           895       20985
213           9080          2615            15       22415

[214 rows x 6 columns]
```

How did we know to skip three rows? We looked at the data. The first three rows of the data had information we didn't need to import:

```
Data source: https://ttimbers.github.io/canlang/
Data originally published in: Statistics Canada Census of Population 2016.
Reproduced and distributed on an as-is basis with their permission.
```

The column names began at row 4, so we skipped the first three rows.

2.4.3 Using the **sep** argument for different separators

Another common way data is stored is with tabs as the separator. Notice the data file, `can_lang.tsv`, has tabs in between the columns instead of commas.

```
category        language        mother_tongue        most_at_home        most_at_
↪work        lang_known
Aboriginal languages        Aboriginal languages, n.o.s.
↪        590        235        30        665
Non-Official & Non-Aboriginal␣
↪languages        Afrikaans        10260        4785        85        23415
Non-Official & Non-Aboriginal languages        Afro-Asiatic languages, n.i.e.
↪        1150        445        10        2775
Non-Official & Non-Aboriginal languages        Akan␣
↪(Twi)        13460        5985        25        22150
Non-Official & Non-Aboriginal␣
↪languages        Albanian        26895        13135        345        31930
Aboriginal languages        Algonquian languages, n.i.e.
↪        45        10        0        120
Aboriginal languages        Algonquin        1260        370        40        2480
Non-Official & Non-Aboriginal languages        American Sign␣
↪Language        2685        3020        1145        21930
Non-Official & Non-Aboriginal␣
↪languages        Amharic        22465        12785        200        33670
```

To read in `.tsv` (**t**ab **s**eparated **v**alues) files, we can set the `sep` argument in the `read_csv` function to the *tab character* `\t`.

Note: `\t` is an example of an *escaped character*, which always starts with a backslash (`\`). Escaped characters are used to represent non-printing characters (like the tab) or characters with special meanings (such as quotation marks).

```
canlang_data = pd.read_csv("data/can_lang.tsv", sep="\t")
canlang_data
```

```
                                    category                         language ␣
↪\
0                       Aboriginal languages    Aboriginal languages, n.o.s.
1    Non-Official & Non-Aboriginal languages                        Afrikaans
2    Non-Official & Non-Aboriginal languages  Afro-Aitic languages, n.i.e.
3    Non-Official & Non-Aboriginal languages                      Akan (Twi)
4    Non-Official & Non-Aboriginal languages                        Albanian
..                                       ...                             ...
209  Non-Official & Non-Aboriginal languages                           Wolof
210                     Aboriginal languages                     Woods Cree
211  Non-Official & Non-Aboriginal languages                Wu (Shanghainese)
212  Non-Official & Non-Aboriginal languages                         Yiddish
213  Non-Official & Non-Aboriginal languages                           Yoruba

     mother_tongue  most_at_home  most_at_work  lang_known
0              590           235            30         665
1            10260          4785            85       23415
2             1150           445            10        2775
3            13460          5985            25       22150
4            26895         13135           345       31930
..             ...           ...           ...         ...
209           3990          1385            10        8240
210           1840           800            75        2665
```

(continues on next page)

(continued from previous page)

```
211          12915          7650          105       16530
212          13555          7085          895       20985
213           9080          2615           15       22415

[214 rows x 6 columns]
```

If you compare the data frame here to the data frame we obtained in Section 2.4.1 using `read_csv`, you'll notice that they look identical: they have the same number of columns and rows, the same column names, and the same entries. So even though we needed to use different arguments depending on the file format, our resulting data frame (`canlang_data`) in both cases was the same.

2.4.4 Using the `header` argument to handle missing column names

The `can_lang_no_names.tsv` file contains a slightly different version of this data set, except with no column names, and tabs for separators. Here is how the file looks in a text editor:

```
Aboriginal languages        Aboriginal languages, n.o.s.
 ↪        590        235        30        665
Non-Official & Non-Aboriginal␣
 ↪languages        Afrikaans        10260        4785        85        23415
Non-Official & Non-Aboriginal languages        Afro-Asiatic languages, n.i.e.
 ↪        1150        445        10        2775
Non-Official & Non-Aboriginal languages        Akan␣
 ↪(Twi)        13460        5985        25        22150
Non-Official & Non-Aboriginal␣
 ↪languages        Albanian        26895        13135        345        31930
Aboriginal languages        Algonquian languages, n.i.e.
 ↪        45        10        0        120
Aboriginal languages        Algonquin        1260        370        40        2480
Non-Official & Non-Aboriginal languages        American Sign␣
 ↪Language        2685        3020        1145        21930
Non-Official & Non-Aboriginal␣
 ↪languages        Amharic        22465        12785        200        33670
```

Data frames in Python need to have column names. Thus if you read in data without column names, Python will assign names automatically. In this example, Python assigns the column names `0, 1, 2, 3, 4, 5`. To read this data into Python, we specify the first argument as the path to the file (as done with `read_csv`), and then provide values to the `sep` argument (here a tab, which we represent by `"\t"`), and finally set `header = None` to tell `pandas` that the data file does not contain its own column names.

```
canlang_data = pd.read_csv(
    "data/can_lang_no_names.tsv",
    sep="\t",
    header=None
)
canlang_data
```

```
                                                    0                                 1 ␣
↪\
0                          Aboriginal languages    Aboriginal languages, n.o.s.
1    Non-Official & Non-Aboriginal languages                          Afrikaans
2    Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3    Non-Official & Non-Aboriginal languages                        Akan (Twi)
4    Non-Official & Non-Aboriginal languages                          Albanian
..                                       ...                               ...
209  Non-Official & Non-Aboriginal languages                            Wolof
210                        Aboriginal languages                     Woods Cree
211  Non-Official & Non-Aboriginal languages                Wu (Shanghainese)
212  Non-Official & Non-Aboriginal languages                          Yiddish
213  Non-Official & Non-Aboriginal languages                            Yoruba

          2      3    4      5
0       590    235   30    665
1     10260   4785   85  23415
2      1150    445   10   2775
3     13460   5985   25  22150
4     26895  13135  345  31930
..      ...    ...  ...    ...
209    3990   1385   10   8240
210    1840    800   75   2665
211   12915   7650  105  16530
212   13555   7085  895  20985
213    9080   2615   15  22415

[214 rows x 6 columns]
```

It is best to rename your columns manually in this scenario. The current column names (`0, 1`, etc.) are problematic for two reasons: first, because they not very descriptive names, which will make your analysis confusing; and second, because your column names should generally be *strings*, but are currently *integers*. To rename your columns, you can use the `rename` function from the pandas package[1]. The argument of the `rename` function is `columns`, which takes a mapping between the old column names and the new column names. In this case, we want to rename the old columns (`0, 1, ..., 5`) in the `canlang_data` data frame to more descriptive names.

To specify the mapping, we create a *dictionary*: a Python object that represents a mapping from *keys* to *values*. We can create a dictionary by using a pair of curly braces `{ }`, and inside the braces placing pairs of `key : value` separated by commas. Below, we create a dictionary called `col_map` that maps the old column names in `canlang_data` to new column names, and then pass it to the `rename` function.

```
col_map = {
    0 : "category",
    1 : "language",
    2 : "mother_tongue",
    3 : "most_at_home",
```

(continues on next page)

[1] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.rename.html#

(continued from previous page)

```
    4 : "most_at_work",
    5 : "lang_known"
}
canlang_data_renamed = canlang_data.rename(columns=col_map)
canlang_data_renamed
```

```
                                    category                         language␣
↪\
0                        Aboriginal languages      Aboriginal languages, n.o.s.
1    Non-Official & Non-Aboriginal languages                         Afrikaans
2    Non-Official & Non-Aboriginal languages    Afro-Asiatic languages, n.i.e.
3    Non-Official & Non-Aboriginal languages                       Akan (Twi)
4    Non-Official & Non-Aboriginal languages                         Albanian
..                                      ...                              ...
209  Non-Official & Non-Aboriginal languages                           Wolof
210                      Aboriginal languages                     Woods Cree
211  Non-Official & Non-Aboriginal languages                 Wu (Shanghainese)
212  Non-Official & Non-Aboriginal languages                          Yiddish
213  Non-Official & Non-Aboriginal languages                           Yoruba

     mother_tongue  most_at_home  most_at_work  lang_known
0              590           235            30         665
1            10260          4785            85       23415
2             1150           445            10        2775
3            13460          5985            25       22150
4            26895         13135           345       31930
..             ...           ...           ...         ...
209           3990          1385            10        8240
210           1840           800            75        2665
211          12915          7650           105       16530
212          13555          7085           895       20985
213           9080          2615            15       22415

[214 rows x 6 columns]
```

The column names can also be assigned to the data frame immediately upon reading it from the file by passing a list of column names to the `names` argument in `read_csv`.

```
canlang_data = pd.read_csv(
    "data/can_lang_no_names.tsv",
    sep="\t",
    header=None,
    names=[
        "category",
        "language",
        "mother_tongue",
        "most_at_home",
        "most_at_work",
        "lang_known",
    ],
)
canlang_data
```

```
                                    category                         language␣
↪\
0                        Aboriginal languages      Aboriginal languages, n.o.s.
```

(continues on next page)

(continued from previous page)

```
1    Non-Official & Non-Aboriginal languages                      Afrikaans
2    Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3    Non-Official & Non-Aboriginal languages                    Akan (Twi)
4    Non-Official & Non-Aboriginal languages                      Albanian
..                                       ...                           ...
209  Non-Official & Non-Aboriginal languages                        Wolof
210                  Aboriginal languages                    Woods Cree
211  Non-Official & Non-Aboriginal languages            Wu (Shanghainese)
212  Non-Official & Non-Aboriginal languages                      Yiddish
213  Non-Official & Non-Aboriginal languages                       Yoruba

     mother_tongue  most_at_home  most_at_work  lang_known
0              590           235            30         665
1            10260          4785            85       23415
2             1150           445            10        2775
3            13460          5985            25       22150
4            26895         13135           345       31930
..             ...           ...           ...         ...
209           3990          1385            10        8240
210           1840           800            75        2665
211          12915          7650           105       16530
212          13555          7085           895       20985
213           9080          2615            15       22415

[214 rows x 6 columns]
```

2.4.5 Reading tabular data directly from a URL

We can also use `read_csv` to read in data directly from a **U**niform **R**esource **L**ocator (URL) that contains tabular data. Here, we provide the URL of a remote file to `read_csv`, instead of a path to a local file on our computer. We need to surround the URL with quotes similar to when we specify a path on our local computer. All other arguments that we use are the same as when using these functions with a local file on our computer.

```
url = "https://raw.githubusercontent.com/UBC-DSCI/introduction-to-datascience-
↪python/reading/source/data/can_lang.csv"
pd.read_csv(url)
canlang_data = pd.read_csv(url)

canlang_data
```

```
                                  category                      language ␣
↪\
0                    Aboriginal languages    Aboriginal languages, n.o.s.
1    Non-Official & Non-Aboriginal languages                     Afrikaans
2    Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3    Non-Official & Non-Aboriginal languages                   Akan (Twi)
4    Non-Official & Non-Aboriginal languages                     Albanian
..                                       ...                           ...
209  Non-Official & Non-Aboriginal languages                       Wolof
210                  Aboriginal languages                   Woods Cree
211  Non-Official & Non-Aboriginal languages           Wu (Shanghainese)
212  Non-Official & Non-Aboriginal languages                     Yiddish
213  Non-Official & Non-Aboriginal languages                      Yoruba
```

(continues on next page)

(continued from previous page)

```
     mother_tongue  most_at_home  most_at_work  lang_known
0              590           235            30         665
1            10260          4785            85       23415
2             1150           445            10        2775
3            13460          5985            25       22150
4            26895         13135           345       31930
..             ...           ...           ...         ...
209           3990          1385            10        8240
210           1840           800            75        2665
211          12915          7650           105       16530
212          13555          7085           895       20985
213           9080          2615            15       22415

[214 rows x 6 columns]
```

2.4.6 Previewing a data file before reading it into Python

In many of the examples above, we gave you previews of the data file before we read it into Python. Previewing data is essential to see whether or not there are column names, what the separators are, and if there are rows you need to skip. You should do this yourself when trying to read in data files: open the file in whichever text editor you prefer to inspect its contents prior to reading it into Python.

2.5 Reading tabular data from a Microsoft Excel file

There are many other ways to store tabular data sets beyond plain text files, and similarly, many ways to load those data sets into Python. For example, it is very common to encounter, and need to load into Python, data stored as a Microsoft Excel spreadsheet (with the file name extension `.xlsx`). To be able to do this, a key thing to know is that even though `.csv` and `.xlsx` files look almost identical when loaded into Excel, the data themselves are stored completely differently. While `.csv` files are plain text files, where the characters you see when you open the file in a text editor are exactly the data they represent, this is not the case for `.xlsx` files. Take a look at a snippet of what a `.xlsx` file would look like in a text editor:

```
,?'O
    _rels/.rels???J1??>E?{7?
<?V????w8?'J???'QrJ???Tf?d??d?o?wZ'???@>?4'?|??hlIo??F
t                                                    8f??3wn
????t??u"/
          %~Ed2??<?w??
                        ?Pd(??J-?E???7?'t(?-GZ?????y???c~N?g[^_r?4
                                                              yG?O
```

(continues on next page)

(continued from previous page)

```
                                                                    ?K??G?


    ]TUEe??O??c[???????6q??s??d?m???\???H?^????3} ?rZY? ?:L60?^?????XTP+?|?
X?a??4VT?,D?Jq
```

This type of file representation allows Excel files to store additional things that you cannot store in a `.csv` file, such as fonts, text formatting, graphics, multiple sheets, and more. And despite looking odd in a plain text editor, we can read Excel spreadsheets into Python using the `pandas` package's `read_excel` function developed specifically for this purpose.

```
canlang_data = pd.read_excel("data/can_lang.xlsx")
canlang_data
```

```
                                    category                         language␣
↪\
0                       Aboriginal languages    Aboriginal languages, n.o.s.
1    Non-Official & Non-Aboriginal languages                       Afrikaans
2    Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3    Non-Official & Non-Aboriginal languages                     Akan (Twi)
4    Non-Official & Non-Aboriginal languages                       Albanian
..                                      ...                            ...
209  Non-Official & Non-Aboriginal languages                         Wolof
210                     Aboriginal languages                   Woods Cree
211  Non-Official & Non-Aboriginal languages              Wu (Shanghainese)
212  Non-Official & Non-Aboriginal languages                        Yiddish
213  Non-Official & Non-Aboriginal languages                         Yoruba

     mother_tongue  most_at_home  most_at_work  lang_known
0              590           235            30         665
1            10260          4785            85       23415
2             1150           445            10        2775
3            13460          5985            25       22150
4            26895         13135           345       31930
..             ...           ...           ...         ...
209           3990          1385            10        8240
210           1840           800            75        2665
211          12915          7650           105       16530
212          13555          7085           895       20985
213           9080          2615            15       22415

[214 rows x 6 columns]
```

If the `.xlsx` file has multiple sheets, you have to use the `sheet_name` argument to specify the sheet number or name. This functionality is useful when a single sheet contains multiple tables (a sad thing that happens to many Excel spreadsheets since this makes reading in data more difficult). You can also specify cell ranges using the `usecols` argument (e.g., `usecols="A:D"` for including columns from `A` to `D`).

As with plain text files, you should always explore the data file before importing it into Python. Exploring the data beforehand helps you decide which

arguments you need to load the data into Python successfully. If you do not have the Excel program on your computer, you can use other programs to preview the file. Examples include Google Sheets and Libre Office.

In Table 2.1 we summarize the `read_csv` and `read_excel` functions we covered in this chapter. We also include the arguments for data separated by semicolons `;`, which you may run into with data sets where the decimal is represented by a comma instead of a period (as with some data sets from European countries).

TABLE 2.1 Summary of read_csv and read_excel

Data File Type	Python Function	Arguments
Comma (`,`) separated files	`read_csv`	just the file path
Tab (`\t`) separated files	`read_csv`	`sep="\t"`
Missing header	`read_csv`	`header=None`
European-style numbers, semicolon (`;`) separators	`read_csv`	`sep=";", thousands=".", decimal=","`
Excel files (`.xlsx`)	`read_excel`	`sheet_name, usecols`

2.6 Reading data from a database

Another very common form of data storage is the relational database. Databases are great when you have large data sets or multiple users working on a project. There are many relational database management systems, such as SQLite, MySQL, PostgreSQL, Oracle, and many more. These different relational database management systems each have their own advantages and limitations. Almost all employ SQL (*structured query language*) to obtain data from the database. But you don't need to know SQL to analyze data from a database; several packages have been written that allow you to connect to relational databases and use the Python programming language to obtain data. In this book, we will give examples of how to do this using Python with SQLite and PostgreSQL databases.

2.6.1 Reading data from a SQLite database

SQLite is probably the simplest relational database system that one can use in combination with Python. SQLite databases are self-contained, and are usually stored and accessed locally on one computer from a file with a `.db`

extension (or sometimes a `.sqlite` extension). Similar to Excel files, these are not plain text files and cannot be read in a plain text editor.

The first thing you need to do to read data into Python from a database is to connect to the database. For an SQLite database, we will do that using the `connect` function from the `sqlite` backend in the `ibis` package. This command does not read in the data, but simply tells Python where the database is and opens up a communication channel that Python can use to send SQL commands to the database.

Note: There is another database package in python called `sqlalchemy`. That package is a bit more mature than `ibis`, so if you want to dig deeper into working with databases in Python, that is a good next package to learn about. We will work with `ibis` in this book, as it provides a more modern and friendlier syntax that is more like `pandas` for data analysis code.

```
import ibis

conn = ibis.sqlite.connect("data/can_lang.db")
```

Often relational databases have many tables; thus, in order to retrieve data from a database, you need to know the name of the table in which the data is stored. You can get the names of all the tables in the database using the `list_tables` function:

```
tables = conn.list_tables()
tables
```

```
['can_lang']
```

The `list_tables` function returned only one name—`"can_lang"`—which tells us that there is only one table in this database. To reference a table in the database (so that we can perform operations like selecting columns and filtering rows), we use the `table` function from the `conn` object. The object returned by the `table` function allows us to work with data stored in databases as if they were just regular `pandas` data frames; but secretly, behind the scenes, `ibis` will turn your commands into SQL queries.

```
canlang_table = conn.table("can_lang")
canlang_table
```

```
DatabaseTable: can_lang
  category      string
  language      string
```

(continues on next page)

(continued from previous page)

```
  mother_tongue float64
  most_at_home  float64
  most_at_work  float64
  lang_known    float64
```

Although it looks like we might have obtained the whole data frame from the database, we didn't. It's a *reference*; the data is still stored only in the SQLite database. The `canlang_table` object is a `DatabaseTable`, which, when printed, tells you which columns are available in the table. But unlike a usual `pandas` data frame, we do not immediately know how many rows are in the table. In order to find out how many rows there are, we have to send an SQL *query* (i.e., command) to the data base. In `ibis`, we can do that using the `count` function from the table object.

```
canlang_table.count()
```

```
r0 := DatabaseTable: can_lang
  category      string
  language      string
  mother_tongue float64
  most_at_home  float64
  most_at_work  float64
  lang_known    float64

CountStar(can_lang): CountStar(r0)
```

Wait a second … this isn't the number of rows in the database. In fact, we haven't actually sent our SQL query to the database yet. We need to explicitly tell `ibis` when we want to send the query. The reason for this is that databases are often more efficient at working with (i.e., selecting, filtering, joining, etc.) large data sets than Python. And typically, the database will not even be stored on your computer, but rather a more powerful machine somewhere on the web. So `ibis` is lazy and waits to bring this data into memory until you explicitly tell it to using the `execute` function. The `execute` function actually sends the SQL query to the database, and gives you the result. Let's look at the number of rows in the table by executing the `count` command.

```
canlang_table.count().execute()
```

```
214
```

There we go. There are 214 rows in the `can_lang` table. If you are interested in seeing the *actual* text of the SQL query that `ibis` sends to the database, you can use the `compile` function instead of `execute`. But note that you have to pass the result of `compile` to the `str` function to turn it into a human-readable string first.

```
str(canlang_table.count().compile())
```

```
'SELECT count(*) AS "CountStar(can_lang)" \nFROM can_lang AS t0'
```

The output above shows the SQL code that is sent to the database. When we write `canlang_table.count().execute()` in Python, in the background, the `execute` function is translating the Python code into SQL, sending that SQL to the database, and then translating the response for us. So `ibis` does all the hard work of translating from Python to SQL and back for us; we can just stick with Python.

The `ibis` package provides lots of `pandas`-like tools for working with database tables. For example, we can look at the first few rows of the table by using the `head` function, followed by `execute` to retrieve the response.

```
canlang_table.head(10).execute()
```

```
                                category                          language  \
0                     Aboriginal languages    Aboriginal languages, n.o.s.
1  Non-Official & Non-Aboriginal languages                       Afrikaans
2  Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3  Non-Official & Non-Aboriginal languages                     Akan (Twi)
4  Non-Official & Non-Aboriginal languages                       Albanian
5                     Aboriginal languages    Algonquian languages, n.i.e.
6                     Aboriginal languages                       Algonquin
7  Non-Official & Non-Aboriginal languages          American Sign Language
8  Non-Official & Non-Aboriginal languages                        Amharic
9  Non-Official & Non-Aboriginal languages                         Arabic

   mother_tongue  most_at_home  most_at_work  lang_known
0          590.0         235.0          30.0       665.0
1        10260.0        4785.0          85.0     23415.0
2         1150.0         445.0          10.0      2775.0
3        13460.0        5985.0          25.0     22150.0
4        26895.0       13135.0         345.0     31930.0
5           45.0          10.0           0.0       120.0
6         1260.0         370.0          40.0      2480.0
7         2685.0        3020.0        1145.0     21930.0
8        22465.0       12785.0         200.0     33670.0
9       419890.0      223535.0        5585.0    629055.0
```

You can see that `ibis` actually returned a `pandas` data frame to us after we executed the query, which is very convenient for working with the data after getting it from the database. So now that we have the `canlang_table` table reference for the 2016 Canadian Census data in hand, we can mostly continue onward as if it were a regular data frame. For example, let's do the same exercise from Chapter 1: we will obtain only those rows corresponding to Aboriginal languages, and keep only the `language` and `mother_tongue` columns. We can use the `[]` operation with a logical statement to obtain only certain rows. Below we filter the data to include only Aboriginal languages.

```
canlang_table_filtered = canlang_table[canlang_table["category"] == "Aboriginal␣
↪languages"]
canlang_table_filtered
```

```
r0 := DatabaseTable: can_lang
  category      string
  language      string
  mother_tongue float64
  most_at_home  float64
  most_at_work  float64
  lang_known    float64

Selection[r0]
  predicates:
    r0.category == 'Aboriginal languages'
```

Above you can see that we have not yet executed this command; `canlang_table_filtered` is just showing the first part of our query (the part that starts with `Selection[r0]` above). We didn't call `execute` because we are not ready to bring the data into Python yet. We can still use the database to do some work to obtain *only* the small amount of data we want to work with locally in Python. Let's add the second part of our SQL query: selecting only the `language` and `mother_tongue` columns.

```
canlang_table_selected = canlang_table_filtered[["language", "mother_tongue"]]
canlang_table_selected
```

```
r0 := DatabaseTable: can_lang
  category      string
  language      string
  mother_tongue float64
  most_at_home  float64
  most_at_work  float64
  lang_known    float64

r1 := Selection[r0]
  predicates:
    r0.category == 'Aboriginal languages'

Selection[r1]
  selections:
    language:      r1.language
    mother_tongue: r1.mother_tongue
```

Now you can see that the `ibis` query will have two steps: it will first find rows corresponding to Aboriginal languages, then it will extract only the `language` and `mother_tongue` columns that we are interested in. Let's actually execute the query now to bring the data into Python as a `pandas` data frame, and print the result.

```
aboriginal_lang_data = canlang_table_selected.execute()
aboriginal_lang_data
```

```
                          language  mother_tongue
0   Aboriginal languages, n.o.s.          590.0
1   Algonquian languages, n.i.e.           45.0
2                       Algonquin         1260.0
3   Athabaskan languages, n.i.e.           50.0
4                       Atikamekw         6150.0
..                            ...           ...
62        Thompson (Ntlakapamux)          335.0
63                       Tlingit           95.0
64                      Tsimshian          200.0
65    Wakashan languages, n.i.e.           10.0
66                    Woods Cree         1840.0

[67 rows x 2 columns]
```

`ibis` provides many more functions (not just the `[]` operation) that you can use to manipulate the data within the database before calling `execute` to obtain the data in Python. But `ibis` does not provide *every* function that we need for analysis; we do eventually need to call `execute`. For example, `ibis` does not provide the `tail` function to look at the last rows in a database, even though `pandas` does.

```
canlang_table_selected.tail(6)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[24], line 1
----> 1 canlang_table_selected.tail(6)

File /opt/conda/lib/python3.11/site-packages/ibis/expr/types/relations.py:645,
 ↪ in Table.__getattr__(self, key)
    641     hint = common_typos[key]
    642     raise AttributeError(
    643         f"{type(self).__name__} object has no attribute {key!r}, did␣
 ↪you mean {hint!r}"
    644     )
--> 645 raise AttributeError(f"'Table' object has no attribute {key!r}")

AttributeError: 'Table' object has no attribute 'tail'
```

```
aboriginal_lang_data.tail(6)
```

```
                      language  mother_tongue
61                     Tahltan          95.0
62      Thompson (Ntlakapamux)         335.0
63                     Tlingit          95.0
64                   Tsimshian         200.0
65  Wakashan languages, n.i.e.          10.0
66                 Woods Cree        1840.0
```

So once you have finished your data wrangling of the database reference object, it is advisable to bring it into Python as a `pandas` data frame using the `execute` function. But be very careful using `execute`: databases are often *very* big, and reading an entire table into Python might take a long time to run

or even possibly crash your machine. So make sure you select and filter the database table to reduce the data to a reasonable size before using `execute` to read it into Python.

2.6.2 Reading data from a PostgreSQL database

PostgreSQL (also called Postgres) is a very popular and open-source option for relational database software. Unlike SQLite, PostgreSQL uses a client–server database engine, as it was designed to be used and accessed on a network. This means that you have to provide more information to Python when connecting to Postgres databases. The additional information that you need to include when you call the `connect` function is listed below:

- `database`: the name of the database (a single PostgreSQL instance can host more than one database)
- `host`: the URL pointing to where the database is located (`localhost` if it is on your local machine)
- `port`: the communication endpoint between Python and the PostgreSQL database (usually `5432`)
- `user`: the username for accessing the database
- `password`: the password for accessing the database

Below we demonstrate how to connect to a version of the `can_mov_db` database, which contains information about Canadian movies. Note that the `host` (`fakeserver.stat.ubc.ca`), `user` (`user0001`), and `password` (`abc123`) below are *not real*; you will not actually be able to connect to a database using this information.

```
conn = ibis.postgres.connect(
    database="can_mov_db",
    host="fakeserver.stat.ubc.ca",
    port=5432,
    user="user0001",
    password="abc123"
)
```

Aside from needing to provide that additional information, `ibis` makes it so that connecting to and working with a Postgres database is identical to connecting to and working with an SQLite database. For example, we can again use `list_tables` to find out what tables are in the `can_mov_db` database:

```
conn.list_tables()
```

```
["themes", "medium", "titles", "title_aliases", "forms", "episodes", "names",
↪"names_occupations", "occupation", "ratings"]
```

We see that there are 10 tables in this database. Let's first look at the `"ratings"` table to find the lowest rating that exists in the `can_mov_db` database.

```
ratings_table = conn.table("ratings")
ratings_table
```

```
AlchemyTable: ratings
  title           string
  average_rating  float64
  num_votes       int64
```

To find the lowest rating that exists in the data base, we first need to select the `average_rating` column:

```
avg_rating = ratings_table[["average_rating"]]
avg_rating
```

```
r0 := AlchemyTable: ratings
  title           string
  average_rating  float64
  num_votes       int64

Selection[r0]
  selections:
    average_rating: r0.average_rating
```

Next, we use the `order_by` function from `ibis` order the table by `average_rating`, and then the `head` function to select the first row (i.e., the lowest score).

```
lowest = avg_rating.order_by("average_rating").head(1)
lowest.execute()
```

```
   average_rating
0             1.0
```

We see the lowest rating given to a movie is 1, indicating that it must have been a really bad movie …

2.6.3 Why should we bother with databases at all?

Opening a database involved a lot more effort than just opening a `.csv`, or any of the other plain text or Excel formats. We had to open a connection to the database, then use `ibis` to translate `pandas`-like commands (the `[]` operation, `head`, etc.) into SQL queries that the database understands, and then finally `execute` them. And not all `pandas` commands can currently be

translated via `ibis` into database queries. So you might be wondering: why should we use databases at all?

Databases are beneficial in a large-scale setting:

- They enable storing large data sets across multiple computers with backups.
- They provide mechanisms for ensuring data integrity and validating input.
- They provide security and data access control.
- They allow multiple users to access data simultaneously and remotely without conflicts and errors. For example, there are billions of Google searches conducted daily in 2021 [Real Time Statistics Project, 2021]. Can you imagine if Google stored all of the data from those searches in a single `.csv` file!? Chaos would ensue.

2.7 Writing data from Python to a `.csv` file

At the middle and end of a data analysis, we often want to write a data frame that has changed (through selecting columns, filtering rows, etc.) to a file to share it with others or use it for another step in the analysis. The most straightforward way to do this is to use the `to_csv` function from the `pandas` package. The default arguments are to use a comma (`,`) as the separator, and to include column names in the first row. We also specify `index = False` to tell `pandas` not to print row numbers in the `.csv` file. Below we demonstrate creating a new version of the Canadian languages data set without the "Official languages" category according to the Canadian 2016 Census, and then writing this to a `.csv` file:

```
no_official_lang_data = canlang_data[canlang_data["category"] != "Official␣
↪languages"]
no_official_lang_data.to_csv("data/no_official_languages.csv", index=False)
```

2.8 Obtaining data from the web

Note: This section is not required reading for the remainder of the textbook. It is included for those readers interested in learning a little bit more about how to obtain different types of data from the web.

Data doesn't just magically appear on your computer; you need to get it from somewhere. Earlier in the chapter we showed you how to access data stored in a plain text, spreadsheet-like format (e.g., comma- or tab-separated) from a web URL using the `read_csv` function from `pandas`. But as time goes on, it is increasingly uncommon to find data (especially large amounts of data) in this format available for download from a URL. Instead, websites now often offer something known as an **a**pplication **p**rogramming **i**nterface (API), which provides a programmatic way to ask for subsets of a data set. This allows the website owner to control *who* has access to the data, *what portion* of the data they have access to, and *how much* data they can access. Typically, the website owner will give you a *token* or *key* (a secret string of characters somewhat like a password) that you have to provide when accessing the API.

Another interesting thought: websites themselves *are* data. When you type a URL into your browser window, your browser asks the *web server* (another computer on the internet whose job it is to respond to requests for the website) to give it the website's data, and then your browser translates that data into something you can see. If the website shows you some information that you're interested in, you could *create* a data set for yourself by copying and pasting that information into a file. This process of taking information directly from what a website displays is called *web scraping* (or sometimes *screen scraping*). Now, of course, copying and pasting information manually is a painstaking and error-prone process, especially when there is a lot of information to gather. So instead of asking your browser to translate the information that the web server provides into something you can see, you can collect that data programmatically—in the form of **h**yper**t**ext **m**arkup **l**anguage (HTML) and **c**ascading **s**tyle **s**heet (CSS) code—and process it to extract useful information. HTML provides the basic structure of a site and tells the webpage how to display the content (e.g., titles, paragraphs, bullet lists, etc.), whereas CSS helps style the content and tells the webpage how the HTML elements should be presented (e.g., colors, layouts, fonts, etc.).

This subsection will show you the basics of both web scraping with the `BeautifulSoup` Python package[2] [Richardson, 2007] and accessing the NASA "Astronomy Picture of the Day" API using the `requests` Python package[3] [Reitz and The Python Software Foundation, Accessed Online: 2023].

[2] https://beautiful-soup-4.readthedocs.io/en/latest/
[3] https://requests.readthedocs.io/en/latest/

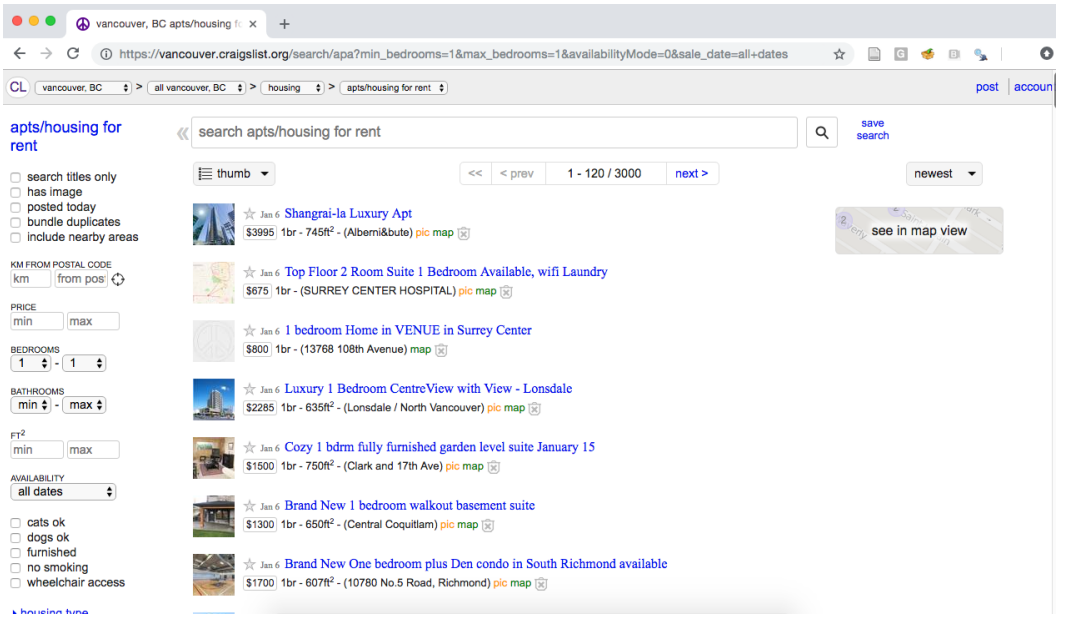


FIGURE 2.2 Craigslist webpage of advertisements for one-bedroom apartments.

2.8.1 Web scraping

2.8.1.1 HTML and CSS selectors

When you enter a URL into your browser, your browser connects to the web server at that URL and asks for the *source code* for the website. This is the data that the browser translates into something you can see; so if we are going to create our own data by scraping a website, we have to first understand what that data looks like. For example, let's say we are interested in knowing the average rental price (per square foot) of the most recently available one-bedroom apartments in Vancouver on Craiglist[4]. When we visit the Vancouver Craigslist website and search for one-bedroom apartments, we should see something similar to Fig. 2.2.

Based on what our browser shows us, it's pretty easy to find the size and price for each apartment listed. But we would like to be able to obtain that information using Python, without any manual human effort or copying and pasting. We do this by examining the *source code* that the web server actually sent our browser to display for us. We show a snippet of it below; the entire source is included with the code for this book[5]:

[4] https://vancouver.craigslist.org

[5] https://github.com/UBC-DSCI/introduction-to-datascience-python/blob/main/source/data/website_source.txt

```
<span class="result-meta">
        <span class="result-price">$800</span>
        <span class="housing">
            1br -
        </span>
        <span class="result-hood"> (13768 108th Avenue)</span>
        <span class="result-tags">
            <span class="maptag" data-pid="6786042973">map</span>
        </span>
        <span class="banish icon icon-trash" role="button">
            <span class="screen-reader-text">hide this posting</span>
        </span>
    <span class="unbanish icon icon-trash red" role="button"></span>
    <a href="#" class="restore-link">
        <span class="restore-narrow-text">restore</span>
        <span class="restore-wide-text">restore this posting</span>
    </a>
    <span class="result-price">$2285</span>
</span>
```

Oof ... you can tell that the source code for a web page is not really designed for humans to understand easily. However, if you look through it closely, you will find that the information we're interested in is hidden among the muck. For example, near the top of the snippet above you can see a line that looks like

```
<span class="result-price">$800</span>
```

That snippet is definitely storing the price of a particular apartment. With some more investigation, you should be able to find things like the date and time of the listing, the address of the listing, and more. So this source code most likely contains all the information we are interested in.

Let's dig into that line above a bit more. You can see that that bit of code has an *opening tag* (words between < and >, like `<span>`) and a *closing tag* (the same with a slash, like `</span>`). HTML source code generally stores its data between opening and closing tags like these. Tags are keywords that tell the web browser how to display or format the content. Above you can see that the information we want (`$800`) is stored between an opening and closing tag (`<span>` and `</span>`). In the opening tag, you can also see a very useful "class" (a special word that is sometimes included with opening tags): `class="result-price"`. Since we want R to programmatically sort through all of the source code for the website to find apartment prices, maybe we can look for all the tags with the `"result-price"` class, and grab the information between the opening and closing tag. Indeed, take a look at another line of the source snippet above:

```
<span class="result-price">$2285</span>
```

It's yet another price for an apartment listing, and the tags surrounding it have the `"result-price"` class. Wonderful! Now that we know what pattern we are looking for—a dollar amount between opening and closing tags that have the `"result-price"` class—we should be able to use code to pull out all of the matching patterns from the source code to obtain our data. This sort of "pattern" is known as a *CSS selector* (where CSS stands for **c**ascading **s**tyle **s**heet).

The above was a simple example of "finding the pattern to look for"; many websites are quite a bit larger and more complex, and so is their website source code. Fortunately, there are tools available to make this process easier. For example, SelectorGadget[6] is an open-source tool that simplifies identifying the generating and finding of CSS selectors. At the end of the chapter in the additional resources section, we include a link to a short video on how to install and use the SelectorGadget tool to obtain CSS selectors for use in web scraping. After installing and enabling the tool, you can click the website element for which you want an appropriate selector. For example, if we click the price of an apartment listing, we find that SelectorGadget shows us the selector `.result-price` in its toolbar, and highlights all the other apartment prices that would be obtained using that selector (Fig. 2.3).

If we then click the size of an apartment listing, SelectorGadget shows us the `span` selector, and highlights many of the lines on the page; this indicates that the `span` selector is not specific enough to capture only apartment sizes (Fig. 2.4).

To narrow the selector, we can click one of the highlighted elements that we *do not* want. For example, we can deselect the "pic/map" links, resulting in only the data we want highlighted using the `.housing` selector (Fig. 2.5).

So to scrape information about the square footage and rental price of apartment listings, we need to use the two CSS selectors `.housing` and `.result-price`, respectively. The selector gadget returns them to us as a comma-separated list (here `.housing , .result-price`), which is exactly the format we need to provide to Python if we are using more than one CSS selector.

Caution: are you allowed to scrape that website?

Before scraping data from the web, you should always check whether or not you are *allowed* to scrape it. There are two documents that are important

[6] https://selectorgadget.com/

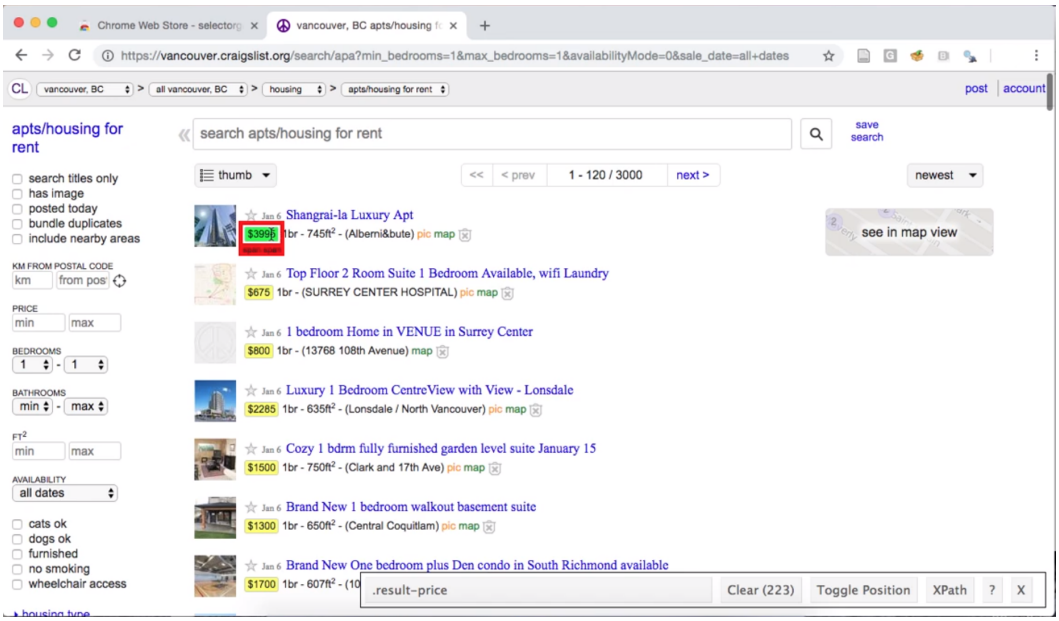


FIGURE 2.3 Using the SelectorGadget on a Craigslist webpage to obtain the CCS selector useful for obtaining apartment prices.

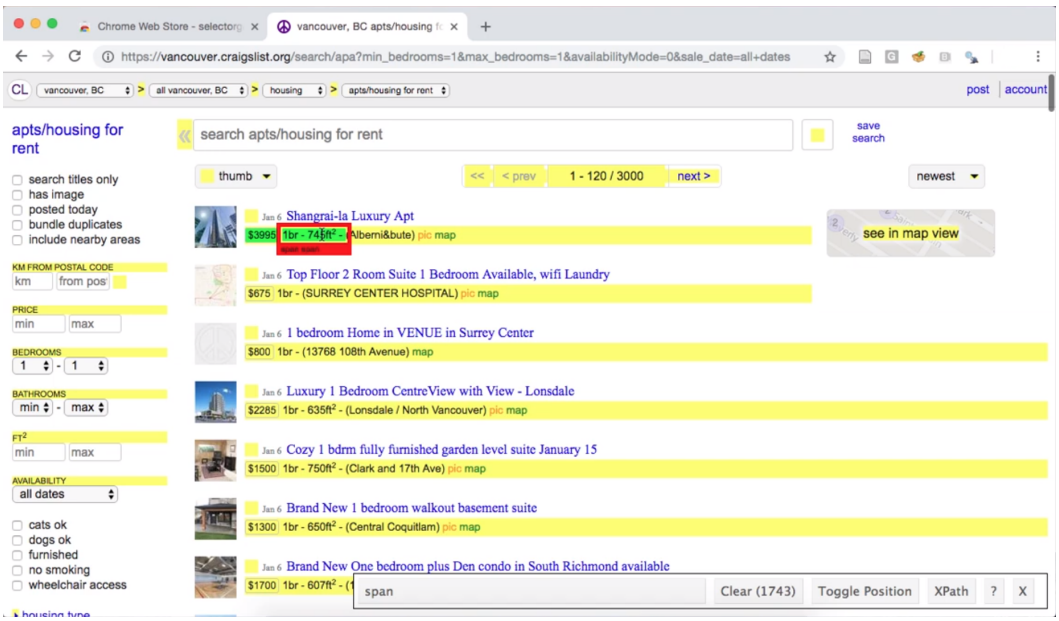


FIGURE 2.4 Using the SelectorGadget on a Craigslist webpage to obtain a CCS selector useful for obtaining apartment sizes.

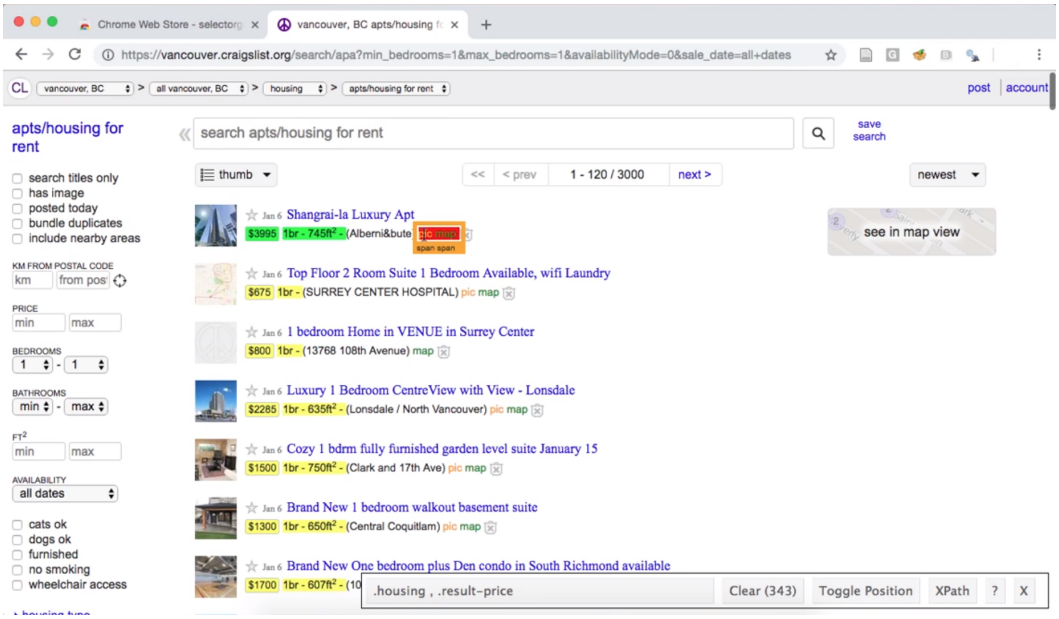


FIGURE 2.5 Using the SelectorGadget on a Craigslist webpage to refine the CCS selector to one that is most useful for obtaining apartment sizes.

for this: the `robots.txt` file and the Terms of Service document. If we take a look at Craigslist's Terms of Service document[7], we find the following text: *"You agree not to copy/collect CL content via robots, spiders, scripts, scrapers, crawlers, or any automated or manual equivalent (e.g., by hand)"*. So unfortunately, without explicit permission, we are not allowed to scrape the website.

What to do now? Well, we *could* ask the owner of Craigslist for permission to scrape. However, we are not likely to get a response, and even if we did they would not likely give us permission. The more realistic answer is that we simply cannot scrape Craigslist. If we still want to find data about rental prices in Vancouver, we must go elsewhere. To continue learning how to scrape data from the web, let's instead scrape data on the population of Canadian cities from Wikipedia. We have checked the Terms of Service document[8], and it does not mention that web scraping is disallowed. We will use the SelectorGadget tool to pick elements that we are interested in (city names and population counts) and deselect others to indicate that we are not interested in them (province names), as shown in Fig. 2.6.

[7] https://www.craigslist.org/about/terms.of.use
[8] https://foundation.wikimedia.org/wiki/Terms_of_Use/en

FIGURE 2.6 Using the SelectorGadget on a Wikipedia webpage.

We include a link to a short video tutorial on this process at the end of the chapter in the additional resources section. SelectorGadget provides in its toolbar the following list of CSS selectors to use:

```
td:nth-child(8) ,
td:nth-child(4) ,
.largestCities-cell-background+ td a
```

Now that we have the CSS selectors that describe the properties of the elements that we want to target, we can use them to find certain elements in web pages and extract data.

2.8.1.2 Scraping with `BeautifulSoup`

We will use the `requests` and `BeautifulSoup` Python packages to scrape data from the Wikipedia page. After loading those packages, we tell Python which page we want to scrape by providing its URL in quotations to the `requests.get` function. This function obtains the raw HTML of the page, which we then pass to the `BeautifulSoup` function for parsing:

```
import requests
import bs4

wiki = requests.get("https://en.wikipedia.org/wiki/Canada")
page = bs4.BeautifulSoup(wiki.content, "html.parser")
```

The `requests.get` function downloads the HTML source code for the page at the URL you specify, just like your browser would if you navigated to this

site. But instead of displaying the website to you, the `requests.get` function just returns the HTML source code itself—stored in the `wiki.content` variable—which we then parse using `BeautifulSoup` and store in the `page` variable. Next, we pass the CSS selectors we obtained from SelectorGadget to the `select` method of the `page` object. Make sure to surround the selectors with quotation marks; `select` expects that argument is a string. We store the result of the `select` function in the `population_nodes` variable. Note that `select` returns a list; below we slice the list to print only the first 5 elements for clarity.

```
population_nodes = page.select(
    "td:nth-child(8) , td:nth-child(4) , .largestCities-cell-background+ td a"
)
population_nodes[:5]
```

```
[<a href="/wiki/Greater_Toronto_Area" title="Greater Toronto Area">Toronto</a>
 ↪,
 <td style="text-align:right;">6,202,225</td>,
 <a href="/wiki/London,_Ontario" title="London, Ontario">London</a>,
 <td style="text-align:right;">543,551
 </td>,
 <a href="/wiki/Greater_Montreal" title="Greater Montreal">Montreal</a>]
```

Each of the items in the `population_nodes` list is a *node* from the HTML document that matches the CSS selectors you specified. A *node* is an HTML tag pair (e.g., `<td>` and `</td>` which defines the cell of a table) combined with the content stored between the tags. For our CSS selector `td:nth-child(4)`, an example node that would be selected would be:

```
<td style="text-align:left;">
<a href="/wiki/London,_Ontario" title="London, Ontario">London</a>
</td>
```

Next, we extract the meaningful data—in other words, we get rid of the HTML code syntax and tags—from the nodes using the `get_text` function. In the case of the example node above, `get_text` function returns `"London"`. Once again we show only the first 5 elements for clarity.

```
[row.get_text() for row in population_nodes[:5]]
```

```
['Toronto', '6,202,225', 'London', '543,551\n', 'Montreal']
```

Fantastic! We seem to have extracted the data of interest from the raw HTML source code. But we are not quite done; the data is not yet in an optimal format for data analysis. Both the city names and population are encoded as characters in a single vector, instead of being in a data frame with one character column for city and one numeric column for population (like a spreadsheet). Additionally, the populations contain commas (not useful for

programmatically dealing with numbers), and some even contain a line break character at the end (`\n`). In Chapter 3, we will learn more about how to *wrangle* data such as this into a more useful format for data analysis using Python.

2.8.1.3 Scraping with `read_html`

Using `requests` and `BeautifulSoup` to extract data based on CSS selectors is a very general way to scrape data from the web, albeit perhaps a little bit complicated. Fortunately, `pandas` provides the `read_html`[9] function, which is easier method to try when the data appear on the webpage already in a tabular format. The `read_html` function takes one argument—the URL of the page to scrape—and will return a list of data frames corresponding to all the tables it finds at that URL. We can see below that `read_html` found 17 tables on the Wikipedia page for Canada.

```
canada_wiki_tables = pd.read_html("https://en.wikipedia.org/wiki/Canada")
len(canada_wiki_tables)
```

```
17
```

After manually searching through these, we find that the table containing the population counts of the largest metropolitan areas in Canada is contained in index 1. We use the `droplevel` method to simplify the column names in the resulting data frame:

```
canada_wiki_df = canada_wiki_tables[1]
canada_wiki_df.columns = canada_wiki_df.columns.droplevel()
canada_wiki_df
```

```
   Rank                          Name          Province     Pop.  Rank.1  \
0     1                       Toronto           Ontario  6202225      11
1     2                      Montreal            Quebec  4291732      12
2     3                     Vancouver  British Columbia  2642825      13
3     4               Ottawa-Gatineau    Ontario-Quebec  1488307      14
4     5                       Calgary           Alberta  1481806      15
5     6                      Edmonton           Alberta  1418118      16
6     7                   Quebec City            Quebec   839311      17
7     8                      Winnipeg          Manitoba   834678      18
8     9                      Hamilton           Ontario   785184      19
9    10  Kitchener-Cambridge-Waterloo           Ontario   575847      20

                   Name.1        Province.1  Pop..1  Unnamed: 8_level_1  \
0                  London           Ontario  543551                 NaN
1                 Halifax       Nova Scotia  465703                 NaN
2  St. Catharines-Niagara           Ontario  433604                 NaN
3                 Windsor           Ontario  422630                 NaN
4                  Oshawa           Ontario  415311                 NaN
5                Victoria  British Columbia  397237                 NaN
```

(continues on next page)

[9] https://pandas.pydata.org/docs/reference/api/pandas.read_html.html

(continued from previous page)

```
6                Saskatoon      Saskatchewan  317480                NaN
7                   Regina      Saskatchewan  249217                NaN
8               Sherbrooke            Quebec  227398                NaN
9                  Kelowna  British Columbia  222162                NaN

   Unnamed: 9_level_1
0                 NaN
1                 NaN
2                 NaN
3                 NaN
4                 NaN
5                 NaN
6                 NaN
7                 NaN
8                 NaN
9                 NaN
```

Once again, we have managed to extract the data of interest from the raw HTML source code—but this time using the convenient `read_html` function, without needing to explicitly use CSS selectors. However, once again, we still need to do some cleaning of this result. Referring back to Fig. 2.6, we can see that the table is formatted with two sets of columns (e.g., `Name` and `Name.1`) that we will need to somehow merge. In Chapter 3, we will learn more about how to *wrangle* data into a useful format for data analysis.

2.8.2 Using an API

Rather than posting a data file at a URL for you to download, many websites these days provide an API that can be accessed through a programming language like Python. The benefit of using an API is that data owners have much more control over the data they provide to users. However, unlike web scraping, there is no consistent way to access an API across websites. Every website typically has its own API designed especially for its own use case. Therefore, we will just provide one example of accessing data through an API in this book, with the hope that it gives you enough of a basic idea that you can learn how to use another API if needed. In particular, in this book we will show you the basics of how to use the `requests` package in Python to access data from the NASA "Astronomy Picture of the Day" API (a great source of desktop backgrounds, by the way—take a look at the stunning picture of the Rho-Ophiuchi cloud complex [NASA *et al.*, Accessed Online: 2023] in Fig. 2.7 from July 13, 2023!).

First, you will need to visit the NASA APIs page[10] and generate an API key (i.e., a password used to identify you when accessing the API). Note that a valid email address is required to associate with the key. The signup form

[10] https://api.nasa.gov/

FIGURE 2.7 The James Webb Space Telescope's NIRCam image of the Rho Ophiuchi molecular cloud complex.

looks something like Fig. 2.8. After filling out the basic information, you will receive the token via email. Make sure to store the key in a safe place, and keep it private.

Caution: think about your API usage carefully.

When you access an API, you are initiating a transfer of data from a web server to your computer. Web servers are expensive to run and do not have infinite resources. If you try to ask for *too much data* at once, you can use up a huge amount of the server's bandwidth. If you try to ask for data *too*

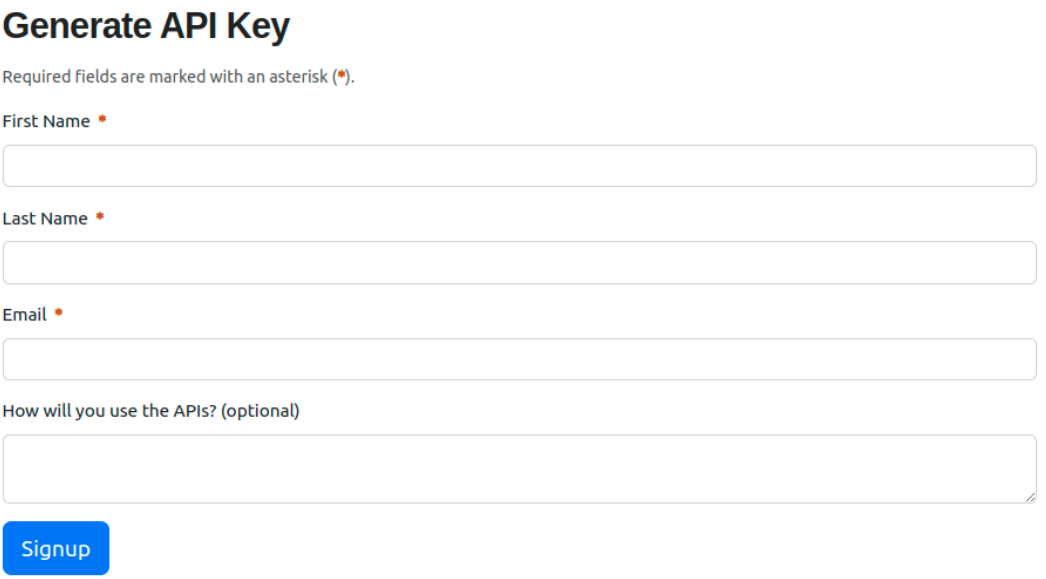


FIGURE 2.8 Generating the API access token for the NASA API.

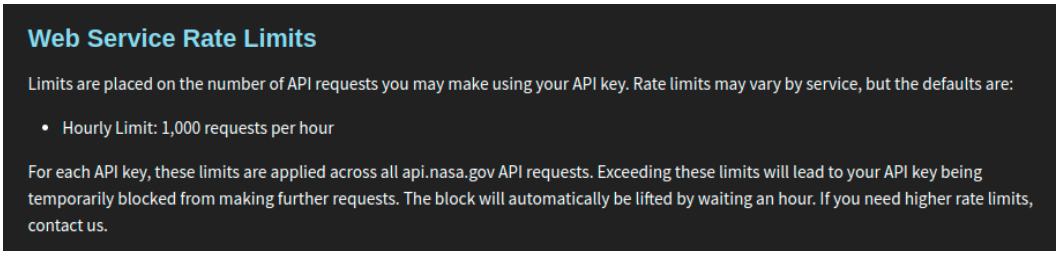


FIGURE 2.9 The NASA website specifies an hourly limit of 1,000 requests.

frequently—e.g., if you make many requests to the server in quick succession—you can also bog the server down and make it unable to talk to anyone else. Most servers have mechanisms to revoke your access if you are not careful, but you should try to prevent issues from happening in the first place by being extra careful with how you write and run your code. You should also keep in mind that when a website owner grants you API access, they also usually specify a limit (or *quota*) of how much data you can ask for. Be careful not to overrun your quota! So *before* we try to use the API, we will first visit the NASA website[11] to see what limits we should abide by when using the API. These limits are outlined in Fig. 2.9.

After checking the NASA website, it seems like we can send at most 1,000 requests per hour. That should be more than enough for our purposes in this section.

2.8.2.1 Accessing the NASA API

The NASA API is what is known as an *HTTP API*: this is a particularly common kind of API, where you can obtain data simply by accessing a particular URL as if it were a regular website. To make a query to the NASA API, we need to specify three things. First, we specify the URL *endpoint* of the API, which is simply a URL that helps the remote server understand which API you are trying to access. NASA offers a variety of APIs, each with its own endpoint; in the case of the NASA "Astronomy Picture of the Day" API, the URL endpoint is `https://api.nasa.gov/planetary/apod`. Second, we write `?`, which denotes that a list of *query parameters* will follow. And finally, we specify a list of query parameters of the form `parameter=value`, separated by `&` characters. The NASA "Astronomy Picture of the Day" API accepts the parameters shown in Fig. 2.10.

So, for example, to obtain the image of the day from July 13, 2023, the API query would have two parameters: `api_key=YOUR_API_KEY` and `date=2023-07-13`. Remember to replace `YOUR_API_KEY` with the API key

[11] https://api.nasa.gov/

Parameter	Type	Default	Description
date	YYYY-MM-DD	*today*	The date of the APOD image to retrieve
start_date	YYYY-MM-DD	none	The start of a date range, when requesting date for a range of dates. Cannot be used with `date`.
end_date	YYYY-MM-DD	*today*	The end of the date range, when used with `start_date`.
count	int	none	If this is specified then `count` randomly chosen images will be returned. Cannot be used with `date` or `start_date` and `end_date`.
thumbs	bool	False	Return the URL of video thumbnail. If an APOD is not a video, this parameter is ignored.
api_key	string	DEMO_KEY	api.nasa.gov key for expanded usage

FIGURE 2.10 The set of parameters that you can specify when querying the NASA "Astronomy Picture of the Day" API, along with syntax, default settings, and a description of each.

you received from NASA in your email. Putting it all together, the query will look like the following:

```
https://api.nasa.gov/planetary/apod?api_key=YOUR_API_KEY&date=2023-07-13
```

If you try putting this URL into your web browser, you'll actually find that the server responds to your request with some text:

```
{"date":"2023-07-13","explanation":"A mere 390 light-years away, Sun-like stars
and future planetary systems are forming in the Rho Ophiuchi molecular cloud
complex, the closest star-forming region to our fair planet. The James Webb
Space Telescope's NIRCam peered into the nearby natal chaos to capture this
infrared image at an inspiring scale. The spectacular cosmic snapshot was
released to celebrate the successful first year of Webb's exploration of the
Universe. The frame spans less than a light-year across the Rho Ophiuchi region
and contains about 50 young stars. Brighter stars clearly sport Webb's
characteristic pattern of diffraction spikes. Huge jets of shocked molecular
hydrogen blasting from newborn stars are red in the image, with the large,
yellowish dusty cavity carved out by the energetic young star near its center.
Near some stars in the stunning image are shadows cast by their protoplanetary
disks.","hdurl":"https://apod.nasa.gov/apod/image/2307/STScI-01_RhoOph.png",
"media_type":"image","service_version":"v1","title":"Webb's
Rho Ophiuchi","url":"https://apod.nasa.gov/apod/image/2307/STScI-01_RhoOph1024.png
↪"}
```

Neat! There is definitely some data there, but it's a bit hard to see what it all is. As it turns out, this is a common format for data called *JSON* (JavaScript Object Notation). We won't encounter this kind of data much in this book, but for now you can interpret this data just like you'd interpret a Python dictionary: these are `key : value` pairs separated by commas. For example,

if you look closely, you'll see that the first entry is `"date":"2023-07-13"`, which indicates that we indeed successfully received data corresponding to July 13, 2023.

So now our job is to do all of this programmatically in Python. We will load the `requests` package, and make the query using the `get` function, which takes a single URL argument; you will recognize the same query URL that we pasted into the browser earlier. We will then obtain a JSON representation of the response using the `json` method.

```
import requests

nasa_data_single = requests.get(
    "https://api.nasa.gov/planetary/apod?api_key=YOUR_API_KEY&date=2023-07-13"
).json()
nasa_data_single
```

```
{'date': '2023-07-13',
 'explanation': "A mere 390 light-years away, Sun-like stars and future␣
↪planetary systems are forming in the Rho Ophiuchi molecular cloud complex,␣
↪the closest star-forming region to our fair planet. The James Webb Space␣
↪Telescope's NIRCam peered into the nearby natal chaos to capture this␣
↪infrared image at an inspiring scale. The spectacular cosmic snapshot was␣
↪released to celebrate the successful first year of Webb's exploration of␣
↪the Universe. The frame spans less than a light-year across the Rho␣
↪Ophiuchi region and contains about 50 young stars. Brighter stars clearly␣
↪sport Webb's characteristic pattern of diffraction spikes. Huge jets of␣
↪shocked molecular hydrogen blasting from newborn stars are red in the image,
↪ with the large, yellowish dusty cavity carved out by the energetic young␣
↪star near its center. Near some stars in the stunning image are shadows␣
↪cast by their protoplanetary disks.",
 'hdurl': 'https://apod.nasa.gov/apod/image/2307/STScI-01_RhoOph.png',
 'media_type': 'image',
 'service_version': 'v1',
 'title': "Webb's Rho Ophiuchi",
 'url': 'https://apod.nasa.gov/apod/image/2307/STScI-01_RhoOph1024.png'}
```

We can obtain more records at once by using the `start_date` and `end_date` parameters, as shown in the table of parameters in Fig. 2.10. Let's obtain all the records between May 1, 2023, and July 13, 2023, and store the result in an object called `nasa_data`; now the response will take the form of a Python list. Each item in the list will correspond to a single day's record (just like the `nasa_data_single` object), and there will be 74 items total, one for each day between the start and end dates:

```
nasa_data = requests.get(
    "https://api.nasa.gov/planetary/apod?api_key=YOUR_API_KEY&start_date=2023-05-
↪01&end_date=2023-07-13"
    ).json()
len(nasa_data)
```

```
74
```

For further data processing using the techniques in this book, you'll need to turn this list of dictionaries into a `pandas` data frame. Here we will extract the `date`, `title`, `copyright`, and `url` variables from the JSON data, and construct a `pandas` DataFrame using the extracted information.

Note: Understanding this code is not required for the remainder of the textbook. It is included for those readers who would like to parse JSON data into a `pandas` data frame in their own data analyses.

```
data_dict = {
    "date":[],
    "title": [],
    "copyright" : [],
    "url": []
}

for item in nasa_data:
    if "copyright" not in item:
        item["copyright"] = None
    for entry in ["url", "title", "date", "copyright"]:
        data_dict[entry].append(item[entry])

nasa_df = pd.DataFrame(data_dict)
nasa_df
```

```
          date                                              title  \
0   2023-05-01                              Carina Nebula North
1   2023-05-02                            Flat Rock Hills on Mars
2   2023-05-03       Centaurus A: A Peculiar Island of Stars
3   2023-05-04  The Galaxy, the Jet, and a Famous Black Hole
4   2023-05-05                         Shackleton from ShadowCam
..         ...                                               ...
69  2023-07-09                           Doomed Star Eta Carinae
70  2023-07-10             Stars, Dust and Nebula in NGC 6559
71  2023-07-11                          Sunspots on an Active Sun
72  2023-07-12       Rings and Bar of Spiral Galaxy NGC 1398
73  2023-07-13                              Webb's Rho Ophiuchi

                                            copyright  \
0                                    \nCarlos Taylor\n
1   \nNASA, \nJPL-Caltech, \nMSSS;\nProcessing: Ne...
2   \nMarco Lorenzi,\nAngus Lau & Tommy Tse; \nTex...
3                                                None
4                                                None
..                                                ...
69  \nNASA, \nESA, \nHubble;\n Processing & \nLice...
70                    \nAdam Block,\nTelescope Live\n
71                                               None
72                                               None
73                                               None

                                                  url
0   https://apod.nasa.gov/apod/image/2305/CarNorth...
1   https://apod.nasa.gov/apod/image/2305/FlatMars...
2   https://apod.nasa.gov/apod/image/2305/NGC5128_...
```

(continues on next page)

(continued from previous page)

```
3   https://apod.nasa.gov/apod/image/2305/pia23122...
4   https://apod.nasa.gov/apod/image/2305/shacklet...
..                                                ...
69  https://apod.nasa.gov/apod/image/2307/EtaCarin...
70  https://apod.nasa.gov/apod/image/2307/NGC6559_...
71  https://apod.nasa.gov/apod/image/2307/SpottedS...
72  https://apod.nasa.gov/apod/image/2307/Ngc1398_...
73  https://apod.nasa.gov/apod/image/2307/STScI-01...

[74 rows x 4 columns]
```

Success—we have created a small data set using the NASA API. This data is also quite different from what we obtained from web scraping; the extracted information is readily available in a JSON format, as opposed to raw HTML code (although not *every* API will provide data in such a nice format). From this point onward, the `nasa_df` data frame is stored on your machine, and you can play with it to your heart's content. For example, you can use `pandas.to_csv` to save it to a file and `pandas.read_csv` to read it into Python again later; and after reading the next few chapters you will have the skills to do even more interesting things. If you decide that you want to ask any of the various NASA APIs for more data (see the list of awesome NASA APIS here[12] for more examples of what is possible), just be mindful as usual about how much data you are requesting and how frequently you are making requests.

2.9 Exercises

Practice exercises for the material covered in this chapter can be found in the accompanying worksheets repository[13] in the "Reading in data locally and from the web" row. You can launch an interactive version of the worksheet in your browser by clicking the "launch binder" button. You can also preview a non-interactive version of the worksheet by clicking "view worksheet". If you instead decide to download the worksheet and run it on your own machine, make sure to follow the instructions for computer setup found in Chapter 13. This will ensure that the automated feedback and guidance that the worksheets provide will function as intended.

[12] https://api.nasa.gov/

[13] https://worksheets.python.datasciencebook.ca

2.10 Additional resources

- The `pandas` documentation[14] provides the documentation for the functions we cover in this chapter. It is where you should look if you want to learn more about these functions, the full set of arguments you can use, and other related functions.
- Sometimes you might run into data in such poor shape that the reading functions we cover in this chapter do not work. In that case, you can consult the data loading chapter[15] from *Python for Data Analysis*[16] [McKinney, 2012], which goes into a lot more detail about how Python parses text from files into data frames.
- A video[17] from the Udacity course *Linux Command Line Basics* provides a good explanation of absolute versus relative paths.
- If you read the subsection on obtaining data from the web via scraping and APIs, we provide two companion tutorial video links for how to use the SelectorGadget tool to obtain desired CSS selectors for:
  - extracting the data for apartment listings on Craigslist[18], and
  - extracting Canadian city names and populations from Wikipedia[19].

[14] https://pandas.pydata.org/docs/getting_started/index.html
[15] https://wesmckinney.com/book/accessing-data.html#io_flat_files
[16] https://wesmckinney.com/book/
[17] https://www.youtube.com/embed/ephId3mYu9o
[18] https://www.youtube.com/embed/YdIWI6K64zo
[19] https://www.youtube.com/embed/O9HKbdhqYzk

3

Cleaning and wrangling data

3.1 Overview

This chapter is centered around defining tidy data—a data format that is suitable for analysis—and the tools needed to transform raw data into this format. This will be presented in the context of a real-world data science application, providing more practice working through a whole case study.

3.2 Chapter learning objectives

By the end of the chapter, readers will be able to do the following:

- Define the term "tidy data".
- Discuss the advantages of storing data in a tidy data format.
- Define what series and data frames are in Python, and describe how they relate to each other.
- Describe the common types of data in Python and their uses.
- Use the following functions for their intended data wrangling tasks:
  - `melt`
  - `pivot`
  - `reset_index`
  - `str.split`
  - `agg`
  - `assign` and regular column assignment
  - `groupby`
  - `merge`

DOI: 10.1201/9781003438397-3

- Use the following operators for their intended data wrangling tasks:
  - `==`, `!=`, `<`, `>`, `<=`, and `>=`
  - `isin`
  - `&` and `|`
  - `[]`, `loc[]`, and `iloc[]`

3.3 Data frames and series

In Chapters 1 and 2, *data frames* were the focus: we learned how to import data into Python as a data frame, and perform basic operations on data frames in Python. In the remainder of this book, this pattern continues. The vast majority of tools we use will require that data are represented as a `pandas` **data frame** in Python. Therefore, in this section, we will dig more deeply into what data frames are and how they are represented in Python. This knowledge will be helpful in effectively utilizing these objects in our data analyses.

3.3.1 What is a data frame?

A data frame is a table-like structure for storing data in Python. Data frames are important to learn about because most data that you will encounter in practice can be naturally stored as a table. In order to define data frames precisely, we need to introduce a few technical terms:

- **variable:** a characteristic, number, or quantity that can be measured.
- **observation:** all of the measurements for a given entity.
- **value:** a single measurement of a single variable for a given entity.

Given these definitions, a **data frame** is a tabular data structure in Python that is designed to store observations, variables, and their values. Most commonly, each column in a data frame corresponds to a variable, and each row corresponds to an observation. For example, Fig. 3.1 displays a data set of city populations. Here, the variables are "region, year, population"; each of these are properties that can be collected or measured. The first observation is "Toronto, 2016, 2235145"; these are the values that the three variables take for the first entity in the data set. There are 13 entities in the data set in total, corresponding to the 13 rows in Fig. 3.1.

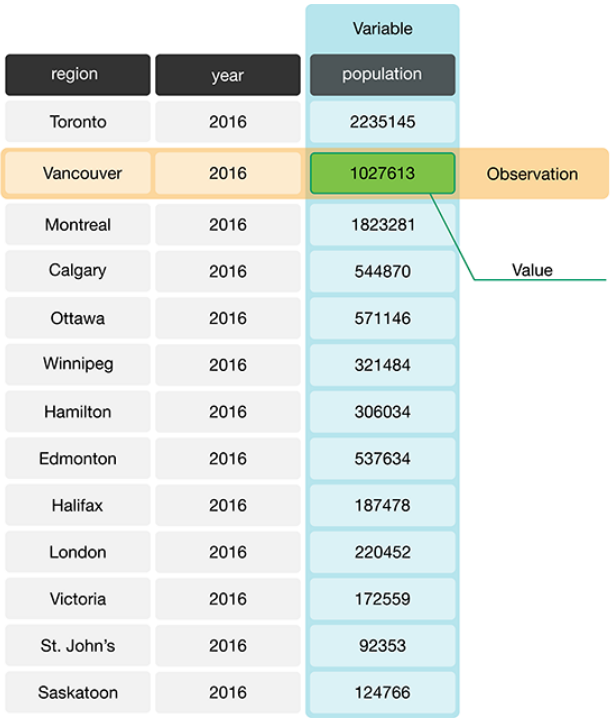


FIGURE 3.1 A data frame storing data regarding the population of various regions in Canada. In this example data frame, the row that corresponds to the observation for the city of Vancouver is colored yellow, and the column that corresponds to the population variable is colored blue.

3.3.2 What is a series?

In Python, `pandas` **series** are objects that can contain one or more elements (like a list). They are a single column, are ordered, can be indexed, and can contain any data type. The `pandas` package uses `Series` objects to represent the columns in a data frame. `Series` can contain a mix of data types, but it is good practice to only include a single type in a series because all observations of one variable should be the same type. Python has several different basic data types, as shown in Table 3.1. You can create a `pandas` series using the `pd.Series()` function. For example, to create the series `region` as shown in Fig. 3.2, you can write the following.

```
import pandas as pd

region = pd.Series(["Toronto", "Montreal", "Vancouver", "Calgary", "Ottawa"])
region
```

```
0       Toronto
1     Montreal
```

(continues on next page)

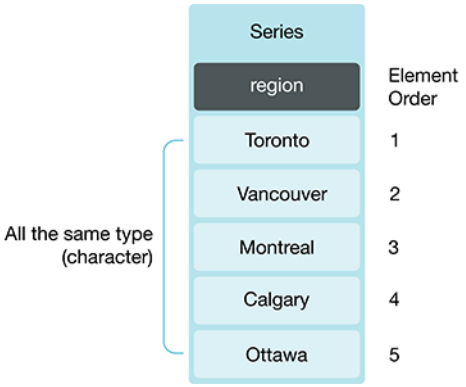


FIGURE 3.2 Example of a `pandas` series whose type is string.

(continued from previous page)

```
2    Vancouver
3      Calgary
4       Ottawa
dtype: object
```

It is important in Python to make sure you represent your data with the correct type. Many of the `pandas` functions we use in this book treat the various data types differently. You should use `int` and `float` types to represent numbers and perform arithmetic. The `int` type is for integers that have no decimal point, while the `float` type is for numbers that have a decimal point. The `bool` type are boolean variables that can only take on one of two values: `True` or `False`. The `string` type is used to represent data that should be thought of as "text", such as words, names, paths, URLs, and more. A `NoneType` is a special type in Python that is used to indicate no value; this can occur, for example, when you have missing data. There are other basic data types in Python, but we will generally not use these in this textbook.

TABLE 3.1 Basic data types in Python

Data type	Abbreviation	Description	Example
integer	`int`	positive/negative/zero whole numbers	`42`
floating point number	`float`	real number in decimal form	`3.14159`
boolean	`bool`	true or false	`True`
string	`str`	text	`"Hello World"`
none	`NoneType`	represents no value	`None`

Series of type character	Series of type integer	Series of type logical
region	year	voted
Toronto	2016	TRUE
Vancouver	2016	TRUE
Montreal	2016	TRUE
Calgary	2016	TRUE
Ottawa	2016	TRUE
Winnipeg	2016	TRUE
Hamilton	2016	TRUE
Edmonton	2016	TRUE
Halifax	2016	TRUE
London	2016	TRUE
Victoria	2016	TRUE
St. John's	2016	TRUE
Saskatoon	2016	TRUE

FIGURE 3.3 Data frame and series types.

3.3.3 What does this have to do with data frames?

A data frame is really just a collection of series that are stuck together, where each series corresponds to one column and all must have the same length. But not all columns in a data frame need to be of the same type. Fig. 3.3 shows a data frame where the columns are series of different types. But each element *within* one column should usually be the same type, since the values for a single variable are usually all of the same type. For example, if the variable is the name of a city, that name should be a string, whereas if the variable is a year, that should be an integer. So even though series let you put different types in them, it is most common (and good practice!) to have just one type per column.

Note: You can use the function `type` on a data object. For example we can check the class of the Canadian languages data set, `can_lang`, we worked with in the previous chapters and we see it is a `pandas.core.frame.DataFrame`.

```
can_lang = pd.read_csv("data/can_lang.csv")
type(can_lang)
```

```
pandas.core.frame.DataFrame
```

3.3.4 Data structures in Python

The `Series` and `DataFrame` types are *data structures* in Python, which are core to most data analyses. The functions from `pandas` that we use often give us back a `DataFrame` or a `Series` depending on the operation. Because `Series` are essentially simple `DataFrames`, we will refer to both `DataFrames` and `Series` as "data frames" in the text. There are other types that represent data structures in Python. We summarize the most common ones in Table 3.2.

TABLE 3.2 Basic data structures in Python

Data Structure	Description
list	An ordered collection of values that can store multiple data types at once.
dict	A labeled data structure where `keys` are paired with `values`.
Series	An ordered collection of values *with labels* that can store multiple data types at once.
DataFrame	A labeled data structure with `Series` columns of potentially different types.

A `list` is an ordered collection of values. To create a list, we put the contents of the list in between square brackets `[]`, where each item of the list is separated by a comma. A `list` can contain values of different types. The example below contains six `str` entries.

```
cities = ["Toronto", "Vancouver", "Montreal", "Calgary", "Ottawa", "Winnipeg"]
cities
```

```
['Toronto', 'Vancouver', 'Montreal', 'Calgary', 'Ottawa', 'Winnipeg']
```

A list can directly be converted to a pandas `Series`.

```
cities_series = pd.Series(cities)
cities_series
```

```
0       Toronto
1     Vancouver
2      Montreal
3       Calgary
4        Ottawa
5     Winnipeg
dtype: object
```

A `dict`, or dictionary, contains pairs of "keys" and "values". You use a key to look up its corresponding value. Dictionaries are created using curly brackets

`{}`. Each entry starts with the key on the left, followed by a colon symbol `:`, and then the value. A dictionary can have multiple key-value pairs, each separated by a comma. Keys can take a wide variety of types (`int` and `str` are commonly used), and values can take any type; the key-value pairs in a dictionary can all be of different types, too. In the example below, we create a dictionary that has two keys: `"cities"` and `"population"`. The values associated with each are lists.

```
population_in_2016 = {
  "cities": ["Toronto", "Vancouver", "Montreal", "Calgary", "Ottawa", "Winnipeg
↪"],
  "population": [2235145, 1027613, 1823281, 544870, 571146, 321484]
}
population_in_2016
```

```
{'cities': ['Toronto',
  'Vancouver',
  'Montreal',
  'Calgary',
  'Ottawa',
  'Winnipeg'],
 'population': [2235145, 1027613, 1823281, 544870, 571146, 321484]}
```

A dictionary can be converted to a data frame. Keys become the column names, and the values become the entries in those columns. Dictionaries on their own are quite simple objects; it is preferable to work with a data frame because then we have access to the built-in functionality in `pandas` (e.g. `loc[]`, `[]`, and many functions that we will discuss in the upcoming sections).

```
population_in_2016_df = pd.DataFrame(population_in_2016)
population_in_2016_df
```

```
      cities  population
0    Toronto     2235145
1  Vancouver     1027613
2   Montreal     1823281
3    Calgary      544870
4     Ottawa      571146
5   Winnipeg      321484
```

Of course, there is no need to name the dictionary separately before passing it to `pd.DataFrame`; we can instead construct the dictionary right inside the call. This is often the most convenient way to create a new data frame.

```
population_in_2016_df = pd.DataFrame({
  "cities": ["Toronto", "Vancouver", "Montreal", "Calgary", "Ottawa", "Winnipeg
↪"],
  "population": [2235145, 1027613, 1823281, 544870, 571146, 321484]
})
population_in_2016_df
```

```
      cities  population
0    Toronto     2235145
1  Vancouver     1027613
2   Montreal     1823281
3    Calgary      544870
4     Ottawa      571146
5   Winnipeg      321484
```

3.4 Tidy data

There are many ways a tabular data set can be organized. The data frames we have looked at so far have all been using the **tidy data** format of organization. This chapter will focus on introducing the tidy data format and how to make your raw (and likely messy) data tidy. A tidy data frame satisfies the following three criteria [Wickham, 2014]:

- each row is a single observation,
- each column is a single variable, and
- each value is a single cell (i.e., its entry in the data frame is not shared with another value).

Fig. 3.4 demonstrates a tidy data set that satisfies these three criteria.

There are many good reasons for making sure your data are tidy as a first step in your analysis. The most important is that it is a single, consistent format that nearly every function in the `pandas` recognizes. No matter what the variables and observations in your data represent, as long as the data frame is tidy, you can manipulate it, plot it, and analyze it using the same tools. If your data is *not* tidy, you will have to write special bespoke code in your analysis that will not only be error-prone, but hard for others to understand. Beyond making your analysis more accessible to others and less error-prone, tidy data is also typically easy for humans to interpret. Given these benefits, it is well worth spending the time to get your data into a tidy format upfront. Fortunately, there are many well-designed `pandas` data cleaning/wrangling tools to help you easily tidy your data. Let's explore them below.

Note: Is there only one shape for tidy data for a given data set? Not necessarily! It depends on the statistical question you are asking and what the variables are for that question. For tidy data, each variable should be its own column. So, just as it's essential to match your statistical question with the appropriate data analysis tool, it's important to match your statistical

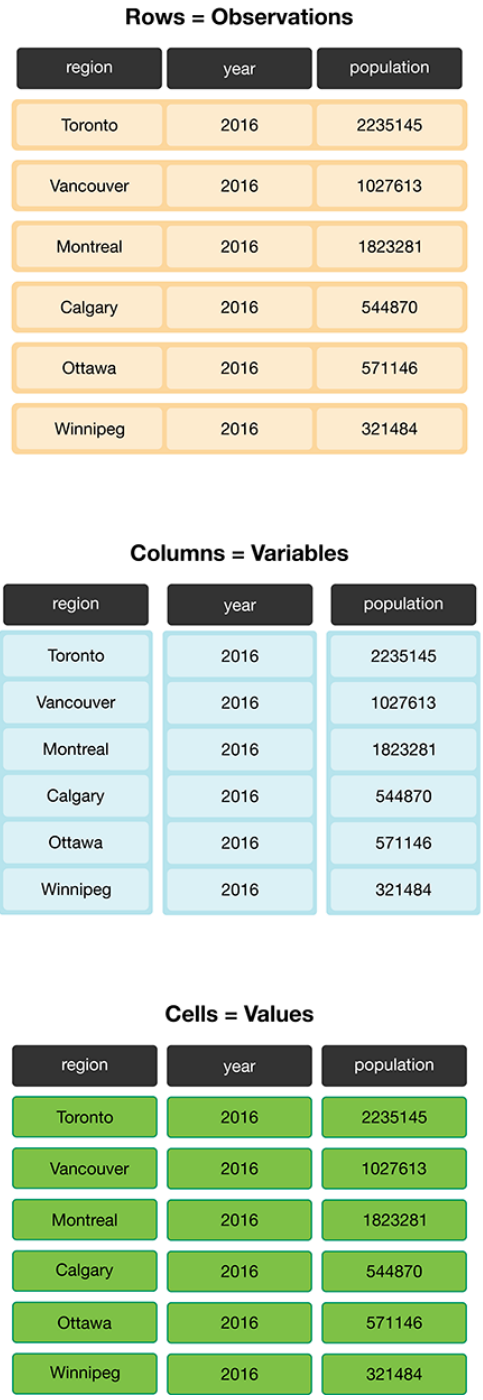


FIGURE 3.4 Tidy data satisfies three criteria.

question with the appropriate variables and ensure they are represented as individual columns to make the data tidy.

3.4.1 Tidying up: going from wide to long using `melt`

One task that is commonly performed to get data into a tidy format is to combine values that are stored in separate columns, but are really part of the same variable, into one. Data is often stored this way because this format is sometimes more intuitive for human readability and understanding, and humans create data sets. In Fig. 3.5, the table on the left is in an untidy, "wide" format because the year values (2006, 2011, 2016) are stored as column names. And as a consequence, the values for population for the various cities over these years are also split across several columns.

For humans, this table is easy to read, which is why you will often find data stored in this wide format. However, this format is difficult to work with when performing data visualization or statistical analysis using Python. For example, if we wanted to find the latest year it would be challenging because the year values are stored as column names instead of as values in a single column. So before we could apply a function to find the latest year (for example, by using `max`), we would have to first extract the column names to get them as a list and then apply a function to extract the latest year. The problem only gets worse if you would like to find the value for the population for a given region for the latest year. Both of these tasks are greatly simplified once the data is tidied.

Another problem with data in this format is that we don't know what the numbers under each year actually represent. Do those numbers represent population size? Land area? It's not clear. To solve both of these problems, we can reshape this data set to a tidy data format by creating a column called "year" and a column called "population". This transformation—which makes the data "longer"—is shown as the right table in Fig. 3.5. Note that the number of entries in our data frame can change in this transformation. The "untidy" data has 5 rows and 3 columns for a total of 15 entries, whereas the "tidy" data on the right has 15 rows and 2 columns for a total of 30 entries.

We can achieve this effect in Python using the `melt` function from the `pandas` package. The `melt` function combines columns, and is usually used during tidying data when we need to make the data frame longer and narrower. To learn how to use `melt`, we will work through an example with the `region_lang_top5_cities_wide.csv` data set. This data set contains the counts of how many Canadians cited each language as their mother tongue for five major Canadian cities (Toronto, Montréal, Vancouver, Calgary, and

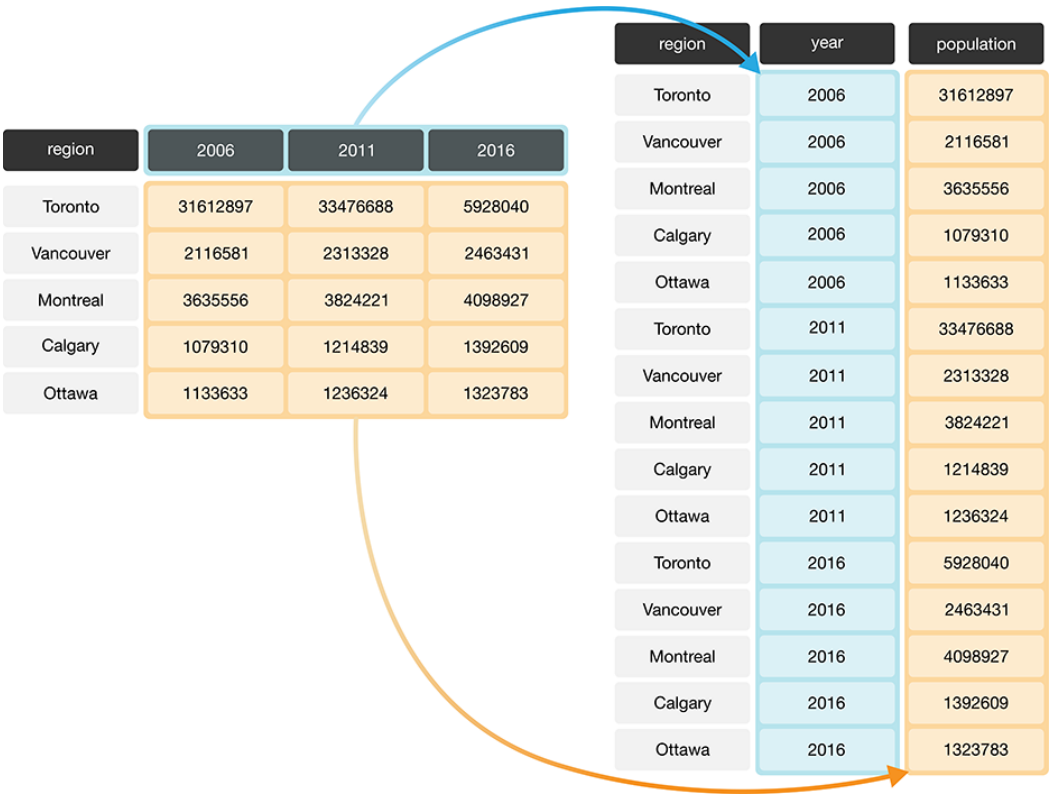


FIGURE 3.5 Melting data from a wide to long data format.

Edmonton) from the 2016 Canadian census. To get started, we will use `pd.read_csv` to load the (untidy) data.

```
lang_wide = pd.read_csv("data/region_lang_top5_cities_wide.csv")
lang_wide
```

```
                                  category                        language ␣
↪\
0                     Aboriginal languages    Aboriginal languages, n.o.s.
1    Non-Official & Non-Aboriginal languages                       Afrikaans
2    Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3    Non-Official & Non-Aboriginal languages                     Akan (Twi)
4    Non-Official & Non-Aboriginal languages                       Albanian
..                                      ...                             ...
209  Non-Official & Non-Aboriginal languages                          Wolof
210                   Aboriginal languages                    Woods Cree
211  Non-Official & Non-Aboriginal languages             Wu (Shanghainese)
212  Non-Official & Non-Aboriginal languages                       Yiddish
213  Non-Official & Non-Aboriginal languages                        Yoruba

     Toronto  Montréal  Vancouver  Calgary  Edmonton
0         80        30         70       20        25
1        985        90       1435      960       575
2        360       240         45       45        65
3       8485      1015        400      705       885
```

(continues on next page)

(continued from previous page)

```
4       13260       2450        1090       1365        770
..        ...        ...         ...        ...        ...
209       165       2440          30        120        130
210         5          0          20         10        155
211      5290       1025        4330        380        235
212      3355       8960         220         80         55
213      3380        210         190       1430        700

[214 rows x 7 columns]
```

What is wrong with the untidy format above? The table on the left in Fig. 3.6 represents the data in the "wide" (messy) format. From a data analysis perspective, this format is not ideal because the values of the variable *region* (Toronto, Montréal, Vancouver, Calgary, and Edmonton) are stored as column names. Thus they are not easily accessible to the data analysis functions we will apply to our data set. Additionally, the *mother tongue* variable values are spread across multiple columns, which will prevent us from doing any desired visualization or statistical tasks until we combine them into one column. For instance, suppose we want to know the languages with the highest number of Canadians reporting it as their mother tongue among all five regions. This question would be tough to answer with the data in its current format. We *could* find the answer with the data in this format, though it would be much easier to answer if we tidy our data first. If mother tongue were instead stored as one column, as shown in the tidy data on the right in Fig. 3.6, we could simply use one line of code (`df["mother_tongue"].max()`) to get the maximum value.

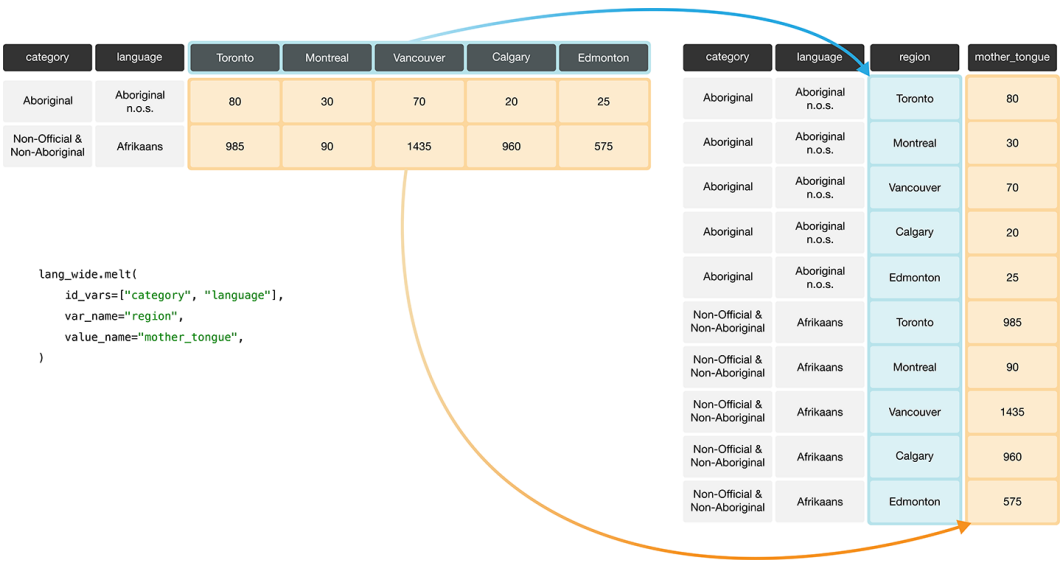


FIGURE 3.6 Going from wide to long with the `melt` function.

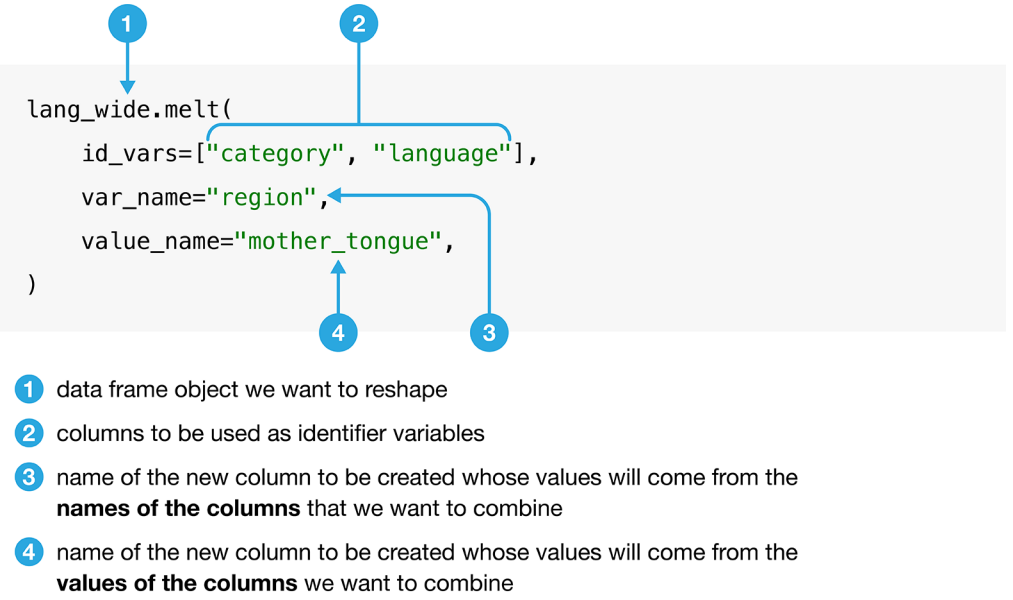


FIGURE 3.7 Syntax for the `melt` function.

Fig. 3.7 details the arguments that we need to specify in the `melt` function to accomplish this data transformation.

We use `melt` to combine the Toronto, Montréal, Vancouver, Calgary, and Edmonton columns into a single column called `region`, and create a column called `mother_tongue` that contains the count of how many Canadians report each language as their mother tongue for each metropolitan area.

```
lang_mother_tidy = lang_wide.melt(
    id_vars=["category", "language"],
    var_name="region",
    value_name="mother_tongue",
)
lang_mother_tidy
```

```
                                    category                          language␣
↪ \
0                        Aboriginal languages    Aboriginal languages, n.o.s.
1     Non-Official & Non-Aboriginal languages                       Afrikaans
2     Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3     Non-Official & Non-Aboriginal languages                    Akan (Twi)
4     Non-Official & Non-Aboriginal languages                      Albanian
...                                      ...                             ...
1065  Non-Official & Non-Aboriginal languages                         Wolof
1066                     Aboriginal languages                   Woods Cree
1067  Non-Official & Non-Aboriginal languages            Wu (Shanghainese)
1068  Non-Official & Non-Aboriginal languages                      Yiddish
1069  Non-Official & Non-Aboriginal languages                        Yoruba

        region  mother_tongue
0      Toronto             80
```

(continues on next page)

(continued from previous page)

```
1       Toronto            985
2       Toronto            360
3       Toronto           8485
4       Toronto          13260
...         ...            ...
1065  Edmonton             130
1066  Edmonton             155
1067  Edmonton             235
1068  Edmonton              55
1069  Edmonton             700

[1070 rows x 4 columns]
```

Note: In the code above, the call to the `melt` function is split across several lines. Recall from Chapter 1 that this is allowed in certain cases. For example, when calling a function as above, the input arguments are between parentheses `()` and Python knows to keep reading on the next line. Each line ends with a comma `,` making it easier to read. Splitting long lines like this across multiple lines is encouraged as it helps significantly with code readability. Generally speaking, you should limit each line of code to about 80 characters.

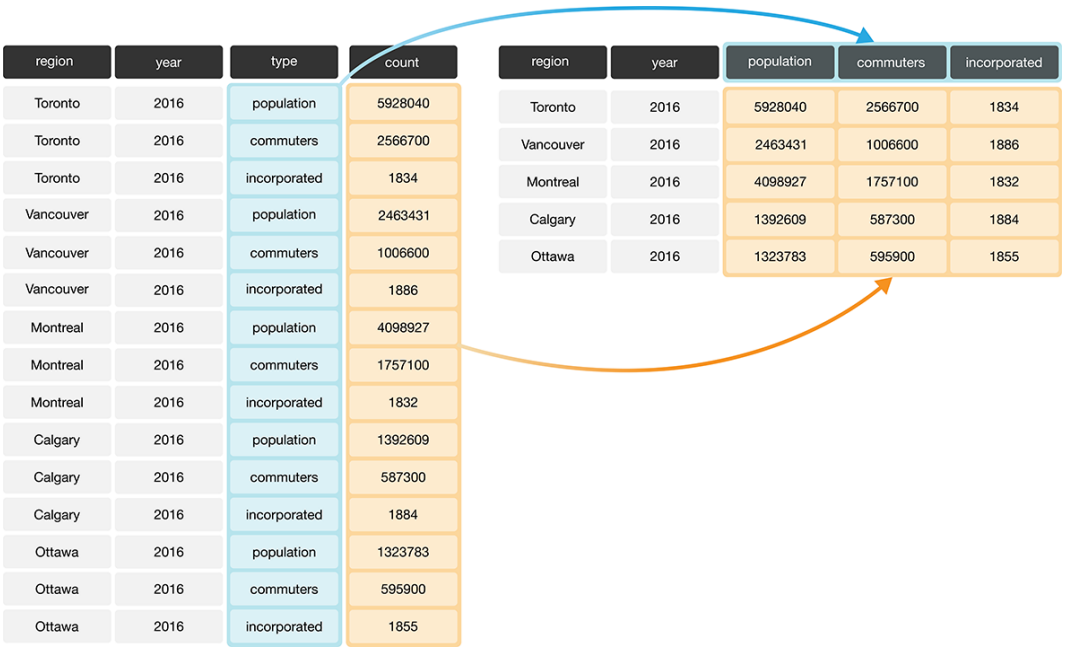
The data above is now tidy because all three criteria for tidy data have now been met:

1. All the variables (`category`, `language`, `region` and `mother_tongue`) are now their own columns in the data frame.
2. Each observation, i.e., each `category`, `language`, `region`, and count of Canadians where that language is the mother tongue, are in a single row.
3. Each value is a single cell, i.e., its row, column position in the data frame is not shared with another value.

3.4.2 Tidying up: going from long to wide using `pivot`

Suppose we have observations spread across multiple rows rather than in a single row. For example, in Fig. 3.8, the table on the left is in an untidy, long format because the `count` column contains three variables (population, commuter, and incorporated count) and information about each observation (here, population, commuter, and incorporated counts for a region) is split across three rows. Remember: one of the criteria for tidy data is that each observation must be in a single row.

Using data in this format—where two or more variables are mixed together in a single column—makes it harder to apply many usual `pandas` functions.



region	year	type	count
Toronto	2016	population	5928040
Toronto	2016	commuters	2566700
Toronto	2016	incorporated	1834
Vancouver	2016	population	2463431
Vancouver	2016	commuters	1006600
Vancouver	2016	incorporated	1886
Montreal	2016	population	4098927
Montreal	2016	commuters	1757100
Montreal	2016	incorporated	1832
Calgary	2016	population	1392609
Calgary	2016	commuters	587300
Calgary	2016	incorporated	1884
Ottawa	2016	population	1323783
Ottawa	2016	commuters	595900
Ottawa	2016	incorporated	1855

region	year	population	commuters	incorporated
Toronto	2016	5928040	2566700	1834
Vancouver	2016	2463431	1006600	1886
Montreal	2016	4098927	1757100	1832
Calgary	2016	1392609	587300	1884
Ottawa	2016	1323783	595900	1855

FIGURE 3.8 Going from long to wide data.

For example, finding the maximum number of commuters would require an additional step of filtering for the commuter values before the maximum can be computed. In comparison, if the data were tidy, all we would have to do is compute the maximum value for the commuter column. To reshape this untidy data set to a tidy (and in this case, wider) format, we need to create columns called "population", "commuters", and "incorporated". This is illustrated in the right table of Fig. 3.8.

To tidy this type of data in Python, we can use the `pivot` function. The `pivot` function generally increases the number of columns (widens) and decreases the number of rows in a data set. To learn how to use `pivot`, we will work through an example with the `region_lang_top5_cities_long.csv` data set. This data set contains the number of Canadians reporting the primary language at home and work for five major cities (Toronto, Montréal, Vancouver, Calgary, and Edmonton).

```
lang_long = pd.read_csv("data/region_lang_top5_cities_long.csv")
lang_long
```

```
         region                              category  \
0      Montréal                 Aboriginal languages
1      Montréal                 Aboriginal languages
2       Toronto                 Aboriginal languages
3       Toronto                 Aboriginal languages
4       Calgary                 Aboriginal languages
```

(continues on next page)

(continued from previous page)

```
...         ...                                        ...
2135    Calgary  Non-Official & Non-Aboriginal languages
2136   Edmonton  Non-Official & Non-Aboriginal languages
2137   Edmonton  Non-Official & Non-Aboriginal languages
2138  Vancouver  Non-Official & Non-Aboriginal languages
2139  Vancouver  Non-Official & Non-Aboriginal languages

                          language          type  count
0     Aboriginal languages, n.o.s.  most_at_home     15
1     Aboriginal languages, n.o.s.  most_at_work      0
2     Aboriginal languages, n.o.s.  most_at_home     50
3     Aboriginal languages, n.o.s.  most_at_work      0
4     Aboriginal languages, n.o.s.  most_at_home      5
...                            ...           ...    ...
2135                        Yoruba  most_at_work      0
2136                        Yoruba  most_at_home    280
2137                        Yoruba  most_at_work      0
2138                        Yoruba  most_at_home     40
2139                        Yoruba  most_at_work      0

[2140 rows x 5 columns]
```

What makes the data set shown above untidy? In this example, each observation is a language in a region. However, each observation is split across multiple rows: one where the count for `most_at_home` is recorded, and the other where the count for `most_at_work` is recorded. Suppose the goal with this data was to visualize the relationship between the number of Canadians reporting their primary language at home and work. Doing that would be difficult with this data in its current form, since these two variables are stored in the same column. Fig. 3.9 shows how this data will be tidied using the `pivot` function.

Fig. 3.10 details the arguments that we need to specify in the `pivot` function.

We will apply the function as detailed in Fig. 3.10, and then rename the columns.

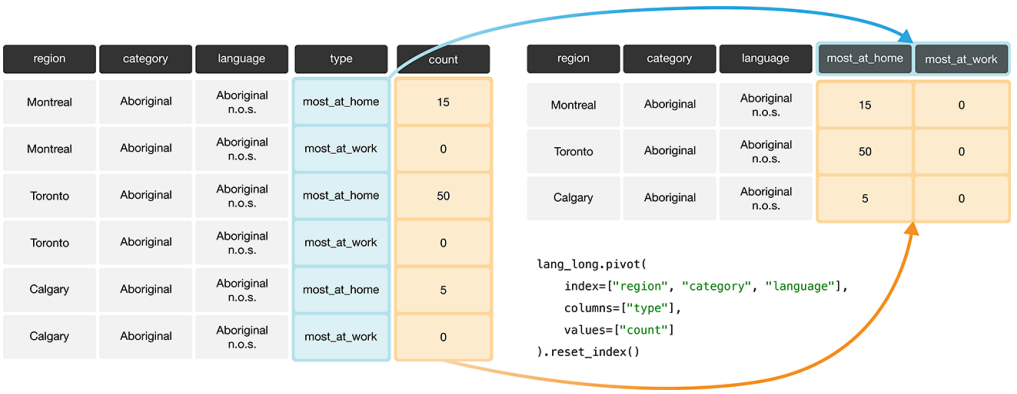


FIGURE 3.9 Going from long to wide with the `pivot` function.

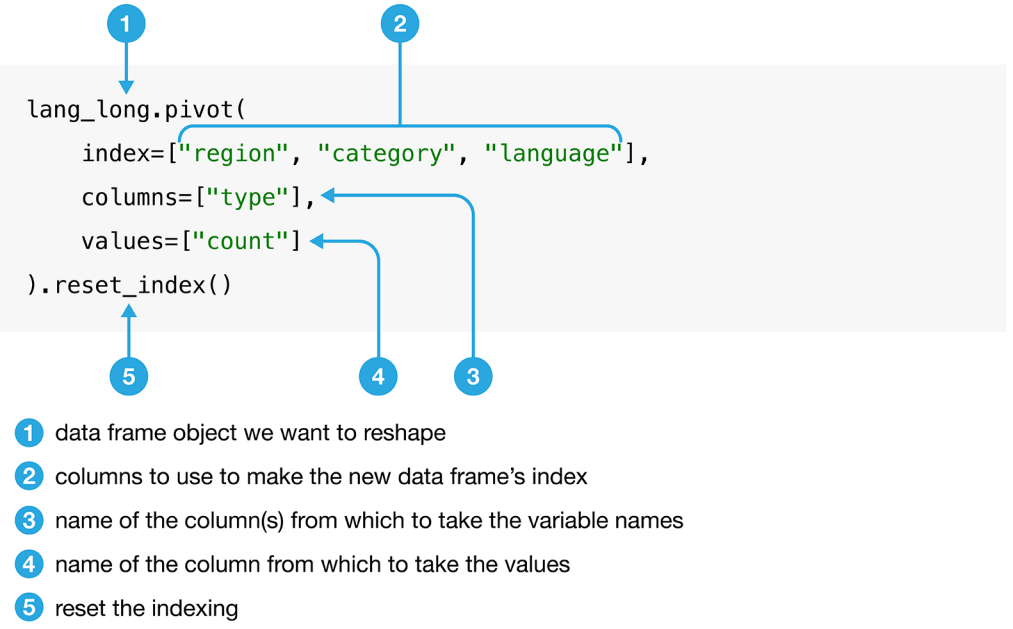


FIGURE 3.10 Syntax for the `pivot` function.

```
lang_home_tidy = lang_long.pivot(
    index=["region", "category", "language"],
    columns=["type"],
    values=["count"]
).reset_index()

lang_home_tidy.columns = [
    "region",
    "category",
    "language",
    "most_at_home",
    "most_at_work",
]
lang_home_tidy
```

```
         region                                  category  \
0       Calgary                     Aboriginal languages
1       Calgary                     Aboriginal languages
2       Calgary                     Aboriginal languages
3       Calgary                     Aboriginal languages
4       Calgary                     Aboriginal languages
...         ...                                       ...
1065  Vancouver  Non-Official & Non-Aboriginal languages
1066  Vancouver  Non-Official & Non-Aboriginal languages
1067  Vancouver  Non-Official & Non-Aboriginal languages
1068  Vancouver                       Official languages
1069  Vancouver                       Official languages

                          language  most_at_home  most_at_work
0     Aboriginal languages, n.o.s.             5             0
1     Algonquian languages, n.i.e.             0             0
2                        Algonquin             0             0
```

(continues on next page)

(continued from previous page)

```
3     Athabaskan languages, n.i.e.              0              0
4                          Atikamekw             0              0
...                              ...           ...            ...
1065             Wu (Shanghainese)          2495             45
1066                         Yiddish            10              0
1067                          Yoruba            40              0
1068                         English       1622735        1330555
1069                          French          8630           3245

[1070 rows x 5 columns]
```

In the first step, note that we added a call to `reset_index`. When `pivot` is called with multiple column names passed to the `index`, those entries become the "name" of each row that would be used when you filter rows with `[]` or `loc` rather than just simple numbers. This can be confusing … What `reset_index` does is sets us back with the usual expected behavior where each row is "named" with an integer. This is a subtle point, but the main take-away is that when you call `pivot`, it is a good idea to call `reset_index` afterwards.

The second operation we applied is to rename the columns. When we perform the `pivot` operation, it keeps the original column name `"count"` and adds the `"type"` as a second column name. Having two names for a column can be confusing. So we rename giving each column only one name.

We can print out some useful information about our data frame using the `info` function. In the first row it tells us the `type` of `lang_home_tidy` (it is a `pandas DataFrame`). The second row tells us how many rows there are: 1070, and to index those rows, you can use numbers between 0 and 1069 (remember that Python starts counting at 0!). Next, there is a print out about the data columns. Here there are 5 columns total. The little table it prints out tells you the name of each column, the number of non-null values (e.g. the number of entries that are not missing values), and the type of the entries. Finally the last two rows summarize the types of each column and how much memory the data frame is using on your computer.

```
lang_home_tidy.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1070 entries, 0 to 1069
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   region        1070 non-null   object
 1   category      1070 non-null   object
 2   language      1070 non-null   object
 3   most_at_home  1070 non-null   int64
 4   most_at_work  1070 non-null   int64
```

(continues on next page)

(continued from previous page)

```
dtypes: int64(2), object(3)
memory usage: 41.9+ KB
```

The data is now tidy. We can go through the three criteria again to check that this data is a tidy data set.

1. All the statistical variables are their own columns in the data frame (i.e., `most_at_home`, and `most_at_work` have been separated into their own columns in the data frame).
2. Each observation (i.e., each language in a region) is in a single row.
3. Each value is a single cell (i.e., its row, column position in the data frame is not shared with another value).

You might notice that we have the same number of columns in the tidy data set as we did in the messy one. Therefore `pivot` didn't really "widen" the data. This is just because the original `type` column only had two categories in it. If it had more than two, `pivot` would have created more columns, and we would see the data set "widen".

3.4.3 Tidying up: using `str.split` to deal with multiple separators

Data are also not considered tidy when multiple values are stored in the same cell. The data set we show below is even messier than the ones we dealt with above: the `Toronto`, `Montréal`, `Vancouver`, `Calgary`, and `Edmonton` columns contain the number of Canadians reporting their primary language at home and work in one column separated by the separator (`/`). The column names are the values of a variable, *and* each value does not have its own cell. To turn this messy data into tidy data, we'll have to fix these issues.

```
lang_messy = pd.read_csv("data/region_lang_top5_cities_messy.csv")
lang_messy
```

```
                                    category                          language ␣
↪\
0                       Aboriginal languages    Aboriginal languages, n.o.s.
1    Non-Official & Non-Aboriginal languages                       Afrikaans
2    Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3    Non-Official & Non-Aboriginal languages                     Akan (Twi)
4    Non-Official & Non-Aboriginal languages                       Albanian
..                                      ...                             ...
209  Non-Official & Non-Aboriginal languages                         Wolof
210                     Aboriginal languages                   Woods Cree
211  Non-Official & Non-Aboriginal languages            Wu (Shanghainese)
212  Non-Official & Non-Aboriginal languages                      Yiddish
213  Non-Official & Non-Aboriginal languages                       Yoruba
```

(continues on next page)

(continued from previous page)

```
      Toronto  Montréal Vancouver Calgary Edmonton
0        50/0      15/0      15/0     5/0     10/0
1       265/0      10/0    520/10  505/15    300/0
2      185/10      65/0      10/0    15/0     20/0
3     4045/20     440/0    125/10   330/0    445/0
4    6380/215   1445/20    530/10  620/25   370/10
..        ...       ...       ...     ...      ...
209      75/0     770/0       5/0    65/0    90/10
210      0/10       0/0       5/0     0/0     20/0
211   3130/30    760/15   2495/45   210/0    120/0
212    350/20  6665/860      10/0    10/0      0/0
213   1080/10      45/0      40/0   350/0    280/0

[214 rows x 7 columns]
```

First, we'll use `melt` to create two columns, `region` and `value`, similar to what we did previously. The new `region` columns will contain the region names, and the new column `value` will be a temporary holding place for the data that we need to further separate, i.e., the number of Canadians reporting their primary language at home and work.

```
lang_messy_longer = lang_messy.melt(
    id_vars=["category", "language"],
    var_name="region",
    value_name="value",
)

lang_messy_longer
```

```
                                    category                          language␣
↪ \
0                        Aboriginal languages    Aboriginal languages, n.o.s.
1     Non-Official & Non-Aboriginal languages                       Afrikaans
2     Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3     Non-Official & Non-Aboriginal languages                    Akan (Twi)
4     Non-Official & Non-Aboriginal languages                      Albanian
...                                      ...                             ...
1065  Non-Official & Non-Aboriginal languages                         Wolof
1066                     Aboriginal languages                   Woods Cree
1067  Non-Official & Non-Aboriginal languages           Wu (Shanghainese)
1068  Non-Official & Non-Aboriginal languages                      Yiddish
1069  Non-Official & Non-Aboriginal languages                       Yoruba

        region     value
0      Toronto      50/0
1      Toronto     265/0
2      Toronto    185/10
3      Toronto   4045/20
4      Toronto  6380/215
...        ...       ...
1065  Edmonton     90/10
1066  Edmonton      20/0
1067  Edmonton     120/0
1068  Edmonton       0/0
1069  Edmonton     280/0
```

(continues on next page)

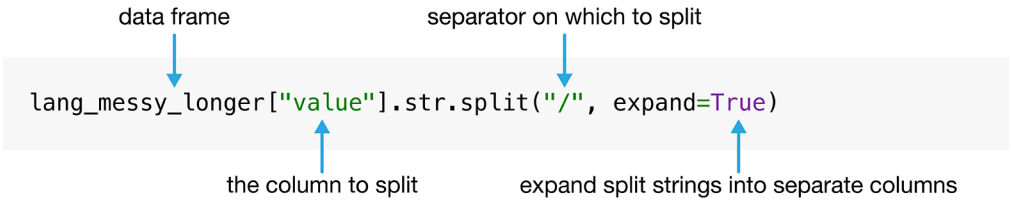


FIGURE 3.11 Syntax for the `str.split` function.

(continued from previous page)

```
[1070 rows x 4 columns]
```

Next, we'll split the `value` column into two columns. In basic Python, if we wanted to split the string `"50/0"` into two numbers `["50", "0"]` we would use the `split` method on the string, and specify that the split should be made on the slash character `"/"`.

```
"50/0".split("/")
```

```
['50', '0']
```

The `pandas` package provides similar functions that we can access by using the `str` method. So to split all of the entries for an entire column in a data frame, we will use the `str.split` method. The output of this method is a data frame with two columns: one containing only the counts of Canadians that speak each language most at home, and the other containing only the counts of Canadians that speak each language most at work for each region. We drop the no-longer-needed `value` column from the `lang_messy_longer` data frame, and then assign the two columns from `str.split` to two new columns. Fig. 3.11 outlines what we need to specify to use `str.split`.

```
tidy_lang = lang_messy_longer.drop(columns=["value"])
tidy_lang[["most_at_home", "most_at_work"]] = lang_messy_longer["value"].str.
↪split("/", expand=True)
tidy_lang
```

```
                                    category                          language␣
↪ \
0                        Aboriginal languages    Aboriginal languages, n.o.s.
1     Non-Official & Non-Aboriginal languages                       Afrikaans
2     Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3     Non-Official & Non-Aboriginal languages                    Akan (Twi)
4     Non-Official & Non-Aboriginal languages                      Albanian
...                                      ...                             ...
1065  Non-Official & Non-Aboriginal languages                         Wolof
1066                     Aboriginal languages                   Woods Cree
1067  Non-Official & Non-Aboriginal languages            Wu (Shanghainese)
```

(continues on next page)

(continued from previous page)

```
1068  Non-Official & Non-Aboriginal languages                    Yiddish
1069  Non-Official & Non-Aboriginal languages                     Yoruba

         region most_at_home most_at_work
0       Toronto           50            0
1       Toronto          265            0
2       Toronto          185           10
3       Toronto         4045           20
4       Toronto         6380          215
...         ...          ...          ...
1065   Edmonton           90           10
1066   Edmonton           20            0
1067   Edmonton          120            0
1068   Edmonton            0            0
1069   Edmonton          280            0

[1070 rows x 5 columns]
```

Is this data set now tidy? If we recall the three criteria for tidy data:

- each row is a single observation,
- each column is a single variable, and
- each value is a single cell.

We can see that this data now satisfies all three criteria, making it easier to analyze. But we aren't done yet. Although we can't see it in the data frame above, all of the variables are actually `object` data types. We can check this using the `info` method.

```
tidy_lang.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1070 entries, 0 to 1069
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   category      1070 non-null   object
 1   language      1070 non-null   object
 2   region        1070 non-null   object
 3   most_at_home  1070 non-null   object
 4   most_at_work  1070 non-null   object
dtypes: object(5)
memory usage: 41.9+ KB
```

Object columns in `pandas` data frames are columns of strings or columns with mixed types. In the previous example in Section 3.4.2, the `most_at_home` and `most_at_work` variables were `int64` (integer), which is a type of numeric data. This change is due to the separator (`/`) when we read in this messy data set. Python read these columns in as string types, and by default, `str.split` will return columns with the `object` data type.

It makes sense for `region`, `category`, and `language` to be stored as an `object` type since they hold categorical values. However, suppose we want to apply any functions that treat the `most_at_home` and `most_at_work` columns as a number (e.g., finding rows above a numeric threshold of a column). That won't be possible if the variable is stored as an `object`. Fortunately, the `astype` method from `pandas` provides a natural way to fix problems like this: it will convert the column to a selected data type. In this case, we choose the `int` data type to indicate that these variables contain integer counts. Note that below we *assign* the new numerical series to the `most_at_home` and `most_at_work` columns in `tidy_lang`; we have seen this syntax before in Section 1.9, and we will discuss it in more depth later in this chapter in Section 3.11.

```
tidy_lang["most_at_home"] = tidy_lang["most_at_home"].astype("int")
tidy_lang["most_at_work"] = tidy_lang["most_at_work"].astype("int")
tidy_lang
```

```
                                    category                         language␣
↪ \
0                        Aboriginal languages    Aboriginal languages, n.o.s.
1     Non-Official & Non-Aboriginal languages                       Afrikaans
2     Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3     Non-Official & Non-Aboriginal languages                     Akan (Twi)
4     Non-Official & Non-Aboriginal languages                       Albanian
...                                      ...                            ...
1065  Non-Official & Non-Aboriginal languages                         Wolof
1066                     Aboriginal languages                   Woods Cree
1067  Non-Official & Non-Aboriginal languages             Wu (Shanghainese)
1068  Non-Official & Non-Aboriginal languages                       Yiddish
1069  Non-Official & Non-Aboriginal languages                        Yoruba

         region  most_at_home  most_at_work
0       Toronto            50             0
1       Toronto           265             0
2       Toronto           185            10
3       Toronto          4045            20
4       Toronto          6380           215
...         ...           ...           ...
1065  Edmonton            90            10
1066  Edmonton            20             0
1067  Edmonton           120             0
1068  Edmonton             0             0
1069  Edmonton           280             0

[1070 rows x 5 columns]
```

```
tidy_lang.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1070 entries, 0 to 1069
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   category      1070 non-null   object
```

(continues on next page)

(continued from previous page)

```
 1   language      1070 non-null   object
 2   region        1070 non-null   object
 3   most_at_home  1070 non-null   int64
 4   most_at_work  1070 non-null   int64
dtypes: int64(2), object(3)
memory usage: 41.9+ KB
```

Now we see `most_at_home` and `most_at_work` columns are of `int64` data types, indicating they are integer data types (i.e., numbers).

3.5 Using `[]` to extract rows or columns

Now that the `tidy_lang` data is indeed *tidy*, we can start manipulating it using the powerful suite of functions from the `pandas`. We will first revisit the `[]` from Chapter 1, which lets us obtain a subset of either the rows **or** the columns of a data frame. This section will highlight more advanced usage of the `[]` function, including an in-depth treatment of the variety of logical statements one can use in the `[]` to select subsets of rows.

3.5.1 Extracting columns by name

Recall that if we provide a list of column names, `[]` returns the subset of columns with those names as a data frame. Suppose we wanted to select the columns `language`, `region`, `most_at_home` and `most_at_work` from the `tidy_lang` data set. Using what we learned in Chapter 1, we can pass all of these column names into the square brackets.

```
tidy_lang[["language", "region", "most_at_home", "most_at_work"]]
```

```
                            language     region  most_at_home  most_at_work
0       Aboriginal languages, n.o.s.    Toronto            50             0
1                           Afrikaans    Toronto           265             0
2     Afro-Asiatic languages, n.i.e.    Toronto           185            10
3                           Akan (Twi)    Toronto          4045            20
4                             Albanian    Toronto          6380           215
...                                ...        ...           ...           ...
1065                            Wolof   Edmonton            90            10
1066                      Woods Cree   Edmonton            20             0
1067               Wu (Shanghainese)   Edmonton           120             0
1068                         Yiddish   Edmonton             0             0
1069                           Yoruba   Edmonton           280             0

[1070 rows x 4 columns]
```

Likewise, if we pass a list containing a single column name, a data frame with this column will be returned.

```
tidy_lang[["language"]]
```

```
                            language
0       Aboriginal languages, n.o.s.
1                           Afrikaans
2     Afro-Asiatic languages, n.i.e.
3                         Akan (Twi)
4                           Albanian
...                               ...
1065                          Wolof
1066                      Woods Cree
1067              Wu (Shanghainese)
1068                        Yiddish
1069                         Yoruba

[1070 rows x 1 columns]
```

When we need to extract only a single column, we can also pass the column name as a string rather than a list. The returned data type will now be a series. Throughout this textbook, we will mostly extract single columns this way, but we will point out a few occasions where it is advantageous to extract single columns as data frames.

```
tidy_lang["language"]
```

```
0         Aboriginal languages, n.o.s.
1                           Afrikaans
2       Afro-Asiatic languages, n.i.e.
3                           Akan (Twi)
4                             Albanian
                     ...
1065                            Wolof
1066                        Woods Cree
1067                Wu (Shanghainese)
1068                          Yiddish
1069                           Yoruba
Name: language, Length: 1070, dtype: object
```

3.5.2 Extracting rows that have a certain value with ==

Suppose we are only interested in the subset of rows in `tidy_lang` corresponding to the official languages of Canada (English and French). We can extract these rows by using the *equivalency operator* (==) to compare the values of the `category` column with the value `"Official languages"`. With these arguments, `[]` returns a data frame with all the columns of the input data frame but only the rows we asked for in the logical statement, i.e., those where the `category` column holds the value `"Official languages"`. We name this data frame `official_langs`.

```
official_langs = tidy_lang[tidy_lang["category"] == "Official languages"]
official_langs
```

```
                category language     region  most_at_home  most_at_work
54   Official languages  English    Toronto       3836770       3218725
59   Official languages   French    Toronto         29800         11940
268  Official languages  English   Montréal        620510        412120
273  Official languages   French   Montréal       2669195       1607550
482  Official languages  English  Vancouver       1622735       1330555
487  Official languages   French  Vancouver          8630          3245
696  Official languages  English    Calgary       1065070        844740
701  Official languages   French    Calgary          8630          2140
910  Official languages  English   Edmonton       1050410        792700
915  Official languages   French   Edmonton         10950          2520
```

3.5.3 Extracting rows that do not have a certain value with `!=`

What if we want all the other language categories in the data set *except* for those in the `"Official languages"` category? We can accomplish this with the `!=` operator, which means "not equal to". So if we want to find all the rows where the `category` does *not* equal `"Official languages"` we write the code below.

```
tidy_lang[tidy_lang["category"] != "Official languages"]
```

```
                                    category                         language␣
↪ \
0                       Aboriginal languages    Aboriginal languages, n.o.s.
1     Non-Official & Non-Aboriginal languages                       Afrikaans
2     Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3     Non-Official & Non-Aboriginal languages                     Akan (Twi)
4     Non-Official & Non-Aboriginal languages                       Albanian
...                                      ...                            ...
1065  Non-Official & Non-Aboriginal languages                         Wolof
1066                    Aboriginal languages                   Woods Cree
1067  Non-Official & Non-Aboriginal languages              Wu (Shanghainese)
1068  Non-Official & Non-Aboriginal languages                       Yiddish
1069  Non-Official & Non-Aboriginal languages                        Yoruba

        region  most_at_home  most_at_work
0      Toronto            50             0
1      Toronto           265             0
2      Toronto           185            10
3      Toronto          4045            20
4      Toronto          6380           215
...        ...           ...           ...
1065  Edmonton            90            10
1066  Edmonton            20             0
1067  Edmonton           120             0
1068  Edmonton             0             0
1069  Edmonton           280             0

[1060 rows x 5 columns]
```

3.5.4 Extracting rows satisfying multiple conditions using `&`

Suppose now we want to look at only the rows for the French language in Montréal. To do this, we need to filter the data set to find rows that satisfy

multiple conditions simultaneously. We can do this with the ampersand symbol (`&`), which is interpreted by Python as "and". We write the code as shown below to filter the `official_langs` data frame to subset the rows where `region == "Montréal"` *and* `language == "French"`.

```
tidy_lang[
  (tidy_lang["region"] == "Montréal") &
  (tidy_lang["language"] == "French")
]
```

```
              category language    region  most_at_home  most_at_work
273  Official languages   French  Montréal        2669195       1607550
```

3.5.5 Extracting rows satisfying at least one condition using `|`

Suppose we were interested in only those rows corresponding to cities in Alberta in the `official_langs` data set (Edmonton and Calgary). We can't use `&` as we did above because `region` cannot be both Edmonton *and* Calgary simultaneously. Instead, we can use the vertical pipe (`|`) logical operator, which gives us the cases where one condition *or* another condition *or* both are satisfied. In the code below, we ask Python to return the rows where the `region` columns are equal to "Calgary" *or* "Edmonton".

```
official_langs[
    (official_langs["region"] == "Calgary") |
    (official_langs["region"] == "Edmonton")
]
```

```
              category language    region  most_at_home  most_at_work
696  Official languages  English   Calgary        1065070        844740
701  Official languages   French   Calgary           8630          2140
910  Official languages  English  Edmonton        1050410        792700
915  Official languages   French  Edmonton          10950          2520
```

3.5.6 Extracting rows with values in a list using `isin`

Next, suppose we want to see the populations of our five cities. Let's read in the `region_data.csv` file that comes from the 2016 Canadian census, as it contains statistics for number of households, land area, population, and number of dwellings for different regions.

```
region_data = pd.read_csv("data/region_data.csv")
region_data
```

```
              region  households        area  population  dwellings
0          Belleville       43002  1354.65121      103472      45050
1          Lethbridge       45696  3046.69699      117394      48317
2         Thunder Bay       52545  2618.26318      121621      57146
3        Peterborough       50533  1636.98336      121721      55662
```

(continues on next page)

(continued from previous page)

```
4           Saint John       52872  3793.42158      126202      58398
..                 ...         ...         ...         ...        ...
30  Ottawa - Gatineau      535499  7168.96442     1323783     571146
31            Calgary      519693  5241.70103     1392609     544870
32          Vancouver      960894  3040.41532     2463431    1027613
33           Montréal     1727310  4638.24059     4098927    1823281
34            Toronto     2135909  6269.93132     5928040    2235145

[35 rows x 5 columns]
```

To get the population of the five cities we can filter the data set using the `isin` method. The `isin` method is used to see if an element belongs to a list. Here we are filtering for rows where the value in the `region` column matches any of the five cities we are interested in: Toronto, Montréal, Vancouver, Calgary, and Edmonton.

```
city_names = ["Toronto", "Montréal", "Vancouver", "Calgary", "Edmonton"]
five_cities = region_data[region_data["region"].isin(city_names)]
five_cities
```

```
       region  households        area  population  dwellings
29   Edmonton      502143  9857.77908     1321426     537634
31    Calgary      519693  5241.70103     1392609     544870
32  Vancouver      960894  3040.41532     2463431    1027613
33   Montréal     1727310  4638.24059     4098927    1823281
34    Toronto     2135909  6269.93132     5928040    2235145
```

Note: What's the difference between == and `isin`? Suppose we have two Series, `seriesA` and `seriesB`. If you type `seriesA == seriesB` into Python it will compare the series element by element. Python checks if the first element of `seriesA` equals the first element of `seriesB`, the second element of `seriesA` equals the second element of `seriesB`, and so on. On the other hand, `seriesA.isin(seriesB)` compares the first element of `seriesA` to all the elements in `seriesB`. Then the second element of `seriesA` is compared to all the elements in `seriesB`, and so on. Notice the difference between == and `isin` in the example below.

```
pd.Series(["Vancouver", "Toronto"]) == pd.Series(["Toronto", "Vancouver"])
```

```
0    False
1    False
dtype: bool
```

```
pd.Series(["Vancouver", "Toronto"]).isin(pd.Series(["Toronto", "Vancouver"]))
```

```
0    True
1    True
dtype: bool
```

3.5.7 Extracting rows above or below a threshold using > and <

We saw in Section 3.5.4 that 2,669,195 people reported speaking French in Montréal as their primary language at home. If we are interested in finding the official languages in regions with higher numbers of people who speak it as their primary language at home compared to French in Montréal, then we can use `[]` to obtain rows where the value of `most_at_home` is greater than 2,669,195. We use the > symbol to look for values *above* a threshold, and the < symbol to look for values *below* a threshold. The >= and <= symbols similarly look for *equal to or above* a threshold and *equal to or below* a threshold.

```
official_langs[official_langs["most_at_home"] > 2669195]
```

	category	language	region	most_at_home	most_at_work
54	Official languages	English	Toronto	3836770	3218725

This operation returns a data frame with only one row, indicating that when considering the official languages, only English in Toronto is reported by more people as their primary language at home than French in Montréal according to the 2016 Canadian census.

3.5.8 Extracting rows using `query`

You can also extract rows above, below, equal or not-equal to a threshold using the `query` method. For example the following gives us the same result as when we used `official_langs[official_langs["most_at_home"] > 2669195]`.

```
official_langs.query("most_at_home > 2669195")
```

	category	language	region	most_at_home	most_at_work
54	Official languages	English	Toronto	3836770	3218725

The query (criteria we are using to select values) is input as a string. The `query` method is less often used than the earlier approaches we introduced, but it can come in handy to make long chains of filtering operations a bit easier to read.

3.6 Using `loc[]` to filter rows and select columns

The `[]` operation is only used when you want to either filter rows **or** select columns; it cannot be used to do both operations at the same time. This is where `loc[]` comes in. For the first example, recall `loc[]` from Chapter 1, which lets us create a subset of the rows and columns in the `tidy_lang` data frame. In the first argument to `loc[]`, we specify a logical statement that filters the rows to only those pertaining to the Toronto region, and the second argument specifies a list of columns to keep by name.

```
tidy_lang.loc[
    tidy_lang["region"] == "Toronto",
    ["language", "region", "most_at_home", "most_at_work"]
]
```

```
                           language   region  most_at_home  most_at_work
0      Aboriginal languages, n.o.s.  Toronto            50             0
1                         Afrikaans  Toronto           265             0
2    Afro-Asiatic languages, n.i.e.  Toronto           185            10
3                      Akan (Twi)  Toronto          4045            20
4                         Albanian  Toronto          6380           215
..                             ...      ...           ...           ...
209                          Wolof  Toronto            75             0
210                    Woods Cree  Toronto             0            10
211              Wu (Shanghainese)  Toronto          3130            30
212                        Yiddish  Toronto           350            20
213                         Yoruba  Toronto          1080            10

[214 rows x 4 columns]
```

In addition to simultaneous subsetting of rows and columns, `loc[]` has two more special capabilities beyond those of `[]`. First, `loc[]` has the ability to specify *ranges* of rows and columns. For example, note that the list of columns `language`, `region`, `most_at_home`, `most_at_work` corresponds to the *range* of columns from `language` to `most_at_work`. Rather than explicitly listing all of the column names as we did above, we can ask for the range of columns `"language":"most_at_work"`; the `:`-syntax denotes a range, and is supported by the `loc[]` function, but not by `[]`.

```
tidy_lang.loc[
    tidy_lang["region"] == "Toronto",
    "language":"most_at_work"
]
```

```
                           language   region  most_at_home  most_at_work
0      Aboriginal languages, n.o.s.  Toronto            50             0
1                         Afrikaans  Toronto           265             0
2    Afro-Asiatic languages, n.i.e.  Toronto           185            10
3                      Akan (Twi)  Toronto          4045            20
```

(continues on next page)

(continued from previous page)

```
4                          Albanian  Toronto          6380           215
..                              ...      ...           ...           ...
209                            Wolof  Toronto            75             0
210                        Woods Cree  Toronto             0            10
211                Wu (Shanghainese)  Toronto          3130            30
212                          Yiddish  Toronto           350            20
213                           Yoruba  Toronto          1080            10

[214 rows x 4 columns]
```

We can pass `:` by itself—without anything before or after—to denote that we want to retrieve everything. For example, to obtain a subset of all rows and only those columns ranging from `language` to `most_at_work`, we could use the following expression.

```
tidy_lang.loc[:, "language":"most_at_work"]
```

```
                            language    region  most_at_home  most_at_work
0       Aboriginal languages, n.o.s.   Toronto            50             0
1                           Afrikaans   Toronto           265             0
2     Afro-Asiatic languages, n.i.e.   Toronto           185            10
3                           Akan (Twi)   Toronto          4045            20
4                             Albanian   Toronto          6380           215
...                                ...       ...           ...           ...
1065                             Wolof  Edmonton            90            10
1066                         Woods Cree  Edmonton            20             0
1067                 Wu (Shanghainese)  Edmonton           120             0
1068                           Yiddish  Edmonton             0             0
1069                            Yoruba  Edmonton           280             0

[1070 rows x 4 columns]
```

We can also omit the beginning or end of the `:` range expression to denote that we want "everything up to" or "everything after" an element. For example, if we want all of the columns including and after `language`, we can write the expression:

```
tidy_lang.loc[:, "language":]
```

```
                            language    region  most_at_home  most_at_work
0       Aboriginal languages, n.o.s.   Toronto            50             0
1                           Afrikaans   Toronto           265             0
2     Afro-Asiatic languages, n.i.e.   Toronto           185            10
3                           Akan (Twi)   Toronto          4045            20
4                             Albanian   Toronto          6380           215
...                                ...       ...           ...           ...
1065                             Wolof  Edmonton            90            10
1066                         Woods Cree  Edmonton            20             0
1067                 Wu (Shanghainese)  Edmonton           120             0
1068                           Yiddish  Edmonton             0             0
1069                            Yoruba  Edmonton           280             0

[1070 rows x 4 columns]
```

By not putting anything after the `:`, Python reads this as "from `language` until the last column". Similarly, we can specify that we want everything up to and including `language` by writing the expression:

```
tidy_lang.loc[:, :"language"]
```

```
                                    category                          language
0                          Aboriginal languages     Aboriginal languages, n.o.s.
1     Non-Official & Non-Aboriginal languages                         Afrikaans
2     Non-Official & Non-Aboriginal languages   Afro-Asiatic languages, n.i.e.
3     Non-Official & Non-Aboriginal languages                       Akan (Twi)
4     Non-Official & Non-Aboriginal languages                         Albanian
...                                      ...                               ...
1065  Non-Official & Non-Aboriginal languages                           Wolof
1066                       Aboriginal languages                     Woods Cree
1067  Non-Official & Non-Aboriginal languages                 Wu (Shanghainese)
1068  Non-Official & Non-Aboriginal languages                           Yiddish
1069  Non-Official & Non-Aboriginal languages                            Yoruba

[1070 rows x 2 columns]
```

By not putting anything before the `:`, Python reads this as "from the first column until `language`". Although the notation for selecting a range using `:` is convenient because less code is required, it must be used carefully. If you were to re-order columns or add a column to the data frame, the output would change. Using a list is more explicit and less prone to potential confusion, but sometimes involves a lot more typing.

The second special capability of `.loc[]` over `[]` is that it enables *selecting columns* using logical statements. The `[]` operator can only use logical statements to filter rows; `.loc[]` can do both. For example, let's say we wanted only to select the columns `most_at_home` and `most_at_work`. We could then use the `.str.startswith` method to choose only the columns that start with the word "most". The `str.startswith` expression returns a list of `True` or `False` values corresponding to the column names that start with the desired characters.

```
tidy_lang.loc[:, tidy_lang.columns.str.startswith("most")]
```

```
      most_at_home  most_at_work
0               50             0
1              265             0
2              185            10
3             4045            20
4             6380           215
...            ...           ...
1065            90            10
1066            20             0
1067           120             0
1068             0             0
1069           280             0
```

(continues on next page)

(continued from previous page)

```
[1070 rows x 2 columns]
```

We could also have chosen the columns containing an underscore _ by using the `.str.contains("_")`, since we notice the columns we want contain underscores and the others don't.

```
tidy_lang.loc[:, tidy_lang.columns.str.contains("_")]
```

```
      most_at_home  most_at_work
0               50             0
1              265             0
2              185            10
3             4045            20
4             6380           215
...            ...           ...
1065            90            10
1066            20             0
1067           120             0
1068             0             0
1069           280             0

[1070 rows x 2 columns]
```

3.7 Using `iloc[]` to extract rows and columns by position

Another approach for selecting rows and columns is to use `iloc[]`, which provides the ability to index with the position rather than the label of the columns. For example, the column labels of the `tidy_lang` data frame are `["category", "language", "region", "most_at_home", "most_at_work"]`. Using `iloc[]`, you can ask for the `language` column by requesting the column at index `1` (remember that Python starts counting at `0`, so the second column `"language"` has index `1`!).

```
tidy_lang.iloc[:, 1]
```

```
0         Aboriginal languages, n.o.s.
1                           Afrikaans
2       Afro-Asiatic languages, n.i.e.
3                           Akan (Twi)
4                             Albanian
                    ...
1065                            Wolof
1066                      Woods Cree
1067                Wu (Shanghainese)
1068                          Yiddish
1069                           Yoruba
Name: language, Length: 1070, dtype: object
```

You can also ask for multiple columns. We pass `1:` after the comma indicating we want columns after and including index 1 (i.e., `language`).

```
tidy_lang.iloc[:, 1:]
```

```
                            language    region  most_at_home  most_at_work
0       Aboriginal languages, n.o.s.   Toronto           50             0
1                          Afrikaans   Toronto          265             0
2     Afro-Asiatic languages, n.i.e.   Toronto          185            10
3                          Akan (Twi)   Toronto         4045            20
4                            Albanian   Toronto         6380           215
...                               ...       ...          ...           ...
1065                            Wolof  Edmonton           90            10
1066                     Woods Cree  Edmonton           20             0
1067              Wu (Shanghainese)  Edmonton          120             0
1068                         Yiddish  Edmonton            0             0
1069                          Yoruba  Edmonton          280             0

[1070 rows x 4 columns]
```

We can also use `iloc[]` to select ranges of rows, or simultaneously select ranges of rows and columns, using a similar syntax. For example, to select the first five rows and columns after and including index 1, we could use the following:

```
tidy_lang.iloc[:5, 1:]
```

```
                          language   region  most_at_home  most_at_work
0    Aboriginal languages, n.o.s.  Toronto           50             0
1                        Afrikaans  Toronto          265             0
2  Afro-Asiatic languages, n.i.e.  Toronto          185            10
3                        Akan (Twi)  Toronto         4045            20
4                          Albanian  Toronto         6380           215
```

Note that the `iloc[]` method is not commonly used, and must be used with care. For example, it is easy to accidentally put in the wrong integer index. If you did not correctly remember that the `language` column was index `1`, and used `2` instead, your code might end up having a bug that is quite hard to track down.

3.8 Aggregating data

3.8.1 Calculating summary statistics on individual columns

As a part of many data analyses, we need to calculate a summary value for the data (a *summary statistic*). Examples of summary statistics we might want to calculate are the number of observations, the average/mean value for a column, the minimum value, etc. Oftentimes, this summary statistic is

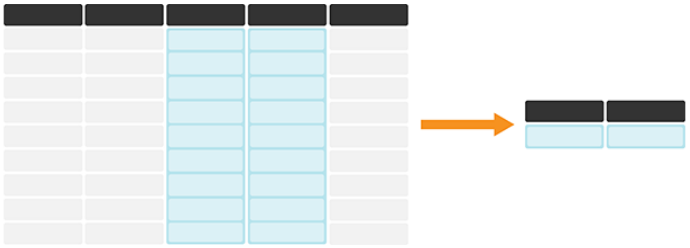


FIGURE 3.12 Calculating summary statistics on one or more column(s) in `pandas` generally creates a series or data frame containing the summary statistic(s) for each column being summarized. The darker, top row of each table represents column headers.

calculated from the values in a data frame column, or columns, as shown in Fig. 3.12.

We will start by showing how to compute the minimum and maximum number of Canadians reporting a particular language as their primary language at home. First, a reminder of what `region_lang` looks like:

```
region_lang = pd.read_csv("data/region_lang.csv")
region_lang
```

```
                  region                                  category  \
0              St. John's                     Aboriginal languages
1                 Halifax                     Aboriginal languages
2                 Moncton                     Aboriginal languages
3              Saint John                     Aboriginal languages
4                Saguenay                     Aboriginal languages
...                   ...                                      ...
7485    Ottawa - Gatineau  Non-Official & Non-Aboriginal languages
7486              Kelowna  Non-Official & Non-Aboriginal languages
7487  Abbotsford - Mission  Non-Official & Non-Aboriginal languages
7488             Vancouver  Non-Official & Non-Aboriginal languages
7489              Victoria  Non-Official & Non-Aboriginal languages

                        language  mother_tongue  most_at_home  most_at_work␣
↪ \
0     Aboriginal languages, n.o.s.             5             0             0
1     Aboriginal languages, n.o.s.             5             0             0
2     Aboriginal languages, n.o.s.             0             0             0
3     Aboriginal languages, n.o.s.             0             0             0
4     Aboriginal languages, n.o.s.             5             5             0
...                            ...           ...           ...           ...
7485                         Yoruba           265            65            10
7486                         Yoruba             5             0             0
7487                         Yoruba            20             0             0
7488                         Yoruba           190            40             0
7489                         Yoruba            20             0             0

      lang_known
0              0
1              0
2              0
```

(continues on next page)

(continued from previous page)

```
3               0
4               0
...           ...
7485          910
7486            0
7487           50
7488          505
7489           90

[7490 rows x 7 columns]
```

We use `.min` to calculate the minimum and `.max` to calculate maximum number of Canadians reporting a particular language as their primary language at home, for any region.

```
region_lang["most_at_home"].min()
```

```
0
```

```
region_lang["most_at_home"].max()
```

```
3836770
```

From this we see that there are some languages in the data set that no one speaks as their primary language at home. We also see that the most commonly spoken primary language at home is spoken by 3,836,770 people. If instead we wanted to know the total number of people in the survey, we could use the `sum` summary statistic method.

```
region_lang["most_at_home"].sum()
```

```
23171710
```

Other handy summary statistics include the `mean`, `median` and `std` for computing the mean, median, and standard deviation of observations, respectively. We can also compute multiple statistics at once using `agg` to "aggregate" results. For example, if we wanted to compute both the `min` and `max` at once, we could use `agg` with the argument `["min", "max"]`. Note that `agg` outputs a `Series` object.

```
region_lang["most_at_home"].agg(["min", "max"])
```

```
min          0
max    3836770
Name: most_at_home, dtype: int64
```

The pandas package also provides the describe method, which is a handy function that computes many common summary statistics at once; it gives us a *summary* of a variable.

```
region_lang["most_at_home"].describe()
```

```
count    7.490000e+03
mean     3.093686e+03
std      6.401258e+04
min      0.000000e+00
25%      0.000000e+00
50%      0.000000e+00
75%      3.000000e+01
max      3.836770e+06
Name: most_at_home, dtype: float64
```

In addition to the summary methods we introduced earlier, the describe method outputs a count (the total number of observations, or rows, in our data frame), as well as the 25th, 50th, and 75th percentiles. Table 3.3 provides an overview of some of the useful summary statistics that you can compute with pandas.

TABLE 3.3 Basic summary statistics

Function	Description
count	The number of observations (rows)
mean	The mean of the observations
median	The median value of the observations
std	The standard deviation of the observations
max	The largest value in a column
min	The smallest value in a column
sum	The sum of all observations
agg	Aggregate multiple statistics together
describe	A summary

Note: In pandas, the value NaN is often used to denote missing data. By default, when pandas calculates summary statistics (e.g., max, min, sum, etc.), it ignores these values. If you look at the documentation for these functions, you will see an input variable skipna, which by default is set to skipna=True. This means that pandas will skip NaN values when computing statistics.

3.8.2 Calculating summary statistics on data frames

What if you want to calculate summary statistics on an entire data frame? Well, it turns out that the functions in Table 3.3 can be applied to a whole data frame. For example, we can ask for the maximum value of each column has using `max`.

```
region_lang.max()
```

```
region                     Winnipeg
category         Official languages
language                     Yoruba
mother_tongue               3061820
most_at_home                3836770
most_at_work                3218725
lang_known                  5600480
dtype: object
```

We can see that for columns that contain string data with words like `"Vancouver"` and `"Halifax"`, the maximum value is determined by sorting the string alphabetically and returning the last value. If we only want the maximum value for numeric columns, we can provide `numeric_only=True`:

```
region_lang.max(numeric_only=True)
```

```
mother_tongue    3061820
most_at_home     3836770
most_at_work     3218725
lang_known       5600480
dtype: int64
```

We could also ask for the `mean` for each columns in the data frame. It does not make sense to compute the mean of the string columns, so in this case we *must* provide the keyword `numeric_only=True` so that the mean is only computed on columns with numeric values.

```
region_lang.mean(numeric_only=True)
```

```
mother_tongue    3200.341121
most_at_home     3093.686248
most_at_work     1853.757677
lang_known       5127.499332
dtype: float64
```

If there are only some columns for which you would like to get summary statistics, you can first use `[]` or `.loc[]` to select those columns, and then ask for the summary statistic as we did for a single column previously. For example, if we want to know the mean and standard deviation of all of the columns between `"mother_tongue"` and `"lang_known"`, we use `.loc[]` to select those columns and then `agg` to ask for both the `mean` and `std`.

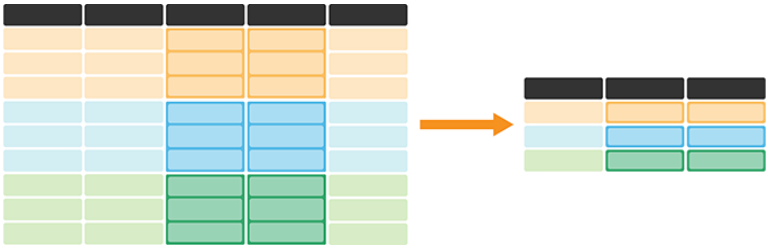


FIGURE 3.13 A summary statistic function paired with `groupby` is useful for calculating that statistic on one or more column(s) for each group. It creates a new data frame with one row for each group and one column for each summary statistic. The darker, top row of each table represents the column headers. The orange, blue, and green colored rows correspond to the rows that belong to each of the three groups being represented in this cartoon example.

```
region_lang.loc[:, "mother_tongue":"lang_known"].agg(["mean", "std"])
```

```
      mother_tongue  most_at_home  most_at_work    lang_known
mean    3200.341121   3093.686248   1853.757677   5127.499332
std    55231.640268  64012.578320  48574.532066  94001.162338
```

3.9 Performing operations on groups of rows using `groupby`

What happens if we want to know how languages vary by region? In this case, we need a new tool that lets us group rows by region. This can be achieved using the `groupby` function in `pandas`. Pairing summary functions with `groupby` lets you summarize values for subgroups within a data set, as illustrated in Fig. 3.13. For example, we can use `groupby` to group the regions of the `tidy_lang` data frame and then calculate the minimum and maximum number of Canadians reporting the language as the primary language at home for each of the regions in the data set.

The `groupby` function takes at least one argument—the columns to use in the grouping. Here we use only one column for grouping (`region`).

```
region_lang.groupby("region")
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fe56d8f9950>
```

Notice that `groupby` converts a `DataFrame` object to a `DataFrameGroupBy` object, which contains information about the groups of the data frame. We

can then apply aggregating functions to the `DataFrameGroupBy` object. Here we first select the `most_at_home` column, and then summarize the grouped data by their minimum and maximum values using `agg`.

```
region_lang.groupby("region")["most_at_home"].agg(["min", "max"])
```

```
                        min      max
region
Abbotsford - Mission      0   137445
Barrie                    0   182390
Belleville                0    97840
Brantford                 0   124560
Calgary                   0  1065070
...                     ...      ...
Trois-Rivières            0   149835
Vancouver                 0  1622735
Victoria                  0   331375
Windsor                   0   270715
Winnipeg                  0   612595

[35 rows x 2 columns]
```

The resulting data frame has `region` as an index name. This is similar to what happened when we used the `pivot` function in Section 3.4.2; and just as we did then, you can use `reset_index` to get back to a regular data frame with `region` as a column name.

```
region_lang.groupby("region")["most_at_home"].agg(["min", "max"]).reset_index()
```

```
                   region  min      max
0   Abbotsford - Mission    0   137445
1                 Barrie    0   182390
2             Belleville    0    97840
3              Brantford    0   124560
4                Calgary    0  1065070
..                   ...  ...      ...
30       Trois-Rivières    0   149835
31            Vancouver    0  1622735
32             Victoria    0   331375
33              Windsor    0   270715
34             Winnipeg    0   612595

[35 rows x 3 columns]
```

You can also pass multiple column names to `groupby`. For example, if we wanted to know about how the different categories of languages (Aboriginal, Non-Official & Non-Aboriginal, and Official) are spoken at home in different regions, we would pass a list including `region` and `category` to `groupby`.

```
region_lang.groupby(["region", "category"])["most_at_home"].agg(["min", "max"]).
↪reset_index()
```

```
                  region                                  category    min  \
0    Abbotsford - Mission                       Aboriginal languages      0
1    Abbotsford - Mission  Non-Official & Non-Aboriginal languages      0
2    Abbotsford - Mission                         Official languages    250
3                  Barrie                       Aboriginal languages      0
4                  Barrie  Non-Official & Non-Aboriginal languages      0
..                    ...                                       ...    ...
100              Windsor  Non-Official & Non-Aboriginal languages      0
101              Windsor                         Official languages   2695
102             Winnipeg                       Aboriginal languages      0
103             Winnipeg  Non-Official & Non-Aboriginal languages      0
104             Winnipeg                         Official languages  11185

        max
0         5
1     23015
2    137445
3         0
4       875
..      ...
100    8235
101  270715
102     365
103   23565
104  612595

[105 rows x 4 columns]
```

You can also ask for grouped summary statistics on the whole data frame.

```
region_lang.groupby("region").agg(["min", "max"]).reset_index()
```

```
                  region              category                     \
                                         min                  max
0   Abbotsford - Mission  Aboriginal languages  Official languages
1                 Barrie  Aboriginal languages  Official languages
2             Belleville  Aboriginal languages  Official languages
3              Brantford  Aboriginal languages  Official languages
4                Calgary  Aboriginal languages  Official languages
..                   ...                   ...                 ...
30       Trois-Rivières  Aboriginal languages  Official languages
31            Vancouver  Aboriginal languages  Official languages
32             Victoria  Aboriginal languages  Official languages
33              Windsor  Aboriginal languages  Official languages
34             Winnipeg  Aboriginal languages  Official languages

                        language         mother_tongue          most_at_home ␣
↪\
                             min     max           min      max          min
0   Aboriginal languages, n.o.s.  Yoruba             0   122100            0
1   Aboriginal languages, n.o.s.  Yoruba             0   168990            0
2   Aboriginal languages, n.o.s.  Yoruba             0    93655            0
3   Aboriginal languages, n.o.s.  Yoruba             0   116645            0
4   Aboriginal languages, n.o.s.  Yoruba             0   937055            0
..                           ...     ...           ...      ...          ...
30  Aboriginal languages, n.o.s.  Yoruba             0   147805            0
31  Aboriginal languages, n.o.s.  Yoruba             0  1316635            0
32  Aboriginal languages, n.o.s.  Yoruba             0   302690            0
33  Aboriginal languages, n.o.s.  Yoruba             0   235990            0
34  Aboriginal languages, n.o.s.  Yoruba             0   530570            0
```

(continues on next page)

(continued from previous page)

```
            most_at_work          lang_known
        max          min      max         min      max
0    137445            0    93495           0   167835
1    182390            0   115125           0   193445
2     97840            0    54150           0   100855
3    124560            0    73910           0   130835
4   1065070            0   844740           0  1343335
..      ...          ...      ...         ...      ...
30   149835            0    78610           0   149805
31  1622735            0  1330555           0  2289515
32   331375            0   211705           0   354470
33   270715            0   166220           0   318540
34   612595            0   437460           0   749285

[35 rows x 13 columns]
```

If you want to ask for only some columns, for example, the columns between `"most_at_home"` and `"lang_known"`, you might think about first applying `groupby` and then `["most_at_home":"lang_known"]`; but `groupby` returns a `DataFrameGroupBy` object, which does not work with ranges inside `[]`. The other option is to do things the other way around: first use `["most_at_home":"lang_known"]`, then use `groupby`. This can work, but you have to be careful! For example, in our case, we get an error.

```
region_lang["most_at_home":"lang_known"].groupby("region").max()
```

```
KeyError: "region"
```

This is because when we use `[]` we selected only the columns between `"most_at_home"` and `"lang_known"`, which doesn't include `"region"`. Instead, we need to use `groupby` first and then call `[]` with a list of column names that includes `region`; this approach always works.

```
region_lang.groupby("region")[["most_at_home", "most_at_work", "lang_known"]].
↪max().reset_index()
```

```
                region  most_at_home  most_at_work  lang_known
0   Abbotsford - Mission        137445         93495      167835
1                Barrie        182390        115125      193445
2            Belleville         97840         54150      100855
3             Brantford        124560         73910      130835
4               Calgary       1065070        844740     1343335
..                  ...           ...           ...         ...
30      Trois-Rivières         149835         78610      149805
31           Vancouver        1622735       1330555     2289515
32            Victoria         331375        211705      354470
33             Windsor         270715        166220      318540
34            Winnipeg         612595        437460      749285

[35 rows x 4 columns]
```

To see how many observations there are in each group, we can use `value_counts`.

```
region_lang.value_counts("region")
```

```
region
Abbotsford - Mission                214
St. Catharines - Niagara            214
Québec                              214
Regina                              214
Saguenay                            214
                                   ...
Kitchener - Cambridge - Waterloo    214
Lethbridge                          214
London                              214
Moncton                             214
Winnipeg                            214
Name: count, Length: 35, dtype: int64
```

Which takes the `normalize` parameter to show the output as proportion instead of a count.

```
region_lang.value_counts("region", normalize=True)
```

```
region
Abbotsford - Mission                0.028571
St. Catharines - Niagara            0.028571
Québec                              0.028571
Regina                              0.028571
Saguenay                            0.028571
                                      ...
Kitchener - Cambridge - Waterloo    0.028571
Lethbridge                          0.028571
London                              0.028571
Moncton                             0.028571
Winnipeg                            0.028571
Name: proportion, Length: 35, dtype: float64
```

3.10 Apply functions across multiple columns

Computing summary statistics is not the only situation in which we need to apply a function across columns in a data frame. There are two other common wrangling tasks that require the application of a function across columns. The first is when we want to apply a transformation, such as a conversion of measurement units, to multiple columns. We illustrate such a data transformation in Fig. 3.14; note that it does not change the shape of the data frame.

For example, imagine that we wanted to convert all the numeric columns in the `region_lang` data frame from `int64` type to `int32` type using the `.`

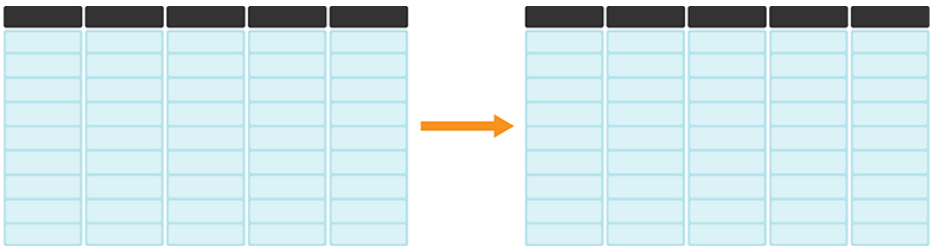


FIGURE 3.14 A transformation applied across many columns. The darker, top row of each table represents the column headers.

`astype` function. When we revisit the `region_lang` data frame, we can see that this would be the columns from `mother_tongue` to `lang_known`.

```
region_lang
```

```
                   region                                  category  \
0             St. John's                       Aboriginal languages
1                Halifax                       Aboriginal languages
2                Moncton                       Aboriginal languages
3             Saint John                       Aboriginal languages
4               Saguenay                       Aboriginal languages
...                  ...                                       ...
7485     Ottawa - Gatineau  Non-Official & Non-Aboriginal languages
7486               Kelowna  Non-Official & Non-Aboriginal languages
7487  Abbotsford - Mission  Non-Official & Non-Aboriginal languages
7488             Vancouver  Non-Official & Non-Aboriginal languages
7489              Victoria  Non-Official & Non-Aboriginal languages

                         language  mother_tongue  most_at_home  most_at_work␣
↪ \
0     Aboriginal languages, n.o.s.              5             0             0
1     Aboriginal languages, n.o.s.              5             0             0
2     Aboriginal languages, n.o.s.              0             0             0
3     Aboriginal languages, n.o.s.              0             0             0
4     Aboriginal languages, n.o.s.              5             5             0
...                            ...            ...           ...           ...
7485                         Yoruba            265            65            10
7486                         Yoruba              5             0             0
7487                         Yoruba             20             0             0
7488                         Yoruba            190            40             0
7489                         Yoruba             20             0             0

      lang_known
0              0
1              0
2              0
3              0
4              0
...          ...
7485         910
7486           0
7487          50
7488         505
7489          90

[7490 rows x 7 columns]
```

We can simply call the `.astype` function to apply it across the desired range of columns.

```
region_lang_nums = region_lang.loc[:, "mother_tongue":"lang_known"].astype("int32
↪")
region_lang_nums.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7490 entries, 0 to 7489
Data columns (total 4 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   mother_tongue  7490 non-null   int32
 1   most_at_home   7490 non-null   int32
 2   most_at_work   7490 non-null   int32
 3   lang_known     7490 non-null   int32
dtypes: int32(4)
memory usage: 117.2 KB
```

You can now see that the columns from `mother_tongue` to `lang_known` are type `int32`, and that we have obtained a data frame with the same number of columns and rows as the input data frame.

The second situation occurs when you want to apply a function across columns within each individual row, i.e., *row-wise*. This operation, illustrated in Fig. 3.15, will produce a single column whose entries summarize each row in the original data frame; this new column can be added back into the original data.

FIGURE 3.15 A function applied row-wise across a data frame, producing a new column. The darker, top row of each table represents the column headers.

For example, suppose we want to know the maximum value between `mother_tongue`, and `lang_known` for each language and region in the `region_lang_nums` data set. In other words, we want to apply the `max` function *row-wise*. In order to tell `max` that we want to work row-wise (as opposed to acting on each column individually, which is the default behavior), we just specify the argument `axis=1`.

```
region_lang_nums.max(axis=1)
```

```
0       5
1       5
2       0
3       0
```

(continues on next page)

(continued from previous page)

```
4         5
        ...
7485    910
7486      5
7487     50
7488    505
7489     90
Length: 7490, dtype: int32
```

We see that we obtain a series containing the maximum value between `mother_tongue`, `most_at_home`, `most_at_work` and `lang_known` for each row in the data frame. It is often the case that we want to include a column result from a row-wise operation as a new column in the data frame, so that we can make plots or continue our analysis. To make this happen, we will use column assignment or the `assign` function to create a new column. This is discussed in the next section.

Note: While `pandas` provides many methods (like `max`, `astype`, etc.) that can be applied to a data frame, sometimes you may want to apply your own function to multiple columns in a data frame. In this case you can use the more general `apply`[1] method.

3.11 Modifying and adding columns

When we compute summary statistics or apply functions, a new data frame or series is created. But what if we want to append that information to an existing data frame? For example, say we wanted to compute the maximum value in each row of the `region_lang_nums` data frame, and to append that as an additional column of the `region_lang` data frame. In this case, we have two options: we can either create a new column within the `region_lang` data frame itself, or create an entirely new data frame with the `assign` method. The first option we have seen already in earlier chapters, and is the more commonly used pattern in practice:

```
region_lang["maximum"] = region_lang_nums.max(axis=1)
region_lang
```

```
                region                                category  \
0          St. John's                 Aboriginal languages
```

(continues on next page)

[1] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.apply.html

(continued from previous page)

```
1                  Halifax                     Aboriginal languages
2                  Moncton                     Aboriginal languages
3               Saint John                     Aboriginal languages
4                 Saguenay                     Aboriginal languages
...                    ...                                      ...
7485     Ottawa - Gatineau  Non-Official & Non-Aboriginal languages
7486               Kelowna  Non-Official & Non-Aboriginal languages
7487  Abbotsford - Mission  Non-Official & Non-Aboriginal languages
7488             Vancouver  Non-Official & Non-Aboriginal languages
7489              Victoria  Non-Official & Non-Aboriginal languages

                          language  mother_tongue  most_at_home  most_at_work␣
↪ \
0     Aboriginal languages, n.o.s.              5             0             0
1     Aboriginal languages, n.o.s.              5             0             0
2     Aboriginal languages, n.o.s.              0             0             0
3     Aboriginal languages, n.o.s.              0             0             0
4     Aboriginal languages, n.o.s.              5             5             0
...                              ...           ...           ...           ...
7485                         Yoruba            265            65            10
7486                         Yoruba              5             0             0
7487                         Yoruba             20             0             0
7488                         Yoruba            190            40             0
7489                         Yoruba             20             0             0

      lang_known  maximum
0              0        5
1              0        5
2              0        0
3              0        0
4              0        5
...          ...      ...
7485         910      910
7486           0        5
7487          50       50
7488         505      505
7489          90       90

[7490 rows x 8 columns]
```

You can see above that the `region_lang` data frame now has an additional column named `maximum`. The `maximum` column contains the maximum value between `mother_tongue`, `most_at_home`, `most_at_work` and `lang_known` for each language and region, just as we specified.

To instead create an entirely new data frame, we can use the `assign` method and specify one argument for each column we want to create. In this case we want to create one new column named `maximum`, so the argument to `assign` begins with `maximum=` . Then after the `=`, we specify what the contents of that new column should be. In this case we use `max` just as we did previously to give us the maximum values. Remember to specify `axis=1` in the `max` method so that we compute the row-wise maximum value.

```
region_lang.assign(
```

(continues on next page)

(continued from previous page)

```
  maximum=region_lang_nums.max(axis=1)
)
```

```
                  region                                  category  \
0              St. John's                     Aboriginal languages
1                 Halifax                     Aboriginal languages
2                 Moncton                     Aboriginal languages
3              Saint John                     Aboriginal languages
4                Saguenay                     Aboriginal languages
...                   ...                                     ...
7485  Ottawa - Gatineau  Non-Official & Non-Aboriginal languages
7486            Kelowna  Non-Official & Non-Aboriginal languages
7487  Abbotsford - Mission  Non-Official & Non-Aboriginal languages
7488          Vancouver  Non-Official & Non-Aboriginal languages
7489           Victoria  Non-Official & Non-Aboriginal languages

                        language  mother_tongue  most_at_home  most_at_work␣
↪ \
0     Aboriginal languages, n.o.s.              5             0             0
1     Aboriginal languages, n.o.s.              5             0             0
2     Aboriginal languages, n.o.s.              0             0             0
3     Aboriginal languages, n.o.s.              0             0             0
4     Aboriginal languages, n.o.s.              5             5             0
...                            ...            ...           ...           ...
7485                         Yoruba            265            65            10
7486                         Yoruba              5             0             0
7487                         Yoruba             20             0             0
7488                         Yoruba            190            40             0
7489                         Yoruba             20             0             0

      lang_known  maximum
0              0        5
1              0        5
2              0        0
3              0        0
4              0        5
...          ...      ...
7485         910      910
7486           0        5
7487          50       50
7488         505      505
7489          90       90

[7490 rows x 8 columns]
```

This data frame looks just like the previous one, except that it is a copy of `region_lang`, not `region_lang` itself; making further changes to this data frame will not impact the original `region_lang` data frame.

As another example, we might ask the question: "What proportion of the population reported English as their primary language at home in the 2016 census?" For example, in Toronto, 3,836,770 people reported speaking English as their primary language at home, and the population of Toronto was reported to be 5,928,040 people. So the proportion of people reporting English as their primary language in Toronto in the 2016 census was 0.65. How could we figure this out starting from the `region_lang` data frame?

First, we need to filter the `region_lang` data frame so that we only keep the rows where the language is English. We will also restrict our attention to the five major cities in the `five_cities` data frame: Toronto, Montréal, Vancouver, Calgary, and Edmonton. We will filter to keep only those rows pertaining to the English language and pertaining to the five aforementioned cities. To combine these two logical statements we will use the `&` symbol. and with the `[]` operation, `"English"` as the `language` and filter the rows, and name the new data frame `english_langs`.

```
english_lang = region_lang[
    (region_lang["language"] == "English") &
    (region_lang["region"].isin(five_cities["region"]))
]
english_lang
```

```
         region            category language  mother_tongue  most_at_home  \
1898   Montréal  Official languages  English         444955        620510
1903    Toronto  Official languages  English        3061820       3836770
1918    Calgary  Official languages  English         937055       1065070
1919   Edmonton  Official languages  English         930405       1050410
1923  Vancouver  Official languages  English        1316635       1622735

      most_at_work  lang_known
1898        412120     2500590
1903       3218725     5600480
1918        844740     1343335
1919        792700     1275265
1923       1330555     2289515
```

Okay, now we have a data frame that pertains only to the English language and the five cities mentioned earlier. In order to compute the proportion of the population speaking English in each of these cities, we need to add the population data from the `five_cities` data frame.

```
five_cities
```

```
       region  households        area  population  dwellings
29   Edmonton      502143  9857.77908     1321426     537634
31    Calgary      519693  5241.70103     1392609     544870
32  Vancouver      960894  3040.41532     2463431    1027613
33   Montréal     1727310  4638.24059     4098927    1823281
34    Toronto     2135909  6269.93132     5928040    2235145
```

The data frame above shows that the populations of the five cities in 2016 were 5928040 (Toronto), 4098927 (Montréal), 2463431 (Vancouver), 1392609 (Calgary), and 1321426 (Edmonton). Next, we will add this information to a new data frame column called `city_pops`. Once again, we will illustrate how to do this using both the `assign` method and regular column assignment. We specify the new column name (`city_pops`) as the argument, followed by the equals symbol =, and finally the data in the column. Note that the order of the rows in the `english_lang` data frame is Montréal, Toronto, Calgary,

Edmonton, Vancouver. So we will create a column called `city_pops` where we list the populations of those cities in that order, and add it to our data frame. And remember that by default, like other `pandas` functions, `assign` does not modify the original data frame directly, so the `english_lang` data frame is unchanged.

```
english_lang.assign(
  city_pops=[4098927, 5928040, 1392609, 1321426, 2463431]
)
```

```
         region            category language  mother_tongue  most_at_home  \
1898   Montréal  Official languages  English         444955        620510
1903    Toronto  Official languages  English        3061820       3836770
1918    Calgary  Official languages  English         937055       1065070
1919   Edmonton  Official languages  English         930405       1050410
1923  Vancouver  Official languages  English        1316635       1622735

      most_at_work  lang_known  city_pops
1898        412120     2500590    4098927
1903       3218725     5600480    5928040
1918        844740     1343335    1392609
1919        792700     1275265    1321426
1923       1330555     2289515    2463431
```

Instead of using the `assign` method we can directly modify the `english_lang` data frame using regular column assignment. This would be a more natural choice in this particular case, since the syntax is more convenient for simple column modifications and additions.

```
english_lang["city_pops"] = [4098927, 5928040, 1392609, 1321426, 2463431]
english_lang
```

```
/tmp/ipykernel_12/2654974267.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/
↪stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  english_lang["city_pops"] = [4098927, 5928040, 1392609, 1321426, 2463431]
```

```
         region            category language  mother_tongue  most_at_home  \
1898   Montréal  Official languages  English         444955        620510
1903    Toronto  Official languages  English        3061820       3836770
1918    Calgary  Official languages  English         937055       1065070
1919   Edmonton  Official languages  English         930405       1050410
1923  Vancouver  Official languages  English        1316635       1622735

      most_at_work  lang_known  city_pops
1898        412120     2500590    4098927
1903       3218725     5600480    5928040
1918        844740     1343335    1392609
1919        792700     1275265    1321426
1923       1330555     2289515    2463431
```

Wait a moment … what is that warning message? It seems to suggest that something went wrong, but if we inspect the `english_lang` data frame above,

it looks like the city populations were added just fine. As it turns out, this is caused by the earlier filtering we did from `region_lang` to produce the original `english_lang`. The details are a little bit technical, but `pandas` sometimes does not like it when you subset a data frame using `[]` or `loc[]` followed by column assignment. For the purposes of your own data analysis, if you ever see a `SettingWithCopyWarning`, just make sure to double check that the result of your column assignment looks the way you expect it to before proceeding. For the rest of the book, we will silence that warning to help with readability.

Note: Inserting the data column `[4098927, 5928040, ...]` manually as we did above is generally very error-prone and is not recommended. We do it here to demonstrate another usage of `assign` and regular column assignment. But in more advanced data wrangling, one would solve this problem in a less error-prone way using the `merge` function, which lets you combine two data frames. We will show you an example using `merge` at the end of the chapter.

Now we have a new column with the population for each city. Finally, we can convert all the numerical columns to proportions of people who speak English by taking the ratio of all the numerical columns with `city_pops`. Let's modify the `english_lang` column directly; in this case we can just assign directly to the data frame. This is similar to what we did in Section 3.4.3, when we first read in the `"region_lang_top5_cities_messy.csv"` data and we needed to convert a few of the variables to numeric types. Here we assign to a range of columns simultaneously using `loc[]`. Note that it is again possible to instead use the `assign` function to produce a new data frame when modifying existing columns, although this is not commonly done. Note also that we use the `div` method with the argument `axis=0` to divide a range of columns in a data frame by the values in a single column—the basic division symbol `/` won't work in this case.

```
english_lang.loc[:, "mother_tongue":"lang_known"] = english_lang.loc[
    :,
    "mother_tongue":"lang_known"
    ].div(english_lang["city_pops"], axis=0)
english_lang
```

```
         region            category language  mother_tongue  most_at_home  \
1898   Montréal  Official languages  English       0.108554      0.151384
1903    Toronto  Official languages  English       0.516498      0.647224
1918    Calgary  Official languages  English       0.672877      0.764802
1919   Edmonton  Official languages  English       0.704092      0.794906
1923  Vancouver  Official languages  English       0.534472      0.658730
```

(continues on next page)

(continued from previous page)

```
      most_at_work  lang_known  city_pops
1898      0.100543    0.610060    4098927
1903      0.542966    0.944744    5928040
1918      0.606588    0.964617    1392609
1919      0.599882    0.965067    1321426
1923      0.540123    0.929401    2463431
```

3.12 Using `merge` to combine data frames

Let's return to the situation right before we added the city populations of Toronto, Montréal, Vancouver, Calgary, and Edmonton to the `english_lang` data frame. Before adding the new column, we had filtered `region_lang` to create the `english_lang` data frame containing only English speakers in the five cities of interest.

```
english_lang
```

```
         region            category language  mother_tongue  most_at_home  \
1898   Montréal  Official languages  English         444955        620510
1903    Toronto  Official languages  English        3061820       3836770
1918    Calgary  Official languages  English         937055       1065070
1919   Edmonton  Official languages  English         930405       1050410
1923  Vancouver  Official languages  English        1316635       1622735

      most_at_work  lang_known
1898        412120     2500590
1903       3218725     5600480
1918        844740     1343335
1919        792700     1275265
1923       1330555     2289515
```

We then added the populations of these cities as a column (Toronto: 5928040, Montréal: 4098927, Vancouver: 2463431, Calgary: 1392609, and Edmonton: 1321426). We had to be careful to add those populations in the right order; this is an error-prone process. An alternative approach, that we demonstrate here is to (1) create a new data frame with the city names and populations, and (2) use `merge` to combine the two data frames, recognizing that the "regions" are the same.

We create a new data frame by calling `pd.DataFrame` with a dictionary as its argument. The dictionary associates each column name in the data frame to be created with a list of entries. Here we list city names in a column called `"region"` and their populations in a column called `"population"`.

```
city_populations = pd.DataFrame({
```

(continues on next page)

(continued from previous page)

```
  "region" : ["Toronto", "Montréal", "Vancouver", "Calgary", "Edmonton"],
  "population" : [5928040, 4098927, 2463431, 1392609, 1321426]
})
city_populations
```

```
      region  population
0    Toronto     5928040
1   Montréal     4098927
2  Vancouver     2463431
3    Calgary     1392609
4   Edmonton     1321426
```

This new data frame has the same `region` column as the `english_lang` data frame. The order of the cities is different, but that is okay. We can use the `merge` function in `pandas` to say we would like to combine the two data frames by matching the `region` between them. The argument `on="region"` tells pandas we would like to use the `region` column to match up the entries.

```
english_lang = english_lang.merge(city_populations, on="region")
english_lang
```

```
      region            category language  mother_tongue  most_at_home  \
0   Montréal  Official languages  English         444955        620510
1    Toronto  Official languages  English        3061820       3836770
2    Calgary  Official languages  English         937055       1065070
3   Edmonton  Official languages  English         930405       1050410
4  Vancouver  Official languages  English        1316635       1622735

   most_at_work  lang_known  population
0        412120     2500590     4098927
1       3218725     5600480     5928040
2        844740     1343335     1392609
3        792700     1275265     1321426
4       1330555     2289515     2463431
```

You can see that the populations for each city are correct (e.g. Montréal: 4098927, Toronto: 5928040), and we can proceed to with our analysis from here.

3.13 Summary

Cleaning and wrangling data can be a very time-consuming process. However, it is a critical step in any data analysis. We have explored many different functions for cleaning and wrangling data into a tidy format. Table 3.4 summarizes some of the key wrangling functions we learned in this chapter. In the following chapters, you will learn how you can take this tidy data and do so much more with it to answer your burning data science questions.

TABLE 3.4 Summary of wrangling functions

Function	Description
`agg`	calculates aggregated summaries of inputs
`assign`	adds or modifies columns in a data frame
`groupby`	allows you to apply function(s) to groups of rows
`iloc`	subsets columns/rows of a data frame using integer indices
`loc`	subsets columns/rows of a data frame using labels
`melt`	generally makes the data frame longer and narrower
`merge`	combine two data frames
`pivot`	generally makes a data frame wider and decreases the number of rows
`str.split`	splits up a string column into multiple columns

3.14 Exercises

Practice exercises for the material covered in this chapter can be found in the accompanying worksheets repository[2] in the "Cleaning and wrangling data" row. You can launch an interactive version of the worksheet in your browser by clicking the "launch binder" button. You can also preview a non-interactive version of the worksheet by clicking "view worksheet". If you instead decide to download the worksheet and run it on your own machine, make sure to follow the instructions for computer setup found in Chapter 13. This will ensure that the automated feedback and guidance that the worksheets provide will function as intended.

3.15 Additional resources

- The `pandas` package documentation[3] is another resource to learn more about the functions in this chapter, the full set of arguments you can use, and other related functions.
- *Python for Data Analysis*[4] [McKinney, 2012] has a few chapters related to

[2] https://worksheets.python.datasciencebook.ca
[3] https://pandas.pydata.org/docs/reference/index.html
[4] https://wesmckinney.com/book/

data wrangling that go into more depth than this book. For example, the data wrangling chapter[5] covers tidy data, `melt` and `pivot`, but also covers missing values and additional wrangling functions (like `stack`). The data aggregation chapter[6] covers `groupby`, aggregating functions, `apply`, etc.

- You will occasionally encounter a case where you need to iterate over items in a data frame, but none of the above functions are flexible enough to do what you want. In that case, you may consider using a for loop[7] [McKinney, 2012].

[5] https://wesmckinney.com/book/data-wrangling.html
[6] https://wesmckinney.com/book/data-aggregation.html
[7] https://wesmckinney.com/book/python-basics.html#control_for

4

Effective data visualization

4.1 Overview

This chapter will introduce concepts and tools relating to data visualization beyond what we have seen and practiced so far. We will focus on guiding principles for effective data visualization and explaining visualizations independent of any particular tool or programming language. In the process, we will cover some specifics of creating visualizations (scatter plots, bar plots, line plots, and histograms) for data using Python.

4.2 Chapter learning objectives

By the end of the chapter, readers will be able to do the following:

- Describe when to use the following kinds of visualizations to answer specific questions using a data set:
  - scatter plots
  - line plots
  - bar plots
  - histogram plots
- Given a data set and a question, select from the above plot types and use Python to create a visualization that best answers the question.
- Evaluate the effectiveness of a visualization and suggest improvements to better answer a given question.
- Referring to the visualization, communicate the conclusions in non-technical terms.
- Identify rules of thumb for creating effective visualizations.

DOI: 10.1201/9781003438397-4

- Use the `altair` library in Python to create and refine the above visualizations using:
  - graphical marks: `mark_point`, `mark_line`, `mark_circle`, `mark_bar`, `mark_rule`
  - encoding channels: `x`, `y`, `color`, `shape`
  - labeling: `title`
  - transformations: `scale`
  - subplots: `facet`
- Define the two key aspects of `altair` charts:
  - graphical marks
  - encoding channels
- Describe the difference in raster and vector output formats.
- Use `chart.save()` to save visualizations in `.png` and `.svg` format.

4.3 Choosing the visualization

Ask a question, and answer it

The purpose of a visualization is to answer a question about a data set of interest. So naturally, the first thing to do **before** creating a visualization is to formulate the question about the data you are trying to answer. A good visualization will clearly answer your question without distraction; a *great* visualization will suggest even what the question was itself without additional explanation. Imagine your visualization as part of a poster presentation for a project; even if you aren't standing at the poster explaining things, an effective visualization will convey your message to the audience.

Recall the different data analysis questions from Chapter 1. With the visualizations we will cover in this chapter, we will be able to answer *only descriptive and exploratory* questions. Be careful to not answer any *predictive, inferential, causal or mechanistic* questions with the visualizations presented here, as we have not learned the tools necessary to do that properly just yet.

As with most coding tasks, it is totally fine (and quite common) to make mistakes and iterate a few times before you find the right visualization for your data and question. There are many different kinds of plotting graphics

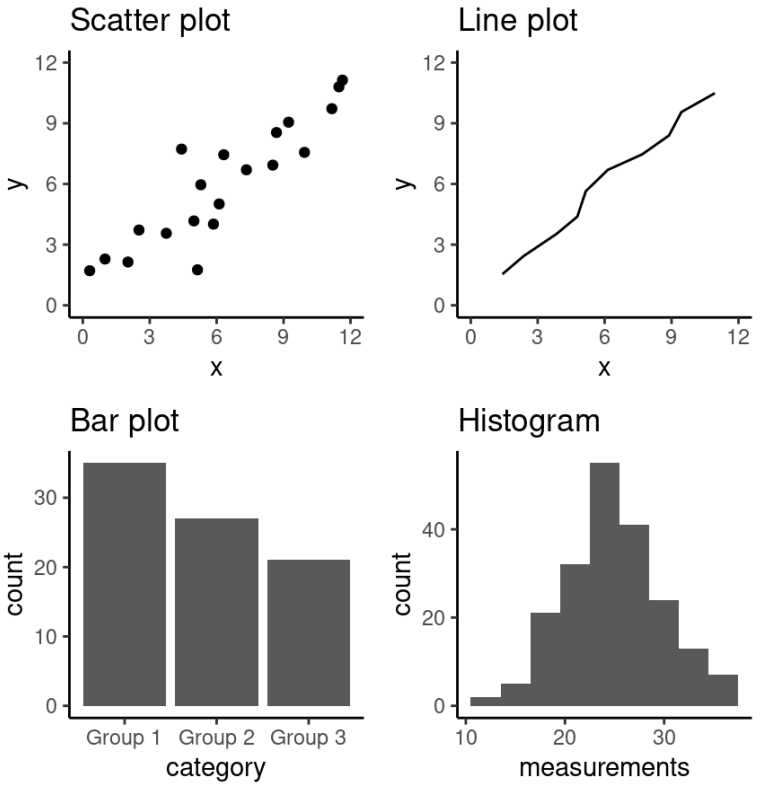


FIGURE 4.1 Examples of scatter, line and bar plots, as well as histograms.

available to use (see Chapter 5 of *Fundamentals of Data Visualization* [Wilke, 2019] for a directory). The types of plots that we introduce in this book are shown in Fig. 4.1; which one you should select depends on your data and the question you want to answer. In general, the guiding principles of when to use each type of plot are as follows:

- **scatter plots** visualize the relationship between two quantitative variables
- **line plots** visualize trends with respect to an independent, ordered quantity (e.g., time)
- **bar plots** visualize comparisons of amounts
- **histograms** visualize the distribution of one quantitative variable (i.e., all its possible values and how often they occur)

All types of visualization have their (mis)uses, but three kinds are usually hard to understand or are easily replaced with an oft-better alternative. In particular, you should avoid **pie charts**; it is generally better to use bars, as it is easier to compare bar heights than pie slice sizes. You should also not use **3D visualizations**, as they are typically hard to understand when converted

to a static 2D image format. Finally, do not use tables to make numerical comparisons; humans are much better at quickly processing visual information than text and math. Bar plots are again typically a better alternative.

4.4 Refining the visualization

Convey the message, minimize noise

Just being able to make a visualization in Python with `altair` (or any other tool for that matter) doesn't mean that it effectively communicates your message to others. Once you have selected a broad type of visualization to use, you will have to refine it to suit your particular need. Some rules of thumb for doing this are listed below. They generally fall into two classes: you want to *make your visualization convey your message*, and you want to *reduce visual noise* as much as possible. Humans have limited cognitive ability to process information; both of these types of refinement aim to reduce the mental load on your audience when viewing your visualization, making it easier for them to understand and remember your message quickly.

Convey the message

- Make sure the visualization answers the question you have asked most simply and plainly as possible.
- Use legends and labels so that your visualization is understandable without reading the surrounding text.
- Ensure the text, symbols, lines, etc., on your visualization are big enough to be easily read.
- Ensure the data are clearly visible; don't hide the shape/distribution of the data behind other objects (e.g., a bar).
- Make sure to use color schemes that are understandable by those with colorblindness (a surprisingly large fraction of the overall population—from about 1% to 10%, depending on sex and ancestry [Deeb, 2005]). For example, Color Schemes[1] provides the ability to pick such color schemes, and you can check your visualizations after you have created them by uploading to online tools such as a color blindness simulator[2].

[1] https://altair-viz.github.io/user_guide/customization.html#customizing-colors
[2] https://www.color-blindness.com/coblis-color-blindness-simulator/

- Redundancy can be helpful; sometimes conveying the same message in multiple ways reinforces it for the audience.

Minimize noise

- Use colors sparingly. Too many different colors can be distracting, create false patterns, and detract from the message.
- Be wary of overplotting. Overplotting is when marks that represent the data overlap, and is problematic as it prevents you from seeing how many data points are represented in areas of the visualization where this occurs. If your plot has too many dots or lines and starts to look like a mess, you need to do something different.
- Only make the plot area (where the dots, lines, bars are) as big as needed. Simple plots can be made small.
- Don't adjust the axes to zoom in on small differences. If the difference is small, show that it's small.

4.5 Creating visualizations with `altair`

Build the visualization iteratively

This section will cover examples of how to choose and refine a visualization given a data set and a question that you want to answer, and then how to create the visualization in Python using `altair`. To use the `altair` package, we need to first import it. We will also import `pandas` to use for reading in the data.

```
import pandas as pd
import altair as alt
```

Note: In this chapter, we will provide example visualizations using relatively small data sets, so we are fine using the default settings in `altair`. However, `altair` will raise an error if you try to plot with a data frame that has more than 5,000 rows. The simplest way to plot larger data sets is to enable the `vegafusion` data transformer right after you import the `altair` package: `alt.data_transformers.enable("vegafusion")`. This will allow you to

plot up to 100,000 graphical objects (e.g., a scatter plot with 100,000 points). To visualize *even larger* data sets, see the `altair` documentation[3].

4.5.1 Scatter plots and line plots: the Mauna Loa CO_2 data set

The Mauna Loa CO_2 data set[4], curated by Dr. Pieter Tans, NOAA/GML and Dr. Ralph Keeling, Scripps Institution of Oceanography, records the atmospheric concentration of carbon dioxide (CO_2, in parts per million) at the Mauna Loa research station in Hawaii from 1959 onward [Tans and Keeling, 2020]. For this book, we are going to focus on the years 1980–2020.

Question: Does the concentration of atmospheric CO_2 change over time, and are there any interesting patterns to note?

To get started, we will read and inspect the data:

```
# mauna loa carbon dioxide data
co2_df = pd.read_csv(
    "data/mauna_loa_data.csv",
    parse_dates=["date_measured"]
)
co2_df
```

```
    date_measured     ppm
0      1980-02-01  338.34
1      1980-03-01  340.01
2      1980-04-01  340.93
3      1980-05-01  341.48
4      1980-06-01  341.33
..            ...     ...
479    2020-02-01  414.11
480    2020-03-01  414.51
481    2020-04-01  416.21
482    2020-05-01  417.07
483    2020-06-01  416.39

[484 rows x 2 columns]
```

```
co2_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 484 entries, 0 to 483
Data columns (total 2 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   date_measured  484 non-null    datetime64[ns]
 1   ppm            484 non-null    float64
dtypes: datetime64[ns](1), float64(1)
memory usage: 7.7 KB
```

[3] https://altair-viz.github.io/user_guide/large_datasets
[4] https://www.esrl.noaa.gov/gmd/ccgg/trends/data.html

We see that there are two columns in the `co2_df` data frame; `date_measured` and `ppm`. The `date_measured` column holds the date the measurement was taken, and is of type `datetime64`. The `ppm` column holds the value of CO_2 in parts per million that was measured on each date, and is type `float64`; this is the usual type for decimal numbers.

Note: `read_csv` was able to parse the `date_measured` column into the `datetime` vector type because it was entered in the international standard date format, called ISO 8601, which lists dates as `year-month-day` and we used `parse_dates=True`. `datetime` vectors are `double` vectors with special properties that allow them to handle dates correctly. For example, `datetime` type vectors allow functions like `altair` to treat them as numeric dates and not as character vectors, even though they contain non-numeric characters (e.g., in the `date_measured` column in the `co2_df` data frame). This means Python will not accidentally plot the dates in the wrong order (i.e., not alphanumerically as would happen if it was a character vector). More about dates and times can be viewed here[5].

Since we are investigating a relationship between two variables (CO_2 concentration and date), a scatter plot is a good place to start. Scatter plots show the data as individual points with `x` (horizontal axis) and `y` (vertical axis) coordinates. Here, we will use the measurement date as the `x` coordinate and the CO_2 concentration as the `y` coordinate. We create a chart with the `alt.Chart()` function. There are a few basic aspects of a plot that we need to specify:

- The name of the **data frame** to visualize.
  - Here, we specify the `co2_df` data frame as an argument to `alt.Chart`
- The **graphical mark**, which specifies how the mapped data should be displayed.
  - To create a graphical mark, we use `Chart.mark_*` methods (see the altair reference[6] for a list of graphical mark).
  - Here, we use the `mark_point` function to visualize our data as a scatter plot.
- The **encoding channels**, which tells `altair` how the columns in the data frame map to visual properties in the chart.

[5] https://wesmckinney.com/book/time-series.html
[6] https://altair-viz.github.io/user_guide/marks/index.html

- To create an encoding, we use the `encode` function.
- The `encode` method builds a key-value mapping between encoding channels (such as x, y) to fields in the data set, accessed by field name (column names)
- Here, we set the `x` axis of the plot to the `date_measured` variable, and on the `y` axis, we plot the `ppm` variable.
- For the y-axis, we also provided the method `scale(zero=False)`. By default, `altair` chooses the y-limits based on the data and will keep `y=0` in view. This is often a helpful default, but here it makes it difficult to see any trends in our data since the smallest value is >300 ppm. So by providing `scale(zero=False)`, we tell altair to choose a reasonable lower bound based on our data, and that lower bound doesn't have to be zero.
- To change the properties of the encoding channels, we need to leverage the helper functions `alt.Y` and `alt.X`. These helpers have the role of customizing things like order, titles, and scales. Here, we use `alt.Y` to change the domain of the y-axis, so that it starts from the lowest value in the `date_measured` column rather than from zero.

```
co2_scatter = alt.Chart(co2_df).mark_point().encode(
    x="date_measured",
    y=alt.Y("ppm").scale(zero=False)
)
```

The visualization in Fig. 4.2 shows a clear upward trend in the atmospheric concentration of CO_2 over time. This plot answers the first part of our question in the affirmative, but that appears to be the only conclusion one can make from the scatter visualization.

One important thing to note about this data is that one of the variables we are exploring is time. Time is a special kind of quantitative variable because it forces additional structure on the data—the data points have a natural order. Specifically, each observation in the data set has a predecessor and a successor, and the order of the observations matters; changing their order alters their meaning. In situations like this, we typically use a line plot to visualize the data. Line plots connect the sequence of `x` and `y` coordinates of the observations with line segments, thereby emphasizing their order.

We can create a line plot in `altair` using the `mark_line` function. Let's now try to visualize the `co2_df` as a line plot with just the default arguments:

```
co2_line = alt.Chart(co2_df).mark_line().encode(
```

(continues on next page)

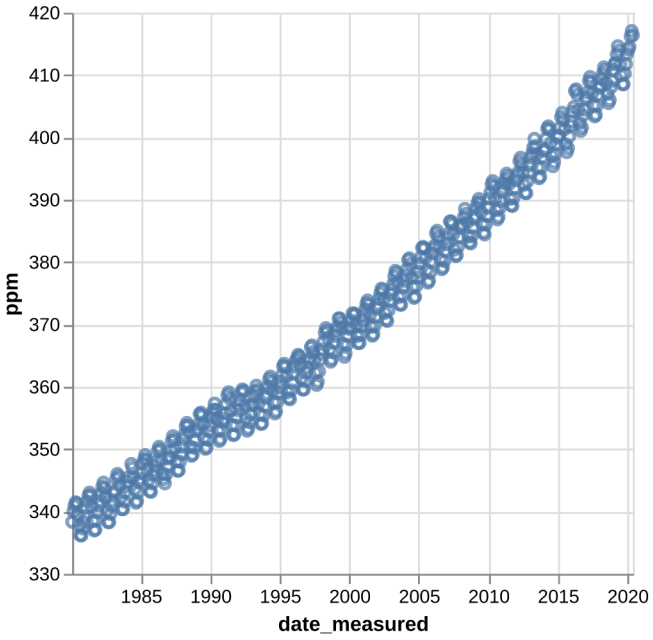


FIGURE 4.2 Scatter plot of atmospheric concentration of CO_2 over time.

(continued from previous page)

```
    x="date_measured",
    y=alt.Y("ppm").scale(zero=False)
)
```

Aha! Fig. 4.3 shows us there *is* another interesting phenomenon in the data: in addition to increasing over time, the concentration seems to oscillate as well. Given the visualization as it is now, it is still hard to tell how fast the oscillation is, but nevertheless, the line seems to be a better choice for answering the question than the scatter plot was. The comparison between these two visualizations also illustrates a common issue with scatter plots: often, the points are shown too close together or even on top of one another, muddling information that would otherwise be clear (*overplotting*).

Now that we have settled on the rough details of the visualization, it is time to refine things. This plot is fairly straightforward, and there is not much visual noise to remove. But there are a few things we must do to improve clarity, such as adding informative axis labels and making the font a more readable size. To add axis labels, we use the `title` method along with `alt.X` and `alt.Y` functions. To change the font size, we use the `configure_axis` function with the `titleFontSize` argument (Fig. 4.4).

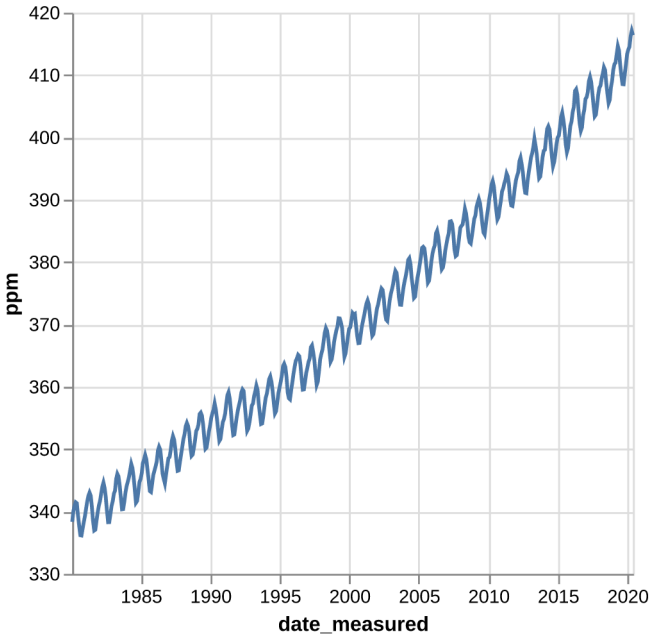


FIGURE 4.3 Line plot of atmospheric concentration of CO_2 over time.

```
co2_line_labels = alt.Chart(co2_df).mark_line().encode(
    x=alt.X("date_measured").title("Year"),
    y=alt.Y("ppm").scale(zero=False).title("Atmospheric CO2 (ppm)")
).configure_axis(titleFontSize=12)
```

Note: The `configure_*` functions in `altair` support additional customization, such as updating the size of the plot, changing the font color, and many other options that can be viewed here[7].

Finally, let's see if we can better understand the oscillation by changing the visualization slightly. Note that it is totally fine to use a small number of visualizations to answer different aspects of the question you are trying to answer. We will accomplish this by using *scale*, another important feature of `altair` that easily transforms the different variables and set limits. In particular, here, we will use the `alt.Scale` function to zoom in on just a few years of data (say, 1990–1995) (Fig. 4.5). The `domain` argument takes a list of length two to specify the upper and lower bounds to limit the axis. We also added the argument `clip=True` to `mark_line`. This tells `altair` to "clip" (remove) the data outside of the specified domain that we set so that it doesn't extend past the plot area. Since we are using both the `scale` and

[7] https://altair-viz.github.io/user_guide/configuration.html

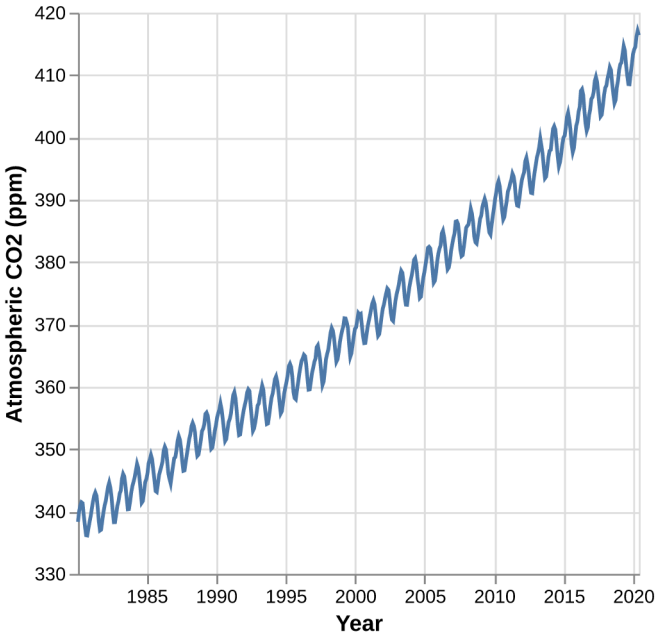


FIGURE 4.4 Line plot of atmospheric concentration of CO_2 over time with clearer axes and labels.

`title` method on the encodings we stack them on separate lines to make the code easier to read (Fig. 4.5).

```
co2_line_scale = alt.Chart(co2_df).mark_line(clip=True).encode(
    x=alt.X("date_measured")
        .scale(domain=["1990", "1995"])
        .title("Measurement Date"),
    y=alt.Y("ppm")
        .scale(zero=False)
        .title("Atmospheric CO2 (ppm)")
).configure_axis(titleFontSize=12)
```

Interesting! It seems that each year, the atmospheric CO_2 increases until it reaches its peak somewhere around April, decreases until around late September, and finally increases again until the end of the year. In Hawaii, there are two seasons: summer from May through October, and winter from November through April. Therefore, the oscillating pattern in CO_2 matches up fairly closely with the two seasons.

A useful analogy to constructing a data visualization is painting a picture. We start with a blank canvas, and the first thing we do is prepare the surface for our painting by adding primer. In our data visualization this is akin to calling `alt.Chart` and specifying the data set we will be using. Next, we sketch out the background of the painting. In our data visualization, this would be when we map data to the axes in the `encode` function. Then we add our key visual

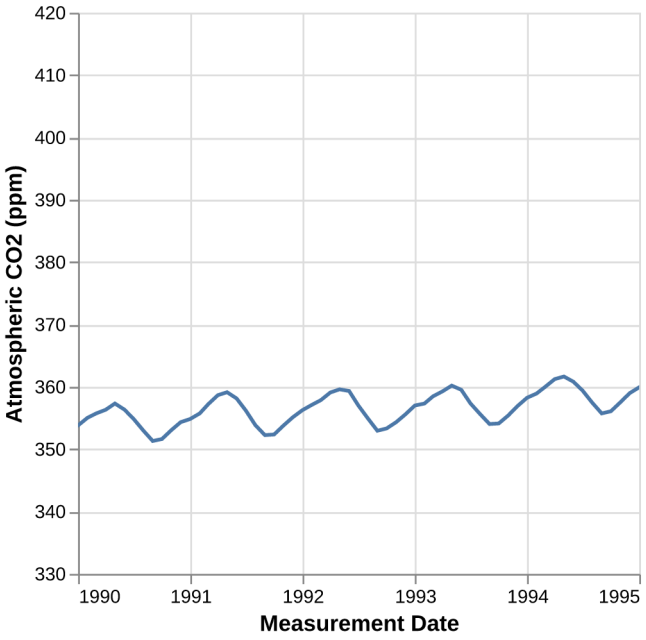


FIGURE 4.5 Line plot of atmospheric concentration of CO_2 from 1990 to 1995.

subjects to the painting. In our data visualization, this would be the graphical marks (e.g., `mark_point`, `mark_line`, etc.). And finally, we work on adding details and refinements to the painting. In our data visualization this would be when we fine tune axis labels, change the font, adjust the point size, and do other related things.

4.5.2 Scatter plots: the Old Faithful eruption time data set

The `faithful` data set contains measurements of the waiting time between eruptions and the subsequent eruption duration (in minutes) of the Old Faithful geyser in Yellowstone National Park, Wyoming, United States. First, we will read the data and then answer the following question:

Question: Is there a relationship between the waiting time before an eruption and the duration of the eruption?

```
faithful = pd.read_csv("data/faithful.csv")
faithful
```

```
     eruptions  waiting
0        3.600       79
1        1.800       54
2        3.333       74
```

(continues on next page)

(continued from previous page)

```
3         2.283        62
4         4.533        85
..          ...       ...
267       4.117        81
268       2.150        46
269       4.417        90
270       1.817        46
271       4.467        74

[272 rows x 2 columns]
```

Here again, we investigate the relationship between two quantitative variables (waiting time and eruption time). But if you look at the output of the data frame, you'll notice that unlike time in the Mauna Loa CO_2 data set, neither of the variables here have a natural order to them. So a scatter plot is likely to be the most appropriate visualization. Let's create a scatter plot using the `altair` package with the `waiting` variable on the horizontal axis, the `eruptions` variable on the vertical axis, and `mark_point` as the graphical mark. The result is shown in Fig. 4.6.

```
faithful_scatter = alt.Chart(faithful).mark_point().encode(
    x="waiting",
    y="eruptions"
)
```

We can see in Fig. 4.6 that the data tend to fall into two groups: one with short waiting and eruption times, and one with long waiting and eruption times.

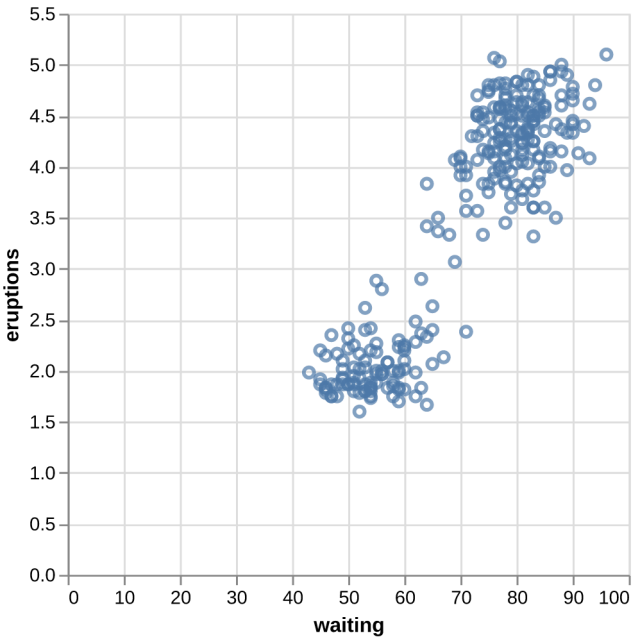


FIGURE 4.6 Scatter plot of waiting time and eruption time.

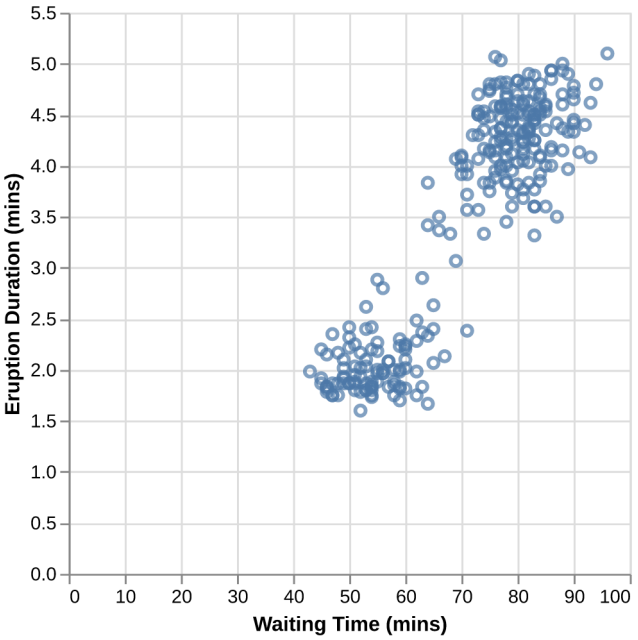


FIGURE 4.7 Scatter plot of waiting time and eruption time with clearer axes and labels.

Note that in this case, there is no overplotting: the points are generally nicely visually separated, and the pattern they form is clear. In order to refine the visualization, we need only to add axis labels and make the font more readable (Fig. 4.7).

```
faithful_scatter_labels = alt.Chart(faithful).mark_point().encode(
    x=alt.X("waiting").title("Waiting Time (mins)"),
    y=alt.Y("eruptions").title("Eruption Duration (mins)")
)
```

We can change the size of the point and color of the plot by specifying `mark_point(size=10, color="black")` (Fig. 4.8).

```
faithful_scatter_labels_black = alt.Chart(faithful).mark_point(size=10, color=
↪"black").encode(
    x=alt.X("waiting").title("Waiting Time (mins)"),
    y=alt.Y("eruptions").title("Eruption Duration (mins)")
)
```

4.5.3 Axis transformation and colored scatter plots: the Canadian languages data set

Recall the `can_lang` data set [Timbers, 2020] from Chapters 1, 2, and 3. It contains counts of languages from the 2016 Canadian census.

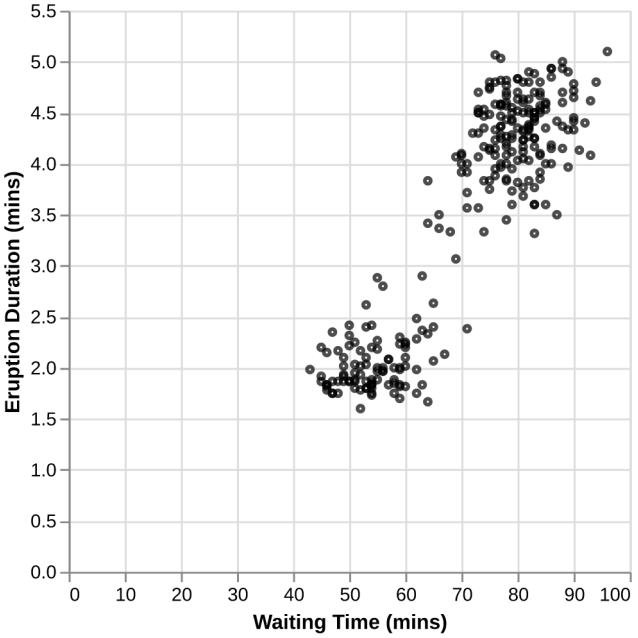


FIGURE 4.8 Scatter plot of waiting time and eruption time with black points.

Question: Is there a relationship between the percentage of people who speak a language as their mother tongue and the percentage for whom that is the primary language spoken at home? And is there a pattern in the strength of this relationship in the higher-level language categories (Official languages, Aboriginal languages, or non-official and non-Aboriginal languages)?

To get started, we will read and inspect the data:

```
can_lang = pd.read_csv("data/can_lang.csv")
can_lang
```

```
                                    category                         language ␣
↪\
0                       Aboriginal languages    Aboriginal languages, n.o.s.
1    Non-Official & Non-Aboriginal languages                       Afrikaans
2    Non-Official & Non-Aboriginal languages  Afro-Asiatic languages, n.i.e.
3    Non-Official & Non-Aboriginal languages                     Akan (Twi)
4    Non-Official & Non-Aboriginal languages                       Albanian
..                                      ...                            ...
209  Non-Official & Non-Aboriginal languages                          Wolof
210                     Aboriginal languages                   Woods Cree
211  Non-Official & Non-Aboriginal languages              Wu (Shanghainese)
212  Non-Official & Non-Aboriginal languages                        Yiddish
213  Non-Official & Non-Aboriginal languages                         Yoruba

     mother_tongue  most_at_home  most_at_work  lang_known
0              590           235            30         665
```

(continues on next page)

(continued from previous page)

```
1            10260          4785           85       23415
2             1150           445           10        2775
3            13460          5985           25       22150
4            26895         13135          345       31930
..             ...           ...          ...         ...
209           3990          1385           10        8240
210           1840           800           75        2665
211          12915          7650          105       16530
212          13555          7085          895       20985
213           9080          2615           15       22415

[214 rows x 6 columns]
```

We will begin with a scatter plot of the `mother_tongue` and `most_at_home` columns from our data frame. As we have seen in the scatter plots in the previous section, the default behavior of `mark_point` is to draw the outline of each point. If we would like to fill them in, we can pass the argument `filled=True` to `mark_point` or use the shortcut `mark_circle`. Whether to fill points or not is mostly a matter of personal preferences, although hollow points can make it easier to see individual points when there are many overlapping points in a chart. The resulting plot is shown in Fig. 4.9.

```
can_lang_plot = alt.Chart(can_lang).mark_circle().encode(
    x="most_at_home",
    y="mother_tongue"
)
```

To make an initial improvement in the interpretability of Fig. 4.9, we should replace the default axis names with more informative labels. To make the axes labels on the plots more readable, we can print long labels over multiple lines. To achieve this, we specify the title as a list of strings where each string in the list will correspond to a new line of text. We can also increase the font size to further improve readability.

```
can_lang_plot_labels = alt.Chart(can_lang).mark_circle().encode(
    x=alt.X("most_at_home")
        .title(["Language spoken most at home", "(number of Canadian residents)
↪"]),
    y=alt.Y("mother_tongue")
        .scale(zero=False)
        .title(["Mother tongue", "(number of Canadian residents)"])
).configure_axis(titleFontSize=12)
```

Okay! The axes and labels in Fig. 4.10 are much more readable and interpretable now. However, the scatter points themselves could use some work; most of the 214 data points are bunched up in the lower left-hand side of the visualization. The data is clumped because many more people in Canada speak English or French (the two points in the upper right corner) than other languages. In particular, the most common mother tongue language has

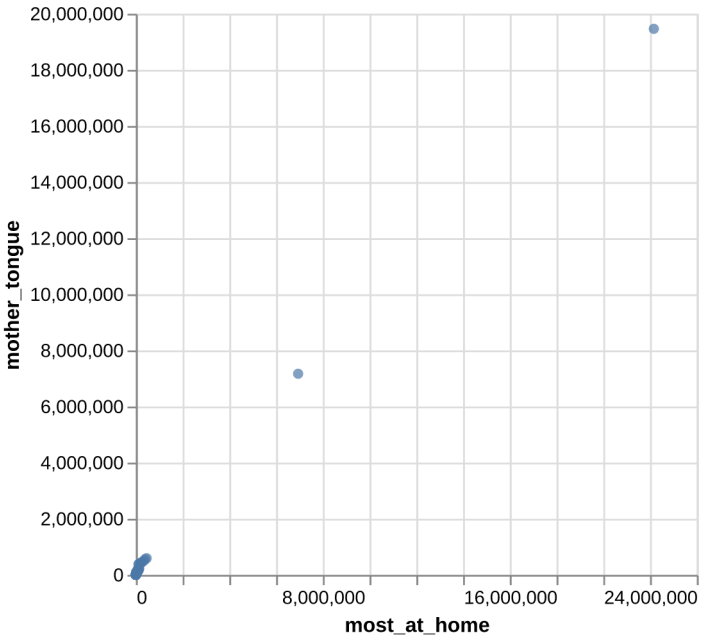


FIGURE 4.9 Scatter plot of number of Canadians reporting a language as their mother tongue vs the primary language at home.

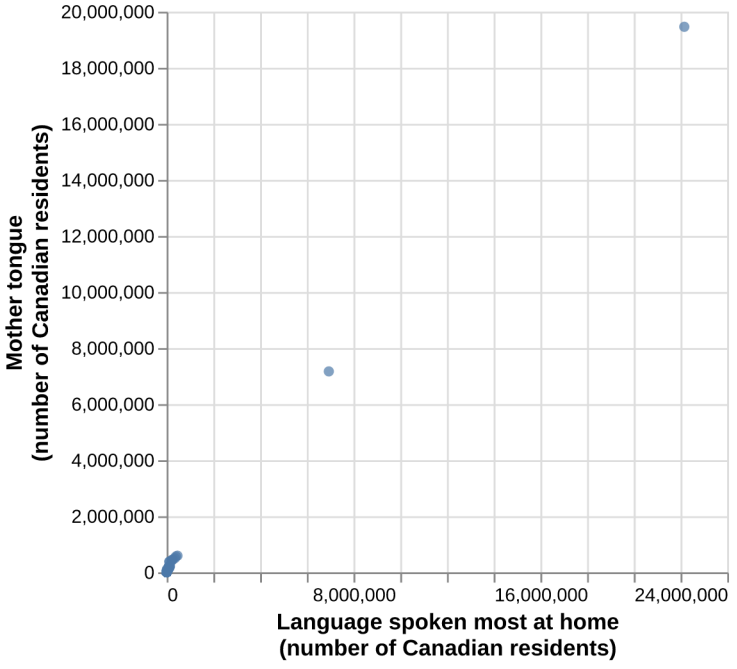


FIGURE 4.10 Scatter plot of number of Canadians reporting a language as their mother tongue vs the primary language at home with x and y labels.

19,460,850 speakers, while the least common has only 10. That's a six-decimal-place difference in the magnitude of these two numbers. We can confirm that the two points in the upper right-hand corner correspond to Canada's two official languages by filtering the data:

```
can_lang.loc[
    (can_lang["language"]=="English")
    | (can_lang["language"]=="French")
]
```

```
              category language  mother_tongue  most_at_home  most_at_work  \
54  Official languages  English       19460850      22162865      15265335
59  Official languages   French        7166700       6943800       3825215

    lang_known
54    29748265
59    10242945
```

Recall that our question about this data pertains to *all* languages; so to properly answer our question, we will need to adjust the scale of the axes so that we can clearly see all of the scatter points. In particular, we will improve the plot by adjusting the horizontal and vertical axes so that they are on a **logarithmic** (or **log**) scale. Log scaling is useful when your data take both *very large* and *very small* values, because it helps space out small values and squishes larger values together. For example, $\log_{10}(1) = 0$, $\log_{10}(10) = 1$, $\log_{10}(100) = 2$, and $\log_{10}(1000) = 3$; on the logarithmic scale, the values 1, 10, 100, and 1000 are all the same distance apart. So we see that applying this function is moving big values closer together and moving small values farther apart. Note that if your data can take the value 0, logarithmic scaling may not be appropriate (since `log10(0)` is `-inf` in Python). There are other ways to transform the data in such a case, but these are beyond the scope of the book.

We can accomplish logarithmic scaling in the `altair` visualization using the argument `type="log"` in the scale method.

```
can_lang_plot_log = alt.Chart(can_lang).mark_circle().encode(
    x=alt.X("most_at_home")
        .scale(type="log")
        .title(["Language spoken most at home", "(number of Canadian residents)
↪"]),
    y=alt.Y("mother_tongue")
        .scale(type="log")
        .title(["Mother tongue", "(number of Canadian residents)"])
).configure_axis(titleFontSize=12)
```

You will notice two things in the chart in Fig. 4.11 above, changing the axis to log creates many axis ticks and gridlines, which makes the appearance of the chart rather noisy and it is hard to focus on the data. You can also see that the second last tick label is missing on the x-axis; Altair dropped it because

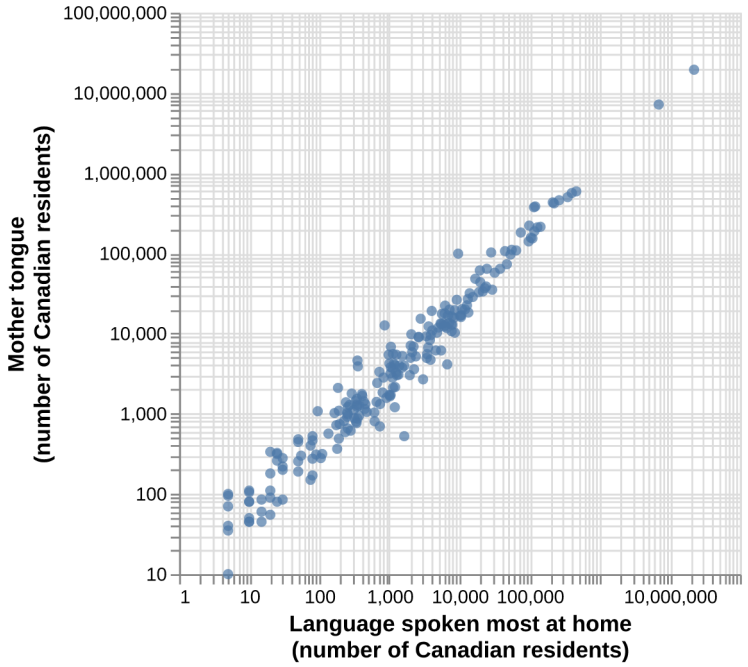


FIGURE 4.11 Scatter plot of number of Canadians reporting a language as their mother tongue vs the primary language at home with log-adjusted x and y axes.

there wasn't space to fit in all the large numbers next to each other. It is also hard to see if the label for 100,000,000 is for the last or second last tick. To fix these issue, we can limit the number of ticks and gridlines to only include the seven major ones, and change the number formatting to include a suffix which makes the labels shorter (Fig. 4.12).

```
can_lang_plot_log_revised = alt.Chart(can_lang).mark_circle().encode(
    x=alt.X("most_at_home")
        .scale(type="log")
        .title(["Language spoken most at home", "(number of Canadian residents)
↪"])
        .axis(tickCount=7, format="s"),
    y=alt.Y("mother_tongue")
        .scale(type="log")
        .title(["Mother tongue", "(number of Canadian residents)"])
        .axis(tickCount=7, format="s")
).configure_axis(titleFontSize=12)
```

Similar to some of the examples in Chapter 3, we can convert the counts to percentages to give them context and make them easier to understand. We can do this by dividing the number of people reporting a given language as their mother tongue or primary language at home by the number of people who live in Canada and multiplying by 100%. For example, the percentage

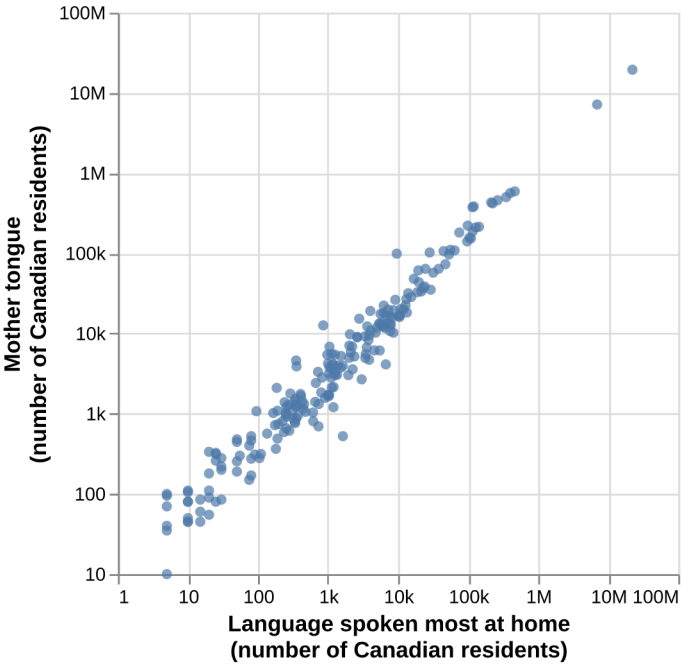


FIGURE 4.12 Scatter plot of number of Canadians reporting a language as their mother tongue vs the primary language at home with log-adjusted x and y axes. Only the major gridlines are shown. The suffix "k" indicates 1,000 ("kilo"), while the suffix "M" indicates 1,000,000 ("million").

of people who reported that their mother tongue was English in the 2016 Canadian census was 19,460,850 / 35,151,728 × 100% = 55.36%

Below we assign the percentages of people reporting a given language as their mother tongue and primary language at home to two new columns in the `can_lang` data frame. Since the new columns are appended to the end of the data table, we selected the new columns after the transformation so you can clearly see the mutated output from the table. Note that we formatted the number for the Canadian population using _ so that it is easier to read; this does not affect how Python interprets the number and is just added for readability.

```
canadian_population = 35_151_728
can_lang["mother_tongue_percent"] = can_lang["mother_tongue"]/canadian_
↪population*100
can_lang["most_at_home_percent"] = can_lang["most_at_home"]/canadian_
↪population*100
can_lang[["mother_tongue_percent", "most_at_home_percent"]]
```

```
   mother_tongue_percent  most_at_home_percent
0                0.001678              0.000669
```

(continues on next page)

(continued from previous page)

```
1                 0.029188              0.013612
2                 0.003272              0.001266
3                 0.038291              0.017026
4                 0.076511              0.037367
..                     ...                   ...
209               0.011351              0.003940
210               0.005234              0.002276
211               0.036741              0.021763
212               0.038561              0.020155
213               0.025831              0.007439

[211 rows x 2 columns]
```

Next, we will edit the visualization to use the percentages we just computed (and change our axis labels to reflect this change in units). Fig. 4.13 displays the final result. Here all the tick labels fit by default so we are not changing the labels to include suffixes. Note that suffixes can also be harder to understand, so it is often advisable to avoid them (particularly for small quantities) unless you are communicating to a technical audience.

```
can_lang_plot_percent = alt.Chart(can_lang).mark_circle().encode(
    x=alt.X("most_at_home_percent")
        .scale(type="log")
        .axis(tickCount=7)
        .title(["Language spoken most at home", "(percentage of Canadian␣
↪residents)"]),
    y=alt.Y("mother_tongue_percent")
        .scale(type="log")
        .axis(tickCount=7)
        .title(["Mother tongue", "(percentage of Canadian residents)"]),
).configure_axis(titleFontSize=12)
```

Fig. 4.13 is the appropriate visualization to use to answer the first question in this section, i.e., whether there is a relationship between the percentage of people who speak a language as their mother tongue and the percentage for whom that is the primary language spoken at home. To fully answer the question, we need to use Fig. 4.13 to assess a few key characteristics of the data:

- **Direction:** if the y variable tends to increase when the x variable increases, then y has a **positive** relationship with x. If y tends to decrease when x increases, then y has a **negative** relationship with x. If y does not meaningfully increase or decrease as x increases, then y has **little or no** relationship with x.
- **Strength:** if the y variable *reliably* increases, decreases, or stays flat as x increases, then the relationship is **strong**. Otherwise, the relationship is **weak**. Intuitively, the relationship is strong when the scatter points are close together and look more like a "line" or "curve" than a "cloud".

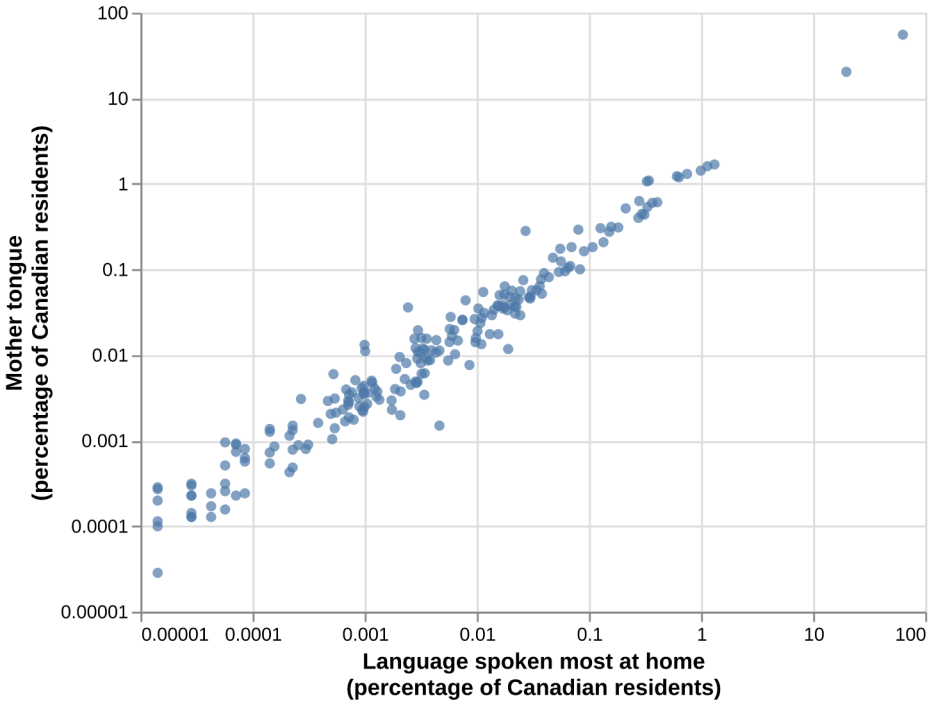


FIGURE 4.13 Scatter plot of percentage of Canadians reporting a language as their mother tongue vs the primary language at home.

- **Shape:** if you can draw a straight line roughly through the data points, the relationship is **linear**. Otherwise, it is **nonlinear**.

In Fig. 4.13, we see that as the percentage of people who have a language as their mother tongue increases, so does the percentage of people who speak that language at home. Therefore, there is a **positive** relationship between these two variables. Furthermore, because the points in Fig. 4.13 are fairly close together, and the points look more like a "line" than a "cloud", we can say that this is a **strong** relationship. And finally, because drawing a straight line through these points in Fig. 4.13 would fit the pattern we observe quite well, we say that the relationship is **linear**.

Onto the second part of our exploratory data analysis question. Recall that we are interested in knowing whether the strength of the relationship we uncovered in Fig. 4.13 depends on the higher-level language category (Official languages, Aboriginal languages, and non-official, non-Aboriginal languages). One common way to explore this is to color the data points on the scatter plot we have already created by group. For example, given that we have the higher-level language category for each language recorded in the 2016

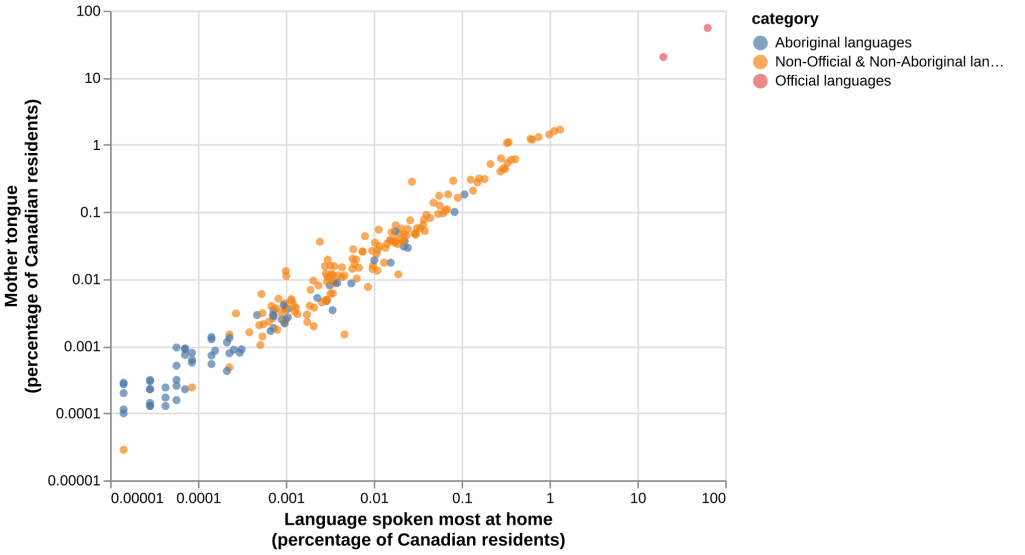


FIGURE 4.14 Scatter plot of percentage of Canadians reporting a language as their mother tongue vs the primary language at home colored by language category.

Canadian census, we can color the points in our previous scatter plot to represent each language's higher-level language category.

Here we want to distinguish the values according to the `category` group with which they belong. We can add the argument `color` to the `encode` method, specifying that the `category` column should color the points. Adding this argument will color the points according to their group and add a legend at the side of the plot. Since the labels of the language category as descriptive of their own, we can remove the title of the legend to reduce visual clutter without reducing the effectiveness of the chart (Fig. 4.14).

```
can_lang_plot_category=alt.Chart(can_lang).mark_circle().encode(
    x=alt.X("most_at_home_percent")
        .scale(type="log")
        .axis(tickCount=7)
        .title(["Language spoken most at home", "(percentage of Canadian␣
↪residents)"]),
    y=alt.Y("mother_tongue_percent")
        .scale(type="log")
        .axis(tickCount=7)
        .title(["Mother tongue", "(percentage of Canadian residents)"]),
    color="category"
).configure_axis(titleFontSize=12)
```

Another thing we can adjust is the location of the legend. This is a matter of preference and not critical for the visualization. We move the legend title using the `alt.Legend` method and specify that we want it on the top of the

chart. This automatically changes the legend items to be laid out horizontally instead of vertically, but we could also keep the vertical layout by specifying `direction="vertical"` inside `alt.Legend`.

```
can_lang_plot_legend = alt.Chart(can_lang).mark_circle().encode(
    x=alt.X("most_at_home_percent")
        .scale(type="log")
        .axis(tickCount=7)
        .title(["Language spoken most at home", "(percentage of Canadian␣
↪residents)"]),
    y=alt.Y("mother_tongue_percent")
        .scale(type="log")
        .axis(tickCount=7)
        .title(["Mother tongue", "(percentage of Canadian residents)"]),
    color=alt.Color("category")
        .legend(orient="top")
        .title("")
).configure_axis(titleFontSize=12)
```

In Fig. 4.15, the points are colored with the default `altair` color scheme, which is called `"tableau10"`. This is an appropriate choice for most situations and is also easy to read for people with reduced color vision. In general, the

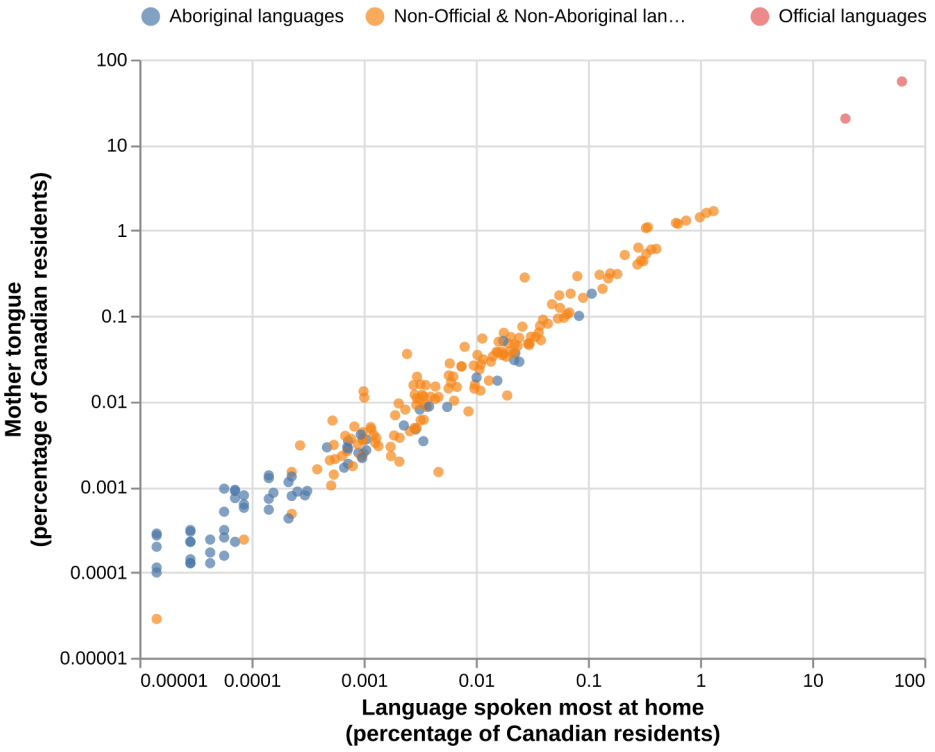


FIGURE 4.15 Scatter plot of percentage of Canadians reporting a language as their mother tongue vs the primary language at home colored by language category with the legend edited.

color schemes that are used by default in Altair are adapted to the type of data that is displayed and selected to be easy to interpret both for people with good and reduced color vision. If you are unsure about a certain color combination, you can use this color blindness simulator[8] to check if your visualizations are color-blind friendly.

All the available color schemes and information on how to create your own can be viewed in the Altair documentation[9]. To change the color scheme of our chart, we can add the `scheme` argument in the `scale` of the `color` encoding. Below we pick the `"dark2"` theme, with the result shown in Fig. 4.16. We also set the `shape` aesthetic mapping to the `category` variable as well; this makes the scatter point shapes different for each language category. This kind of visual redundancy—i.e., conveying the same information with both scatter point color and shape—can further improve the clarity and accessibility of your visualization, but can add visual noise if there are many different shapes and colors, so it should be used with care. Note that we are switching back to the use of `mark_point` here since `mark_circle` does not support the `shape` encoding and will always show up as a filled circle.

```
can_lang_plot_theme = alt.Chart(can_lang).mark_point(filled=True).encode(
    x=alt.X("most_at_home_percent")
        .scale(type="log")
        .axis(tickCount=7)
        .title(["Language spoken most at home", "(percentage of Canadian␣
↪residents)"]),
    y=alt.Y("mother_tongue_percent")
        .scale(type="log")
        .axis(tickCount=7)
        .title(["Mother tongue", "(percentage of Canadian residents)"]),
    color=alt.Color("category")
        .legend(orient="top")
        .title("")
        .scale(scheme="dark2"),
    shape="category"
).configure_axis(titleFontSize=12)
```

The chart above gives a good indication of how the different language categories differ, and this information is sufficient to answer our research question. But what if we want to know exactly which language correspond to which point in the chart? With a regular visualization library this would not be possible, as adding text labels for each individual language would add a lot of visual noise and make the chart difficult to interpret. However, since Altair is an interactive visualization library we can add information on demand via the `Tooltip` encoding channel, so that text labels for each point show up once we hover over it with the mouse pointer. Here we also add the exact values of the variables on the x and y-axis to the tooltip.

[8] https://www.color-blindness.com/coblis-color-blindness-simulator/
[9] https://altair-viz.github.io/user_guide/customization.html#customizing-colors

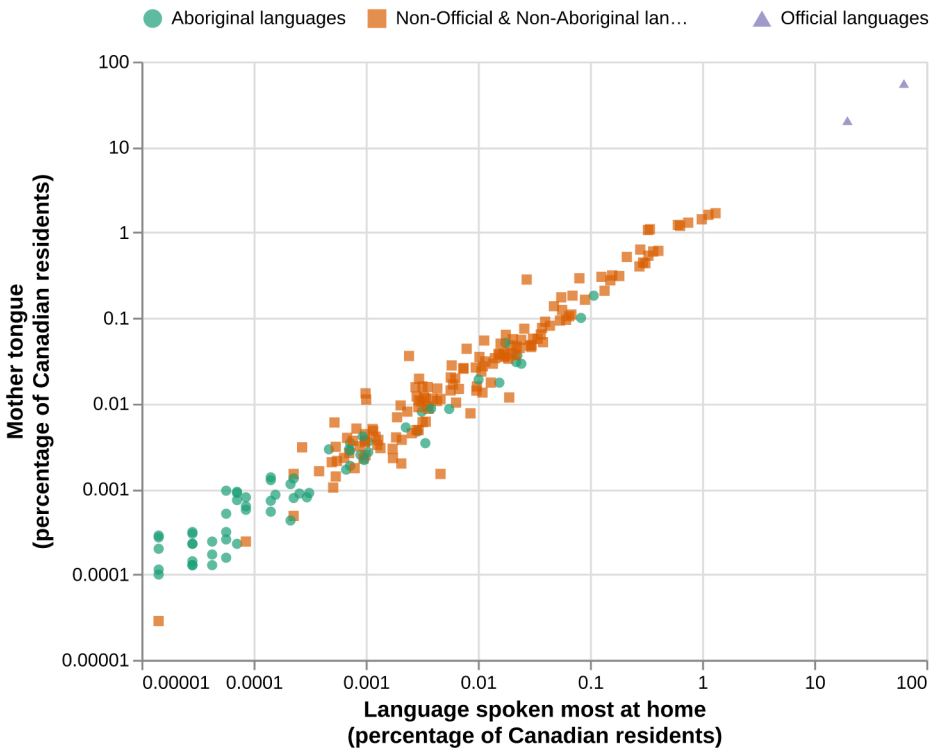


FIGURE 4.16 Scatter plot of percentage of Canadians reporting a language as their mother tongue vs the primary language at home colored by language category with custom colors and shapes.

```
can_lang_plot_tooltip = alt.Chart(can_lang).mark_point(filled=True).encode(
    x=alt.X("most_at_home_percent")
        .scale(type="log")
        .axis(tickCount=7)
        .title(["Language spoken most at home", "(percentage of Canadian␣
↪residents)"]),
    y=alt.Y("mother_tongue_percent")
        .scale(type="log")
        .axis(tickCount=7)
        .title(["Mother tongue", "(percentage of Canadian residents)"]),
    color=alt.Color("category")
        .legend(orient="top")
        .title("")
        .scale(scheme="dark2"),
    shape="category",
    tooltip=alt.Tooltip(["language", "mother_tongue", "most_at_home"])
).configure_axis(titleFontSize=12)
```

From the visualization in Fig. 4.17, we can now clearly see that the vast majority of Canadians reported one of the official languages as their mother tongue and as the language they speak most often at home. What do we see when considering the second part of our exploratory question? Do we see a

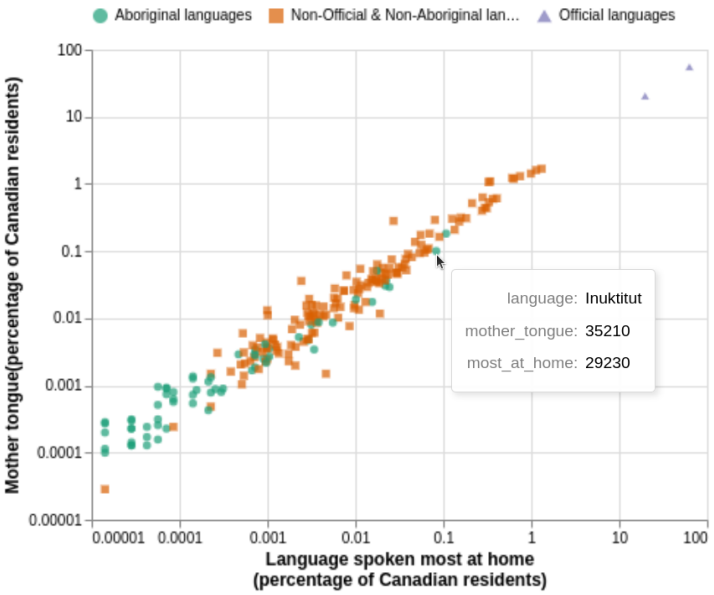


FIGURE 4.17 Scatter plot of percentage of Canadians reporting a language as their mother tongue vs the primary language at home colored by language category with custom colors and mouse hover tooltip.

difference in the relationship between languages spoken as a mother tongue and as a primary language at home across the higher-level language categories? Based on Fig. 4.17, there does not appear to be much of a difference. For each higher-level language category, there appears to be a strong, positive, and linear relationship between the percentage of people who speak a language as their mother tongue and the percentage who speak it as their primary language at home. The relationship looks similar regardless of the category.

Does this mean that this relationship is positive for all languages in the world? And further, can we use this data visualization on its own to predict how many people have a given language as their mother tongue if we know how many people speak it as their primary language at home? The answer to both these questions is "no!" However, with exploratory data analysis, we can create new hypotheses, ideas, and questions (like the ones at the beginning of this paragraph). Answering those questions often involves doing more complex analyses, and sometimes even gathering additional data. We will see more of such complex analyses later on in this book.

4.5.4 Bar plots: the island landmass data set

The `islands.csv` data set contains a list of Earth's landmasses as well as their area (in thousands of square miles) [McNeil, 1977].

Question: Are the continents (North / South America, Africa, Europe, Asia, Australia, Antarctica) Earth's seven largest landmasses? If so, what are the next few largest landmasses after those?

To get started, we will read and inspect the data:

```
islands_df = pd.read_csv("data/islands.csv")
islands_df
```

	landmass	size	landmass_type
0	Africa	11506	Continent
1	Antarctica	5500	Continent
2	Asia	16988	Continent
3	Australia	2968	Continent
4	Axel Heiberg	16	Other
5	Baffin	184	Other
6	Banks	23	Other
7	Borneo	280	Other
8	Britain	84	Other
9	Celebes	73	Other
10	Celon	25	Other
11	Cuba	43	Other
12	Devon	21	Other
13	Ellesmere	82	Other
14	Europe	3745	Continent
15	Greenland	840	Other
16	Hainan	13	Other
17	Hispaniola	30	Other
18	Hokkaido	30	Other
19	Honshu	89	Other
20	Iceland	40	Other
21	Ireland	33	Other
22	Java	49	Other
23	Kyushu	14	Other
24	Luzon	42	Other
25	Madagascar	227	Other
26	Melville	16	Other
27	Mindanao	36	Other
28	Moluccas	29	Other
29	New Britain	15	Other
30	New Guinea	306	Other
31	New Zealand (N)	44	Other
32	New Zealand (S)	58	Other
33	Newfoundland	43	Other
34	North America	9390	Continent
35	Novaya Zemlya	32	Other
36	Prince of Wales	13	Other
37	Sakhalin	29	Other
38	South America	6795	Continent
39	Southampton	16	Other
40	Spitsbergen	15	Other
41	Sumatra	183	Other
42	Taiwan	14	Other
43	Tasmania	26	Other
44	Tierra del Fuego	19	Other
45	Timor	13	Other
46	Vancouver	12	Other
47	Victoria	82	Other

Here, we have a data frame of Earth's landmasses, and are trying to compare their sizes. The right type of visualization to answer this question is a bar

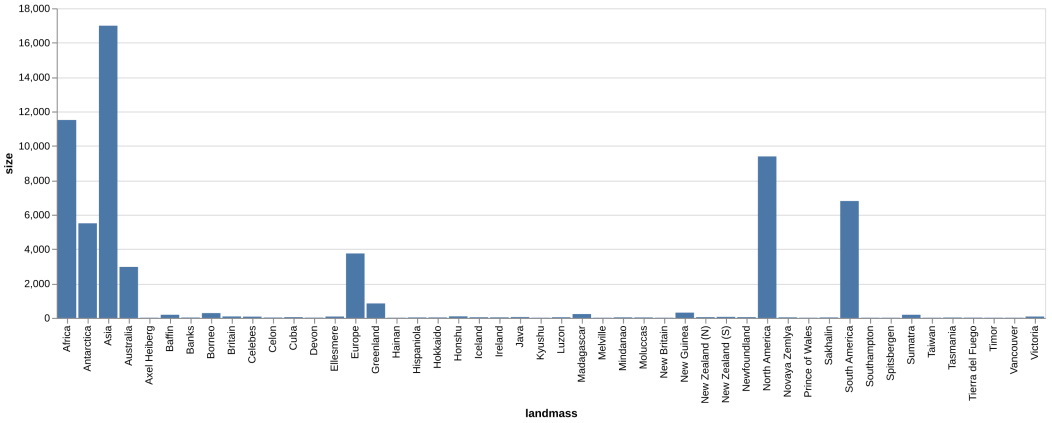


FIGURE 4.18 Bar plot of Earth's landmass sizes. The plot is too wide with the default settings.

plot. In a bar plot, the height of each bar represents the value of an *amount* (a size, count, proportion, percentage, etc.). They are particularly useful for comparing counts or proportions across different groups of a categorical variable. Note, however, that bar plots should generally not be used to display mean or median values, as they hide important information about the variation of the data. Instead it's better to show the distribution of all the individual data points, e.g., using a histogram, which we will discuss further in Section 4.5.5.

We specify that we would like to use a bar plot via the `mark_bar` function in `altair`. The result is shown in Fig. 4.18.

```
islands_bar = alt.Chart(islands_df).mark_bar().encode(
    x="landmass",
    y="size"
)
```

Alright, not bad! The plot in Fig. 4.18 is definitely the right kind of visualization, as we can clearly see and compare sizes of landmasses. The major issues are that the smaller landmasses' sizes are hard to distinguish, and the plot is so wide that we can't compare them all. But remember that the question we asked was only about the largest landmasses; let's make the plot a little bit clearer by keeping only the largest 12 landmasses. We do this using the `nlargest` function: the first argument is the number of rows we want and the second is the name of the column we want to use for comparing which is largest. Then to help make the landmass labels easier to read we'll swap the `x` and `y` variables, so that the labels are on the y-axis and we don't have to tilt our head to read them.

Note: Recall that in Chapter 1, we used `sort_values` followed by `head` to obtain the ten rows with the largest values of a variable. We could have instead used the `nlargest` function from `pandas` for this purpose. The `nsmallest` and `nlargest` functions achieve the same goal as `sort_values` followed by `head`, but are slightly more efficient because they are specialized for this purpose. In general, it is good to use more specialized functions when they are available.

```
islands_top12 = islands_df.nlargest(12, "size")

islands_bar_top = alt.Chart(islands_top12).mark_bar().encode(
    x="size",
    y="landmass"
)
```

The plot in Fig. 4.19 is definitely clearer now, and allows us to answer our initial questions: "Are the seven continents Earth's largest landmasses?" and "Which are the next few largest landmasses?". However, we could still improve this visualization by coloring the bars based on whether they correspond to a continent, and by organizing the bars by landmass size rather than by alphabetical order. The data for coloring the bars is stored in the `landmass_type` column, so we set the `color` encoding to `landmass_type`. To organize the landmasses by their `size` variable, we will use the altair `sort` function in the y-encoding of the chart. Since the `size` variable is encoded in the x channel of the chart, we specify `sort("x")` on `alt.Y`. This plots the values on `y` axis

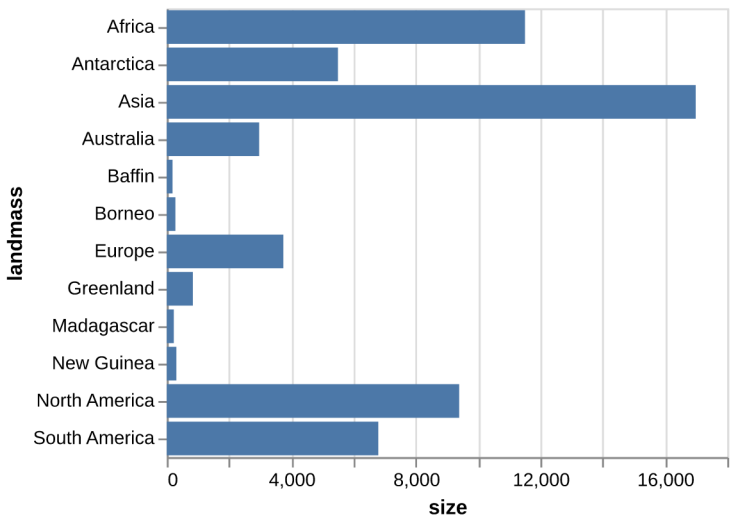


FIGURE 4.19 Bar plot of size for Earth's largest 12 landmasses.

in the ascending order of `x` axis values. This creates a chart where the largest bar is the closest to the axis line, which is generally the most visually appealing when sorting bars. If instead we wanted to sort the values on `y-axis` in descending order of `x-axis`, we could add a minus sign to reverse the order and specify `sort="-x"`.

To finalize this plot we will customize the axis and legend labels using the `title` method, and add a title to the chart by specifying the `title` argument of `alt.Chart`. Plot titles are not always required, especially when it would be redundant with an already-existing caption or surrounding context (e.g., in a slide presentation with annotations). But if you decide to include one, a good plot title should provide the take home message that you want readers to focus on, e.g., "Earth's seven largest landmasses are continents", or a more general summary of the information displayed, e.g., "Earth's twelve largest landmasses".

```
islands_plot_sorted = alt.Chart(
        islands_top12,
        title="Earth's seven largest landmasses are continents"
).mark_bar().encode(
    x=alt.X("size").title("Size (1000 square mi)"),
    y=alt.Y("landmass").sort("x").title("Landmass"),
    color=alt.Color("landmass_type").title("Type")
)
```

The plot in Fig. 4.20 is now an effective visualization for answering our original questions. Landmasses are organized by their size, and continents are colored

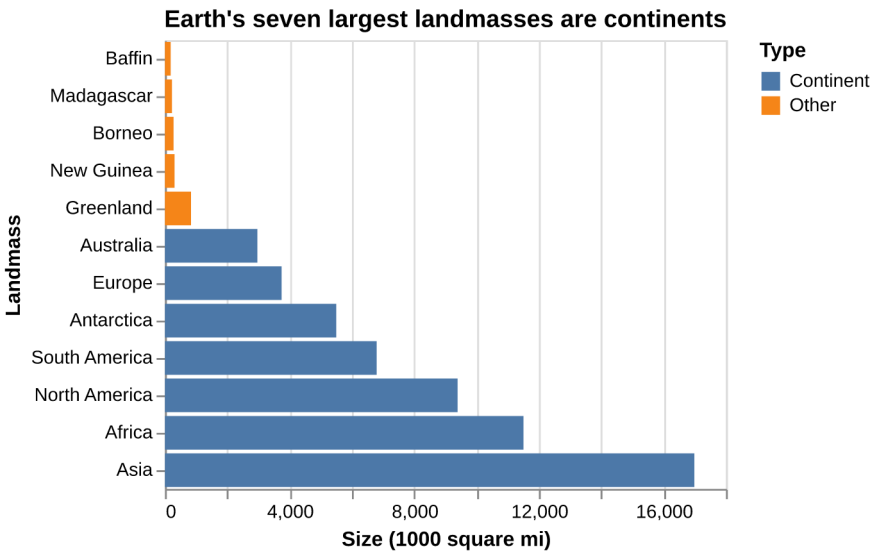


FIGURE 4.20 Bar plot of size for Earth's largest 12 landmasses, colored by landmass type, with clearer axes and labels.

differently than other landmasses, making it quite clear that all the seven largest landmasses are continents.

4.5.5 Histograms: the Michelson speed of light data set

The `morley` data set contains measurements of the speed of light collected in experiments performed in 1879. Five experiments were performed, and in each experiment, 20 runs were performed—meaning that 20 measurements of the speed of light were collected in each experiment [Michelson, 1882]. Because the speed of light is a very large number (the true value is 299,792.458 km/sec), the data is coded to be the measured speed of light minus 299,000. This coding allows us to focus on the variations in the measurements, which are generally much smaller than 299,000. If we used the full large speed measurements, the variations in the measurements would not be noticeable, making it difficult to study the differences between the experiments.

Question: Given what we know now about the speed of light (299,792.458 kilometers per second), how accurate were each of the experiments?

First, we read in the data.

```
morley_df = pd.read_csv("data/morley.csv")
morley_df
```

```
    Expt  Run  Speed
0      1    1    850
1      1    2    740
2      1    3    900
3      1    4   1070
4      1    5    930
..   ...  ...    ...
95     5   16    940
96     5   17    950
97     5   18    800
98     5   19    810
99     5   20    870

[100 rows x 3 columns]
```

In this experimental data, Michelson was trying to measure just a single quantitative number (the speed of light). The data set contains many measurements of this single quantity. To tell how accurate the experiments were, we need to visualize the distribution of the measurements (i.e., all their possible values and how often each occurs). We can do this using a *histogram*. A histogram helps us visualize how a particular variable is distributed in a data set by grouping the values into bins, and then using vertical bars to show how many data points fell in each bin.

To understand how to create a histogram in `altair`, let's start by creating a bar chart just like we did in the previous section. Note that this time, we are

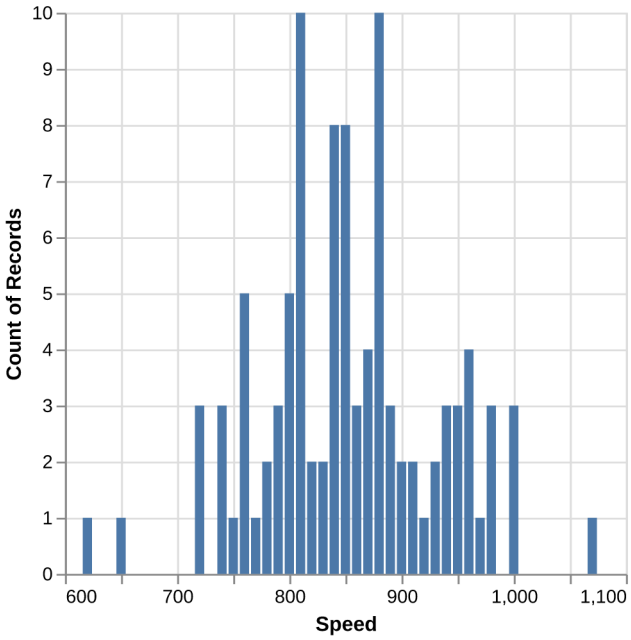


FIGURE 4.21 A bar chart of Michelson's speed of light data.

setting the `y` encoding to `"count()"`. There is no `"count()"` column-name in `morley_df`; we use `"count()"` to tell `altair` that we want to count the number of occurrences of each value in along the x-axis (which we encoded as the `Speed` column) (Fig. 4.21).

```
morley_bars = alt.Chart(morley_df).mark_bar().encode(
    x="Speed",
    y="count()"
)
```

The bar chart above gives us an indication of which values are more common than others, but because the bars are so thin it's hard to get a sense for the overall distribution of the data. We don't really care about how many occurrences there are of each exact `Speed` value, but rather where most of the `Speed` values fall in general. To more effectively communicate this information we can group the x-axis into bins (or "buckets") using the `bin` method and then count how many `Speed` values fall within each bin. A bar chart that represent the count of values for a binned quantitative variable is called a histogram.

```
morley_hist = alt.Chart(morley_df).mark_bar().encode(
    x=alt.X("Speed").bin(),
    y="count()"
)
```

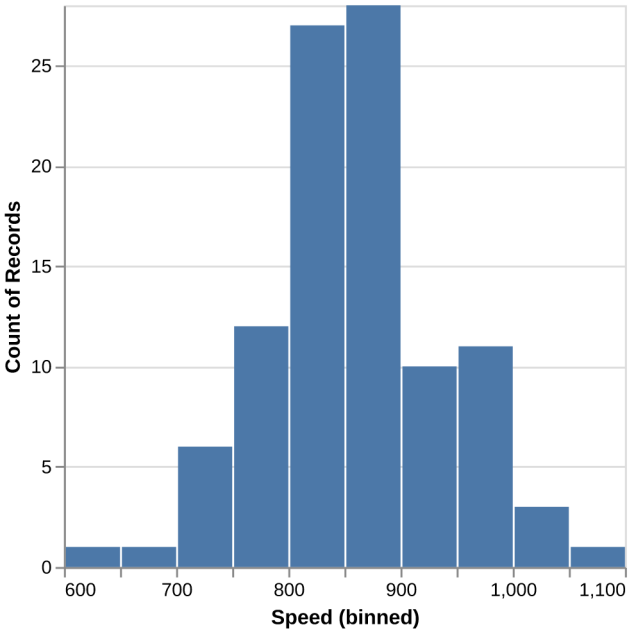


FIGURE 4.22 Histogram of Michelson's speed of light data.

4.5.5.1 Adding layers to an `altair` chart

Fig. 4.22 is a great start. However, we cannot tell how accurate the measurements are using this visualization unless we can see the true value. In order to visualize the true speed of light, we will add a vertical line with the `mark_rule` function. To draw a vertical line with `mark_rule`, we need to specify where on the x-axis the line should be drawn. We can do this by providing `x=alt.datum(792.458)`, where the value `792.458` is the true speed of light minus 299,000 and `alt.datum` tells altair that we have a single datum (number) that we would like plotted (rather than a column in the data frame). Similarly, a horizontal line can be plotted using the `y` axis encoding and the data frame with one value, which would act as the be the y-intercept. Note that *vertical lines* are used to denote quantities on the *horizontal axis*, while *horizontal lines* are used to denote quantities on the *vertical axis*.

To fine tune the appearance of this vertical line, we can change it from a solid to a dashed line with `strokeDash=[5]`, where `5` indicates the length of each dash. We also change the thickness of the line by specifying `size=2`. To add the dashed line on top of the histogram, we **add** the `mark_rule` chart to the `morley_hist` using the `+` operator. Adding features to a plot using the `+` operator is known as *layering* in `altair`. This is a powerful feature of `altair`; you can continue to iterate on a single chart, adding and refining one layer at a time. If you stored your chart as a variable using the assignment

symbol (=), you can add to it using the + operator. Below we add a vertical line created using `mark_rule` to the `morley_hist` we created previously.

Note: Technically we could have left out the data argument when creating the rule chart since we're not using any values from the `morley_df` data frame, but we will need it later when we facet this layered chart, so we are including it here already.

```
v_line = alt.Chart(morley_df).mark_rule(strokeDash=[6], size=1.5).encode(
    x=alt.datum(792.458)
)

morley_hist_line = morley_hist + v_line
```

In Fig. 4.23, we still cannot tell which experiments (denoted by the `Expt` column) led to which measurements; perhaps some experiments were more accurate than others. To fully answer our question, we need to separate the measurements from each other visually. We can try to do this using a *colored* histogram, where counts from different experiments are stacked on top of each other in different colors. We can create a histogram colored by the `Expt` variable by adding it to the `color` argument.

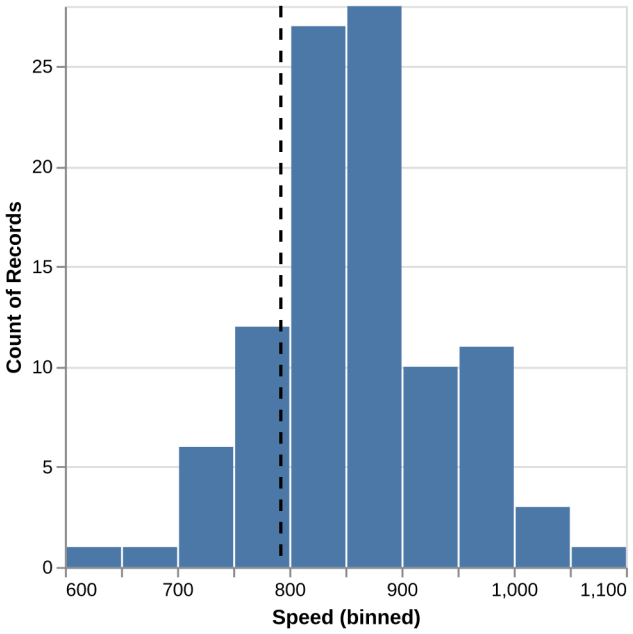


FIGURE 4.23 Histogram of Michelson's speed of light data with vertical line indicating the true speed of light.

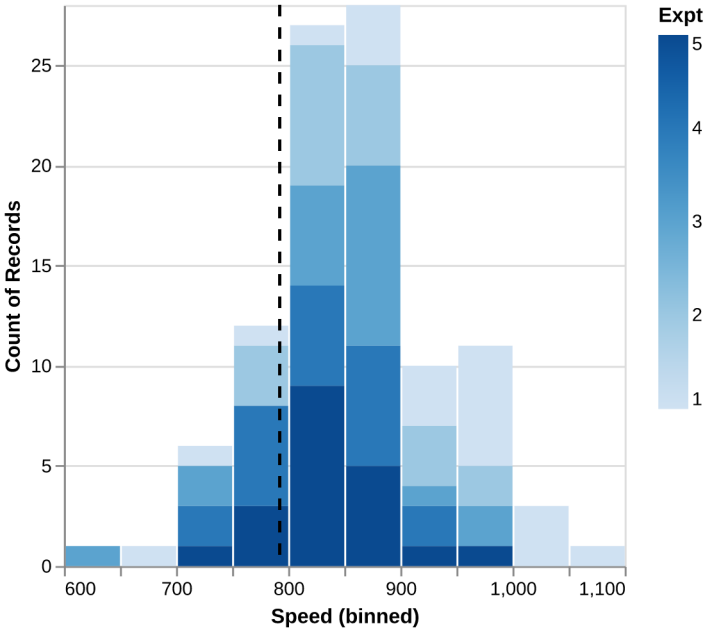


FIGURE 4.24 Histogram of Michelson's speed of light data colored by experiment.

```
morley_hist_colored = alt.Chart(morley_df).mark_bar().encode(
    x=alt.X("Speed").bin(),
    y="count()",
    color="Expt"
)

morley_hist_colored = morley_hist_colored + v_line
```

Alright great, Fig. 4.24 looks ... wait a second! We are not able to easily distinguish between the colors of the different Experiments in the histogram. What is going on here? Well, if you recall from Chapter 3, the *data type* you use for each variable can influence how Python and `altair` treats it. Here, we indeed have an issue with the data types in the `morley` data frame. In particular, the `Expt` column is currently an *integer*—specifically, an `int64` type. But we want to treat it as a *category*, i.e., there should be one category per type of experiment.

```
morley_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Expt    100 non-null    int64
 1   Run     100 non-null    int64
 2   Speed   100 non-null    int64
```

(continues on next page)

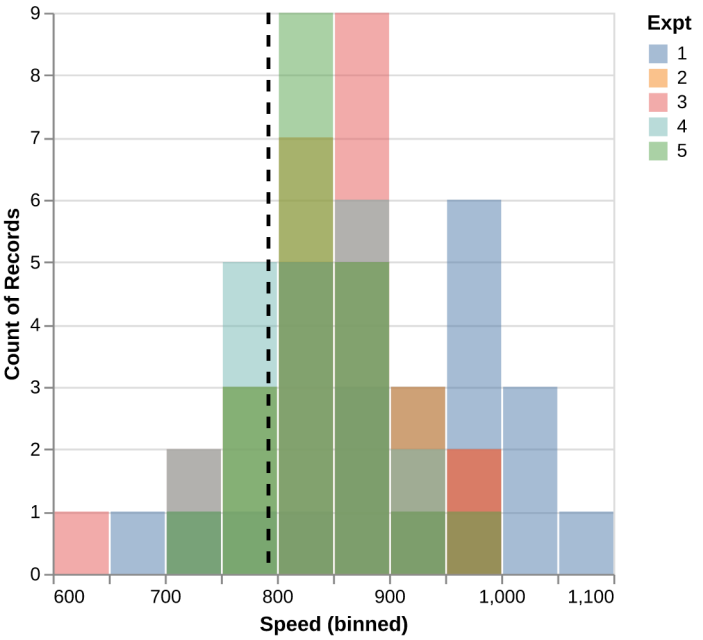


FIGURE 4.25 Histogram of Michelson's speed of light data colored by experiment as a categorical variable.

(continued from previous page)

```
dtypes: int64(3)
memory usage: 2.5 KB
```

To fix this issue we can convert the `Expt` variable into a `nominal` (i.e., categorical) type variable by adding a suffix `:N` to the `Expt` variable. Adding the `:N` suffix ensures that `altair` will treat a variable as a categorical variable, and hence use a discrete color map in visualizations (read more about data types in the altair documentation[10]). We also add the `stack(False)` method on the `y` encoding so that the bars are not stacked on top of each other, but instead share the same baseline. We try to ensure that the different colors can be seen despite them sitting in front of each other by setting the `opacity` argument in `mark_bar` to `0.5` to make the bars slightly translucent.

```
morley_hist_categorical = alt.Chart(morley_df).mark_bar(opacity=0.5).encode(
    x=alt.X("Speed").bin(),
    y=alt.Y("count()").stack(False),
    color="Expt:N"
)

morley_hist_categorical = morley_hist_categorical + v_line
```

Unfortunately, the attempt to separate out the experiment number visually has created a bit of a mess. All of the colors in Fig. 4.25 are blending together,

[10] https://altair-viz.github.io/user_guide/encodings/index.html#encoding-data-types

and although it is possible to derive *some* insight from this (e.g., experiments 1 and 3 had some of the most incorrect measurements), it isn't the clearest way to convey our message and answer the question. Let's try a different strategy of creating grid of separate histogram plots.

We can use the `facet` function to create a chart that has multiple subplots arranged in a grid. The argument to `facet` specifies the variable(s) used to split the plot into subplots (`Expt` in the code below), and how many columns there should be in the grid. In this example, we chose to arrange our plots in a single column (`columns=1`) since this makes it easier for us to compare the location of the histograms along the `x`-axis in the different subplots. We also reduce the height of each chart so that they all fit in the same view. Note that we are re-using the chart we created just above, instead of re-creating the same chart from scratch. We also explicitly specify that `facet` is a categorical variable since faceting should only be done with categorical variables.

```
morley_hist_facet = morley_hist_categorical.properties(
    height=100
).facet(
    "Expt:N",
    columns=1
)
```

The visualization in Fig. 4.26 makes it clear how accurate the different experiments were with respect to one another. The most variable measurements came from Experiment 1, where the measurements ranged from about 650–1050 km/sec. The least variable measurements came from Experiment 2, where the measurements ranged from about 750–950 km/sec. The most different experiments still obtained quite similar overall results.

There are three finishing touches to make this visualization even clearer. First and foremost, we need to add informative axis labels using the `alt.X` and `alt.Y` function, and increase the font size to make it readable using the `configure_axis` function. We can also add a title; for a `facet` plot, this is done by providing the `title` to the facet function. Finally, and perhaps most subtly, even though it is easy to compare the experiments on this plot to one another, it is hard to get a sense of just how accurate all the experiments were overall. For example, how accurate is the value 800 on the plot, relative to the true speed of light? To answer this question, we'll transform our data to a relative measure of error rather than an absolute measurement.

```
speed_of_light = 299792.458
morley_df["RelativeError"] = (
    100 * (299000 + morley_df["Speed"] - speed_of_light) / speed_of_light
)
morley_df
```

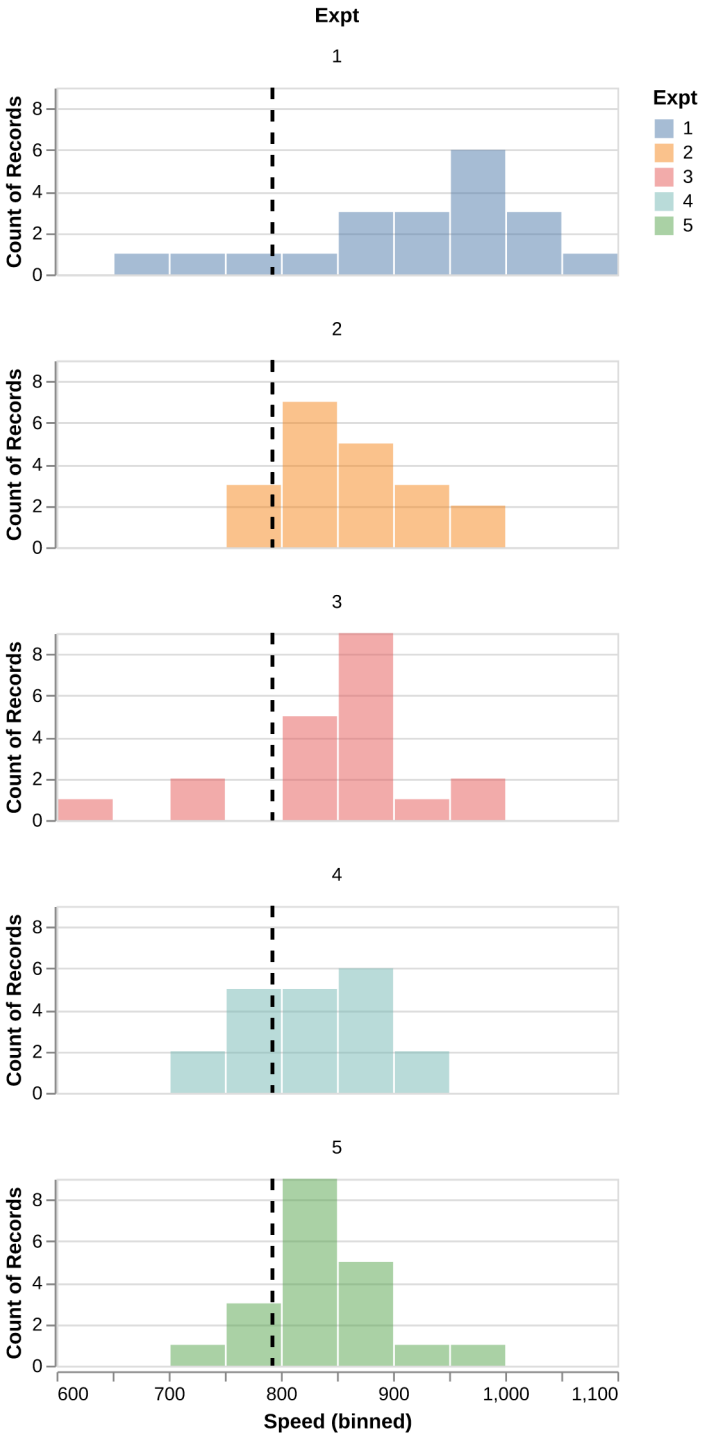


FIGURE 4.26 Histogram of Michelson's speed of light data split vertically by experiment.

```
    Expt  Run  Speed  RelativeError
0      1    1    850       0.019194
1      1    2    740      -0.017498
2      1    3    900       0.035872
3      1    4   1070       0.092578
4      1    5    930       0.045879
..   ...  ...    ...            ...
95     5   16    940       0.049215
96     5   17    950       0.052550
97     5   18    800       0.002516
98     5   19    810       0.005851
99     5   20    870       0.025865

[100 rows x 4 columns]
```

```
morley_hist_rel = alt.Chart(morley_df).mark_bar().encode(
    x=alt.X("RelativeError")
        .bin()
        .title("Relative Error (%)"),
    y=alt.Y("count()").title("# Measurements"),
    color=alt.Color("Expt:N").title("Experiment ID")
)

# Recreating v_line to indicate that the speed of light is at 0% relative error
v_line = alt.Chart(morley_df).mark_rule(strokeDash=[6], size=1.5).encode(
    x=alt.datum(0)
)

morley_hist_relative = (morley_hist_rel + v_line).properties(
    height=100
).facet(
    "Expt:N",
    columns=1,
    title="Histogram of relative error of Michelson's speed of light data"
)
```

Wow, impressive! These measurements of the speed of light from 1879 had errors around *0.05%* of the true speed. Fig. 4.27 shows you that even though experiments 2 and 5 were perhaps the most accurate, all of the experiments did quite an admirable job given the technology available at the time.

4.5.5.2 Choosing a binwidth for histograms

When you create a histogram in `altair`, it tries to choose a reasonable number of bins. We can change the number of bins by using the `maxbins` parameter inside the `bin` method (Fig. 4.28).

```
morley_hist_maxbins = alt.Chart(morley_df).mark_bar().encode(
    x=alt.X("RelativeError").bin(maxbins=30),
    y="count()"
)
```

But what number of bins is the right one to use? Unfortunately there is no hard rule for what the right bin number or width is. It depends entirely on your problem; the *right* number of bins or bin width is the one that *helps you*

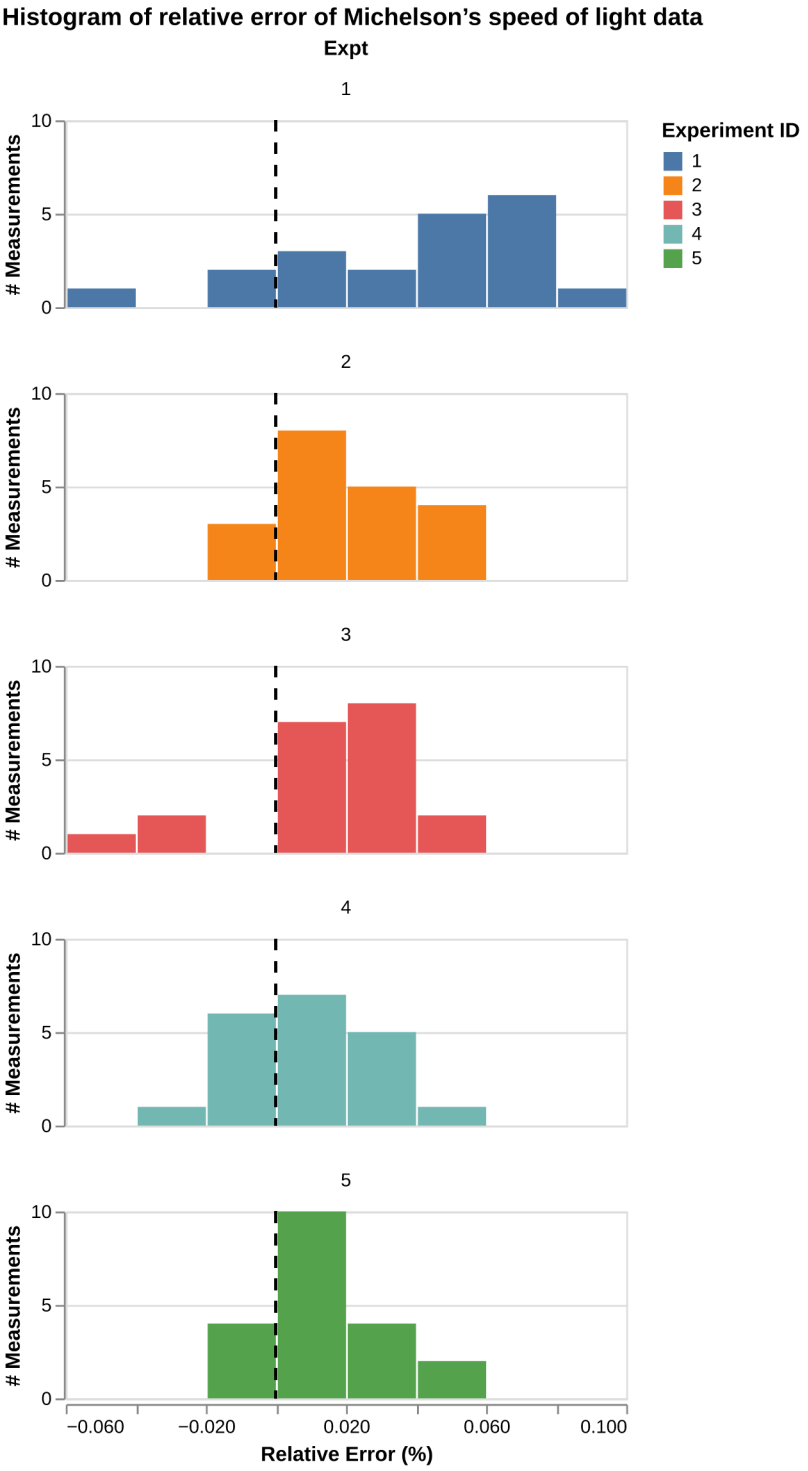


FIGURE 4.27 Histogram of relative error split vertically by experiment with clearer axes and labels.

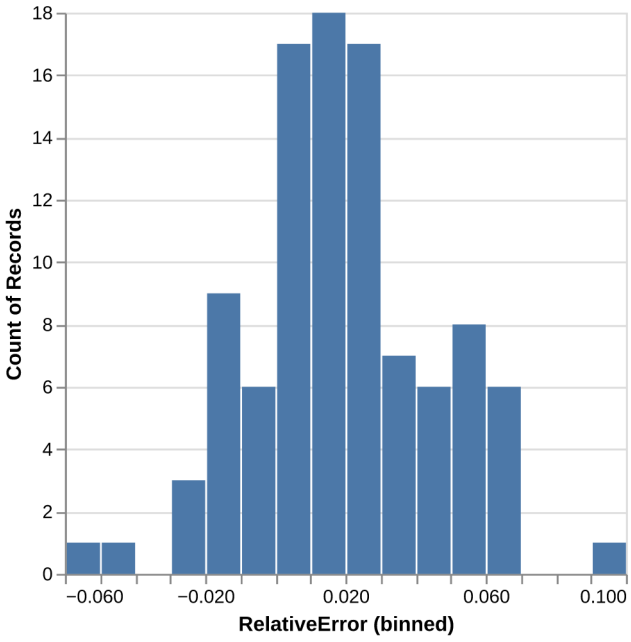


FIGURE 4.28 Histogram of Michelson's speed of light data.

answer the question you asked. Choosing the correct setting for your problem is something that commonly takes iteration. It's usually a good idea to try out several `maxbins` to see which one most clearly captures your data in the context of the question you want to answer.

To get a sense for how different bin affect visualizations, let's experiment with the histogram that we have been working on in this section. In Fig. 4.29, we compare the default setting with three other histograms where we set the `maxbins` to 200, 70, and 5. In this case, we can see that both the default number of bins and the `maxbins=70` of are effective for helping to answer our question. On the other hand, the `maxbins=200` and `maxbins=5` are too small and too big, respectively.

4.6 Explaining the visualization

Tell a story

Typically, your visualization will not be shown entirely on its own, but rather it will be part of a larger presentation. Further, visualizations can provide supporting information for any aspect of a presentation, from opening to conclusion. For example, you could use an exploratory visualization in the

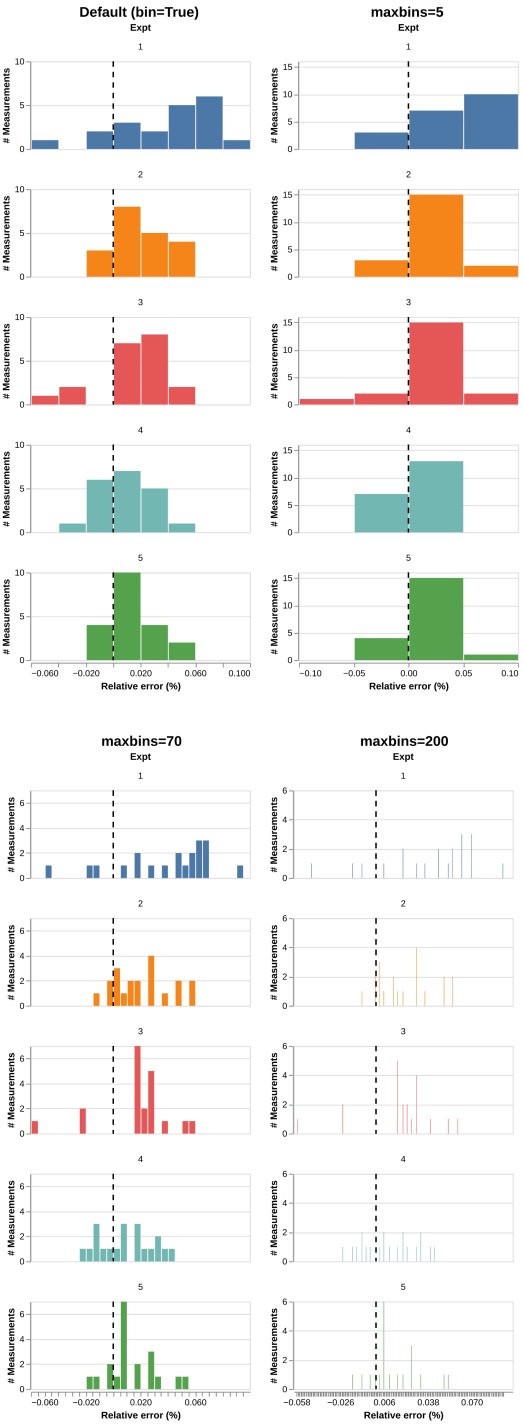


FIGURE 4.29 Effect of varying number of max bins on histograms.

opening of the presentation to motivate your choice of a more detailed data analysis / model, a visualization of the results of your analysis to show what your analysis has uncovered, or even one at the end of a presentation to help suggest directions for future work.

Regardless of where it appears, a good way to discuss your visualization is as a story:

1) Establish the setting and scope, and describe why you did what you did.

2) Pose the question that your visualization answers. Justify why the question is important to answer.

3) Answer the question using your visualization. Make sure you describe *all* aspects of the visualization (including describing the axes). But you can emphasize different aspects based on what is important to answer your question:

    - **trends (lines):** Does a line describe the trend well? If so, the trend is *linear*, and if not, the trend is *nonlinear*. Is the trend increasing, decreasing, or neither? Is there a periodic oscillation (wiggle) in the trend? Is the trend noisy (does the line "jump around" a lot) or smooth?

    - **distributions (scatters, histograms):** How spread out are the data? Where are they centered, roughly? Are there any obvious "clusters" or "subgroups", which would be visible as multiple bumps in the histogram?

    - **distributions of two variables (scatters):** Is there a clear / strong relationship between the variables (points fall in a distinct pattern), a weak one (points fall in a pattern but there is some noise), or no discernible relationship (the data are too noisy to make any conclusion)?

    - **amounts (bars):** How large are the bars relative to one another? Are there patterns in different groups of bars?

4) Summarize your findings, and use them to motivate whatever you will discuss next.

Below are two examples of how one might take these four steps in describing the example visualizations that appeared earlier in this chapter. Each of the steps is denoted by its numeral in parentheses, e.g. (3).

Mauna Loa Atmospheric CO_2 Measurements: (1) Many current forms of energy generation and conversion—from automotive engines to natural gas

power plants—rely on burning fossil fuels and produce greenhouse gases, typically primarily carbon dioxide (CO_2), as a byproduct. Too much of these gases in the Earth's atmosphere will cause it to trap more heat from the sun, leading to global warming. (2) In order to assess how quickly the atmospheric concentration of CO_2 is increasing over time, we (3) used a data set from the Mauna Loa observatory in Hawaii, consisting of CO_2 measurements from 1980 to 2020. We plotted the measured concentration of CO_2 (on the vertical axis) over time (on the horizontal axis). From this plot, you can see a clear, increasing, and generally linear trend over time. There is also a periodic oscillation that occurs once per year and aligns with Hawaii's seasons, with an amplitude that is small relative to the growth in the overall trend. This shows that atmospheric CO_2 is clearly increasing over time, and (4) it is perhaps worth investigating more into the causes.

Michelson Light Speed Experiments: (1) Our modern understanding of the physics of light has advanced significantly from the late 1800s when Michelson and Morley's experiments first demonstrated that it had a finite speed. We now know, based on modern experiments, that it moves at roughly 299,792.458 kilometers per second. (2) But how accurately were we first able to measure this fundamental physical constant, and did certain experiments produce more accurate results than others? (3) To better understand this, we plotted data from 5 experiments by Michelson in 1879, each with 20 trials, as histograms stacked on top of one another. The horizontal axis shows the error of the measurements relative to the true speed of light as we know it today, expressed as a percentage. From this visualization, you can see that most results had relative errors of at most 0.05%. You can also see that experiments 1 and 3 had measurements that were the farthest from the true value, and experiment 5 tended to provide the most consistently accurate result. (4) It would be worth further investigating the differences between these experiments to see why they produced different results.

4.7 Saving the visualization

Choose the right output format for your needs

Just as there are many ways to store data sets, there are many ways to store visualizations and images. Which one you choose can depend on several factors, such as file size/type limitations (e.g., if you are submitting your visualization as part of a conference paper or to a poster printing shop) and where it will be displayed (e.g., online, in a paper, on a poster, on a billboard, in talk slides).

Generally speaking, images come in two flavors: *raster* formats and *vector* formats.

Raster images are represented as a 2D grid of square pixels, each with its own color. Raster images are often *compressed* before storing so they take up less space. A compressed format is *lossy* if the image cannot be perfectly re-created when loading and displaying, with the hope that the change is not noticeable. *Lossless* formats, on the other hand, allow a perfect display of the original image.

- *Common file types:*
  - JPEG[11] (`.jpg`, `.jpeg`): lossy, usually used for photographs
  - PNG[12] (`.png`): lossless, usually used for plots / line drawings
  - BMP[13] (`.bmp`): lossless, raw image data, no compression (rarely used)
  - TIFF[14] (`.tif`, `.tiff`): typically lossless, no compression, used mostly in graphic arts, publishing
- *Open-source software:* GIMP[15]

Vector images are represented as a collection of mathematical objects (lines, surfaces, shapes, curves). When the computer displays the image, it redraws all of the elements using their mathematical formulas.

- *Common file types:*
  - SVG[16] (`.svg`): general-purpose use
  - EPS[17] (`.eps`), general-purpose use (rarely used)
- *Open-source software:* Inkscape[18]

Raster and vector images have opposing advantages and disadvantages. A raster image of a fixed width / height takes the same amount of space and time to load regardless of what the image shows (the one caveat is that the compression algorithms may shrink the image more or run faster for certain images). A vector image takes space and time to load corresponding to how complex the image is, since the computer has to draw all the elements each

[11] https://en.wikipedia.org/wiki/JPEG
[12] https://en.wikipedia.org/wiki/Portable_Network_Graphics
[13] https://en.wikipedia.org/wiki/BMP_file_format
[14] https://en.wikipedia.org/wiki/TIFF
[15] https://www.gimp.org/
[16] https://en.wikipedia.org/wiki/Scalable_Vector_Graphics
[17] https://en.wikipedia.org/wiki/Encapsulated_PostScript
[18] https://inkscape.org/

time it is displayed. For example, if you have a scatter plot with 1 million points stored as an SVG file, it may take your computer some time to open the image. On the other hand, you can zoom into / scale up vector graphics as much as you like without the image looking bad, while raster images eventually start to look "pixelated".

Note: The portable document format PDF[19] (`.pdf`) is commonly used to store *both* raster and vector formats. If you try to open a PDF and it's taking a long time to load, it may be because there is a complicated vector graphics image that your computer is rendering.

Let's learn how to save plot images to `.png` and `.svg` file formats using the `faithful_scatter_labels` scatter plot of the Old Faithful data set[20] [Hardle, 1991] that we created earlier, shown in Fig. 4.7. To save the plot to a file, we can use the `save` method. The `save` method takes the path to the filename where you would like to save the file (e.g., `img/viz/filename.png` to save a file named `filename.png` to the `img/viz/` directory). The kind of image to save is specified by the file extension. For example, to create a PNG image file, we specify that the file extension is `.png`. Below we demonstrate how to save PNG and SVG file types for the `faithful_scatter_labels` plot.

```
faithful_scatter_labels.save("img/viz/faithful_plot.png")
faithful_scatter_labels.save("img/viz/faithful_plot.svg")
```

TABLE 4.1 File sizes of the scatter plot of the Old Faithful data set when saved as different file formats.

Image type	File type	Image size
Raster	PNG	0.07 MB
Vector	SVG	0.09 MB

Take a look at the file sizes in Table 4.1. Wow, that's quite a difference! In this case, the `.png` image is almost 4 times smaller than the `.svg` image. Since there are a decent number of points in the plot, the vector graphics format image (`.svg`) is bigger than the raster image (`.png`), which just stores the image data itself. In Fig. 4.30, we show what the images look like when we zoom in to a rectangle with only 3 data points. You can see why vector graphics formats are so useful: because they're just based on mathematical

[19] https://en.wikipedia.org/wiki/PDF
[20] https://www.stat.cmu.edu/~larry/all-of-statistics/=data/faithful.dat

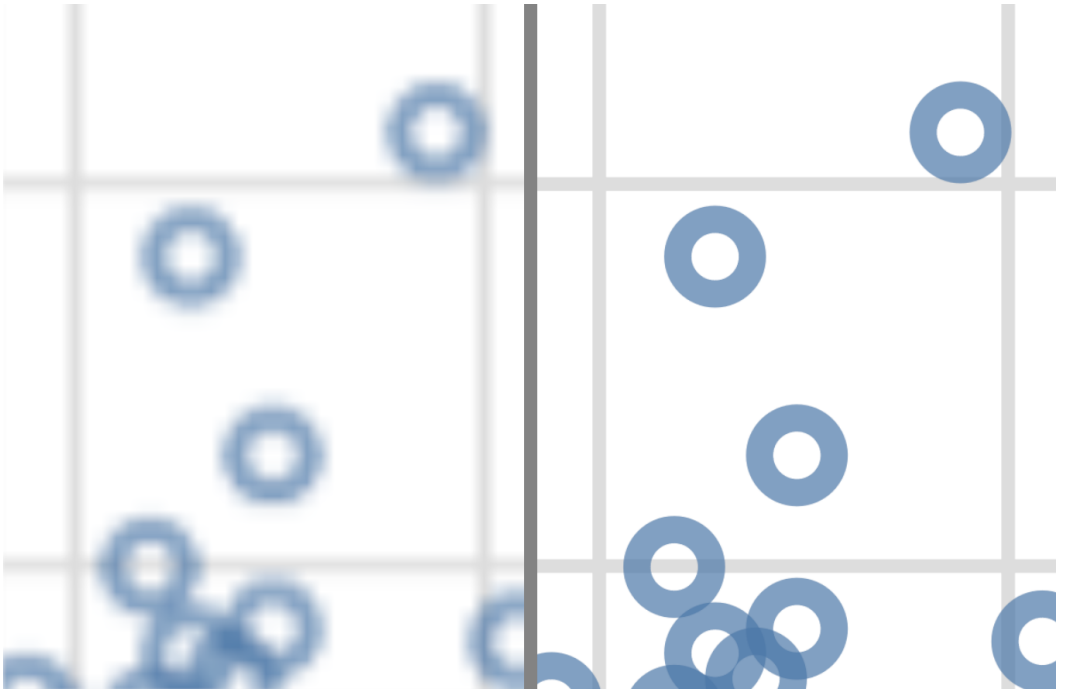


FIGURE 4.30 Zoomed in `faithful`, raster (PNG, left) and vector (SVG, right) formats.

formulas, vector graphics can be scaled up to arbitrary sizes. This makes them great for presentation media of all sizes, from papers to posters to billboards.

4.8 Exercises

Practice exercises for the material covered in this chapter can be found in the accompanying worksheets repository[21] in the "Effective data visualization" row. You can launch an interactive version of the worksheet in your browser by clicking the "launch binder" button. You can also preview a non-interactive version of the worksheet by clicking "view worksheet". If you instead decide to download the worksheet and run it on your own machine, make sure to follow the instructions for computer setup found in Chapter 13. This will ensure that the automated feedback and guidance that the worksheets provide will function as intended.

[21] https://worksheets.python.datasciencebook.ca

4.9 Additional resources

- The altair documentation[22] [VanderPlas *et al.*, 2018] is where you should look if you want to learn more about the functions in this chapter, the full set of arguments you can use, and other related functions.
- The *Fundamentals of Data Visualization*[23] [Wilke, 2019] has a wealth of information on designing effective visualizations. It is not specific to any particular programming language or library. If you want to improve your visualization skills, this is the next place to look.
- The dates and times[24] chapter of *Python for Data Analysis*[25] [McKinney, 2012] is where you should look if you want to learn about `date` and `time`, including how to create them, and how to use them to effectively handle durations, etc.

[22] https://altair-viz.github.io/
[23] https://clauswilke.com/dataviz/
[24] https://wesmckinney.com/book/time-series.html
[25] https://wesmckinney.com/book/

5

Classification I: training & predicting

5.1 Overview

In previous chapters, we focused solely on descriptive and exploratory data analysis questions. This chapter and the next together serve as our first foray into answering *predictive* questions about data. In particular, we will focus on *classification*, i.e., using one or more variables to predict the value of a categorical variable of interest. This chapter will cover the basics of classification, how to preprocess data to make it suitable for use in a classifier, and how to use our observed data to make predictions. The next chapter will focus on how to evaluate how accurate the predictions from our classifier are, as well as how to improve our classifier (where possible) to maximize its accuracy.

5.2 Chapter learning objectives

By the end of the chapter, readers will be able to do the following:

- Recognize situations where a classifier would be appropriate for making predictions.
- Describe what a training data set is and how it is used in classification.
- Interpret the output of a classifier.
- Compute, by hand, the straight-line (Euclidean) distance between points on a graph when there are two predictor variables.
- Explain the K-nearest neighbors classification algorithm.
- Perform K-nearest neighbors classification in Python using `scikit-learn`.

DOI: 10.1201/9781003438397-5

- Use methods from `scikit-learn` to center, scale, balance, and impute data as a preprocessing step.
- Combine preprocessing and model training into a `Pipeline` using `make_pipeline`.

5.3 The classification problem

In many situations, we want to make predictions based on the current situation as well as past experiences. For instance, a doctor may want to diagnose a patient as either diseased or healthy based on their symptoms and the doctor's past experience with patients; an email provider might want to tag a given email as "spam" or "not spam" based on the email's text and past email text data; or a credit card company may want to predict whether a purchase is fraudulent based on the current purchase item, amount, and location as well as past purchases. These tasks are all examples of **classification**, i.e., predicting a categorical class (sometimes called a *label*) for an observation given its other variables (sometimes called *features*).

Generally, a classifier assigns an observation without a known class (e.g., a new patient) to a class (e.g., diseased or healthy) on the basis of how similar it is to other observations for which we do know the class (e.g., previous patients with known diseases and symptoms). These observations with known classes that we use as a basis for prediction are called a **training set**; this name comes from the fact that we use these data to train, or teach, our classifier. Once taught, we can use the classifier to make predictions on new data for which we do not know the class.

There are many possible methods that we could use to predict a categorical class/label for an observation. In this book, we will focus on the widely used **K-nearest neighbors** algorithm [Cover and Hart, 1967; Fix and Hodges, 1951]. In your future studies, you might encounter decision trees, support vector machines (SVMs), logistic regression, neural networks, and more; see the additional resources section at the end of the next chapter for where to begin learning more about these other methods. It is also worth mentioning that there are many variations on the basic classification problem. For example, we focus on the setting of **binary classification** where only two classes are involved (e.g., a diagnosis of either healthy or diseased), but you may also run into multiclass classification problems with more than two categories (e.g., a diagnosis of healthy, bronchitis, pneumonia, or a common cold).

5.4 Exploring a data set

In this chapter and the next, we will study a data set of digitized breast cancer image features[1], created by Dr. William H. Wolberg, W. Nick Street, and Olvi L. Mangasarian [Street *et al.*, 1993]. Each row in the data set represents an image of a tumor sample, including the diagnosis (benign or malignant) and several other measurements (nucleus texture, perimeter, area, and more). Diagnosis for each image was conducted by physicians.

As with all data analyses, we first need to formulate a precise question that we want to answer. Here, the question is *predictive*: can we use the tumor image measurements available to us to predict whether a future tumor image (with unknown diagnosis) shows a benign or malignant tumor? Answering this question is important because traditional, non-data-driven methods for tumor diagnosis are quite subjective and dependent upon how skilled and experienced the diagnosing physician is. Furthermore, benign tumors are not normally dangerous; the cells stay in the same place, and the tumor stops growing before it gets very large. By contrast, in malignant tumors, the cells invade the surrounding tissue and spread into nearby organs, where they can cause serious damage [Stanford Health Care, 2021]. Thus, it is important to quickly and accurately diagnose the tumor type to guide patient treatment.

5.4.1 Loading the cancer data

Our first step is to load, wrangle, and explore the data using visualizations in order to better understand the data we are working with. We start by loading the `pandas` and `altair` packages needed for our analysis.

```
import pandas as pd
import altair as alt
```

In this case, the file containing the breast cancer data set is a `.csv` file with headers. We'll use the `read_csv` function with no additional arguments, and then inspect its contents:

```
cancer = pd.read_csv("data/wdbc.csv")
cancer
```

```
          ID Class    Radius   Texture  Perimeter      Area  Smoothness  \
0     842302     M  1.096100 -2.071512   1.268817  0.983510    1.567087
1     842517     M  1.828212 -0.353322   1.684473  1.907030   -0.826235
```

(continues on next page)

[1] https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29

(continued from previous page)

```
2    84300903     M  1.578499  0.455786   1.565126  1.557513    0.941382
3    84348301     M -0.768233  0.253509  -0.592166 -0.763792    3.280667
4    84358402     M  1.748758 -1.150804   1.775011  1.824624    0.280125
..        ...   ...       ...       ...        ...       ...         ...
564    926424     M  2.109139  0.720838   2.058974  2.341795    1.040926
565    926682     M  1.703356  2.083301   1.614511  1.722326    0.102368
566    926954     M  0.701667  2.043775   0.672084  0.577445   -0.839745
567    927241     M  1.836725  2.334403   1.980781  1.733693    1.524426
568     92751     B -1.806811  1.220718  -1.812793 -1.346604   -3.109349

     Compactness  Concavity  Concave_Points  Symmetry  Fractal_Dimension
0       3.280628   2.650542        2.530249  2.215566           2.253764
1      -0.486643  -0.023825        0.547662  0.001391          -0.867889
2       1.052000   1.362280        2.035440  0.938859          -0.397658
3       3.399917   1.914213        1.450431  2.864862           4.906602
4       0.538866   1.369806        1.427237 -0.009552          -0.561956
..           ...        ...             ...       ...                ...
564     0.218868   1.945573        2.318924 -0.312314          -0.930209
565    -0.017817   0.692434        1.262558 -0.217473          -1.057681
566    -0.038646   0.046547        0.105684 -0.808406          -0.894800
567     3.269267   3.294046        2.656528  2.135315           1.042778
568    -1.149741  -1.113893       -1.260710 -0.819349          -0.560539

[569 rows x 12 columns]
```

5.4.2 Describing the variables in the cancer data set

Breast tumors can be diagnosed by performing a *biopsy*, a process where tissue is removed from the body and examined for the presence of disease. Traditionally these procedures were quite invasive; modern methods such as fine needle aspiration, used to collect the present data set, extract only a small amount of tissue and are less invasive. Based on a digital image of each breast tissue sample collected for this data set, ten different variables were measured for each cell nucleus in the image (items 3–12 of the list of variables below), and then the mean for each variable across the nuclei was recorded. As part of the data preparation, these values have been *standardized (centered and scaled)*; we will discuss what this means and why we do it later in this chapter. Each image additionally was given a unique ID and a diagnosis by a physician. Therefore, the total set of variables per image in this data set is:

1. ID: identification number
2. Class: the diagnosis (M = malignant or B = benign)
3. Radius: the mean of distances from center to points on the perimeter
4. Texture: the standard deviation of gray-scale values
5. Perimeter: the length of the surrounding contour
6. Area: the area inside the contour

7. Smoothness: the local variation in radius lengths
8. Compactness: the ratio of squared perimeter and area
9. Concavity: severity of concave portions of the contour
10. Concave Points: the number of concave portions of the contour
11. Symmetry: how similar the nucleus is when mirrored
12. Fractal Dimension: a measurement of how "rough" the perimeter is

Below we use the `info` method to preview the data frame. This method can make it easier to inspect the data when we have a lot of columns: it prints only the column names down the page (instead of across), as well as their data types and the number of non-missing entries.

```
cancer.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   ID                 569 non-null    int64
 1   Class              569 non-null    object
 2   Radius             569 non-null    float64
 3   Texture            569 non-null    float64
 4   Perimeter          569 non-null    float64
 5   Area               569 non-null    float64
 6   Smoothness         569 non-null    float64
 7   Compactness        569 non-null    float64
 8   Concavity          569 non-null    float64
 9   Concave_Points     569 non-null    float64
 10  Symmetry           569 non-null    float64
 11  Fractal_Dimension  569 non-null    float64
dtypes: float64(10), int64(1), object(1)
memory usage: 53.5+ KB
```

From the summary of the data above, we can see that `Class` is of type `object`. We can use the `unique` method on the `Class` column to see all unique values present in that column. We see that there are two diagnoses: benign, represented by `"B"`, and malignant, represented by `"M"`.

```
cancer["Class"].unique()
```

```
array(['M', 'B'], dtype=object)
```

We will improve the readability of our analysis by renaming `"M"` to `"Malignant"` and `"B"` to `"Benign"` using the `replace` method. The `replace` method takes one argument: a dictionary that maps previous values to desired new values. We will verify the result using the `unique` method.

```
cancer["Class"] = cancer["Class"].replace({
    "M" : "Malignant",
    "B" : "Benign"
})

cancer["Class"].unique()
```

```
array(['Malignant', 'Benign'], dtype=object)
```

5.4.3 Exploring the cancer data

Before we start doing any modeling, let's explore our data set. Below we use the `groupby` and `size` methods to find the number and percentage of benign and malignant tumor observations in our data set. When paired with `groupby`, `size` counts the number of observations for each value of the `Class` variable. Then we calculate the percentage in each group by dividing by the total number of observations and multiplying by 100. The total number of observations equals the number of rows in the data frame, which we can access via the `shape` attribute of the data frame (`shape[0]` is the number of rows and `shape[1]` is the number of columns). We have 357 (63%) benign and 212 (37%) malignant tumor observations.

```
100 * cancer.groupby("Class").size() / cancer.shape[0]
```

```
Class
Benign       62.741652
Malignant    37.258348
dtype: float64
```

The `pandas` package also has a more convenient specialized `value_counts` method for counting the number of occurrences of each value in a column. If we pass no arguments to the method, it outputs a series containing the number of occurrences of each value. If we instead pass the argument `normalize=True`, it instead prints the fraction of occurrences of each value.

```
cancer["Class"].value_counts()
```

```
Class
Benign       357
Malignant    212
Name: count, dtype: int64
```

```
cancer["Class"].value_counts(normalize=True)
```

```
Class
Benign       0.627417
Malignant    0.372583
Name: proportion, dtype: float64
```

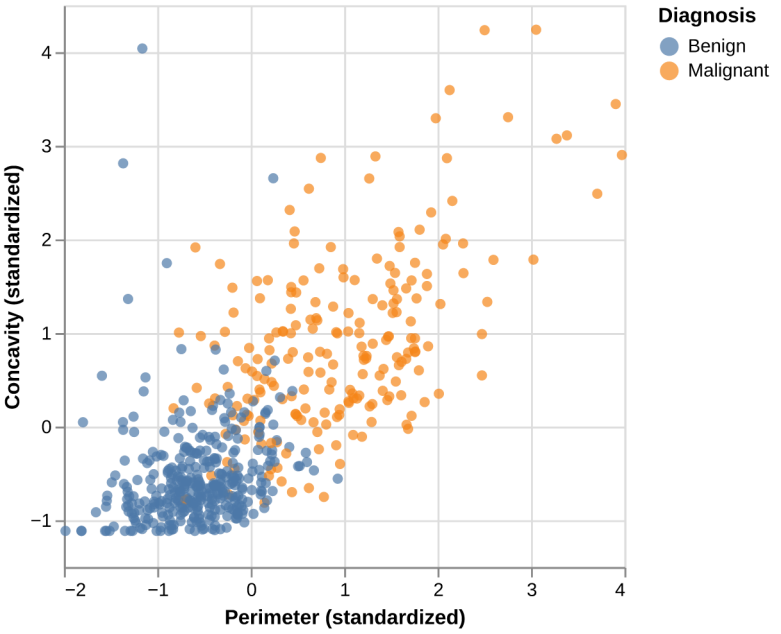


FIGURE 5.1 Scatter plot of concavity versus perimeter colored by diagnosis label.

Next, let's draw a colored scatter plot to visualize the relationship between the perimeter and concavity variables. Recall that the default palette in `altair` is colorblind-friendly, so we can stick with that here.

```
perim_concav = alt.Chart(cancer).mark_circle().encode(
    x=alt.X("Perimeter").title("Perimeter (standardized)"),
    y=alt.Y("Concavity").title("Concavity (standardized)"),
    color=alt.Color("Class").title("Diagnosis")
)
perim_concav
```

In Fig. 5.1, we can see that malignant observations typically fall in the upper right-hand corner of the plot area. By contrast, benign observations typically fall in the lower left-hand corner of the plot. In other words, benign observations tend to have lower concavity and perimeter values, and malignant ones tend to have larger values. Suppose we obtain a new observation not in the current data set that has all the variables measured *except* the label (i.e., an image without the physician's diagnosis for the tumor class). We could compute the standardized perimeter and concavity values, resulting in values of, say, 1 and 1. Could we use this information to classify that observation as benign or malignant? Based on the scatter plot, how might you classify that new observation? If the standardized concavity and perimeter values are 1 and 1 respectively, the point would lie in the middle of the orange cloud of malignant points and thus we could probably classify it as malignant. Based

on our visualization, it seems like it may be possible to make accurate predictions of the `Class` variable (i.e., a diagnosis) for tumor images with unknown diagnoses.

5.5 Classification with K-nearest neighbors

In order to actually make predictions for new observations in practice, we will need a classification algorithm. In this book, we will use the K-nearest neighbors classification algorithm. To predict the label of a new observation (here, classify it as either benign or malignant), the K-nearest neighbors classifier generally finds the K "nearest" or "most similar" observations in our training set, and then uses their diagnoses to make a prediction for the new observation's diagnosis. K is a number that we must choose in advance; for now, we will assume that someone has chosen K for us. We will cover how to choose K ourselves in the next chapter.

To illustrate the concept of K-nearest neighbors classification, we will walk through an example. Suppose we have a new observation, with standardized perimeter of 2.0 and standardized concavity of 4.0, whose diagnosis "Class" is unknown. This new observation is depicted by the red, diamond point in Fig. 5.2.

Fig. 5.3 shows that the nearest point to this new observation is **malignant** and located at the coordinates (2.1, 3.6). The idea here is that if a point is close to another in the scatter plot, then the perimeter and concavity values are similar, and so we may expect that they would have the same diagnosis.

Suppose we have another new observation with standardized perimeter 0.2 and concavity of 3.3. Looking at the scatter plot in Fig. 5.4, how would you classify this red, diamond observation? The nearest neighbor to this new point is a **benign** observation at (0.2, 2.7). Does this seem like the right prediction to make for this observation? Probably not, if you consider the other nearby points.

To improve the prediction we can consider several neighboring points, say $K = 3$, that are closest to the new observation to predict its diagnosis class. Among those 3 closest points, we use the *majority class* as our prediction for the new observation. As shown in Fig. 5.5, we see that the diagnoses of 2 of the 3 nearest neighbors to our new observation are malignant. Therefore we take majority vote and classify our new red, diamond observation as malignant.

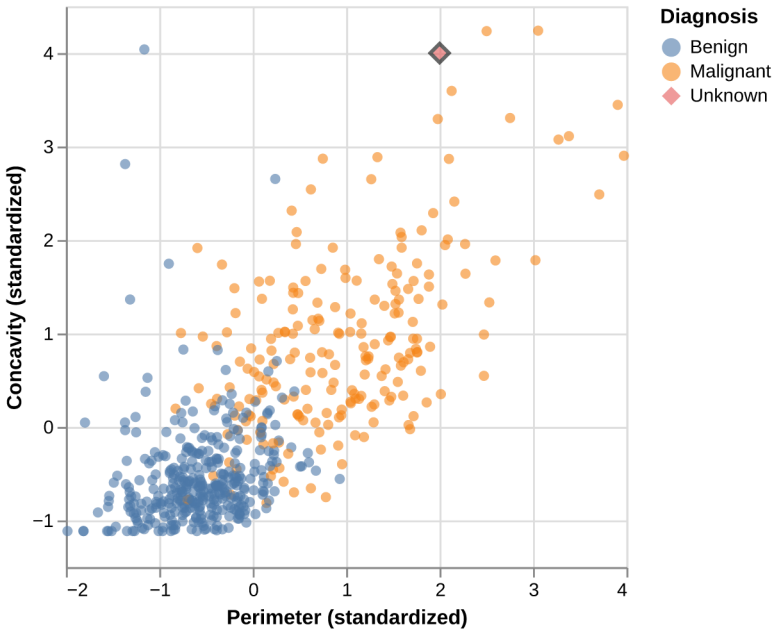


FIGURE 5.2 Scatter plot of concavity versus perimeter with new observation represented as a red diamond.

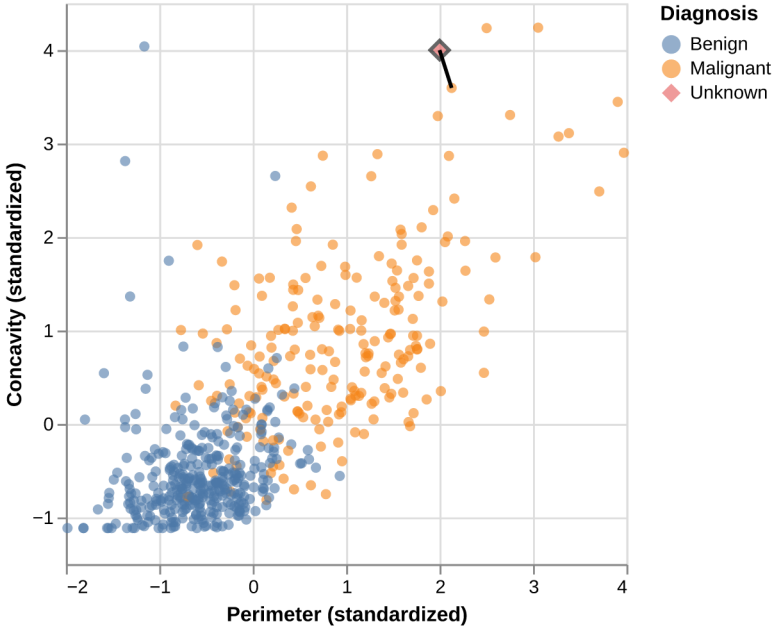


FIGURE 5.3 Scatter plot of concavity versus perimeter. The new observation is represented as a red diamond with a line to the one nearest neighbor, which has a malignant label.

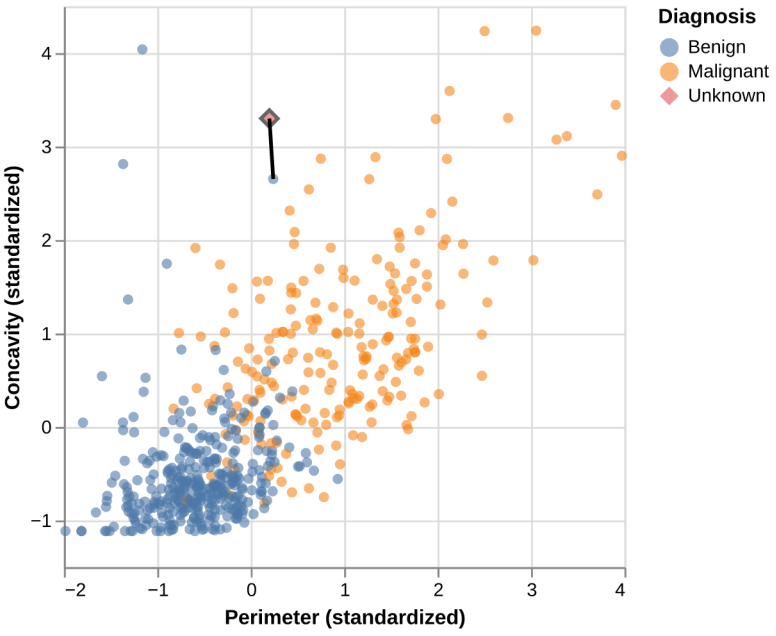


FIGURE 5.4 Scatter plot of concavity versus perimeter. The new observation is represented as a red diamond with a line to the one nearest neighbor, which has a benign label.

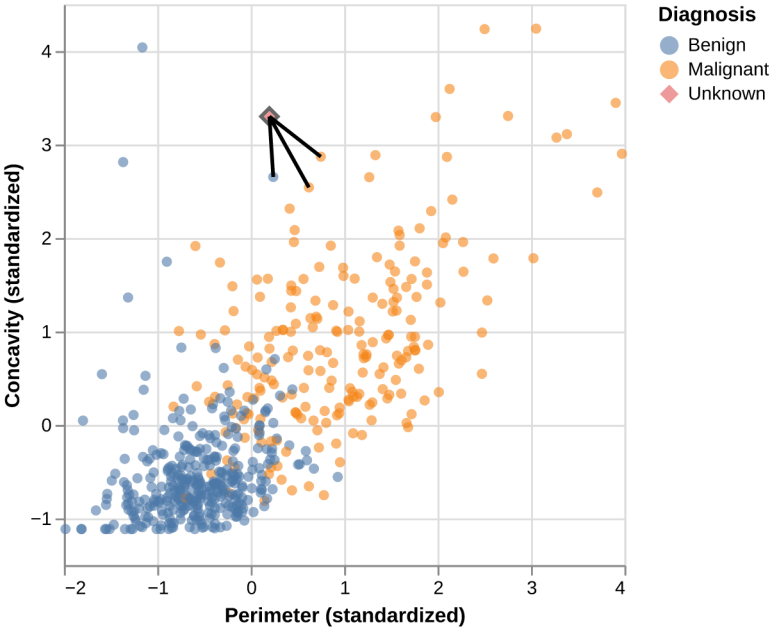


FIGURE 5.5 Scatter plot of concavity versus perimeter with three nearest neighbors.

Here we chose the $K = 3$ nearest observations, but there is nothing special about $K = 3$. We could have used $K = 4, 5$ or more (though we may want to choose an odd number to avoid ties). We will discuss more about choosing K in the next chapter.

5.5.1 Distance between points

We decide which points are the K "nearest" to our new observation using the *straight-line distance* (we will often just refer to this as *distance*). Suppose we have two observations a and b, each having two predictor variables, x and y. Denote a_x and a_y to be the values of variables x and y for observation a; b_x and b_y have similar definitions for observation b. Then the straight-line distance between observation a and b on the x-y plane can be computed using the following formula:

$$\text{Distance} = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$

To find the K nearest neighbors to our new observation, we compute the distance from that new observation to each observation in our training data, and select the K observations corresponding to the K *smallest* distance values. For example, suppose we want to use $K = 5$ neighbors to classify a new observation with perimeter 0.0 and concavity 3.5, shown as a red diamond in Fig. 5.6. Let's calculate the distances between our new point and each of the observations in the training set to find the $K = 5$ neighbors that are nearest to our new point. You will see in the code below, we compute the straight-line distance using the formula above: we square the differences between the two observations' perimeter and concavity coordinates, add the squared differences, and then take the square root. In order to find the $K = 5$ nearest neighbors, we will use the `nsmallest` function from `pandas`.

```
new_obs_Perimeter = 0
new_obs_Concavity = 3.5
cancer["dist_from_new"] = (
       (cancer["Perimeter"] - new_obs_Perimeter) ** 2
     + (cancer["Concavity"] - new_obs_Concavity) ** 2
)**(1/2)
cancer.nsmallest(5, "dist_from_new")[[
    "Perimeter",
    "Concavity",
    "Class",
    "dist_from_new"
]]
```

	Perimeter	Concavity	Class	dist_from_new
112	0.241202	2.653051	Benign	0.880626
258	0.750277	2.870061	Malignant	0.979663
351	0.622700	2.541410	Malignant	1.143088
430	0.416930	2.314364	Malignant	1.256806
152	-1.160091	4.039155	Benign	1.279258

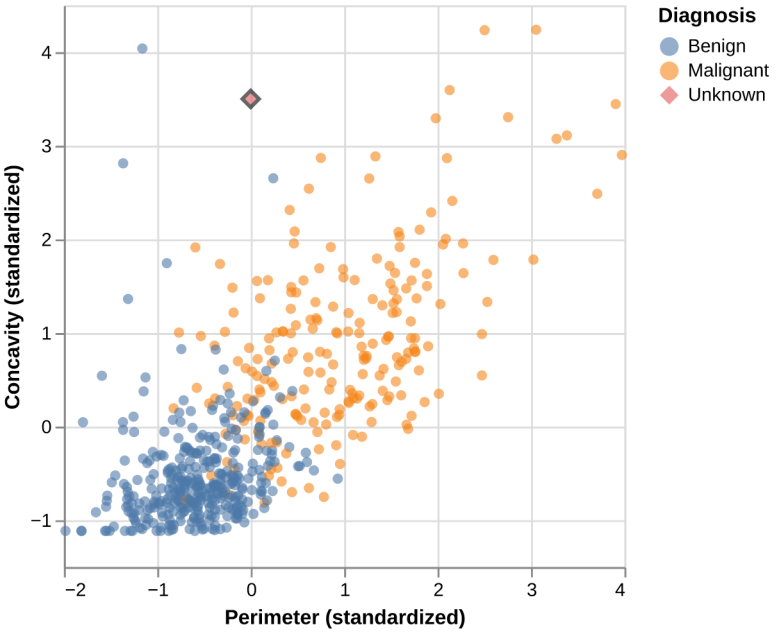


FIGURE 5.6 Scatter plot of concavity versus perimeter with new observation represented as a red diamond.

In Table 5.1 we show in mathematical detail how we computed the `dist_from_new` variable (the distance to the new observation) for each of the 5 nearest neighbors in the training data.

TABLE 5.1 Evaluating the distances from the new observation to each of its 5 nearest neighbors

Perimeter	Concavity	Distance	Class
0.24	2.65	$\sqrt{(0-0.24)^2+(3.5-2.65)^2} = 0.88$	Benign
0.75	2.87	$\sqrt{(0-0.75)^2+(3.5-2.87)^2} = 0.98$	Malignant
0.62	2.54	$\sqrt{(0-0.62)^2+(3.5-2.54)^2} = 1.14$	Malignant
0.42	2.31	$\sqrt{(0-0.42)^2+(3.5-2.31)^2} = 1.26$	Malignant
-1.16	4.04	$\sqrt{(0-(-1.16))^2+(3.5-4.04)^2} = 1.28$	Benign

The result of this computation shows that 3 of the 5 nearest neighbors to our new observation are malignant; since this is the majority, we classify our new observation as malignant. These 5 neighbors are circled in Fig. 5.7.

5.5.2 More than two explanatory variables

Although the above description is directed toward two predictor variables, exactly the same K-nearest neighbors algorithm applies when you have a higher

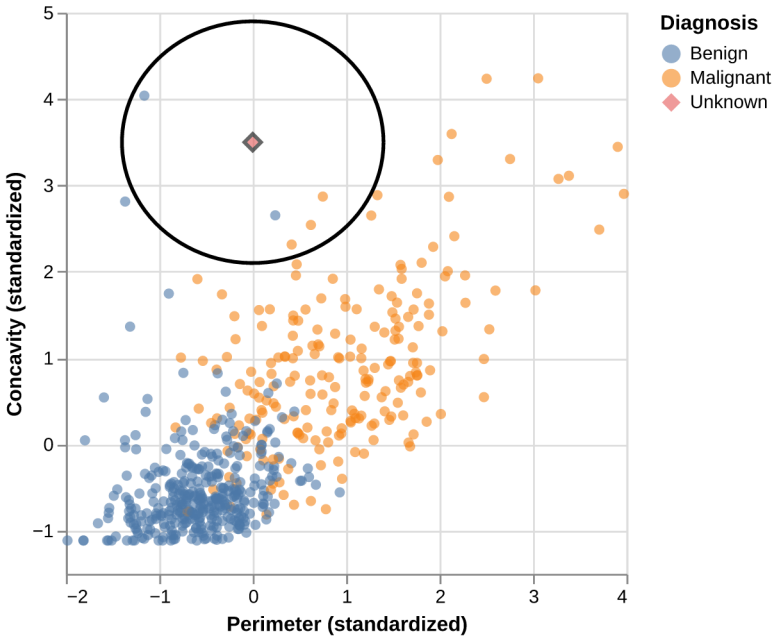


FIGURE 5.7 Scatter plot of concavity versus perimeter with 5 nearest neighbors circled.

number of predictor variables. Each predictor variable may give us new information to help create our classifier. The only difference is the formula for the distance between points. Suppose we have m predictor variables for two observations a and b, i.e., $a = (a_1, a_2, \ldots, a_m)$ and $b = (b_1, b_2, \ldots, b_m)$.

The distance formula becomes

$$\text{Distance} = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \cdots + (a_m - b_m)^2}.$$

This formula still corresponds to a straight-line distance, just in a space with more dimensions. Suppose we want to calculate the distance between a new observation with a perimeter of 0, concavity of 3.5, and symmetry of 1, and another observation with a perimeter, concavity, and symmetry of 0.417, 2.31, and 0.837 respectively. We have two observations with three predictor variables: perimeter, concavity, and symmetry. Previously, when we had two variables, we added up the squared difference between each of our (two) variables, and then took the square root. Now we will do the same, except for our three variables. We calculate the distance as follows

$$\text{Distance} = \sqrt{(0 - 0.417)^2 + (3.5 - 2.31)^2 + (1 - 0.837)^2} = 1.27.$$

Let's calculate the distances between our new observation and each of the observations in the training set to find the $K = 5$ neighbors when we have these three predictors.

```
new_obs_Perimeter = 0
new_obs_Concavity = 3.5
new_obs_Symmetry = 1
cancer["dist_from_new"] = (
      (cancer["Perimeter"] - new_obs_Perimeter) ** 2
    + (cancer["Concavity"] - new_obs_Concavity) ** 2
    + (cancer["Symmetry"] - new_obs_Symmetry) ** 2
)**(1/2)
cancer.nsmallest(5, "dist_from_new")[[
    "Perimeter",
    "Concavity",
    "Symmetry",
    "Class",
    "dist_from_new"
]]
```

	Perimeter	Concavity	Symmetry	Class	dist_from_new
430	0.416930	2.314364	0.836722	Malignant	1.267368
400	1.334664	2.886368	1.099359	Malignant	1.472326
562	0.470430	2.084810	1.154075	Malignant	1.499268
68	-1.365450	2.812359	1.092064	Benign	1.531594
351	0.622700	2.541410	2.055065	Malignant	1.555575

Based on $K = 5$ nearest neighbors with these three predictors we would classify the new observation as malignant since 4 out of 5 of the nearest neighbors are malignant class. Fig. 5.8 shows what the data look like when we visualize them as a 3D scatter with lines from the new observation to its five nearest neighbors.

5.5.3 Summary of K-nearest neighbors algorithm

In order to classify a new observation using a K-nearest neighbors classifier, we have to do the following:

1. Compute the distance between the new observation and each observation in the training set.
2. Find the K rows corresponding to the K smallest distances.
3. Classify the new observation based on a majority vote of the neighbor classes.

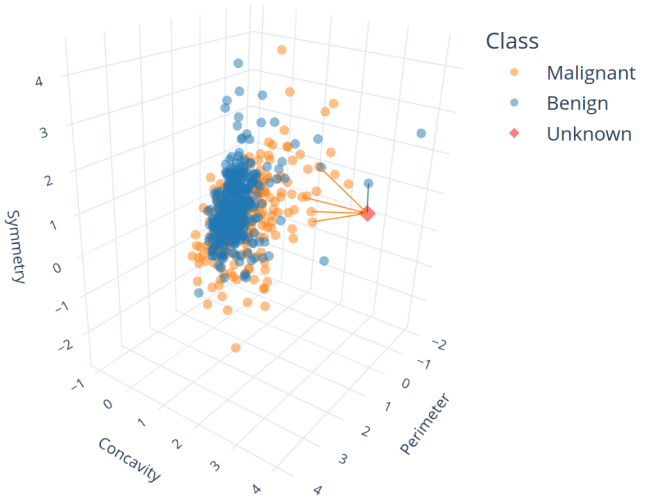


FIGURE 5.8 3D scatter plot of the standardized symmetry, concavity, and perimeter variables. Note that in general we recommend against using 3D visualizations; here we show the data in 3D only to illustrate what higher dimensions and nearest neighbors look like, for learning purposes.

5.6 K-nearest neighbors with `scikit-learn`

Coding the K-nearest neighbors algorithm in Python ourselves can get complicated, especially if we want to handle multiple classes, more than two variables, or predict the class for multiple new observations. Thankfully, in Python, the K-nearest neighbors algorithm is implemented in the `scikit-learn` Python package[2] [Buitinck *et al.*, 2013] along with many other models[3] that you will encounter in this and future chapters of the book. Using the functions in the `scikit-learn` package (named `sklearn` in Python) will help keep our code simple, readable and accurate; the less we have to code ourselves, the fewer mistakes we will likely make. Before getting started with K-nearest neighbors, we need to tell the `sklearn` package that we prefer using `pandas` data frames over regular arrays via the `set_config` function.

Note: You will notice a new way of importing functions in the code below: `from ... import ...`. This lets us import *just* `set_config` from `sklearn`, and then call `set_config` without any package prefix. We will import functions using `from` extensively throughout this and subsequent chapters to avoid

[2] https://scikit-learn.org/stable/index.html
[3] https://scikit-learn.org/stable/user_guide.html

very long names from `scikit-learn` that clutter the code (like `sklearn.neighbors.KNeighborsClassifier`, which has 38 characters!).

```
from sklearn import set_config

# Output dataframes instead of arrays
set_config(transform_output="pandas")
```

We can now get started with K-nearest neighbors. The first step is to import the `KNeighborsClassifier` from the `sklearn.neighbors` module.

```
from sklearn.neighbors import KNeighborsClassifier
```

Let's walk through how to use `KNeighborsClassifier` to perform K-nearest neighbors classification. We will use the `cancer` data set from above, with perimeter and concavity as predictors and $K = 5$ neighbors to build our classifier. Then we will use the classifier to predict the diagnosis label for a new observation with perimeter 0, concavity 3.5, and an unknown diagnosis label. Let's pick out our two desired predictor variables and class label and store them with the name `cancer_train`:

```
cancer_train = cancer[["Class", "Perimeter", "Concavity"]]
cancer_train
```

```
         Class  Perimeter  Concavity
0    Malignant   1.268817   2.650542
1    Malignant   1.684473  -0.023825
2    Malignant   1.565126   1.362280
3    Malignant  -0.592166   1.914213
4    Malignant   1.775011   1.369806
..         ...        ...        ...
564  Malignant   2.058974   1.945573
565  Malignant   1.614511   0.692434
566  Malignant   0.672084   0.046547
567  Malignant   1.980781   3.294046
568     Benign  -1.812793  -1.113893

[569 rows x 3 columns]
```

Next, we create a *model object* for K-nearest neighbors classification by creating a `KNeighborsClassifier` instance, specifying that we want to use $K = 5$ neighbors; we will discuss how to choose K in the next chapter.

Note: You can specify the `weights` argument in order to control how neighbors vote when classifying a new observation. The default is `"uniform"`, where each of the K nearest neighbors gets exactly 1 vote as described above.

Other choices, which weigh each neighbor's vote differently, can be found on the `scikit-learn` website[4].

```
knn = KNeighborsClassifier(n_neighbors=5)
knn
```

```
KNeighborsClassifier()
```

In order to fit the model on the breast cancer data, we need to call `fit` on the model object. The `X` argument is used to specify the data for the predictor variables, while the `y` argument is used to specify the data for the response variable. So below, we set `X=cancer_train[["Perimeter", "Concavity"]]` and `y=cancer_train["Class"]` to specify that `Class` is the response variable (the one we want to predict), and both `Perimeter` and `Concavity` are to be used as the predictors. Note that the `fit` function might look like it does not do much from the outside, but it is actually doing all the heavy lifting to train the K-nearest neighbors model, and modifies the `knn` model object.

```
knn.fit(X=cancer_train[["Perimeter", "Concavity"]], y=cancer_train["Class"]);
```

After using the `fit` function, we can make a prediction on a new observation by calling `predict` on the classifier object, passing the new observation itself. As above, when we ran the K-nearest neighbors classification algorithm manually, the `knn` model object classifies the new observation as "Malignant". Note that the `predict` function outputs an `array` with the model's prediction; you can actually make multiple predictions at the same time using the `predict` function, which is why the output is stored as an `array`.

```
new_obs = pd.DataFrame({"Perimeter": [0], "Concavity": [3.5]})
knn.predict(new_obs)
```

```
array(['Malignant'], dtype=object)
```

Is this predicted malignant label the actual class for this observation? Well, we don't know because we do not have this observation's diagnosis—that is what we were trying to predict. The classifier's prediction is not necessarily correct, but in the next chapter, we will learn ways to quantify how accurate we think our predictions are.

[4] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html?highlight=kneighborsclassifier#sklearn.neighbors.KNeighborsClassifier

5.7 Data preprocessing with `scikit-learn`

5.7.1 Centering and scaling

When using K-nearest neighbors classification, the *scale* of each variable (i.e., its size and range of values) matters. Since the classifier predicts classes by identifying observations nearest to it, any variables with a large scale will have a much larger effect than variables with a small scale. But just because a variable has a large scale *doesn't mean* that it is more important for making accurate predictions. For example, suppose you have a data set with two features, salary (in dollars) and years of education, and you want to predict the corresponding type of job. When we compute the neighbor distances, a difference of \$1000 is huge compared to a difference of 10 years of education. But for our conceptual understanding and answering of the problem, it's the opposite; 10 years of education is huge compared to a difference of \$1,000 in yearly salary.

In many other predictive models, the *center* of each variable (e.g., its mean) matters as well. For example, if we had a data set with a temperature variable measured in degrees Kelvin, and the same data set with temperature measured in degrees Celsius, the two variables would differ by a constant shift of 273 (even though they contain exactly the same information). Likewise, in our hypothetical job classification example, we would likely see that the center of the salary variable is in the tens of thousands, while the center of the years of education variable is in the single digits. Although this doesn't affect the K-nearest neighbors classification algorithm, this large shift can change the outcome of using many other predictive models.

To scale and center our data, we need to find our variables' *mean* (the average, which quantifies the "central" value of a set of numbers) and *standard deviation* (a number quantifying how spread out values are). For each observed value of the variable, we subtract the mean (i.e., center the variable) and divide by the standard deviation (i.e., scale the variable). When we do this, the data is said to be *standardized*, and all variables in a data set will have a mean of 0 and a standard deviation of 1. To illustrate the effect that standardization can have on the K-nearest neighbors algorithm, we will read in the original, unstandardized Wisconsin breast cancer data set; we have been using a standardized version of the data set up until now. We will apply the same initial wrangling steps as we did earlier, and to keep things simple we will just use the `Area`, `Smoothness`, and `Class` variables:

```
unscaled_cancer = pd.read_csv("data/wdbc_unscaled.csv")[["Class", "Area",
↪"Smoothness"]]
unscaled_cancer["Class"] = unscaled_cancer["Class"].replace({
   "M" : "Malignant",
   "B" : "Benign"
})
unscaled_cancer
```

```
          Class    Area  Smoothness
0     Malignant  1001.0     0.11840
1     Malignant  1326.0     0.08474
2     Malignant  1203.0     0.10960
3     Malignant   386.1     0.14250
4     Malignant  1297.0     0.10030
..          ...     ...         ...
564   Malignant  1479.0     0.11100
565   Malignant  1261.0     0.09780
566   Malignant   858.1     0.08455
567   Malignant  1265.0     0.11780
568      Benign   181.0     0.05263

[569 rows x 3 columns]
```

Looking at the unscaled and uncentered data above, you can see that the differences between the values for area measurements are much larger than those for smoothness. Will this affect predictions? In order to find out, we will create a scatter plot of these two predictors (colored by diagnosis) for both the unstandardized data we just loaded, and the standardized version of that same data. But first, we need to standardize the `unscaled_cancer` data set with `scikit-learn`.

The `scikit-learn` framework provides a collection of *preprocessors* used to manipulate data in the `preprocessing` module[5]. Here we will use the `StandardScaler` transformer to standardize the predictor variables in the `unscaled_cancer` data. In order to tell the `StandardScaler` which variables to standardize, we wrap it in a `ColumnTransformer`[6] object using the `make_column_transformer`[7] function. `ColumnTransformer` objects also enable the use of multiple preprocessors at once, which is especially handy when you want to apply different preprocessing to each of the predictor variables. The primary argument of the `make_column_transformer` function is a sequence of pairs of (1) a preprocessor, and (2) the columns to which you want to apply that preprocessor. In the present case, we just have the one `StandardScaler` preprocessor to apply to the `Area` and `Smoothness` columns.

[5] https://scikit-learn.org/stable/modules/preprocessing.html

[6] https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html#sklearn.compose.ColumnTransformer

[7] https://scikit-learn.org/stable/modules/generated/sklearn.compose.make_column_transformer.html#sklearn.compose.make_column_transformer

```
from sklearn.preprocessing import StandardScaler
from sklearn.compose import make_column_transformer

preprocessor = make_column_transformer(
    (StandardScaler(), ["Area", "Smoothness"]),
)
preprocessor
```

```
ColumnTransformer(transformers=[('standardscaler', StandardScaler(),
                                 ['Area', 'Smoothness'])])
```

You can see that the preprocessor includes a single standardization step that is applied to the `Area` and `Smoothness` columns. Note that here we specified which columns to apply the preprocessing step to by individual names; this approach can become quite difficult, e.g., when we have many predictor variables. Rather than writing out the column names individually, we can instead use the `make_column_selector`[8] function. For example, if we wanted to standardize all *numerical* predictors, we would use `make_column_selector` and specify the `dtype_include` argument to be `"number"`. This creates a preprocessor equivalent to the one we created previously.

```
from sklearn.compose import make_column_selector

preprocessor = make_column_transformer(
    (StandardScaler(), make_column_selector(dtype_include="number")),
)
preprocessor
```

```
ColumnTransformer(transformers=[('standardscaler', StandardScaler(),
                                 <sklearn.compose._column_transformer.make_
↪column_selector object at 0x7f667429b910>)])
```

We are now ready to standardize the numerical predictor columns in the `unscaled_cancer` data frame. This happens in two steps. We first use the `fit` function to compute the values necessary to apply the standardization (the mean and standard deviation of each variable), passing the `unscaled_cancer` data as an argument. Then we use the `transform` function to actually apply the standardization. It may seem a bit unnecessary to use two steps—`fit` *and* `transform`—to standardize the data. However, we do this in two steps so that we can specify a different data set in the `transform` step if we want. This enables us to compute the quantities needed to standardize using one data set, and then apply that standardization to another data set.

```
preprocessor.fit(unscaled_cancer)
scaled_cancer = preprocessor.transform(unscaled_cancer)
scaled_cancer
```

[8] https://scikit-learn.org/stable/modules/generated/sklearn.compose.make_column_selector.html#sklearn.compose.make_column_selector

```
     standardscaler__Area  standardscaler__Smoothness
0                0.984375                    1.568466
1                1.908708                   -0.826962
2                1.558884                    0.942210
3               -0.764464                    3.283553
4                1.826229                    0.280372
..                    ...                         ...
564              2.343856                    1.041842
565              1.723842                    0.102458
566              0.577953                   -0.840484
567              1.735218                    1.525767
568             -1.347789                   -3.112085

[569 rows x 2 columns]
```

It looks like our `Smoothness` and `Area` variables have been standardized. Woohoo! But there are two important things to notice about the new `scaled_cancer` data frame. First, it only keeps the columns from the input to `transform` (here, `unscaled_cancer`) that had a preprocessing step applied to them. The default behavior of the `ColumnTransformer` that we build using `make_column_transformer` is to *drop* the remaining columns. This default behavior works well with the rest of `sklearn` (as we will see below in Section 5.8), but for visualizing the result of preprocessing it can be useful to keep the other columns in our original data frame, such as the `Class` variable here. To keep other columns, we need to set the `remainder` argument to `"passthrough"` in the `make_column_transformer` function. Furthermore, you can see that the new column names—"standardscaler___Area" and "standardscaler___Smoothness"—include the name of the preprocessing step separated by underscores. This default behavior is useful in `sklearn` because we sometimes want to apply multiple different preprocessing steps to the same columns; but again, for visualization it can be useful to preserve the original column names. To keep original column names, we need to set the `verbose_feature_names_out` argument to `False`.

Note: Only specify the `remainder` and `verbose_feature_names_out` arguments when you want to examine the result of your preprocessing step. In most cases, you should leave these arguments at their default values.

```
preprocessor_keep_all = make_column_transformer(
    (StandardScaler(), make_column_selector(dtype_include="number")),
    remainder="passthrough",
    verbose_feature_names_out=False
)
preprocessor_keep_all.fit(unscaled_cancer)
scaled_cancer_all = preprocessor_keep_all.transform(unscaled_cancer)
scaled_cancer_all
```

```
          Area  Smoothness      Class
0     0.984375    1.568466  Malignant
1     1.908708   -0.826962  Malignant
2     1.558884    0.942210  Malignant
3    -0.764464    3.283553  Malignant
4     1.826229    0.280372  Malignant
..         ...         ...        ...
564   2.343856    1.041842  Malignant
565   1.723842    0.102458  Malignant
566   0.577953   -0.840484  Malignant
567   1.735218    1.525767  Malignant
568  -1.347789   -3.112085     Benign

[569 rows x 3 columns]
```

You may wonder why we are doing so much work just to center and scale our variables. Can't we just manually scale and center the `Area` and `Smoothness` variables ourselves before building our K-nearest neighbors model? Well, technically *yes*; but doing so is error-prone. In particular, we might accidentally forget to apply the same centering / scaling when making predictions, or accidentally apply a *different* centering / scaling than what we used while training. Proper use of a `ColumnTransformer` helps keep our code simple, readable, and error-free. Furthermore, note that using `fit` and `transform` on the preprocessor is required only when you want to inspect the result of the preprocessing steps yourself. You will see further on in Section 5.8 that `scikit-learn` provides tools to automatically streamline the preprocesser and the model so that you can call `fit` and `transform` on the `Pipeline` as necessary without additional coding effort.

Fig. 5.9 shows the two scatter plots side-by-side—one for `unscaled_cancer` and one for `scaled_cancer`. Each has the same new observation annotated with its $K = 3$ nearest neighbors. In the original unstandardized data plot, you can see some odd choices for the three nearest neighbors. In particular, the "neighbors" are visually well within the cloud of benign observations, and the neighbors are all nearly vertically aligned with the new observation (which is why it looks like there is only one black line on this plot). Fig. 5.10 shows a close-up of that region on the unstandardized plot. Here the computation of nearest neighbors is dominated by the much larger-scale area variable. The plot for standardized data on the right in Fig. 5.9 shows a much more intuitively reasonable selection of nearest neighbors. Thus, standardizing the data can change things in an important way when we are using predictive algorithms. Standardizing your data should be a part of the preprocessing you do before predictive modeling and you should always think carefully about your problem domain and whether you need to standardize your data.

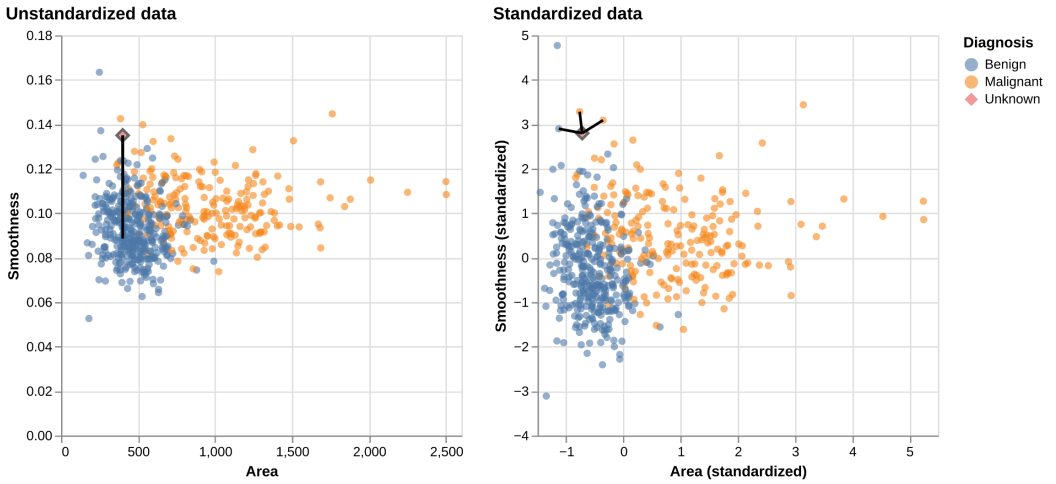


FIGURE 5.9 Comparison of K = 3 nearest neighbors with unstandardized and standardized data.

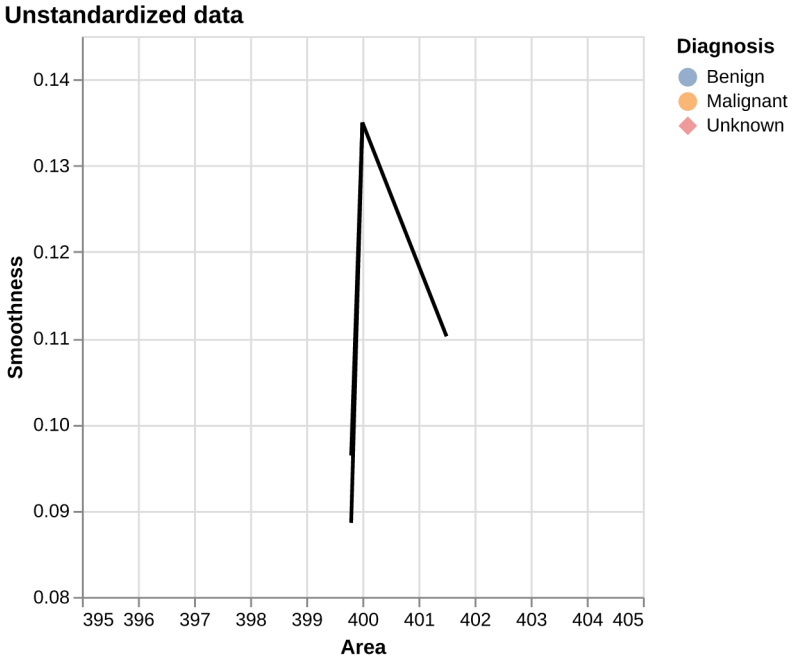


FIGURE 5.10 Close-up of three nearest neighbors for unstandardized data.

5.7.2 Balancing

Another potential issue in a data set for a classifier is *class imbalance*, i.e., when one label is much more common than another. Since classifiers like the K-nearest neighbors algorithm use the labels of nearby points to predict the label of a new point, if there are many more data points with one label overall, the algorithm is more likely to pick that label in general (even if the "pattern" of data suggests otherwise). Class imbalance is actually quite a common and important problem: from rare disease diagnosis to malicious email detection, there are many cases in which the "important" class to identify (presence of disease, malicious email) is much rarer than the "unimportant" class (no disease, normal email).

To better illustrate the problem, let's revisit the scaled breast cancer data, `cancer`; except now we will remove many of the observations of malignant tumors, simulating what the data would look like if the cancer was rare. We will do this by picking only 3 observations from the malignant group, and keeping all of the benign observations. We choose these 3 observations using the `.head()` method, which takes the number of rows to select from the top. We will then use the `concat`[9] function from `pandas` to glue the two resulting filtered data frames back together. The `concat` function *concatenates* data frames along an axis. By default, it concatenates the data frames vertically along `axis=0` yielding a single *taller* data frame, which is what we want to do here. If we instead wanted to concatenate horizontally to produce a *wider* data frame, we would specify `axis=1`. The new imbalanced data is shown in Fig. 5.11, and we print the counts of the classes using the `value_counts` function.

```
rare_cancer = pd.concat((
    cancer[cancer["Class"] == "Benign"],
    cancer[cancer["Class"] == "Malignant"].head(3)
))

rare_plot = alt.Chart(rare_cancer).mark_circle().encode(
    x=alt.X("Perimeter").title("Perimeter (standardized)"),
    y=alt.Y("Concavity").title("Concavity (standardized)"),
    color=alt.Color("Class").title("Diagnosis")
)
rare_plot
```

```
rare_cancer["Class"].value_counts()
```

```
Class
Benign       357
Malignant      3
Name: count, dtype: int64
```

[9] https://pandas.pydata.org/docs/reference/api/pandas.concat.html

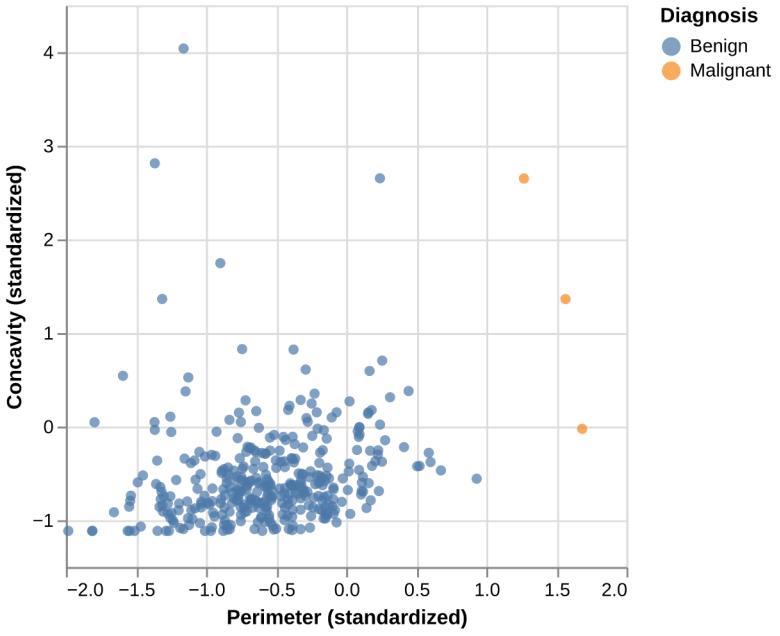


FIGURE 5.11 Imbalanced data.

Suppose we now decided to use $K = 7$ in K-nearest neighbors classification. With only 3 observations of malignant tumors, the classifier will *always predict that the tumor is benign, no matter what its concavity and perimeter are.* This is because in a majority vote of 7 observations, at most 3 will be malignant (we only have 3 total malignant observations), so at least 4 must be benign, and the benign vote will always win. For example, Fig. 5.12 shows what happens for a new tumor observation that is quite close to three observations in the training data that were tagged as malignant.

Fig. 5.13 shows what happens if we set the background color of each area of the plot to the prediction the K-nearest neighbors classifier would make for a new observation at that location. We can see that the decision is always "benign", corresponding to the blue color.

Despite the simplicity of the problem, solving it in a statistically sound manner is actually fairly nuanced, and a careful treatment would require a lot more detail and mathematics than we will cover in this textbook. For the present purposes, it will suffice to rebalance the data by *oversampling* the rare class. In other words, we will replicate rare observations multiple times in our data set to give them more voting power in the K-nearest neighbors algorithm. In order to do this, we will first separate the classes out into their own data frames by filtering. Then, we will use the `sample` method on the rare class data frame to increase the number of `Malignant` observations to be the same as the

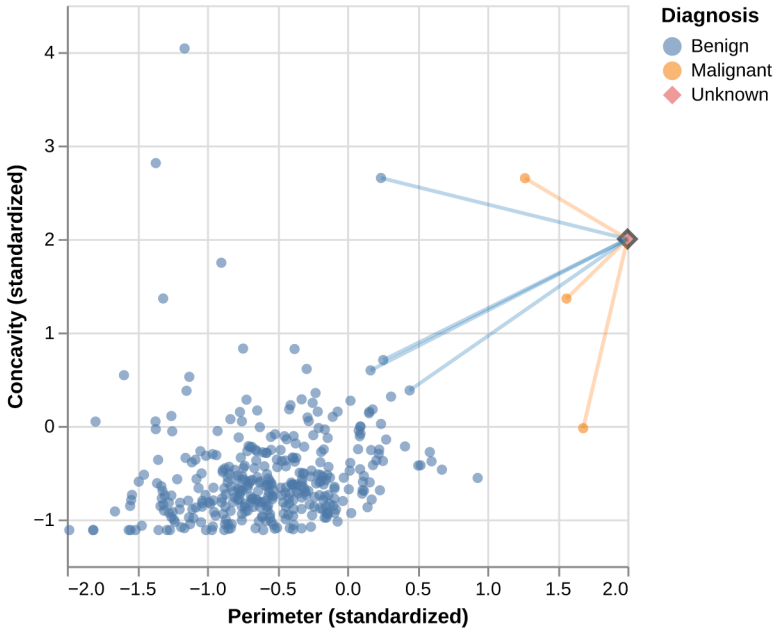


FIGURE 5.12 Imbalanced data with 7 nearest neighbors to a new observation highlighted.

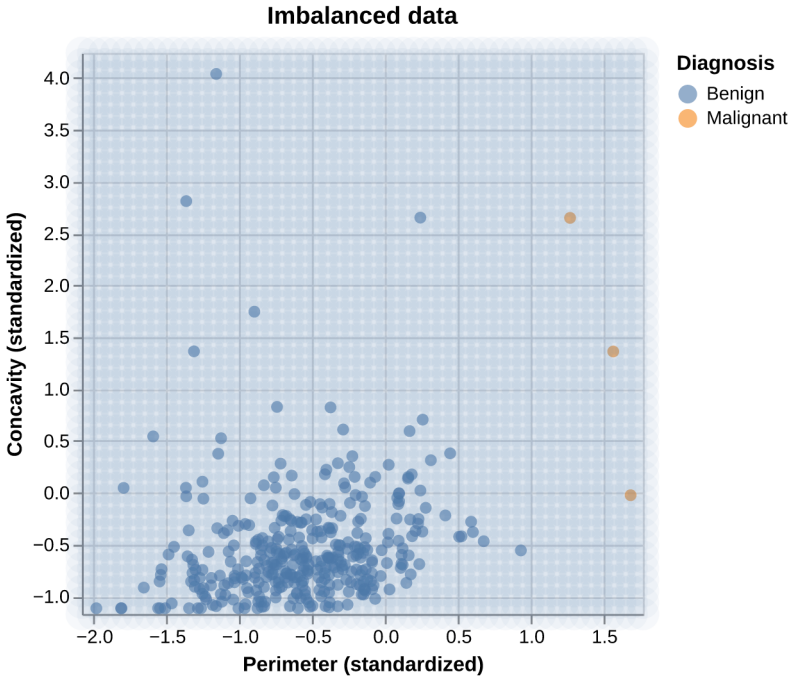


FIGURE 5.13 Imbalanced data with background color indicating the decision of the classifier and the points represent the labeled data.

number of Benign observations. We set the n argument to be the number of Malignant observations we want, and set replace=True to indicate that we are sampling with replacement. Finally, we use the value_counts method to see that our classes are now balanced. Note that sample picks which data to replicate *randomly*; we will learn more about properly handling randomness in data analysis in Chapter 6.

```
malignant_cancer = rare_cancer[rare_cancer["Class"] == "Malignant"]
benign_cancer = rare_cancer[rare_cancer["Class"] == "Benign"]
malignant_cancer_upsample = malignant_cancer.sample(
    n=benign_cancer.shape[0], replace=True
)
upsampled_cancer = pd.concat((malignant_cancer_upsample, benign_cancer))
upsampled_cancer["Class"].value_counts()
```

```
Class
Malignant    357
Benign       357
Name: count, dtype: int64
```

Now suppose we train our K-nearest neighbors classifier with $K = 7$ on this *balanced* data. Fig. 5.14 shows what happens now when we set the background color of each area of our scatter plot to the decision the K-nearest neighbors classifier would make. We can see that the decision is more reasonable; when the points are close to those labeled malignant, the classifier predicts a malignant tumor, and vice versa when they are closer to the benign tumor observations.

5.7.3 Missing data

One of the most common issues in real data sets in the wild is *missing data*, i.e., observations where the values of some of the variables were not recorded. Unfortunately, as common as it is, handling missing data properly is very challenging and generally relies on expert knowledge about the data, setting, and how the data were collected. One typical challenge with missing data is that missing entries can be *informative*: the very fact that an entries were missing is related to the values of other variables. For example, survey participants from a marginalized group of people may be less likely to respond to certain kinds of questions if they fear that answering honestly will come with negative consequences. In that case, if we were to simply throw away data with missing entries, we would bias the conclusions of the survey by inadvertently removing many members of that group of respondents. So ignoring this issue in real problems can easily lead to misleading analyses, with detrimental impacts. In this book, we will cover only those techniques for dealing with missing entries in situations where missing entries are just "randomly missing", i.e., where

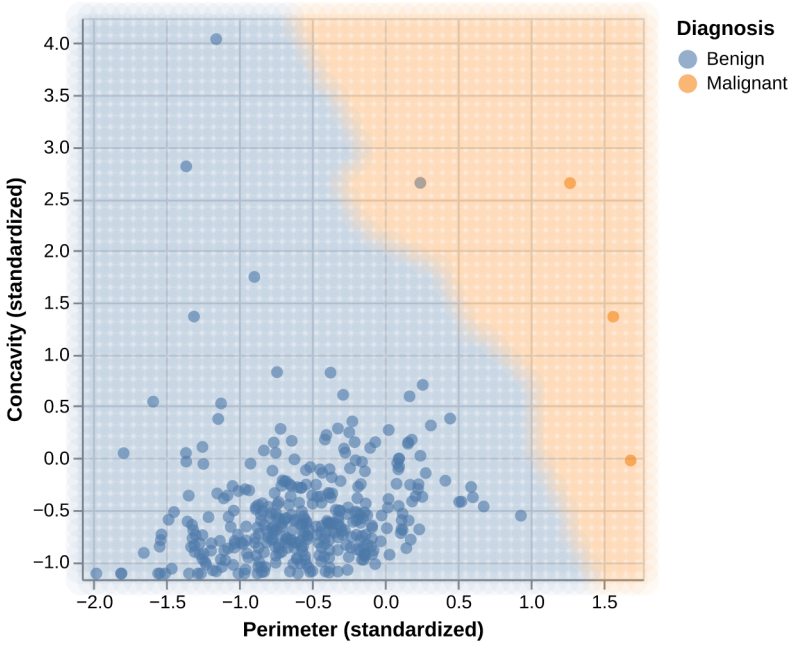


FIGURE 5.14 Upsampled data with background color indicating the decision of the classifier.

the fact that certain entries are missing *isn't related to anything else* about the observation.

Let's load and examine a modified subset of the tumor image data that has a few missing entries:

```
missing_cancer = pd.read_csv("data/wdbc_missing.csv")[["Class", "Radius",
↪"Texture", "Perimeter"]]
missing_cancer["Class"] = missing_cancer["Class"].replace({
   "M" : "Malignant",
   "B" : "Benign"
})
missing_cancer
```

```
       Class    Radius   Texture  Perimeter
0  Malignant       NaN       NaN   1.268817
1  Malignant  1.828212 -0.353322   1.684473
2  Malignant  1.578499       NaN   1.565126
3  Malignant -0.768233  0.253509  -0.592166
4  Malignant  1.748758 -1.150804   1.775011
5  Malignant -0.475956 -0.834601  -0.386808
6  Malignant  1.169878  0.160508   1.137124
```

Recall that K-nearest neighbors classification makes predictions by computing the straight-line distance to nearby training observations, and hence requires access to the values of *all* variables for *all* observations in the training data. So how can we perform K-nearest neighbors classification in the presence of

missing data? Well, since there are not too many observations with missing entries, one option is to simply remove those observations prior to building the K-nearest neighbors classifier. We can accomplish this by using the `dropna` method prior to working with the data.

```
no_missing_cancer = missing_cancer.dropna()
no_missing_cancer
```

```
       Class    Radius   Texture  Perimeter
1  Malignant  1.828212 -0.353322   1.684473
3  Malignant -0.768233  0.253509  -0.592166
4  Malignant  1.748758 -1.150804   1.775011
5  Malignant -0.475956 -0.834601  -0.386808
6  Malignant  1.169878  0.160508   1.137124
```

However, this strategy will not work when many of the rows have missing entries, as we may end up throwing away too much data. In this case, another possible approach is to *impute* the missing entries, i.e., fill in synthetic values based on the other observations in the data set. One reasonable choice is to perform *mean imputation*, where missing entries are filled in using the mean of the present entries in each variable. To perform mean imputation, we use a `SimpleImputer` transformer with the default arguments, and use `make_column_transformer` to indicate which columns need imputation.

```
from sklearn.impute import SimpleImputer

preprocessor = make_column_transformer(
    (SimpleImputer(), ["Radius", "Texture", "Perimeter"]),
    verbose_feature_names_out=False
)
preprocessor
```

```
ColumnTransformer(transformers=[('simpleimputer', SimpleImputer(),
                                 ['Radius', 'Texture', 'Perimeter'])],
                  verbose_feature_names_out=False)
```

To visualize what mean imputation does, let's just apply the transformer directly to the `missing_cancer` data frame using the `fit` and `transform` functions. The imputation step fills in the missing entries with the mean values of their corresponding variables.

```
preprocessor.fit(missing_cancer)
imputed_cancer = preprocessor.transform(missing_cancer)
imputed_cancer
```

```
     Radius   Texture  Perimeter
0  0.846860 -0.384942   1.268817
1  1.828212 -0.353322   1.684473
2  1.578499 -0.384942   1.565126
3 -0.768233  0.253509  -0.592166
```

(continues on next page)

(continued from previous page)

```
4  1.748758 -1.150804   1.775011
5 -0.475956 -0.834601  -0.386808
6  1.169878  0.160508   1.137124
```

Many other options for missing data imputation can be found in the `scikit-learn` documentation[10]. However you decide to handle missing data in your data analysis, it is always crucial to think critically about the setting, how the data were collected, and the question you are answering.

5.8 Putting it together in a `Pipeline`

The `scikit-learn` package collection also provides the `Pipeline`[11], a way to chain together multiple data analysis steps without a lot of otherwise necessary code for intermediate steps. To illustrate the whole workflow, let's start from scratch with the `wdbc_unscaled.csv` data. First, we will load the data, create a model, and specify a preprocessor for the data.

```
# load the unscaled cancer data, make Class readable
unscaled_cancer = pd.read_csv("data/wdbc_unscaled.csv")
unscaled_cancer["Class"] = unscaled_cancer["Class"].replace({
   "M" : "Malignant",
   "B" : "Benign"
})
unscaled_cancer

# create the K-NN model
knn = KNeighborsClassifier(n_neighbors=7)

# create the centering / scaling preprocessor
preprocessor = make_column_transformer(
    (StandardScaler(), ["Area", "Smoothness"]),
)
```

Next, we place these steps in a `Pipeline` using the `make_pipeline`[12] function. The `make_pipeline` function takes a list of steps to apply in your data analysis; in this case, we just have the `preprocessor` and `knn` steps. Finally, we call `fit` on the pipeline. Notice that we do not need to separately call `fit` and `transform` on the `preprocessor`; the pipeline handles doing this properly for us. Also notice that when we call `fit` on the pipeline, we can pass the whole `unscaled_cancer` data frame to the `X` argument, since

[10] https://scikit-learn.org/stable/modules/impute.html

[11] https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html?highlight=pipeline#sklearn.pipeline.Pipeline

[12] https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.make_pipeline.html#sklearn.pipeline.make_pipeline

the preprocessing step drops all the variables except the two we listed: `Area` and `Smoothness`. For the `y` response variable argument, we pass the `unscaled_cancer["Class"]` series as before.

```
from sklearn.pipeline import make_pipeline

knn_pipeline = make_pipeline(preprocessor, knn)
knn_pipeline.fit(
    X=unscaled_cancer,
    y=unscaled_cancer["Class"]
)
knn_pipeline
```

```
Pipeline(steps=[('columntransformer',
                 ColumnTransformer(transformers=[('standardscaler',
                                                  StandardScaler(),
                                                  ['Area', 'Smoothness'])])),
                ('kneighborsclassifier', KNeighborsClassifier(n_
↪neighbors=7))])
```

As before, the fit object lists the function that trains the model. But now the fit object also includes information about the overall workflow, including the standardization preprocessing step. In other words, when we use the `predict` function with the `knn_pipeline` object to make a prediction for a new observation, it will first apply the same preprocessing steps to the new observation. As an example, we will predict the class label of two new observations: one with `Area = 500` and `Smoothness = 0.075`, and one with `Area = 1500` and `Smoothness = 0.1`.

```
new_observation = pd.DataFrame({"Area": [500, 1500], "Smoothness": [0.075, 0.1]})
prediction = knn_pipeline.predict(new_observation)
prediction
```

```
array(['Benign', 'Malignant'], dtype=object)
```

The classifier predicts that the first observation is benign, while the second is malignant. Fig. 5.15 visualizes the predictions that this trained K-nearest neighbors model will make on a large range of new observations. Although you have seen colored prediction map visualizations like this a few times now, we have not included the code to generate them, as it is a little bit complicated. For the interested reader who wants a learning challenge, we now include it below. The basic idea is to create a grid of synthetic new observations using the `meshgrid` function from `numpy`, predict the label of each, and visualize the predictions with a colored scatter having a very high transparency (low `opacity` value) and large point radius. See if you can figure out what each line is doing.

Note: Understanding this code is not required for the remainder of the

textbook. It is included for those readers who would like to use similar visualizations in their own data analyses.

```
import numpy as np

# create the grid of area/smoothness vals, and arrange in a data frame
are_grid = np.linspace(
    unscaled_cancer["Area"].min() * 0.95, unscaled_cancer["Area"].max() * 1.05,␣
↪50
)
smo_grid = np.linspace(
    unscaled_cancer["Smoothness"].min() * 0.95, unscaled_cancer["Smoothness"].
↪max() * 1.05, 50
)
asgrid = np.array(np.meshgrid(are_grid, smo_grid)).reshape(2, -1).T
asgrid = pd.DataFrame(asgrid, columns=["Area", "Smoothness"])

# use the fit workflow to make predictions at the grid points
knnPredGrid = knn_pipeline.predict(asgrid)

# bind the predictions as a new column with the grid points
prediction_table = asgrid.copy()
prediction_table["Class"] = knnPredGrid

# plot:
# 1. the colored scatter of the original data
unscaled_plot = alt.Chart(unscaled_cancer).mark_point(
    opacity=0.6,
    filled=True,
    size=40
).encode(
    x=alt.X("Area")
        .scale(
            nice=False,
            domain=(
                unscaled_cancer["Area"].min() * 0.95,
                unscaled_cancer["Area"].max() * 1.05
            )
        ),
    y=alt.Y("Smoothness")
        .scale(
            nice=False,
            domain=(
                unscaled_cancer["Smoothness"].min() * 0.95,
                unscaled_cancer["Smoothness"].max() * 1.05
            )
        ),
    color=alt.Color("Class").title("Diagnosis")
)

# 2. the faded colored scatter for the grid points
prediction_plot = alt.Chart(prediction_table).mark_point(
    opacity=0.05,
    filled=True,
    size=300
).encode(
    x="Area",
    y="Smoothness",
    color=alt.Color("Class").title("Diagnosis")
)
unscaled_plot + prediction_plot
```

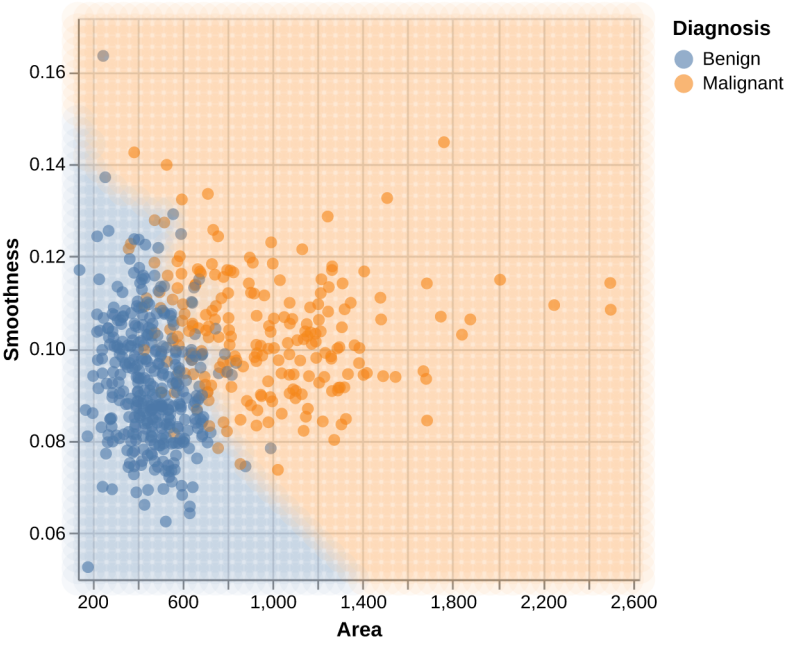


FIGURE 5.15 Scatter plot of smoothness versus area where background color indicates the decision of the classifier.

5.9 Exercises

Practice exercises for the material covered in this chapter can be found in the accompanying worksheets repository[13] in the "Classification I: training and predicting" row. You can launch an interactive version of the worksheet in your browser by clicking the "launch binder" button. You can also preview a non-interactive version of the worksheet by clicking "view worksheet". If you instead decide to download the worksheet and run it on your own machine, make sure to follow the instructions for computer setup found in Chapter 13. This will ensure that the automated feedback and guidance that the worksheets provide will function as intended.

[13] https://worksheets.python.datasciencebook.ca

6

Classification II: evaluation & tuning

6.1 Overview

This chapter continues the introduction to predictive modeling through classification. While the previous chapter covered training and data preprocessing, this chapter focuses on how to evaluate the performance of a classifier, as well as how to improve the classifier (where possible) to maximize its accuracy.

6.2 Chapter learning objectives

By the end of the chapter, readers will be able to do the following:

- Describe what training, validation, and test data sets are and how they are used in classification.
- Split data into training, validation, and test data sets.
- Describe what a random seed is and its importance in reproducible data analysis.
- Set the random seed in Python using the `numpy.random.seed` function.
- Describe and interpret accuracy, precision, recall, and confusion matrices.
- Evaluate classification accuracy, precision, and recall in Python using a test set, a single validation set, and cross-validation.
- Produce a confusion matrix in Python.
- Choose the number of neighbors in a K-nearest neighbors classifier by maximizing estimated cross-validation accuracy.
- Describe underfitting and overfitting, and relate it to the number of neighbors in K-nearest neighbors classification.
- Describe the advantages and disadvantages of the K-nearest neighbors classification algorithm.

DOI: 10.1201/9781003438397-6

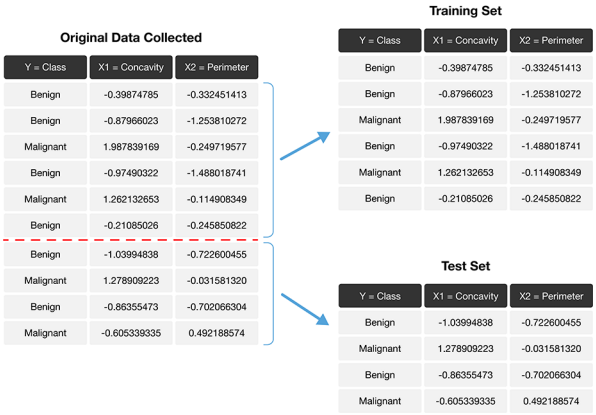


FIGURE 6.1 Splitting the data into training and testing sets.

6.3 Evaluating performance

Sometimes our classifier might make the wrong prediction. A classifier does not need to be right 100% of the time to be useful, though we don't want the classifier to make too many wrong predictions. How do we measure how "good" our classifier is? Let's revisit the breast cancer images data[1] [Street *et al.*, 1993] and think about how our classifier will be used in practice. A biopsy will be performed on a *new* patient's tumor, the resulting image will be analyzed, and the classifier will be asked to decide whether the tumor is benign or malignant. The key word here is *new*: our classifier is "good" if it provides accurate predictions on data *not seen during training*, as this implies that it has actually learned about the relationship between the predictor variables and response variable, as opposed to simply memorizing the labels of individual training data examples. But then, how can we evaluate our classifier without visiting the hospital to collect more tumor images?

The trick is to split the data into a **training set** and **test set** (Fig. 6.1) and use only the **training set** when building the classifier. Then, to evaluate the performance of the classifier, we first set aside the labels from the **test set**, and then use the classifier to predict the labels in the **test set**. If our predictions match the actual labels for the observations in the **test set**, then we have some confidence that our classifier might also accurately predict the class labels for new observations without known class labels.

[1] https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29

Note: If there were a golden rule of machine learning, it might be this: *you cannot use the test data to build the model.* If you do, the model gets to "see" the test data in advance, making it look more accurate than it really is. Imagine how bad it would be to overestimate your classifier's accuracy when predicting whether a patient's tumor is malignant or benign.

How exactly can we assess how well our predictions match the actual labels for the observations in the test set? One way we can do this is to calculate the prediction **accuracy**. This is the fraction of examples for which the classifier made the correct prediction. To calculate this, we divide the number of correct predictions by the number of predictions made. The process for assessing if our predictions match the actual labels in the test set is illustrated in Fig. 6.2.

$$\text{accuracy} = \frac{\text{number of correct predictions}}{\text{total number of predictions}}$$

Accuracy is a convenient, general-purpose way to summarize the performance of a classifier with a single number. But prediction accuracy by itself does not tell the whole story. In particular, accuracy alone only tells us how often the classifier makes mistakes in general, but does not tell us anything about the *kinds* of mistakes the classifier makes. A more comprehensive view of performance can be obtained by additionally examining the **confusion matrix**. The confusion matrix shows how many test set labels of each type are predicted correctly and incorrectly, which gives us more detail about the kinds

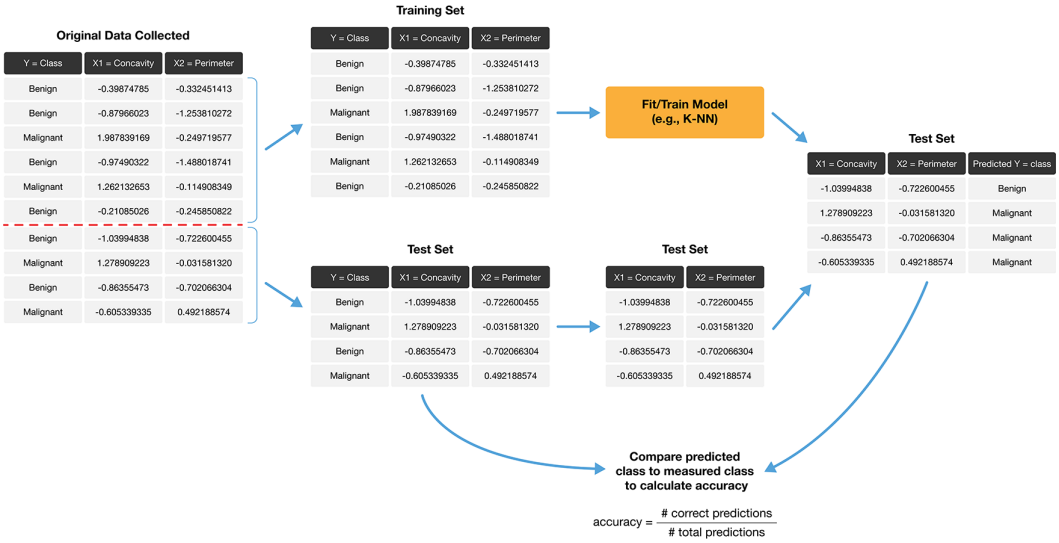


FIGURE 6.2 Process for splitting the data and finding the prediction accuracy.

of mistakes the classifier tends to make. Table 6.1 shows an example of what a confusion matrix might look like for the tumor image data with a test set of 65 observations.

TABLE 6.1 An example confusion matrix for the tumor image data.

	Predicted Malignant	**Predicted Benign**
Actually Malignant	1	3
Actually Benign	4	57

In the example in Table 6.1, we see that there was 1 malignant observation that was correctly classified as malignant (top left corner), and 57 benign observations that were correctly classified as benign (bottom right corner). However, we can also see that the classifier made some mistakes: it classified 3 malignant observations as benign, and 4 benign observations as malignant. The accuracy of this classifier is roughly 89%, given by the formula

$$\text{accuracy} = \frac{\text{number of correct predictions}}{\text{total number of predictions}} = \frac{1+57}{1+57+4+3} = 0.892.$$

But we can also see that the classifier only identified 1 out of 4 total malignant tumors; in other words, it misclassified 75% of the malignant cases present in the data set. In this example, misclassifying a malignant tumor is a potentially disastrous error, since it may lead to a patient who requires treatment not receiving it. Since we are particularly interested in identifying malignant cases, this classifier would likely be unacceptable even with an accuracy of 89%.

Focusing more on one label than the other is common in classification problems. In such cases, we typically refer to the label we are more interested in identifying as the *positive* label, and the other as the *negative* label. In the tumor example, we would refer to malignant observations as *positive*, and benign observations as *negative*. We can then use the following terms to talk about the four kinds of prediction that the classifier can make, corresponding to the four entries in the confusion matrix:

- **True Positive:** A malignant observation that was classified as malignant (top left in Table 6.1).
- **False Positive:** A benign observation that was classified as malignant (bottom left in Table 6.1).
- **True Negative:** A benign observation that was classified as benign (bottom right in Table 6.1).
- **False Negative:** A malignant observation that was classified as benign (top right in Table 6.1).

A perfect classifier would have zero false negatives and false positives (and therefore, 100% accuracy). However, classifiers in practice will almost always make some errors. So you should think about which kinds of error are most important in your application, and use the confusion matrix to quantify and report them. Two commonly used metrics that we can compute using the confusion matrix are the **precision** and **recall** of the classifier. These are often reported together with accuracy. *Precision* quantifies how many of the positive predictions the classifier made were actually positive. Intuitively, we would like a classifier to have a *high* precision: for a classifier with high precision, if the classifier reports that a new observation is positive, we can trust that the new observation is indeed positive. We can compute the precision of a classifier using the entries in the confusion matrix, with the formula

$$\text{precision} = \frac{\text{number of correct positive predictions}}{\text{total number of positive predictions}}.$$

Recall quantifies how many of the positive observations in the test set were identified as positive. Intuitively, we would like a classifier to have a *high* recall: for a classifier with high recall, if there is a positive observation in the test data, we can trust that the classifier will find it. We can also compute the recall of the classifier using the entries in the confusion matrix, with the formula

$$\text{recall} = \frac{\text{number of correct positive predictions}}{\text{total number of positive test set observations}}.$$

In the example presented in Table 6.1, we have that the precision and recall are

$$\text{precision} = \frac{1}{1+4} = 0.20, \quad \text{recall} = \frac{1}{1+3} = 0.25.$$

So even with an accuracy of 89%, the precision and recall of the classifier were both relatively low. For this data analysis context, recall is particularly important: if someone has a malignant tumor, we certainly want to identify it. A recall of just 25% would likely be unacceptable.

Note: It is difficult to achieve both high precision and high recall at the same time; models with high precision tend to have low recall and vice versa. As an example, we can easily make a classifier that has *perfect recall*: just *always* guess positive. This classifier will of course find every positive observation in the test set, but it will make lots of false positive predictions along the way and have low precision. Similarly, we can easily make a classifier that has *perfect precision*: *never* guess positive. This classifier will never incorrectly identify

an observation as positive, but it will make a lot of false negative predictions along the way. In fact, this classifier will have 0% recall. Of course, most real classifiers fall somewhere in between these two extremes. But these examples serve to show that in settings where one of the classes is of interest (i.e., there is a *positive* label), there is a trade-off between precision and recall that one has to make when designing a classifier.

6.4 Randomness and seeds

Beginning in this chapter, our data analyses will often involve the use of *randomness*. We use randomness any time we need to make a decision in our analysis that needs to be fair, unbiased, and not influenced by human input. For example, in this chapter, we need to split a data set into a training set and test set to evaluate our classifier. We certainly do not want to choose how to split the data ourselves by hand, as we want to avoid accidentally influencing the result of the evaluation. So instead, we let Python *randomly* split the data. In future chapters we will use randomness in many other ways, e.g., to help us select a small subset of data from a larger data set, to pick groupings of data, and more.

However, the use of randomness runs counter to one of the main tenets of good data analysis practice: *reproducibility*. Recall that a reproducible analysis produces the same result each time it is run; if we include randomness in the analysis, would we not get a different result each time? The trick is that in Python—and other programming languages—randomness is not actually random. Instead, Python uses a *random number generator* that produces a sequence of numbers that are completely determined by a *seed value*. Once you set the seed value, everything after that point may *look* random, but is actually totally reproducible. As long as you pick the same seed value, you get the same result.

Let's use an example to investigate how randomness works in Python. Say we have a series object containing the integers from 0 to 9. We want to randomly pick 10 numbers from that list, but we want it to be reproducible. Before randomly picking the 10 numbers, we call the `seed` function from the `numpy` package, and pass it any integer as the argument. Below we use the seed number `1`. At that point, Python will keep track of the randomness that occurs throughout the code. For example, we can call the `sample` method on the series of numbers, passing the argument `n=10` to indicate that we want

10 samples. The `to_list` method converts the resulting series into a basic Python list to make the output easier to read.

```
import numpy as np
import pandas as pd

np.random.seed(1)

nums_0_to_9 = pd.Series([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

random_numbers1 = nums_0_to_9.sample(n=10).to_list()
random_numbers1
```

```
[2, 9, 6, 4, 0, 3, 1, 7, 8, 5]
```

You can see that `random_numbers1` is a list of 10 numbers from 0 to 9 that, from all appearances, looks random. If we run the `sample` method again, we will get a fresh batch of 10 numbers that also look random.

```
random_numbers2 = nums_0_to_9.sample(n=10).to_list()
random_numbers2
```

```
[9, 5, 3, 0, 8, 4, 2, 1, 6, 7]
```

If we want to force Python to produce the same sequences of random numbers, we can simply call the `np.random.seed` function with the seed value `1`—the same as before—and then call the `sample` method again.

```
np.random.seed(1)
random_numbers1_again = nums_0_to_9.sample(n=10).to_list()
random_numbers1_again
```

```
[2, 9, 6, 4, 0, 3, 1, 7, 8, 5]
```

```
random_numbers2_again = nums_0_to_9.sample(n=10).to_list()
random_numbers2_again
```

```
[9, 5, 3, 0, 8, 4, 2, 1, 6, 7]
```

Notice that after calling `np.random.seed`, we get the same two sequences of numbers in the same order. `random_numbers1` and `random_numbers1_again` produce the same sequence of numbers, and the same can be said about `random_numbers2` and `random_numbers2_again`. And if we choose a different value for the seed—say, 4235—we obtain a different sequence of random numbers.

```
np.random.seed(4235)
random_numbers1_different = nums_0_to_9.sample(n=10).to_list()
random_numbers1_different
```

```
[6, 7, 2, 3, 5, 9, 1, 4, 0, 8]
```

```
random_numbers2_different = nums_0_to_9.sample(n=10).to_list()
random_numbers2_different
```

```
[6, 0, 1, 3, 2, 8, 4, 9, 5, 7]
```

In other words, even though the sequences of numbers that Python is generating *look* random, they are totally determined when we set a seed value.

So what does this mean for data analysis? Well, `sample` is certainly not the only place where randomness is used in Python. Many of the functions that we use in `scikit-learn` and beyond use randomness—some of them without even telling you about it. Also note that when Python starts up, it creates its own seed to use. So if you do not explicitly call the `np.random.seed` function, your results will likely not be reproducible. Finally, be careful to set the seed *only once* at the beginning of a data analysis. Each time you set the seed, you are inserting your own human input, thereby influencing the analysis. For example, if you use the `sample` many times throughout your analysis but set the seed each time, the randomness that Python uses will not look as random as it should.

In summary: if you want your analysis to be reproducible, i.e., produce *the same result* each time you run it, make sure to use `np.random.seed` exactly once at the beginning of the analysis. Different argument values in `np.random.seed` will lead to different patterns of randomness, but as long as you pick the same value your analysis results will be the same. In the remainder of the textbook, we will set the seed once at the beginning of each chapter.

Note: When you use `np.random.seed`, you are really setting the seed for the `numpy` package's *default random number generator*. Using the global default random number generator is easier than other methods, but has some potential drawbacks. For example, other code that you may not notice (e.g., code buried inside some other package) could potentially *also* call `np.random.seed`, thus modifying your analysis in an undesirable way. Furthermore, not *all* functions use `numpy`'s random number generator; some may use another one entirely. In that case, setting `np.random.seed` may not actually make your whole analysis reproducible.

In this book, we will generally only use packages that play nicely with `numpy`'s default random number generator, so we will stick with `np.random.seed`. You can achieve more careful control over randomness in your analysis by

creating a `numpy Generator` object[2] once at the beginning of your analysis, and passing it to the `random_state` argument that is available in many `pandas` and `scikit-learn` functions. Those functions will then use your `Generator` to generate random numbers instead of `numpy`'s default generator. For example, we can reproduce our earlier example by using a `Generator` object with the `seed` value set to 1; we get the same lists of numbers once again.

```
from numpy.random import Generator, PCG64
rng = Generator(PCG64(seed=1))
random_numbers1_third = nums_0_to_9.sample(n=10, random_state=rng).to_list()
random_numbers1_third
```

```
array([2, 9, 6, 4, 0, 3, 1, 7, 8, 5])
```

```
random_numbers2_third = nums_0_to_9.sample(n=10, random_state=rng).to_list()
random_numbers2_third
```

```
array([9, 5, 3, 0, 8, 4, 2, 1, 6, 7])
```

6.5 Evaluating performance with `scikit-learn`

Back to evaluating classifiers now. In Python, we can use the `scikit-learn` package not only to perform K-nearest neighbors classification, but also to assess how well our classification worked. Let's work through an example of how to use tools from `scikit-learn` to evaluate a classifier using the breast cancer data set from the previous chapter. We begin the analysis by loading the packages we require, reading in the breast cancer data, and then making a quick scatter plot visualization of tumor cell concavity versus smoothness colored by diagnosis in Fig. 6.3. You will also notice that we set the random seed using the `np.random.seed` function, as described in Section 6.4.

```
# load packages
import altair as alt
import pandas as pd
from sklearn import set_config

# Output dataframes instead of arrays
set_config(transform_output="pandas")

# set the seed
np.random.seed(1)
```

(continues on next page)

[2] https://numpy.org/doc/stable/reference/random/generator.html

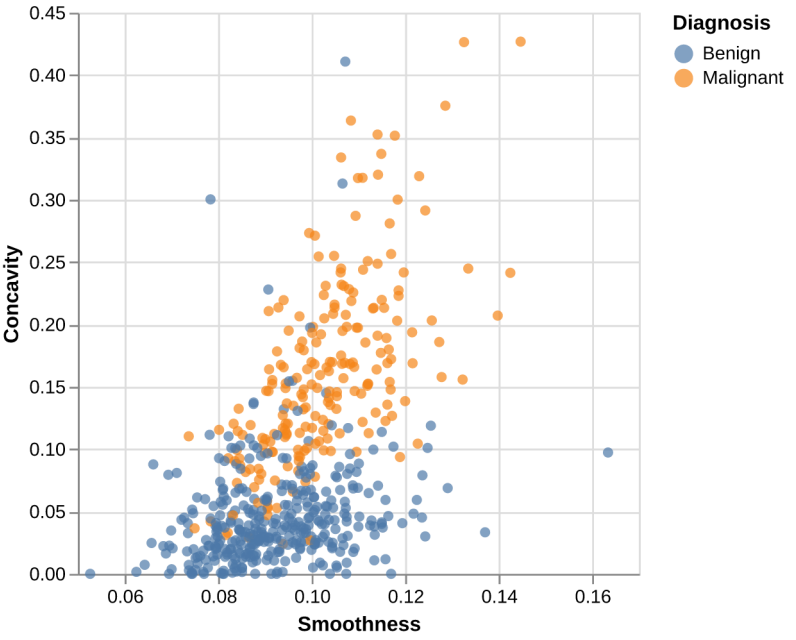


FIGURE 6.3 Scatter plot of tumor cell concavity versus smoothness colored by diagnosis label.

(continued from previous page)

```
# load data
cancer = pd.read_csv("data/wdbc_unscaled.csv")
# re-label Class "M" as "Malignant", and Class "B" as "Benign"
cancer["Class"] = cancer["Class"].replace({
    "M" : "Malignant",
    "B" : "Benign"
})

# create scatter plot of tumor cell concavity versus smoothness,
# labeling the points be diagnosis class

perim_concav = alt.Chart(cancer).mark_circle().encode(
    x=alt.X("Smoothness").scale(zero=False),
    y="Concavity",
    color=alt.Color("Class").title("Diagnosis")
)
perim_concav
```

6.5.1 Create the train / test split

Once we have decided on a predictive question to answer and done some preliminary exploration, the very next thing to do is to split the data into the training and test sets. Typically, the training set is between 50% and 95% of the data, while the test set is the remaining 5% to 50%; the intuition is that you want to trade off between training an accurate model (by using a larger

training data set) and getting an accurate evaluation of its performance (by using a larger test data set). Here, we will use 75% of the data for training, and 25% for testing.

The `train_test_split` function from `scikit-learn` handles the procedure of splitting the data for us. We can specify two very important parameters when using `train_test_split` to ensure that the accuracy estimates from the test data are reasonable. First, setting `shuffle=True` (which is the default) means the data will be shuffled before splitting, which ensures that any ordering present in the data does not influence the data that ends up in the training and testing sets. Second, by specifying the `stratify` parameter to be the response variable in the training set, it **stratifies** the data by the class label, to ensure that roughly the same proportion of each class ends up in both the training and testing sets. For example, in our data set, roughly 63% of the observations are from the benign class (`Benign`), and 37% are from the malignant class (`Malignant`), so specifying `stratify` as the class column ensures that roughly 63% of the training data are benign, 37% of the training data are malignant, and the same proportions exist in the testing data.

Let's use the `train_test_split` function to create the training and testing sets. We first need to import the function from the `sklearn` package. Then we will specify that `train_size=0.75` so that 75% of our original data set ends up in the training set. We will also set the `stratify` argument to the categorical label variable (here, `cancer["Class"]`) to ensure that the training and testing subsets contain the right proportions of each category of observation.

```
from sklearn.model_selection import train_test_split

cancer_train, cancer_test = train_test_split(
    cancer, train_size=0.75, stratify=cancer["Class"]
)
cancer_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 426 entries, 196 to 296
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   ID                 426 non-null    int64
 1   Class              426 non-null    object
 2   Radius             426 non-null    float64
 3   Texture            426 non-null    float64
 4   Perimeter          426 non-null    float64
 5   Area               426 non-null    float64
 6   Smoothness         426 non-null    float64
 7   Compactness        426 non-null    float64
 8   Concavity          426 non-null    float64
 9   Concave_Points     426 non-null    float64
 10  Symmetry           426 non-null    float64
```

(continues on next page)

(continued from previous page)

```
 11  Fractal_Dimension  426 non-null    float64
dtypes: float64(10), int64(1), object(1)
memory usage: 43.3+ KB
```

```
cancer_test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 143 entries, 116 to 15
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   ID                 143 non-null    int64
 1   Class              143 non-null    object
 2   Radius             143 non-null    float64
 3   Texture            143 non-null    float64
 4   Perimeter          143 non-null    float64
 5   Area               143 non-null    float64
 6   Smoothness         143 non-null    float64
 7   Compactness        143 non-null    float64
 8   Concavity          143 non-null    float64
 9   Concave_Points     143 non-null    float64
 10  Symmetry           143 non-null    float64
 11  Fractal_Dimension  143 non-null    float64
dtypes: float64(10), int64(1), object(1)
memory usage: 14.5+ KB
```

We can see from the `info` method above that the training set contains 426 observations, while the test set contains 143 observations. This corresponds to a train / test split of 75% / 25%, as desired. Recall from Chapter 5 that we use the `info` method to preview the number of rows, the variable names, their data types, and missing entries of a data frame.

We can use the `value_counts` method with the `normalize` argument set to `True` to find the percentage of malignant and benign classes in `cancer_train`. We see about 63% of the training data are benign and 37% are malignant, indicating that our class proportions were roughly preserved when we split the data.

```
cancer_train["Class"].value_counts(normalize=True)
```

```
Class
Benign       0.626761
Malignant    0.373239
Name: proportion, dtype: float64
```

6.5.2 Preprocess the data

As we mentioned in the last chapter, K-nearest neighbors is sensitive to the scale of the predictors, so we should perform some preprocessing to standardize

them. An additional consideration we need to take when doing this is that we should create the standardization preprocessor using **only the training data**. This ensures that our test data does not influence any aspect of our model training. Once we have created the standardization preprocessor, we can then apply it separately to both the training and test data sets.

Fortunately, `scikit-learn` helps us handle this properly as long as we wrap our analysis steps in a `Pipeline`, as in Chapter 5. So below we construct and prepare the preprocessor using `make_column_transformer` just as before.

```
from sklearn.preprocessing import StandardScaler
from sklearn.compose import make_column_transformer

cancer_preprocessor = make_column_transformer(
    (StandardScaler(), ["Smoothness", "Concavity"]),
)
```

6.5.3 Train the classifier

Now that we have split our original data set into training and test sets, we can create our K-nearest neighbors classifier with only the training set using the technique we learned in the previous chapter. For now, we will just choose the number K of neighbors to be 3, and use only the concavity and smoothness predictors by selecting them from the `cancer_train` data frame. We will first import the `KNeighborsClassifier` model and `make_pipeline` from `sklearn`. Then as before we will create a model object, combine the model object and preprocessor into a `Pipeline` using the `make_pipeline` function, and then finally use the `fit` method to build the classifier.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline

knn = KNeighborsClassifier(n_neighbors=3)

X = cancer_train[["Smoothness", "Concavity"]]
y = cancer_train["Class"]

knn_pipeline = make_pipeline(cancer_preprocessor, knn)
knn_pipeline.fit(X, y)

knn_pipeline
```

```
Pipeline(steps=[('columntransformer',
                 ColumnTransformer(transformers=[('standardscaler',
                                                  StandardScaler(),
                                                  ['Smoothness',
                                                   'Concavity'])])),
                ('kneighborsclassifier', KNeighborsClassifier(n_
↪neighbors=3))])
```

6.5.4 Predict the labels in the test set

Now that we have a K-nearest neighbors classifier object, we can use it to predict the class labels for our test set and augment the original test data with a column of predictions. The `Class` variable contains the actual diagnoses, while the `predicted` contains the predicted diagnoses from the classifier. Note that below we print out just the `ID`, `Class`, and `predicted` variables in the output data frame.

```
cancer_test["predicted"] = knn_pipeline.predict(cancer_test[["Smoothness",
↪"Concavity"]])
cancer_test[["ID", "Class", "predicted"]]
```

```
           ID      Class  predicted
116    864726     Benign  Malignant
146    869691  Malignant  Malignant
86   86135501  Malignant  Malignant
12     846226  Malignant  Malignant
105    863030  Malignant  Malignant
..        ...        ...        ...
244    884180  Malignant  Malignant
23     851509  Malignant  Malignant
125     86561     Benign     Benign
281   8912055     Benign     Benign
15   84799002  Malignant  Malignant

[143 rows x 3 columns]
```

6.5.5 Evaluate performance

Finally, we can assess our classifier's performance. First, we will examine accuracy. To do this we will use the `score` method, specifying two arguments: predictors and the actual labels. We pass the same test data for the predictors that we originally passed into `predict` when making predictions, and we provide the actual labels via the `cancer_test["Class"]` series.

```
knn_pipeline.score(
    cancer_test[["Smoothness", "Concavity"]],
    cancer_test["Class"]
)
```

```
0.8951048951048951
```

The output shows that the estimated accuracy of the classifier on the test data was 90%. To compute the precision and recall, we can use the `precision_score` and `recall_score` functions from `scikit-learn`. We specify the true labels from the `Class` variable as the `y_true` argument, the predicted labels from the `predicted` variable as the `y_pred` argument, and which label should be considered to be positive via the `pos_label` argument.

```
from sklearn.metrics import recall_score, precision_score

precision_score(
    y_true=cancer_test["Class"],
    y_pred=cancer_test["predicted"],
    pos_label="Malignant"
)
```

```
0.8275862068965517
```

```
recall_score(
    y_true=cancer_test["Class"],
    y_pred=cancer_test["predicted"],
    pos_label="Malignant"
)
```

```
0.9056603773584906
```

The output shows that the estimated precision and recall of the classifier on the test data was 83% and 91%, respectively. Finally, we can look at the *confusion matrix* for the classifier using the `crosstab` function from `pandas`. The `crosstab` function takes two arguments: the actual labels first, then the predicted labels second. Note that `crosstab` orders its columns alphabetically, but the positive label is still `Malignant`, even if it is not in the top left corner as in the example confusion matrix earlier in this chapter.

```
pd.crosstab(
    cancer_test["Class"],
    cancer_test["predicted"]
)
```

```
predicted  Benign  Malignant
Class
Benign         80         10
Malignant       5         48
```

The confusion matrix shows 48 observations were correctly predicted as malignant, and 80 were correctly predicted as benign. It also shows that the classifier made some mistakes; in particular, it classified 5 observations as benign when they were actually malignant, and 10 observations as malignant when they were actually benign. Using our formulas from earlier, we see that the accuracy, precision, and recall agree with what Python reported.

$$\text{accuracy} = \frac{\text{number of correct predictions}}{\text{total number of predictions}} = \frac{80 + 48}{80 + 48 + 10 + 5} = 89.51$$

$$\text{precision} = \frac{\text{number of correct positive predictions}}{\text{total number of positive predictions}} = \frac{48}{48 + 10} = 82.76$$

$$\text{recall} = \frac{\text{number of correct positive predictions}}{\text{total number of positive test set observations}} = \frac{48}{48 + 5} = 90.57$$

6.5.6 Critically analyze performance

We now know that the classifier was 90% accurate on the test data set, and had a precision of 83% and a recall of 91%. That sounds pretty good! Wait, *is* it good? Or do we need something higher?

In general, a *good* value for accuracy (as well as precision and recall, if applicable) depends on the application; you must critically analyze your accuracy in the context of the problem you are solving. For example, if we were building a classifier for a kind of tumor that is benign 99% of the time, a classifier with 99% accuracy is not terribly impressive (just always guess benign!). And beyond just accuracy, we need to consider the precision and recall: as mentioned earlier, the *kind* of mistake the classifier makes is important in many applications as well. In the previous example with 99% benign observations, it might be very bad for the classifier to predict "benign" when the actual class is "malignant" (a false negative), as this might result in a patient not receiving appropriate medical attention. In other words, in this context, we need the classifier to have a *high recall*. On the other hand, it might be less bad for the classifier to guess "malignant" when the actual class is "benign" (a false positive), as the patient will then likely see a doctor who can provide an expert diagnosis. In other words, we are fine with sacrificing some precision in the interest of achieving high recall. This is why it is important not only to look at accuracy, but also the confusion matrix.

However, there is always an easy baseline that you can compare to for any classification problem: the *majority classifier*. The majority classifier *always* guesses the majority class label from the training data, regardless of the predictor variables' values. It helps to give you a sense of scale when considering accuracies. If the majority classifier obtains a 90% accuracy on a problem, then you might hope for your K-nearest neighbors classifier to do better than that. If your classifier provides a significant improvement upon the majority classifier, this means that at least your method is extracting some useful information from your predictor variables. Be careful though: improving on the majority classifier does not *necessarily* mean the classifier is working well enough for your application.

As an example, in the breast cancer data, recall the proportions of benign and malignant observations in the training data are as follows:

```
cancer_train["Class"].value_counts(normalize=True)
```

```
Class
Benign       0.626761
Malignant    0.373239
Name: proportion, dtype: float64
```

Since the benign class represents the majority of the training data, the majority classifier would *always* predict that a new observation is benign. The estimated accuracy of the majority classifier is usually fairly close to the majority class proportion in the training data. In this case, we would suspect that the majority classifier will have an accuracy of around 63%. The K-nearest neighbors classifier we built does quite a bit better than this, with an accuracy of 90%. This means that from the perspective of accuracy, the K-nearest neighbors classifier improved quite a bit on the basic majority classifier. Hooray! But we still need to be cautious; in this application, it is likely very important not to misdiagnose any malignant tumors to avoid missing patients who actually need medical care. The confusion matrix above shows that the classifier does, indeed, misdiagnose a significant number of malignant tumors as benign (5 out of 53 malignant tumors, or 9%!). Therefore, even though the accuracy improved upon the majority classifier, our critical analysis suggests that this classifier may not have appropriate performance for the application.

6.6 Tuning the classifier

The vast majority of predictive models in statistics and machine learning have *parameters*. A *parameter* is a number you have to pick in advance that determines some aspect of how the model behaves. For example, in the K-nearest neighbors classification algorithm, K is a parameter that we have to pick that determines how many neighbors participate in the class vote. By picking different values of K, we create different classifiers that make different predictions.

So then, how do we pick the *best* value of K, i.e., *tune* the model? And is it possible to make this selection in a principled way? In this book, we will focus on maximizing the accuracy of the classifier. Ideally, we want somehow to maximize the accuracy of our classifier on data *it hasn't seen yet*. But we cannot use our test data set in the process of building our model. So we will play the same trick we did before when evaluating our classifier: we'll split our *training data itself* into two subsets, use one to train the model, and then use the other to evaluate it. In this section, we will cover the details of this procedure, as well as how to use it to help you pick a good parameter value for your classifier.

And remember: don't touch the test set during the tuning process. Tuning is a part of model training.

6.6.1 Cross-validation

The first step in choosing the parameter K is to be able to evaluate the classifier using only the training data. If this is possible, then we can compare the classifier's performance for different values of K—and pick the best—using only the training data. As suggested at the beginning of this section, we will accomplish this by splitting the training data, training on one subset, and evaluating on the other. The subset of training data used for evaluation is often called the **validation set**.

There is, however, one key difference from the train/test split that we performed earlier. In particular, we were forced to make only a *single split* of the data. This is because at the end of the day, we have to produce a single classifier. If we had multiple different splits of the data into training and testing data, we would produce multiple different classifiers. But while we are tuning the classifier, we are free to create multiple classifiers based on multiple splits of the training data, evaluate them, and then choose a parameter value based on ***all*** of the different results. If we just split our overall training data *once*, our best parameter choice will depend strongly on whatever data was lucky enough to end up in the validation set. Perhaps using multiple different train/validation splits, we'll get a better estimate of accuracy, which will lead to a better choice of the number of neighbors K for the overall set of training data.

Let's investigate this idea in Python. In particular, we will generate five different train/validation splits of our overall training data, train five different K-nearest neighbors models, and evaluate their accuracy. We will start with just a single split.

```
# create the 25/75 split of the *training data* into sub-training and validation
cancer_subtrain, cancer_validation = train_test_split(
    cancer_train, train_size=0.75, stratify=cancer_train["Class"]
)

# fit the model on the sub-training data
knn = KNeighborsClassifier(n_neighbors=3)
X = cancer_subtrain[["Smoothness", "Concavity"]]
y = cancer_subtrain["Class"]
knn_pipeline = make_pipeline(cancer_preprocessor, knn)
knn_pipeline.fit(X, y)

# compute the score on validation data
acc = knn_pipeline.score(
    cancer_validation[["Smoothness", "Concavity"]],
    cancer_validation["Class"]
)
acc
```

```
0.897196261682243
```

The accuracy estimate using this split is 89.7%. Now we repeat the above code 4 more times, which generates 4 more splits. Therefore we get five different shuffles of the data, and therefore five different values for accuracy: [89.7%, 88.8%, 87.9%, 86.0%, 87.9%]. None of these values are necessarily "more correct" than any other; they're just five estimates of the true, underlying accuracy of our classifier built using our overall training data. We can combine the estimates by taking their average (here 88.0%) to try to get a single assessment of our classifier's accuracy; this has the effect of reducing the influence of any one (un)lucky validation set on the estimate.

In practice, we don't use random splits, but rather use a more structured splitting procedure so that each observation in the data set is used in a validation set only a single time. The name for this strategy is **cross-validation**. In **cross-validation**, we split our **overall training data** into C evenly sized chunks. Then, iteratively use 1 chunk as the **validation set** and combine the remaining $C-1$ chunks as the **training set**. This procedure is shown in Fig. 6.4. Here, $C = 5$ different chunks of the data set are used, resulting in 5 different choices for the **validation set**; we call this *5-fold* cross-validation.

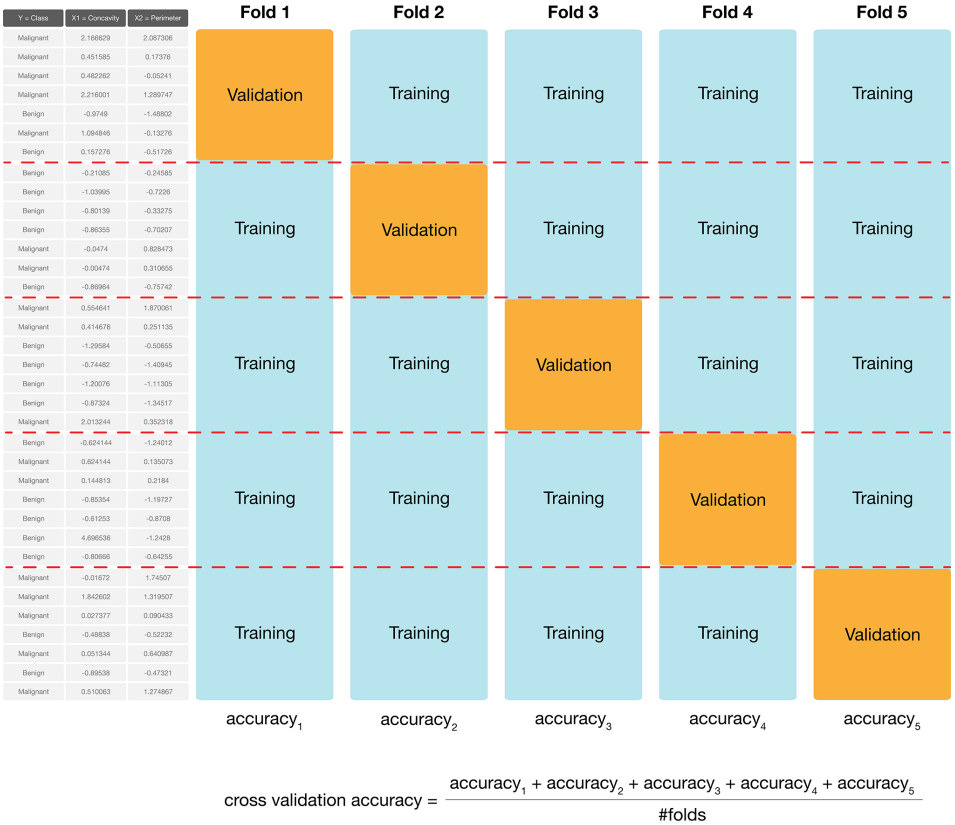


FIGURE 6.4 5-fold cross-validation.

To perform 5-fold cross-validation in Python with `scikit-learn`, we use another function: `cross_validate`. This function requires that we specify a modelling `Pipeline` as the `estimator` argument, the number of folds as the `cv` argument, and the training data predictors and labels as the `X` and `y` arguments. Since the `cross_validate` function outputs a dictionary, we use `pd.DataFrame` to convert it to a `pandas` data frame for better visualization. Note that the `cross_validate` function handles stratifying the classes in each train and validate fold automatically.

```
from sklearn.model_selection import cross_validate

knn = KNeighborsClassifier(n_neighbors=3)
cancer_pipe = make_pipeline(cancer_preprocessor, knn)
X = cancer_train[["Smoothness", "Concavity"]]
y = cancer_train["Class"]
cv_5_df = pd.DataFrame(
    cross_validate(
        estimator=cancer_pipe,
        cv=5,
        X=X,
        y=y
    )
)

cv_5_df
```

	fit_time	score_time	test_score
0	0.007128	0.009620	0.837209
1	0.006880	0.009117	0.870588
2	0.006588	0.009156	0.894118
3	0.006859	0.009262	0.870588
4	0.006076	0.009073	0.882353

The validation scores we are interested in are contained in the `test_score` column. We can then aggregate the *mean* and *standard error* of the classifier's validation accuracy across the folds. You should consider the mean (`mean`) to be the estimated accuracy, while the standard error (`sem`) is a measure of how uncertain we are in that mean value. A detailed treatment of this is beyond the scope of this chapter; but roughly, if your estimated mean is 0.87 and standard error is 0.01, you can expect the *true* average accuracy of the classifier to be somewhere roughly between 86% and 88% (although it may fall outside this range). You may ignore the other columns in the metrics data frame.

```
cv_5_metrics = cv_5_df.agg(["mean", "sem"])
cv_5_metrics
```

	fit_time	score_time	test_score
mean	0.006706	0.009246	0.870971
sem	0.000179	0.000099	0.009501

We can choose any number of folds, and typically the more we use the better our accuracy estimate will be (lower standard error). However, we are limited by computational power: the more folds we choose, the more computation it takes, and hence the more time it takes to run the analysis. So when you do cross-validation, you need to consider the size of the data, the speed of the algorithm (e.g., K-nearest neighbors), and the speed of your computer. In practice, this is a trial-and-error process, but typically C is chosen to be either 5 or 10. Here we will try 10-fold cross-validation to see if we get a lower standard error.

```
cv_10 = pd.DataFrame(
    cross_validate(
        estimator=cancer_pipe,
        cv=10,
        X=X,
        y=y
    )
)

cv_10_df = pd.DataFrame(cv_10)
cv_10_metrics = cv_10_df.agg(["mean", "sem"])
cv_10_metrics
```

```
      fit_time  score_time  test_score
mean  0.006202    0.007249    0.884939
sem   0.000178    0.000156    0.006718
```

In this case, using 10-fold instead of 5-fold cross validation did reduce the standard error very slightly. In fact, due to the randomness in how the data are split, sometimes you might even end up with a *higher* standard error when increasing the number of folds. We can make the reduction in standard error more dramatic by increasing the number of folds by a large amount. In the following code we show the result when $C = 50$; picking such a large number of folds can take a long time to run in practice, so we usually stick to 5 or 10.

```
cv_50_df = pd.DataFrame(
    cross_validate(
        estimator=cancer_pipe,
        cv=50,
        X=X,
        y=y
    )
)
cv_50_metrics = cv_50_df.agg(["mean", "sem"])
cv_50_metrics
```

```
      fit_time  score_time  test_score
mean  0.006529    0.005576    0.888056
sem   0.000104    0.000099    0.003005
```

6.6.2 Parameter value selection

Using 5- and 10-fold cross-validation, we have estimated that the prediction accuracy of our classifier is somewhere around 88%. Whether that is good or not depends entirely on the downstream application of the data analysis. In the present situation, we are trying to predict a tumor diagnosis, with expensive, damaging chemo/radiation therapy or patient death as potential consequences of misprediction. Hence, we might like to do better than 88% for this application.

In order to improve our classifier, we have one choice of parameter: the number of neighbors, K. Since cross-validation helps us evaluate the accuracy of our classifier, we can use cross-validation to calculate an accuracy for each value of K in a reasonable range, and then pick the value of K that gives us the best accuracy. The `scikit-learn` package collection provides built-in functionality, named `GridSearchCV`, to automatically handle the details for us. Before we use `GridSearchCV`, we need to create a new pipeline with a `KNeighborsClassifier` that has the number of neighbors left unspecified.

```
knn = KNeighborsClassifier()
cancer_tune_pipe = make_pipeline(cancer_preprocessor, knn)
```

Next, we specify the grid of parameter values that we want to try for each tunable parameter. We do this in a Python dictionary: the key is the identifier of the parameter to tune, and the value is a list of parameter values to try when tuning. We can find the "identifier" of a parameter by using the `get_params` method on the pipeline.

```
cancer_tune_pipe.get_params()
```

```
{'memory': None,
 'steps': [('columntransformer',
   ColumnTransformer(transformers=[('standardscaler', StandardScaler(),
                                    ['Smoothness', 'Concavity'])])),
  ('kneighborsclassifier', KNeighborsClassifier())],
 'verbose': False,
 'columntransformer': ColumnTransformer(transformers=[('standardscaler',␣
↪StandardScaler(),
                                  ['Smoothness', 'Concavity'])]),
 'kneighborsclassifier': KNeighborsClassifier(),
 'columntransformer__n_jobs': None,
 'columntransformer__remainder': 'drop',
 'columntransformer__sparse_threshold': 0.3,
 'columntransformer__transformer_weights': None,
 'columntransformer__transformers': [('standardscaler',
   StandardScaler(),
   ['Smoothness', 'Concavity'])],
 'columntransformer__verbose': False,
 'columntransformer__verbose_feature_names_out': True,
 'columntransformer__standardscaler': StandardScaler(),
 'columntransformer__standardscaler__copy': True,
```

(continues on next page)

(continued from previous page)

```
'columntransformer__standardscaler__with_mean': True,
'columntransformer__standardscaler__with_std': True,
'kneighborsclassifier__algorithm': 'auto',
'kneighborsclassifier__leaf_size': 30,
'kneighborsclassifier__metric': 'minkowski',
'kneighborsclassifier__metric_params': None,
'kneighborsclassifier__n_jobs': None,
'kneighborsclassifier__n_neighbors': 5,
'kneighborsclassifier__p': 2,
'kneighborsclassifier__weights': 'uniform'}
```

Wow, there's quite a bit of *stuff* there! If you sift through the muck a little bit, you will see one parameter identifier that stands out: `"kneighborsclassifier__n_neighbors"`. This identifier combines the name of the K nearest neighbors classification step in our pipeline, `kneighborsclassifier`, with the name of the parameter, `n_neighbors`. We now construct the `parameter_grid` dictionary that will tell `GridSearchCV` what parameter values to try. Note that you can specify multiple tunable parameters by creating a dictionary with multiple key-value pairs, but here we just have to tune the number of neighbors.

```
parameter_grid = {
    "kneighborsclassifier__n_neighbors": range(1, 100, 5),
}
```

The `range` function in Python that we used above allows us to specify a sequence of values. The first argument is the starting number (here, `1`), the second argument is *one greater than* the final number (here, `100`), and the third argument is the number to values to skip between steps in the sequence (here, `5`). So in this case we generate the sequence 1, 6, 11, 16, ..., 96. If we instead specified `range(0, 100, 5)`, we would get the sequence 0, 5, 10, 15, ..., 90, 95. The number 100 is not included because the third argument is *one greater than* the final possible number in the sequence. There are two additional useful ways to employ `range`. If we call `range` with just one argument, Python counts up to that number starting at 0. So `range(4)` is the same as `range(0, 4, 1)` and generates the sequence 0, 1, 2, 3. If we call `range` with two arguments, Python counts starting at the first number up to the second number. So `range(1, 4)` is the same as `range(1, 4, 1)` and generates the sequence `1, 2, 3`.

Okay! We are finally ready to create the `GridSearchCV` object. First, we import it from the `sklearn` package. Then we pass it the `cancer_tune_pipe` pipeline in the `estimator` argument, the `parameter_grid` in the `param_grid` argument, and specify `cv=10` folds. Note that this does not actually run the tuning yet; just as before, we will have to use the `fit` method.

```
from sklearn.model_selection import GridSearchCV

cancer_tune_grid = GridSearchCV(
    estimator=cancer_tune_pipe,
    param_grid=parameter_grid,
    cv=10
)
```

Now we use the `fit` method on the `GridSearchCV` object to begin the tuning process. We pass the training data predictors and labels as the two arguments to `fit` as usual. The `cv_results_` attribute of the output contains the resulting cross-validation accuracy estimate for each choice of `n_neighbors`, but it isn't in an easily used format. We will wrap it in a `pd.DataFrame` to make it easier to understand, and print the `info` of the result.

```
cancer_tune_grid.fit(
    cancer_train[["Smoothness", "Concavity"]],
    cancer_train["Class"]
)
accuracies_grid = pd.DataFrame(cancer_tune_grid.cv_results_)
accuracies_grid.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20 entries, 0 to 19
Data columns (total 19 columns):
 #   Column                                Non-Null Count  Dtype
---  ------                                --------------  -----
 0   mean_fit_time                         20 non-null     float64
 1   std_fit_time                          20 non-null     float64
 2   mean_score_time                       20 non-null     float64
 3   std_score_time                        20 non-null     float64
 4   param_kneighborsclassifier__n_neighbors  20 non-null  object
 5   params                                20 non-null     object
 6   split0_test_score                     20 non-null     float64
 7   split1_test_score                     20 non-null     float64
 8   split2_test_score                     20 non-null     float64
 9   split3_test_score                     20 non-null     float64
 10  split4_test_score                     20 non-null     float64
 11  split5_test_score                     20 non-null     float64
 12  split6_test_score                     20 non-null     float64
 13  split7_test_score                     20 non-null     float64
 14  split8_test_score                     20 non-null     float64
 15  split9_test_score                     20 non-null     float64
 16  mean_test_score                       20 non-null     float64
 17  std_test_score                        20 non-null     float64
 18  rank_test_score                       20 non-null     int32
dtypes: float64(16), int32(1), object(2)
memory usage: 3.0+ KB
```

There is a lot of information to look at here, but we are most interested in three quantities: the number of neighbors (`param_kneighbors_classifier__n_neighbors`), the cross-validation accuracy estimate (`mean_test_score`), and the standard error of the accuracy estimate. Unfortunately `GridSearchCV` does not directly output the standard error for each cross-validation accuracy; but it *does* output the

standard *deviation* (`std_test_score`). We can compute the standard error from the standard deviation by dividing it by the square root of the number of folds, i.e.,

$$\text{Standard Error} = \frac{\text{Standard Deviation}}{\sqrt{\text{Number of Folds}}}.$$

We will also rename the parameter name column to be a bit more readable, and drop the now unused `std_test_score` column.

```
accuracies_grid["sem_test_score"] = accuracies_grid["std_test_score"] / 10**(1/2)
accuracies_grid = (
    accuracies_grid[[
        "param_kneighborsclassifier__n_neighbors",
        "mean_test_score",
        "sem_test_score"
    ]]
    .rename(columns={"param_kneighborsclassifier__n_neighbors": "n_neighbors"})
)
accuracies_grid
```

	n_neighbors	mean_test_score	sem_test_score
0	1	0.845127	0.019966
1	6	0.873200	0.015680
2	11	0.861517	0.019547
3	16	0.861573	0.017787
4	21	0.866279	0.017889
5	26	0.875637	0.016026
6	31	0.885050	0.015406
7	36	0.887375	0.013694
8	41	0.887375	0.013694
9	46	0.887320	0.013314
10	51	0.882669	0.014523
11	56	0.878018	0.014414
12	61	0.880343	0.014299
13	66	0.873200	0.015416
14	71	0.877962	0.013660
15	76	0.873200	0.014698
16	81	0.873200	0.014698
17	86	0.880288	0.011277
18	91	0.875581	0.012967
19	96	0.875581	0.008193

We can decide which number of neighbors is best by plotting the accuracy versus K, as shown in Fig. 6.5. Here we are using the shortcut `point=True` to layer a point and line chart.

```
accuracy_vs_k = alt.Chart(accuracies_grid).mark_line(point=True).encode(
    x=alt.X("n_neighbors").title("Neighbors"),
    y=alt.Y("mean_test_score")
        .scale(zero=False)
        .title("Accuracy estimate")
)

accuracy_vs_k
```

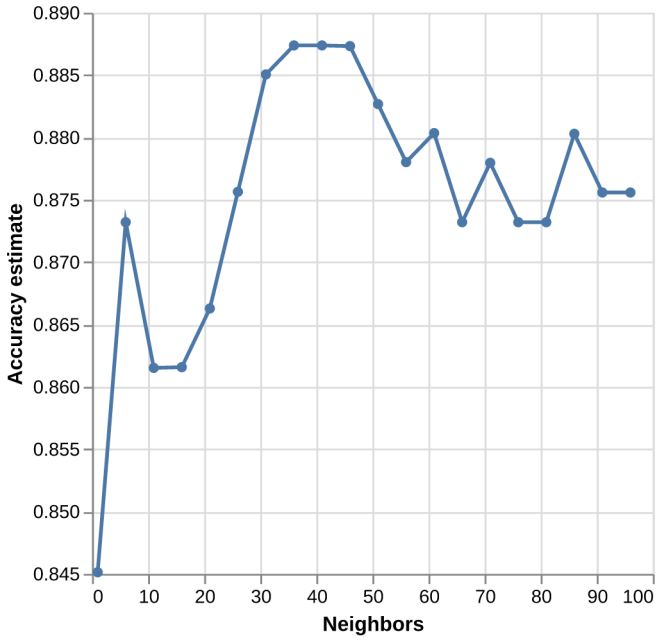


FIGURE 6.5 Plot of estimated accuracy versus the number of neighbors.

We can also obtain the number of neighbors with the highest accuracy programmatically by accessing the `best_params_` attribute of the fit `GridSearchCV` object. Note that it is still useful to visualize the results as we did above since this provides additional information on how the model performance varies.

```
cancer_tune_grid.best_params_
```

```
{'kneighborsclassifier__n_neighbors': 36}
```

Setting the number of neighbors to $K = 36$ provides the highest cross-validation accuracy estimate (88.7%). But there is no exact or perfect answer here; any selection from $K = 30$ to 80 or so would be reasonably justified, as all of these differ in classifier accuracy by a small amount. Remember: the values you see on this plot are *estimates* of the true accuracy of our classifier. Although the $K = 36$ value is higher than the others on this plot, that doesn't mean the classifier is actually more accurate with this parameter value. Generally, when selecting K (and other parameters for other predictive models), we are looking for a value where:

- we get roughly optimal accuracy, so that our model will likely be accurate;
- changing the value to a nearby one (e.g., adding or subtracting a small number) doesn't decrease accuracy too much, so that our choice is reliable in the presence of uncertainty;

- the cost of training the model is not prohibitive (e.g., in our situation, if K is too large, predicting becomes expensive!).

We know that $K = 36$ provides the highest estimated accuracy. Further, Fig. 6.5 shows that the estimated accuracy changes by only a small amount if we increase or decrease K near $K = 36$. And finally, $K = 36$ does not create a prohibitively expensive computational cost of training. Considering these three points, we would indeed select $K = 36$ for the classifier.

6.6.3 Under/Overfitting

To build a bit more intuition, what happens if we keep increasing the number of neighbors K? In fact, the cross-validation accuracy estimate actually starts to decrease. Let's specify a much larger range of values of K to try in the `param_grid` argument of `GridSearchCV`. Fig. 6.6 shows a plot of estimated accuracy as we vary K from 1 to almost the number of observations in the data set.

```
large_param_grid = {
    "kneighborsclassifier__n_neighbors": range(1, 385, 10),
}

large_cancer_tune_grid = GridSearchCV(
    estimator=cancer_tune_pipe,
    param_grid=large_param_grid,
    cv=10
)

large_cancer_tune_grid.fit(
    cancer_train[["Smoothness", "Concavity"]],
    cancer_train["Class"]
)

large_accuracies_grid = pd.DataFrame(large_cancer_tune_grid.cv_results_)

large_accuracy_vs_k = alt.Chart(large_accuracies_grid).mark_line(point=True).
↪encode(
    x=alt.X("param_kneighborsclassifier__n_neighbors").title("Neighbors"),
    y=alt.Y("mean_test_score")
        .scale(zero=False)
        .title("Accuracy estimate")
)

large_accuracy_vs_k
```

Underfitting: What is actually happening to our classifier that causes this? As we increase the number of neighbors, more and more of the training observations (and those that are farther and farther away from the point) get a "say" in what the class of a new observation is. This causes a sort of "averaging effect" to take place, making the boundary between where our classifier would predict a tumor to be malignant versus benign to smooth out and become *simpler*. If you take this to the extreme, setting K to the total training

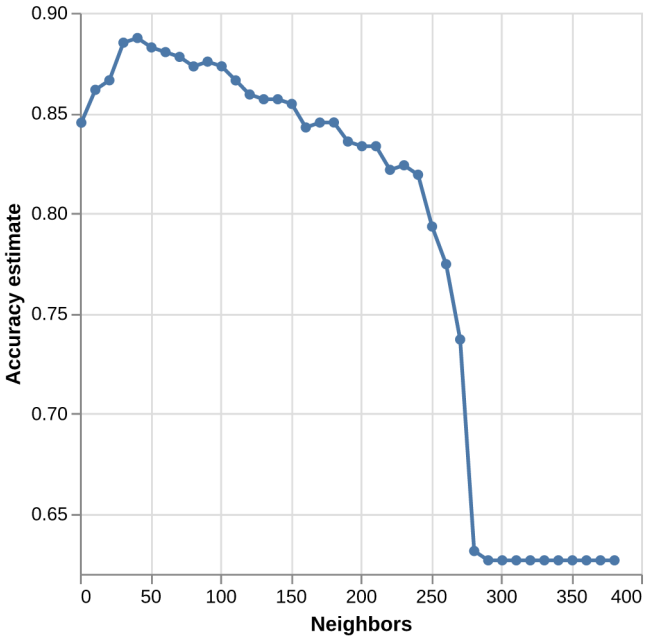


FIGURE 6.6 Plot of accuracy estimate versus number of neighbors for many K values.

data set size, then the classifier will always predict the same label regardless of what the new observation looks like. In general, if the model *isn't influenced enough* by the training data, it is said to **underfit** the data.

Overfitting: In contrast, when we decrease the number of neighbors, each individual data point has a stronger and stronger vote regarding nearby points. Since the data themselves are noisy, this causes a more "jagged" boundary corresponding to a *less simple* model. If you take this case to the extreme, setting $K = 1$, then the classifier is essentially just matching each new observation to its closest neighbor in the training data set. This is just as problematic as the large K case, because the classifier becomes unreliable on new data: if we had a different training set, the predictions would be completely different. In general, if the model *is influenced too much* by the training data, it is said to **overfit** the data.

Both overfitting and underfitting are problematic and will lead to a model that does not generalize well to new data. When fitting a model, we need to strike a balance between the two. You can see these two effects in Fig. 6.7, which shows how the classifier changes as we set the number of neighbors K to 1, 7, 20, and 300.

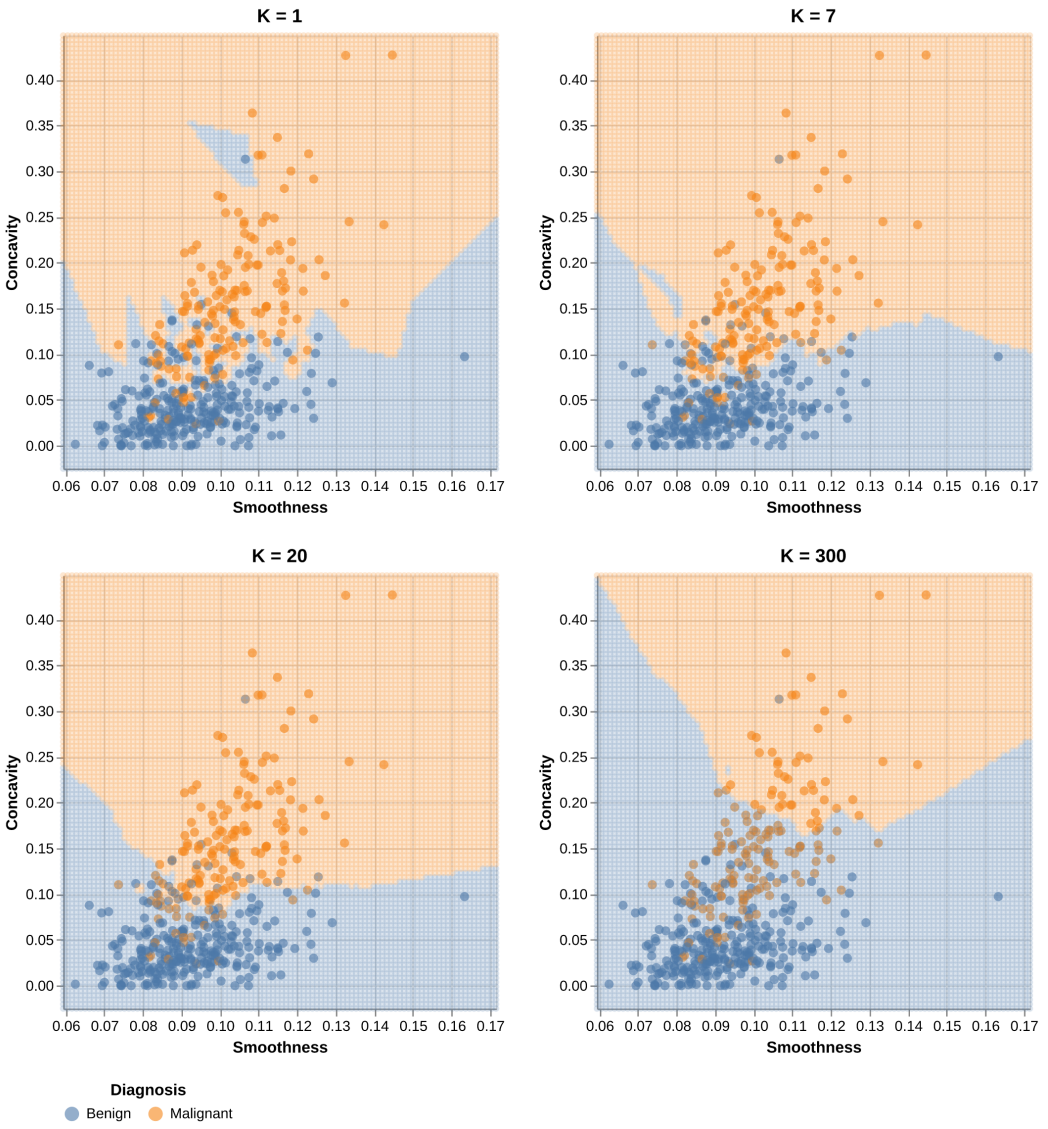


FIGURE 6.7 Effect of K in overfitting and underfitting.

6.6.4 Evaluating on the test set

Now that we have tuned the K-NN classifier and set $K = 36$, we are done building the model and it is time to evaluate the quality of its predictions on the held out test data, as we did earlier in Section 6.5.5. We first need to retrain the K-NN classifier on the entire training data set using the selected number of neighbors. Fortunately we do not have to do this ourselves manually; `scikit-learn` does it for us automatically. To make predictions and assess the estimated accuracy of the best model on the test data, we can use the `score` and `predict` methods of the fit `GridSearchCV` object. We

can then pass those predictions to the `precision`, `recall`, and `crosstab` functions to assess the estimated precision and recall, and print a confusion matrix.

```
cancer_test["predicted"] = cancer_tune_grid.predict(
    cancer_test[["Smoothness", "Concavity"]]
)

cancer_tune_grid.score(
    cancer_test[["Smoothness", "Concavity"]],
    cancer_test["Class"]
)
```

```
0.9090909090909091
```

```
precision_score(
    y_true=cancer_test["Class"],
    y_pred=cancer_test["predicted"],
    pos_label='Malignant'
)
```

```
0.8846153846153846
```

```
recall_score(
    y_true=cancer_test["Class"],
    y_pred=cancer_test["predicted"],
    pos_label='Malignant'
)
```

```
0.8679245283018868
```

```
pd.crosstab(
    cancer_test["Class"],
    cancer_test["predicted"]
)
```

```
predicted  Benign  Malignant
Class
Benign         84          6
Malignant       7         46
```

At first glance, this is a bit surprising: the accuracy of the classifier has not changed much despite tuning the number of neighbors. Our first model with $K = 3$ (before we knew how to tune) had an estimated accuracy of 90%, while the tuned model with $K = 36$ had an estimated accuracy of 91%. Upon examining Fig. 6.5 again to see the cross validation accuracy estimates for a range of neighbors, this result becomes much less surprising. From 1 to around 96 neighbors, the cross validation accuracy estimate varies only by around 3%, with each estimate having a standard error around 1%. Since the cross-validation accuracy estimates the test set accuracy, the fact that

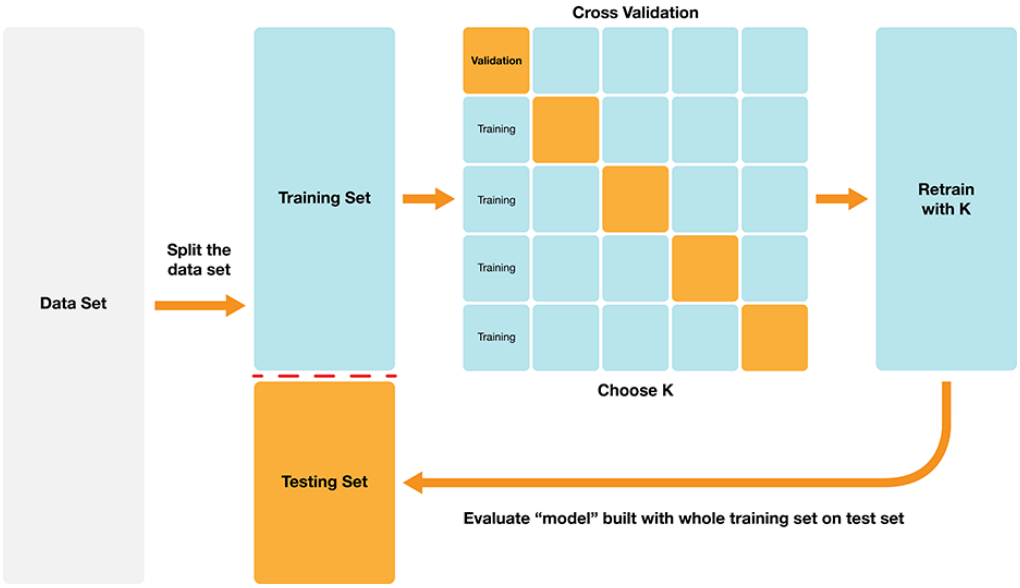


FIGURE 6.8 Overview of K-NN classification.

the test set accuracy also doesn't change much is expected. Also note that the $K = 3$ model had a precision precision of 83% and recall of 91%, while the tuned model had a precision of 88% and recall of 87%. Given that the recall decreased—remember, in this application, recall is critical to making sure we find all the patients with malignant tumors—the tuned model may actually be *less* preferred in this setting. In any case, it is important to think critically about the result of tuning. Models tuned to maximize accuracy are not necessarily better for a given application.

6.7 Summary

Classification algorithms use one or more quantitative variables to predict the value of another categorical variable. In particular, the K-nearest neighbors algorithm does this by first finding the K points in the training data nearest to the new observation, and then returning the majority class vote from those training observations. We can tune and evaluate a classifier by splitting the data randomly into a training and test data set. The training set is used to build the classifier, and we can tune the classifier (e.g., select the number of neighbors in K-nearest neighbors) by maximizing estimated accuracy via cross-validation. After we have tuned the model, we can use the test set to estimate its accuracy. The overall process is summarized in Fig. 6.8.

The overall workflow for performing K-nearest neighbors classification using `scikit-learn` is as follows:

1. Use the `train_test_split` function to split the data into a training and test set. Set the `stratify` argument to the class label column of the data frame. Put the test set aside for now.
2. Create a `Pipeline` that specifies the preprocessing steps and the classifier.
3. Define the parameter grid by passing the set of K values that you would like to tune.
4. Use `GridSearchCV` to estimate the classifier accuracy for a range of K values. Pass the pipeline and parameter grid defined in steps 2. and 3. as the `param_grid` argument and the `estimator` argument, respectively.
5. Execute the grid search by passing the training data to the `fit` method on the `GridSearchCV` instance created in step 4.
6. Pick a value of K that yields a high cross-validation accuracy estimate that doesn't change much if you change K to a nearby value.
7. Create a new model object for the best parameter value (i.e., K), and retrain the classifier by calling the `fit` method.
8. Evaluate the estimated accuracy of the classifier on the test set using the `score` method.

In these last two chapters, we focused on the K-nearest neighbors algorithm, but there are many other methods we could have used to predict a categorical label. All algorithms have their strengths and weaknesses, and we summarize these for the K-NN here.

Strengths: K-nearest neighbors classification

1. is a simple, intuitive algorithm,
2. requires few assumptions about what the data must look like, and
3. works for binary (two-class) and multi-class (more than 2 classes) classification problems.

Weaknesses: K-nearest neighbors classification

1. becomes very slow as the training data gets larger,

2. may not perform well with a large number of predictors, and
3. may not perform well when classes are imbalanced.

6.8 Predictor variable selection

Note: This section is not required reading for the remainder of the textbook. It is included for those readers interested in learning how irrelevant variables can influence the performance of a classifier, and how to pick a subset of useful variables to include as predictors.

Another potentially important part of tuning your classifier is to choose which variables from your data will be treated as predictor variables. Technically, you can choose anything from using a single predictor variable to using every variable in your data; the K-nearest neighbors algorithm accepts any number of predictors. However, it is **not** the case that using more predictors always yields better predictions. In fact, sometimes including irrelevant predictors can actually negatively affect classifier performance.

6.8.1 The effect of irrelevant predictors

Let's take a look at an example where K-nearest neighbors performs worse when given more predictors to work with. In this example, we modified the breast cancer data to have only the `Smoothness`, `Concavity`, and `Perimeter` variables from the original data. Then, we added irrelevant variables that we created ourselves using a random number generator. The irrelevant variables each take a value of 0 or 1 with equal probability for each observation, regardless of what the value `Class` variable takes. In other words, the irrelevant variables have no meaningful relationship with the `Class` variable.

```
cancer_irrelevant[
    ["Class", "Smoothness", "Concavity", "Perimeter", "Irrelevant1", "Irrelevant2
↪"]
]
```

	Class	Smoothness	Concavity	Perimeter	Irrelevant1	Irrelevant2
0	Malignant	0.11840	0.30010	122.80	0	1
1	Malignant	0.08474	0.08690	132.90	0	1
2	Malignant	0.10960	0.19740	130.00	1	0
3	Malignant	0.14250	0.24140	77.58	1	0
4	Malignant	0.10030	0.19800	135.10	1	0
..	...	...	...	...	...	...

(continues on next page)

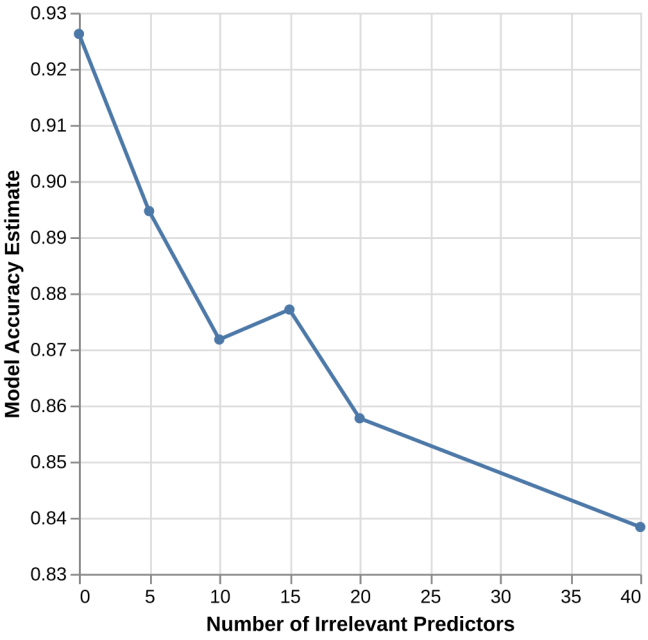


FIGURE 6.9 Effect of inclusion of irrelevant predictors.

(continued from previous page)

```
564  Malignant     0.11100     0.24390     142.00          0          0
565  Malignant     0.09780     0.14400     131.20          0          1
566  Malignant     0.08455     0.09251     108.30          1          1
567  Malignant     0.11780     0.35140     140.10          0          0
568     Benign     0.05263     0.00000      47.92          1          1

[569 rows x 6 columns]
```

Next, we build a sequence of K-NN classifiers that include `Smoothness`, `Concavity`, and `Perimeter` as predictor variables, but also increasingly many irrelevant variables. In particular, we create 6 data sets with 0, 5, 10, 15, 20, and 40 irrelevant predictors. Then we build a model, tuned via 5-fold cross-validation, for each data set. Fig. 6.9 shows the estimated cross-validation accuracy versus the number of irrelevant predictors. As we add more irrelevant predictor variables, the estimated accuracy of our classifier decreases. This is because the irrelevant variables add a random amount to the distance between each pair of observations; the more irrelevant variables there are, the more (random) influence they have, and the more they corrupt the set of nearest neighbors that vote on the class of the new observation to predict.

Although the accuracy decreases as expected, one surprising thing about Fig. 6.9 is that it shows that the method still outperforms the baseline majority classifier (with about 63% accuracy) even with 40 irrelevant variables. How could that be? Fig. 6.10 provides the answer: the tuning procedure for the

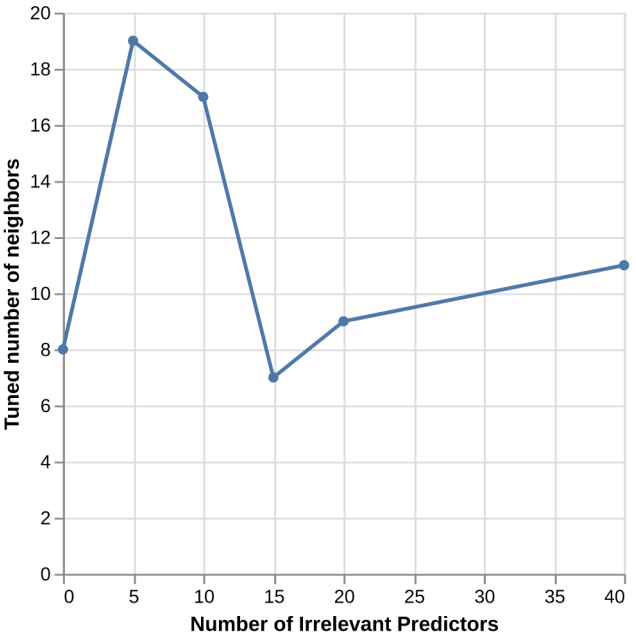


FIGURE 6.10 Tuned number of neighbors for varying number of irrelevant predictors.

K-nearest neighbors classifier combats the extra randomness from the irrelevant variables by increasing the number of neighbors. Of course, because of all the extra noise in the data from the irrelevant variables, the number of neighbors does not increase smoothly; but the general trend is increasing. Fig. 6.11 corroborates this evidence; if we fix the number of neighbors to $K = 3$, the accuracy falls off more quickly.

6.8.2 Finding a good subset of predictors

So then, if it is not ideal to use all of our variables as predictors without consideration, how do we choose which variables we *should* use? A simple method is to rely on your scientific understanding of the data to tell you which variables are not likely to be useful predictors. For example, in the cancer data that we have been studying, the `ID` variable is just a unique identifier for the observation. As it is not related to any measured property of the cells, the `ID` variable should therefore not be used as a predictor. That is, of course, a very clear-cut case. But the decision for the remaining variables is less obvious, as all seem like reasonable candidates. It is not clear which subset of them will create the best classifier. One could use visualizations and other exploratory analyses to try to help understand which variables are potentially relevant, but this process is both time-consuming and error-prone when there are many

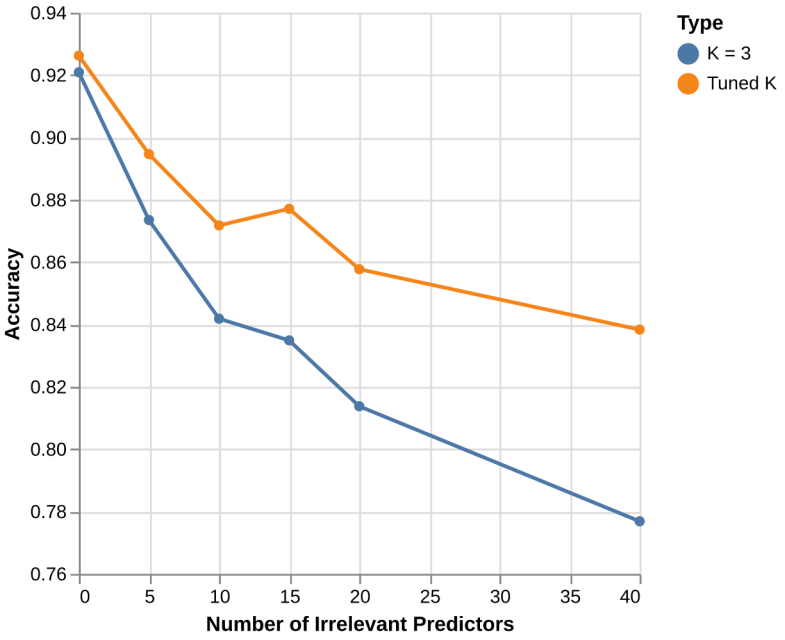


FIGURE 6.11 Accuracy versus number of irrelevant predictors for tuned and untuned number of neighbors.

variables to consider. Therefore we need a more systematic and programmatic way of choosing variables. This is a very difficult problem to solve in general, and there are a number of methods that have been developed that apply in particular cases of interest. Here we will discuss two basic selection methods as an introduction to the topic. See the additional resources at the end of this chapter to find out where you can learn more about variable selection, including more advanced methods.

The first idea you might think of for a systematic way to select predictors is to try all possible subsets of predictors and then pick the set that results in the "best" classifier. This procedure is indeed a well-known variable selection method referred to as *best subset selection* [Beale *et al.*, 1967, Hocking and Leslie, 1967]. In particular, you

1. create a separate model for every possible subset of predictors,
2. tune each one using cross-validation, and
3. pick the subset of predictors that gives you the highest cross-validation accuracy.

Best subset selection is applicable to any classification method (K-NN or otherwise). However, it becomes very slow when you have even a moderate number

of predictors to choose from (say, around 10). This is because the number of possible predictor subsets grows very quickly with the number of predictors, and you have to train the model (itself a slow process!) for each one. For example, if we have 2 predictors—let's call them A and B—then we have 3 variable sets to try: A alone, B alone, and finally A and B together. If we have 3 predictors—A, B, and C—then we have 7 to try: A, B, C, AB, BC, AC, and ABC. In general, the number of models we have to train for m predictors is $2^m - 1$; in other words, when we get to 10 predictors we have over *one thousand* models to train, and at 20 predictors we have over *one million* models to train. So although it is a simple method, best subset selection is usually too computationally expensive to use in practice.

Another idea is to iteratively build up a model by adding one predictor variable at a time. This method—known as *forward selection* [Draper and Smith, 1966, Eforymson, 1966]—is also widely applicable and fairly straightforward. It involves the following steps:

1. Start with a model having no predictors.
2. Run the following 3 steps until you run out of predictors:
    1. For each unused predictor, add it to the model to form a *candidate model*.
    2. Tune all of the candidate models.
    3. Update the model to be the candidate model with the highest cross-validation accuracy.
3. Select the model that provides the best trade-off between accuracy and simplicity.

Say you have m total predictors to work with. In the first iteration, you have to make m candidate models, each with 1 predictor. Then in the second iteration, you have to make $m-1$ candidate models, each with 2 predictors (the one you chose before and a new one). This pattern continues for as many iterations as you want. If you run the method all the way until you run out of predictors to choose, you will end up training $\frac{1}{2}m(m+1)$ separate models. This is a *big* improvement from the $2^m - 1$ models that best subset selection requires you to train. For example, while best subset selection requires training over 1000 candidate models with 10 predictors, forward selection requires training only 55 candidate models. Therefore we will continue the rest of this section using forward selection.

Note: One word of caution before we move on. Every additional model

that you train increases the likelihood that you will get unlucky and stumble on a model that has a high cross-validation accuracy estimate, but a low true accuracy on the test data and other future observations. Since forward selection involves training a lot of models, you run a fairly high risk of this happening. To keep this risk low, only use forward selection when you have a large amount of data and a relatively small total number of predictors. More advanced methods do not suffer from this problem as much; see the additional resources at the end of this chapter for where to learn more about advanced predictor selection methods.

6.8.3 Forward selection in Python

We now turn to implementing forward selection in Python. First, we will extract a smaller set of predictors to work with in this illustrative example—`Smoothness`, `Concavity`, `Perimeter`, `Irrelevant1`, `Irrelevant2`, and `Irrelevant3`—as well as the `Class` variable as the label. We will also extract the column names for the full set of predictors.

```
cancer_subset = cancer_irrelevant[
    [
        "Class",
        "Smoothness",
        "Concavity",
        "Perimeter",
        "Irrelevant1",
        "Irrelevant2",
        "Irrelevant3",
    ]
]

names = list(cancer_subset.drop(
    columns=["Class"]
).columns.values)

cancer_subset
```

```
         Class  Smoothness  Concavity  Perimeter  Irrelevant1  Irrelevant2  \
0    Malignant     0.11840    0.30010     122.80            0            1
1    Malignant     0.08474    0.08690     132.90            0            1
2    Malignant     0.10960    0.19740     130.00            1            0
3    Malignant     0.14250    0.24140      77.58            1            0
4    Malignant     0.10030    0.19800     135.10            1            0
..         ...         ...        ...        ...          ...          ...
564  Malignant     0.11100    0.24390     142.00            0            0
565  Malignant     0.09780    0.14400     131.20            0            1
566  Malignant     0.08455    0.09251     108.30            1            1
567  Malignant     0.11780    0.35140     140.10            0            0
568     Benign     0.05263    0.00000      47.92            1            1

     Irrelevant3
0              0
1              0
```

(continues on next page)

(continued from previous page)

```
2              0
3              0
4              1
..           ...
564            0
565            0
566            0
567            0
568            1

[569 rows x 7 columns]
```

To perform forward selection, we could use the `SequentialFeatureSelector`[3] from `scikit-learn`; but it is difficult to combine this approach with parameter tuning to find a good number of neighbors for each set of features. Instead we will code the forward selection algorithm manually. In particular, we need code that tries adding each available predictor to a model, finding the best, and iterating. If you recall the end of the wrangling chapter, we mentioned that sometimes one needs more flexible forms of iteration than what we have used earlier, and in these cases one typically resorts to a *for loop*; see the control flow section[4] in *Python for Data Analysis* [McKinney, 2012]. Here we will use two for loops: one over increasing predictor set sizes (where you see `for i in range(1, n_total + 1):` below), and another to check which predictor to add in each round (where you see `for j in range(len(names))` below). For each set of predictors to try, we extract the subset of predictors, pass it into a preprocessor, build a `Pipeline` that tunes a K-NN classifier using 10-fold cross-validation, and finally records the estimated accuracy.

```
from sklearn.compose import make_column_selector

accuracy_dict = {"size": [], "selected_predictors": [], "accuracy": []}

# store the total number of predictors
n_total = len(names)

# start with an empty list of selected predictors
selected = []

# create the pipeline and CV grid search objects
param_grid = {
    "kneighborsclassifier__n_neighbors": range(1, 61, 5),
}
cancer_preprocessor = make_column_transformer(
    (StandardScaler(), make_column_selector(dtype_include="number"))
)
cancer_tune_pipe = make_pipeline(cancer_preprocessor, KNeighborsClassifier())
```

(continues on next page)

[3] https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SequentialFeatureSelector.html

[4] https://wesmckinney.com/book/python-basics.html#control_for

(continued from previous page)

```
cancer_tune_grid = GridSearchCV(
    estimator=cancer_tune_pipe,
    param_grid=param_grid,
    cv=10,
    n_jobs=-1
)

# for every possible number of predictors
for i in range(1, n_total + 1):
    accs = np.zeros(len(names))
    # for every possible predictor to add
    for j in range(len(names)):
        # Add remaining predictor j to the model
        X = cancer_subset[selected + [names[j]]]
        y = cancer_subset["Class"]

        # Find the best K for this set of predictors
        cancer_tune_grid.fit(X, y)
        accuracies_grid = pd.DataFrame(cancer_tune_grid.cv_results_)

        # Store the tuned accuracy for this set of predictors
        accs[j] = accuracies_grid["mean_test_score"].max()

    # get the best new set of predictors that maximize cv accuracy
    best_set = selected + [names[accs.argmax()]]

    # store the results for this round of forward selection
    accuracy_dict["size"].append(i)
    accuracy_dict["selected_predictors"].append(", ".join(best_set))
    accuracy_dict["accuracy"].append(accs.max())

    # update the selected & available sets of predictors
    selected = best_set
    del names[accs.argmax()]

accuracies = pd.DataFrame(accuracy_dict)
accuracies
```

	size	selected_predictors	accuracy
0	1	Perimeter	0.891103
1	2	Perimeter, Concavity	0.917450
2	3	Perimeter, Concavity, Smoothness	0.931454
3	4	Perimeter, Concavity, Smoothness, Irrelevant1	0.926253
4	5	Perimeter, Concavity, Smoothness, Irrelevant1,...	0.926253
5	6	Perimeter, Concavity, Smoothness, Irrelevant1,...	0.906955

Interesting! The forward selection procedure first added the three meaningful variables `Perimeter`, `Concavity`, and `Smoothness`, followed by the irrelevant variables. Fig. 6.12 visualizes the accuracy versus the number of predictors in the model. You can see that as meaningful predictors are added, the estimated accuracy increases substantially; and as you add irrelevant variables, the accuracy either exhibits small fluctuations or decreases as the model attempts to tune the number of neighbors to account for the extra noise. In order to pick the right model from the sequence, you have to balance high accuracy and model simplicity (i.e., having fewer predictors and a lower chance of overfitting). The way to find that balance is to look for the *elbow* in Fig.

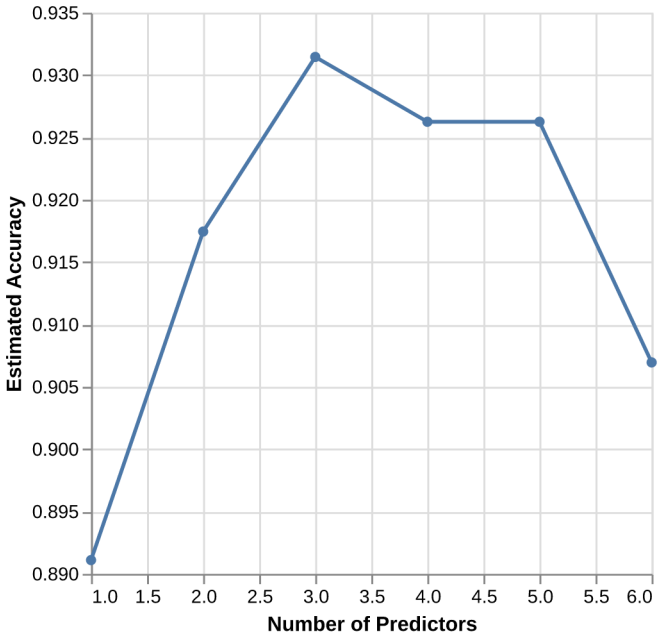


FIGURE 6.12 Estimated accuracy versus the number of predictors for the sequence of models built using forward selection.

6.12, i.e., the place on the plot where the accuracy stops increasing dramatically and levels off or begins to decrease. The elbow in Fig. 6.12 appears to occur at the model with 3 predictors; after that point the accuracy levels off. So here the right trade-off of accuracy and number of predictors occurs with 3 variables: `Perimeter, Concavity, Smoothness`. In other words, we have successfully removed irrelevant predictors from the model. It is always worth remembering, however, that what cross-validation gives you is an *estimate* of the true accuracy; you have to use your judgement when looking at this plot to decide where the elbow occurs, and whether adding a variable provides a meaningful increase in accuracy.

Note: Since the choice of which variables to include as predictors is part of tuning your classifier, you *cannot use your test data* for this process.

6.9 Exercises

Practice exercises for the material covered in this chapter can be found in the accompanying worksheets repository[5] in the "Classification II: evaluation

[5] https://worksheets.python.datasciencebook.ca

and tuning" row. You can launch an interactive version of the worksheet in your browser by clicking the "launch binder" button. You can also preview a non-interactive version of the worksheet by clicking "view worksheet". If you instead decide to download the worksheet and run it on your own machine, make sure to follow the instructions for computer setup found in Chapter 13. This will ensure that the automated feedback and guidance that the worksheets provide will function as intended.

6.10 Additional resources

- The `scikit-learn` website[6] is an excellent reference for more details on, and advanced usage of, the functions and packages in the past two chapters. Aside from that, it also offers many useful tutorials[7] to get you started. It's worth noting that the `scikit-learn` package does a lot more than just classification, and so the examples on the website similarly go beyond classification as well. In the next two chapters, you'll learn about another kind of predictive modeling setting, so it might be worth visiting the website only after reading through those chapters.
- *An Introduction to Statistical Learning*[8] [James *et al.*, 2013] provides a great next stop in the process of learning about classification. Chapter 4 discusses additional basic techniques for classification that we do not cover, such as logistic regression, linear discriminant analysis, and naive Bayes. Chapter 5 goes into much more detail about cross-validation. Chapters 8 and 9 cover decision trees and SVMs, two very popular but more advanced classification methods. Finally, Chapter 6 covers a number of methods for selecting predictor variables. Note that while this book is still a very accessible introductory text, it requires a bit more mathematical background than we require.

[6] https://scikit-learn.org/stable/
[7] https://scikit-learn.org/stable/tutorial/index.html
[8] https://www.statlearning.com/

7

Regression I: K-nearest neighbors

7.1 Overview

This chapter continues our foray into answering predictive questions. Here we will focus on predicting *numerical* variables and will use *regression* to perform this task. This is unlike the past two chapters, which focused on predicting categorical variables via classification. However, regression does have many similarities to classification: for example, just as in the case of classification, we will split our data into training, validation, and test sets, we will use `scikit-learn` workflows, we will use a K-nearest neighbors (K-NN) approach to make predictions, and we will use cross-validation to choose K. Because of how similar these procedures are, make sure to read Chapters 5 and 6 before reading this one—we will move a little bit faster here with the concepts that have already been covered. This chapter will primarily focus on the case where there is a single predictor, but the end of the chapter shows how to perform regression with more than one predictor variable, i.e., *multivariable regression*. It is important to note that regression can also be used to answer inferential and causal questions, however that is beyond the scope of this book.

7.2 Chapter learning objectives

By the end of the chapter, readers will be able to do the following:

- Recognize situations where a regression analysis would be appropriate for making predictions.
- Explain the K-NN regression algorithm and describe how it differs from K-NN classification.
- Interpret the output of a K-NN regression.
- In a data set with two or more variables, perform K-NN regression in Python.

DOI: 10.1201/9781003438397-7

- Evaluate K-NN regression prediction quality in Python using the root mean squared prediction error (RMSPE).
- Estimate the RMSPE in Python using cross-validation or a test set.
- Choose the number of neighbors in K-NN regression by minimizing estimated cross-validation RMSPE.
- Describe underfitting and overfitting, and relate it to the number of neighbors in K-NN regression.
- Describe the advantages and disadvantages of K-NN regression.

7.3 The regression problem

Regression, like classification, is a predictive problem setting where we want to use past information to predict future observations. But in the case of regression, the goal is to predict *numerical* values instead of *categorical* values. The variable that you want to predict is often called the *response variable*. For example, we could try to use the number of hours a person spends on exercise each week to predict their race time in the annual Boston marathon. As another example, we could try to use the size of a house to predict its sale price. Both of these response variables—race time and sale price—are numerical, and so predicting them given past data is considered a regression problem.

Just like in the classification setting, there are many possible methods that we can use to predict numerical response variables. In this chapter, we will focus on the **K-NN** algorithm [Cover and Hart, 1967, Fix and Hodges, 1951], and in the next chapter we will study **linear regression**. In your future studies, you might encounter regression trees, splines, and general local regression methods; see the additional resources section at the end of the next chapter for where to begin learning more about these other methods.

Many of the concepts from classification map over to the setting of regression. For example, a regression model predicts a new observation's response variable based on the response variables for similar observations in the data set of past observations. When building a regression model, we first split the data into training and test sets, in order to ensure that we assess the performance of our method on observations not seen during training. And finally, we can use cross-validation to evaluate different choices of model parameters (e.g., K in a

K-NN model). The major difference is that we are now predicting numerical variables instead of categorical variables.

Note: You can usually tell whether a variable is numerical or categorical—and therefore whether you need to perform regression or classification—by taking the response variable for two observations X and Y from your data, and asking the question, "is response variable X *more* than response variable Y?" If the variable is categorical, the question will make no sense. (Is blue more than red? Is benign more than malignant?) If the variable is numerical, it will make sense. (Is 1.5 hours more than 2.25 hours? Is $500,000 more than $400,000?) Be careful when applying this heuristic, though: sometimes categorical variables will be encoded as numbers in your data (e.g., "1" represents "benign", and "0" represents "malignant"). In these cases you have to ask the question about the *meaning* of the labels ("benign" and "malignant"), not their values ("1" and "0").

7.4 Exploring a data set

In this chapter and the next, we will study a data set of 932 real estate transactions in Sacramento, California[1] originally reported in the *Sacramento Bee* newspaper. We first need to formulate a precise question that we want to answer. In this example, our question is again predictive: Can we use the size of a house in the Sacramento, CA area to predict its sale price? A rigorous, quantitative answer to this question might help a realtor advise a client as to whether the price of a particular listing is fair, or perhaps how to set the price of a new listing. We begin the analysis by loading and examining the data, as well as setting the seed value.

```
import altair as alt
import numpy as np
import pandas as pd
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.compose import make_column_transformer
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn import set_config

# Output dataframes instead of arrays
set_config(transform_output="pandas")
```

(continues on next page)

[1] https://support.spatialkey.com/spatialkey-sample-csv-data/

(continued from previous page)

```
np.random.seed(10)

sacramento = pd.read_csv("data/sacramento.csv")
sacramento
```

```
                city     zip  beds  baths  sqft         type   price  \
0         SACRAMENTO  z95838     2    1.0   836  Residential   59222
1         SACRAMENTO  z95823     3    1.0  1167  Residential   68212
2         SACRAMENTO  z95815     2    1.0   796  Residential   68880
3         SACRAMENTO  z95815     2    1.0   852  Residential   69307
4         SACRAMENTO  z95824     2    1.0   797  Residential   81900
..               ...     ...   ...    ...   ...          ...     ...
927       SACRAMENTO  z95829     4    3.0  2280  Residential  232425
928       SACRAMENTO  z95823     3    2.0  1477  Residential  234000
929   CITRUS_HEIGHTS  z95610     3    2.0  1216  Residential  235000
930        ELK_GROVE  z95758     4    2.0  1685  Residential  235301
931  EL_DORADO_HILLS  z95762     3    2.0  1362  Residential  235738

      latitude   longitude
0    38.631913 -121.434879
1    38.478902 -121.431028
2    38.618305 -121.443839
3    38.616835 -121.439146
4    38.519470 -121.435768
..         ...         ...
927  38.457679 -121.359620
928  38.499893 -121.458890
929  38.708824 -121.256803
930  38.417000 -121.397424
931  38.655245 -121.075915

[932 rows x 9 columns]
```

The scientific question guides our initial exploration: the columns in the data that we are interested in are `sqft` (house size, in livable square feet) and `price` (house sale price, in US dollars (USD)). The first step is to visualize the data as a scatter plot where we place the predictor variable (house size) on the x-axis, and we place the response variable that we want to predict (sale price) on the y-axis.

Note: Given that the y-axis unit is dollars in Fig. 7.1, we format the axis labels to put dollar signs in front of the house prices, as well as commas to increase the readability of the larger numbers. We can do this in `altair` by using `.axis(format="$,.0f")` on the `y` encoding channel.

```
scatter = alt.Chart(sacramento).mark_circle().encode(
    x=alt.X("sqft")
        .scale(zero=False)
        .title("House size (square feet)"),
    y=alt.Y("price")
        .axis(format="$,.0f")
```

(continues on next page)

(continued from previous page)

```
        .title("Price (USD)")
)

scatter
```

The plot is shown in Fig. 7.1. We can see that in Sacramento, CA, as the size of a house increases, so does its sale price. Thus, we can reason that we may be able to use the size of a not-yet-sold house (for which we don't know the sale price) to predict its final sale price. Note that we do not suggest here that a larger house size *causes* a higher sale price; just that house price tends to increase with house size, and that we may be able to use the latter to predict the former.

7.5 K-nearest neighbors regression

Much like in the case of classification, we can use a K-NN-based approach in regression to make predictions. Let's take a small sample of the data in Fig. 7.1 and walk through how K-NN works in a regression context before we dive in to creating our model and assessing how well it predicts house sale

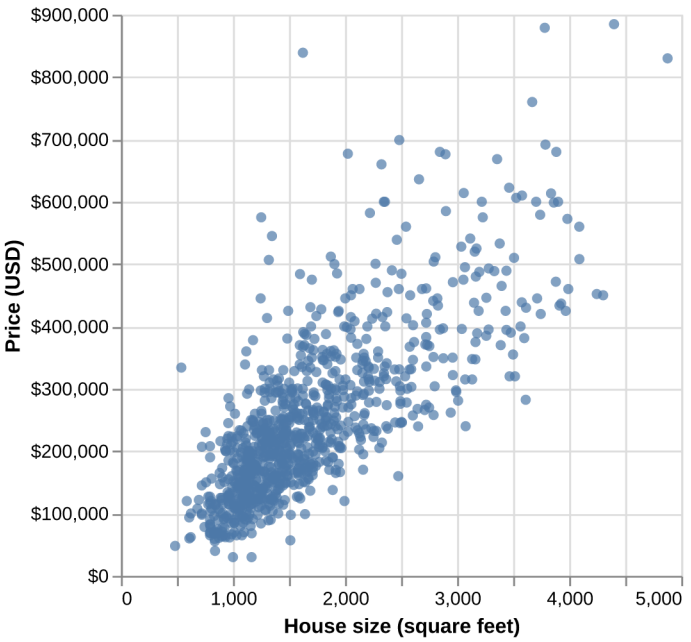


FIGURE 7.1 Scatter plot of price (USD) versus house size (square feet).

price. This subsample is taken to allow us to illustrate the mechanics of K-NN regression with a few data points; later in this chapter we will use all the data.

To take a small random sample of size 30, we'll use the `sample` method on the `sacramento` data frame, specifying that we want to select `n=30` rows.

```
small_sacramento = sacramento.sample(n=30)
```

Next, let's say we come across a 2,000 square-foot house in Sacramento we are interested in purchasing, with an advertised list price of $350,000. Should we offer to pay the asking price for this house, or is it overpriced and we should offer less? Absent any other information, we can get a sense for a good answer to this question by using the data we have to predict the sale price given the sale prices we have already observed. But in Fig. 7.2, you can see that we have no observations of a house of size *exactly* 2,000 square feet. How can we predict the sale price?

```
small_plot = alt.Chart(small_sacramento).mark_circle(opacity=1).encode(
    x=alt.X("sqft")
        .scale(zero=False)
        .title("House size (square feet)"),
    y=alt.Y("price")
        .axis(format="$,.0f")
        .title("Price (USD)")
)

# add an overlay to the base plot
line_df = pd.DataFrame({"x": [2000]})
rule = alt.Chart(line_df).mark_rule(strokeDash=[6], size=1.5, color="black").
↪encode(x="x")

small_plot + rule
```

We will employ the same intuition from Chapters 5 and 6, and use the neighboring points to the new point of interest to suggest/predict what its sale price might be. For the example shown in Fig. 7.2, we find and label the 5 nearest neighbors to our observation of a house that is 2,000 square feet.

```
small_sacramento["dist"] = (2000 - small_sacramento["sqft"]).abs()
nearest_neighbors = small_sacramento.nsmallest(5, "dist")
nearest_neighbors
```

```
              city     zip  beds  baths  sqft         type   price  \
298      SACRAMENTO  z95823     4    2.0  1900  Residential  361745
718        ANTELOPE  z95843     4    2.0  2160  Residential  290000
748       ROSEVILLE  z95678     3    2.0  1744  Residential  326951
252      SACRAMENTO  z95835     3    2.5  1718  Residential  250000
211  RANCHO_CORDOVA  z95670     3    2.0  1671  Residential  175000

      latitude   longitude  dist
298  38.487409 -121.461413   100
718  38.704554 -121.354753   160
```

(continues on next page)

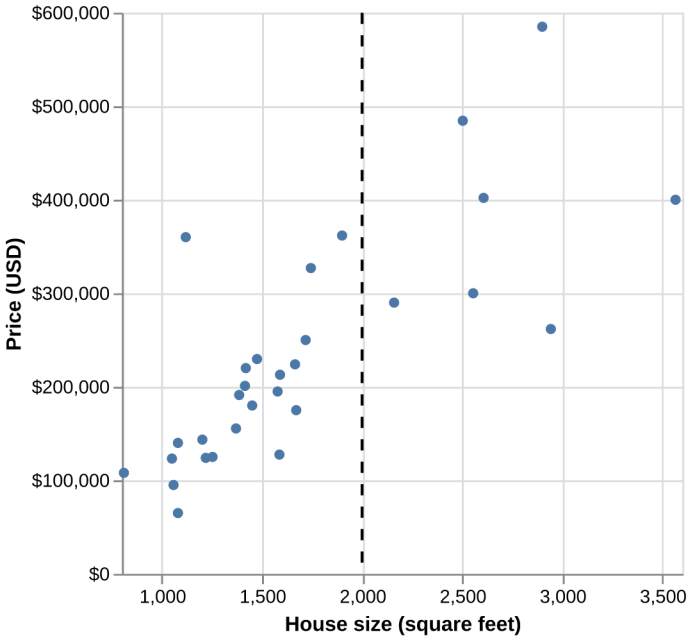


FIGURE 7.2 Scatter plot of price (USD) versus house size (square feet) with vertical line indicating 2,000 square feet on x-axis.

(continued from previous page)

```
748  38.771917 -121.304439   256
252  38.676658 -121.528128   282
211  38.591477 -121.315340   329
```

Fig. 7.3 illustrates the difference between the house sizes of the 5 nearest neighbors (in terms of house size) to our new 2,000 square-foot house of interest. Now that we have obtained these nearest neighbors, we can use their values to predict the sale price for the new home. Specifically, we can take the mean (or average) of these 5 values as our predicted value, as illustrated by the red point in Fig. 7.4.

```
prediction = nearest_neighbors["price"].mean()
prediction
```

```
280739.2
```

Our predicted price is $280,739 (shown as a red point in Fig. 7.4), which is much less than $350,000; perhaps we might want to offer less than the list price at which the house is advertised. But this is only the very beginning of the story. We still have all the same unanswered questions here with K-NN regression that we had with K-NN classification: which K do we choose, and

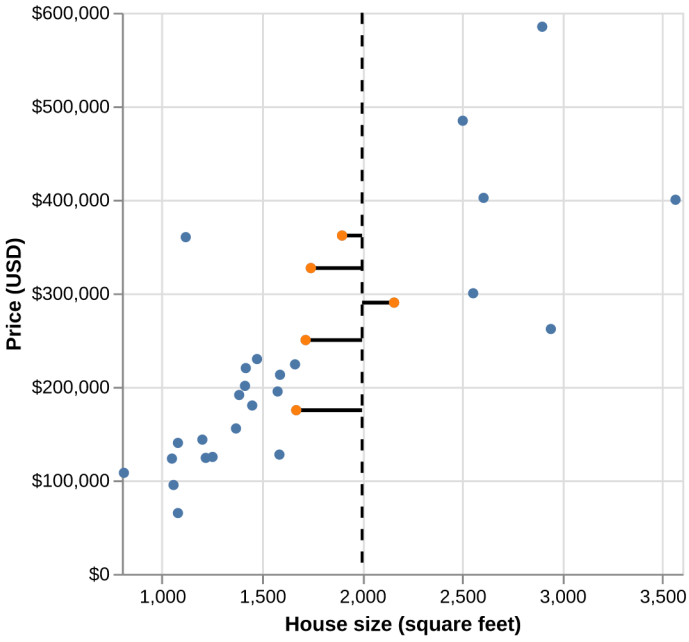


FIGURE 7.3 Scatter plot of price (USD) versus house size (square feet) with lines to 5 nearest neighbors (highlighted in orange).

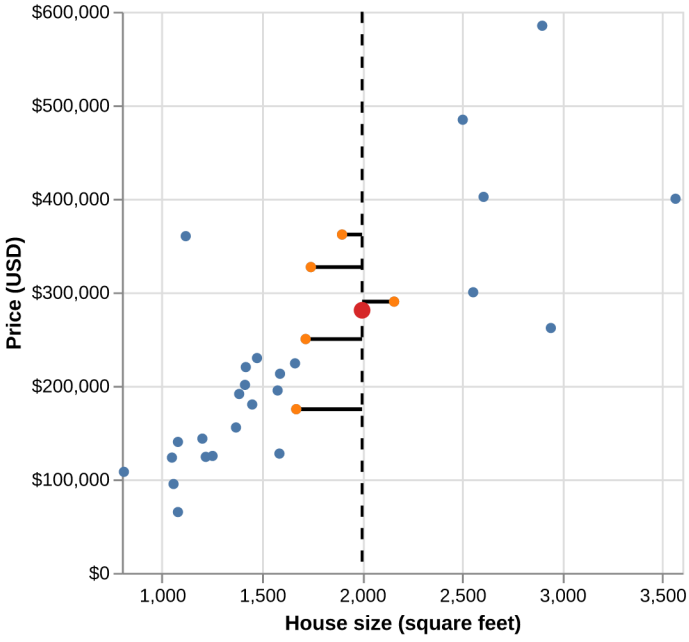


FIGURE 7.4 Scatter plot of price (USD) versus house size (square feet) with predicted price for a 2,000 square-foot house based on 5 nearest neighbors represented as a red dot.

is our model any good at making predictions? In the next few sections, we will address these questions in the context of K-NN regression.

One strength of the K-NN regression algorithm that we would like to draw attention to at this point is its ability to work well with non-linear relationships (i.e., if the relationship is not a straight line). This stems from the use of nearest neighbors to predict values. The algorithm really has very few assumptions about what the data must look like for it to work.

7.6 Training, evaluating, and tuning the model

As usual, we must start by putting some test data away in a lock box that we will come back to only after we choose our final model. Let's take care of that now. Note that for the remainder of the chapter we'll be working with the entire Sacramento data set, as opposed to the smaller sample of 30 points that we used earlier in the chapter (Fig. 7.2).

Note: We are not specifying the `stratify` argument here like we did in Chapter 6, since the `train_test_split` function cannot stratify based on a quantitative variable.

```
sacramento_train, sacramento_test = train_test_split(
    sacramento, train_size=0.75
)
```

Next, we'll use cross-validation to choose K. In K-NN classification, we used accuracy to see how well our predictions matched the true labels. We cannot use the same metric in the regression setting, since our predictions will almost never *exactly* match the true response variable values. Therefore in the context of K-NN regression we will use root mean square prediction error (RMSPE) instead. The mathematical formula for calculating RMSPE is:

$$\text{RMSPE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

where:

- n is the number of observations,
- y_i is the observed value for the i^{th} observation, and
- $\hat{y}_i$ is the forecasted/predicted value for the i^{th} observation.

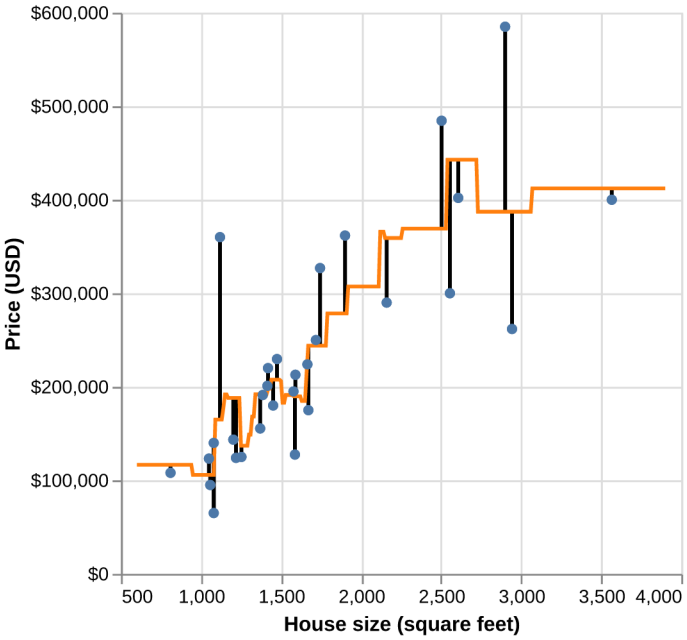


FIGURE 7.5 Scatter plot of price (USD) versus house size (square feet) with example predictions (orange line) and the error in those predictions compared with true response values (vertical lines).

In other words, we compute the *squared* difference between the predicted and true response value for each observation in our test (or validation) set, compute the average, and then finally take the square root. The reason we use the *squared* difference (and not just the difference) is that the differences can be positive or negative, i.e., we can overshoot or undershoot the true response value. Fig. 7.5 illustrates both positive and negative differences between predicted and true response values. So if we want to measure error—a notion of distance between our predicted and true response values—we want to make sure that we are only adding up positive values, with larger positive values representing larger mistakes. If the predictions are very close to the true values, then RMSPE will be small. If, on the other-hand, the predictions are very different from the true values, then RMSPE will be quite large. When we use cross-validation, we will choose the K that gives us the smallest RMSPE.

Note: When using many code packages, the evaluation output we will get to assess the prediction quality of our K-NN regression models is labeled "RMSE", or "root mean squared error". Why is this so, and why not RMSPE? In statistics, we try to be very precise with our language to indicate whether we are calculating the prediction error on the training data (*in-sample* prediction)

versus on the testing data (*out-of-sample* prediction). When predicting and evaluating prediction quality on the training data, we say RMSE. By contrast, when predicting and evaluating prediction quality on the testing or validation data, we say RMSPE. The equation for calculating RMSE and RMSPE is exactly the same; all that changes is whether the ys are training or testing data. But many people just use RMSE for both, and rely on context to denote which data the root mean squared error is being calculated on.

Now that we know how we can assess how well our model predicts a numerical value, let's use Python to perform cross-validation and to choose the optimal K. First, we will create a column transformer for preprocessing our data. Note that we include standardization in our preprocessing to build good habits, but since we only have one predictor, it is technically not necessary; there is no risk of comparing two predictors of different scales. Next, we create a model pipeline for K-NN regression. Note that we use the `KNeighborsRegressor` model object now to denote a regression problem, as opposed to the classification problems from the previous chapters. The use of `KNeighborsRegressor` essentially tells `scikit-learn` that we need to use different metrics (instead of accuracy) for tuning and evaluation. Next, we specify a parameter grid containing numbers of neighbors ranging from 1 to 200. Then we create a 5-fold `GridSearchCV` object, and pass in the pipeline and parameter grid. There is one additional slight complication: unlike classification models in `scikit-learn`—which by default use accuracy for tuning, as desired—regression models in `scikit-learn` do not use the RMSPE for tuning by default. So we need to specify that we want to use the RMSPE for tuning by setting the `scoring` argument to `"neg_root_mean_squared_error"`.

Note: We obtained the identifier of the parameter representing the number of neighbors, `"kneighborsregressor__n_neighbors"` by examining the output of `sacr_pipeline.get_params()`, as we did in Chapter 5.

```
# import the K-NN regression model
from sklearn.neighbors import KNeighborsRegressor

# preprocess the data, make the pipeline
sacr_preprocessor = make_column_transformer((StandardScaler(), ["sqft"]))
sacr_pipeline = make_pipeline(sacr_preprocessor, KNeighborsRegressor())

# create the 5-fold GridSearchCV object
param_grid = {
    "kneighborsregressor__n_neighbors": range(1, 201, 3),
}
sacr_gridsearch = GridSearchCV(
    estimator=sacr_pipeline,
```

(continues on next page)

(continued from previous page)

```
    param_grid=param_grid,
    cv=5,
    scoring="neg_root_mean_squared_error",
)
```

Next, we use the run cross validation by calling the `fit` method on `sacr_gridsearch`. Note the use of two brackets for the input features (`sacramento_train[["sqft"]]`), which creates a data frame with a single column. As we learned in Chapter 3, we can obtain a data frame with a subset of columns by passing a list of column names; `["sqft"]` is a list with one item, so we obtain a data frame with one column. If instead we used just one bracket (`sacramento_train["sqft"]`), we would obtain a series. In `scikit-learn`, it is easier to work with the input features as a data frame rather than a series, so we opt for two brackets here. On the other hand, the response variable can be a series, so we use just one bracket there (`sacramento_train["price"]`).

As in Chapter 6, once the model has been fit we will wrap the `cv_results_` output in a data frame, extract only the relevant columns, compute the standard error based on 5 folds, and rename the parameter column to be more readable.

```
# fit the GridSearchCV object
sacr_gridsearch.fit(
    sacramento_train[["sqft"]],  # A single-column data frame
    sacramento_train["price"]  # A series
)

# Retrieve the CV scores
sacr_results = pd.DataFrame(sacr_gridsearch.cv_results_)
sacr_results["sem_test_score"] = sacr_results["std_test_score"] / 5**(1/2)
sacr_results = (
    sacr_results[[
        "param_kneighborsregressor__n_neighbors",
        "mean_test_score",
        "sem_test_score"
    ]]
    .rename(columns={"param_kneighborsregressor__n_neighbors": "n_neighbors"})
)
sacr_results
```

	n_neighbors	mean_test_score	sem_test_score
0	1	-117365.988307	2715.383001
1	4	-93956.523683	2466.200227
2	7	-89859.401722	2739.713448
3	10	-87893.534919	2958.587153
4	13	-86444.413831	3383.712997
..	...	...	...
62	187	-92909.550051	2562.784826
63	190	-93137.289780	2511.564001
64	193	-93395.588763	2492.272799
65	196	-93671.588088	2473.312705

(continues on next page)

(continued from previous page)

```
66          199    -93986.752272     2473.048651

[67 rows x 3 columns]
```

In the `sacr_results` results data frame, we see that the `n_neighbors` variable contains the values of K, and `mean_test_score` variable contains the value of the RMSPE estimated via cross-validation …Wait a moment! Isn't the RMSPE supposed to be nonnegative? Recall that when we specified the `scoring` argument in the `GridSearchCV` object, we used the value `"neg_root_mean_squared_error"`. See the `neg_` at the start? That stands for *negative*. As it turns out, `scikit-learn` always tries to *maximize* a score when it tunes a model. But we want to *minimize* the RMSPE when we tune a regression model. So `scikit-learn` gets around this by working with the *negative* RMSPE instead. It is a little convoluted, but we need to add one more step to convert the negative RMSPE back to the regular RMSPE.

```
sacr_results["mean_test_score"] = -sacr_results["mean_test_score"]
sacr_results
```

```
   n_neighbors  mean_test_score  sem_test_score
0            1    117365.988307     2715.383001
1            4     93956.523683     2466.200227
2            7     89859.401722     2739.713448
3           10     87893.534919     2958.587153
4           13     86444.413831     3383.712997
..         ...              ...             ...
62         187     92909.550051     2562.784826
63         190     93137.289780     2511.564001
64         193     93395.588763     2492.272799
65         196     93671.588088     2473.312705
66         199     93986.752272     2473.048651

[67 rows x 3 columns]
```

Alright, now the `mean_test_score` variable actually has values of the RMSPE for different numbers of neighbors. Finally, the `sem_test_score` variable contains the standard error of our cross-validation RMSPE estimate, which is a measure of how uncertain we are in the mean value. Roughly, if your estimated mean RMSPE is \$100,000 and standard error is \$1,000, you can expect the *true* RMSPE to be somewhere roughly between \$99,000 and \$101,000 (although it may fall outside this range).

Fig. 7.6 visualizes how the RMSPE varies with the number of neighbors K. We take the *minimum* RMSPE to find the best setting for the number of neighbors. The smallest RMSPE occurs when K is 55.

To see which parameter value corresponds to the minimum RMSPE, we can also access the `best_params_` attribute of the original fit `GridSearchCV`

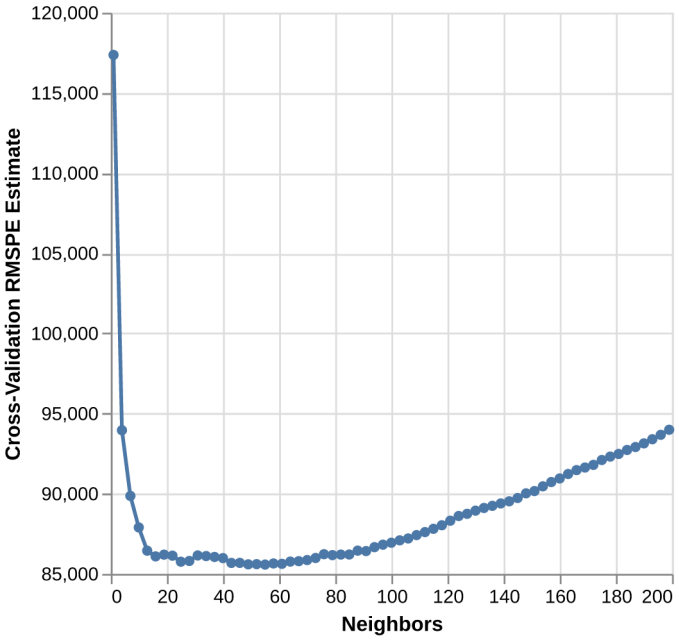


FIGURE 7.6 Effect of the number of neighbors on the RMSPE.

object. Note that it is still useful to visualize the results as we did above since this provides additional information on how the model performance varies.

```
sacr_gridsearch.best_params_
```

```
{'kneighborsregressor__n_neighbors': 55}
```

7.7 Underfitting and overfitting

Similar to the setting of classification, by setting the number of neighbors to be too small or too large, we cause the RMSPE to increase, as shown in Fig. 7.6. What is happening here?

Fig. 7.7 visualizes the effect of different settings of K on the regression model. Each plot shows the predicted values for house sale price from our K-NN regression model for 6 different values for K: 1, 3, 25, 55, 250, and 699 (i.e., all of the training data). For each model, we predict prices for the range of possible home sizes we observed in the data set (here 500 to 5,000 square feet) and we plot the predicted prices as an orange line.

Fig. 7.7 shows that when $K = 1$, the orange line runs perfectly through (almost) all of our training observations. This happens because our predicted

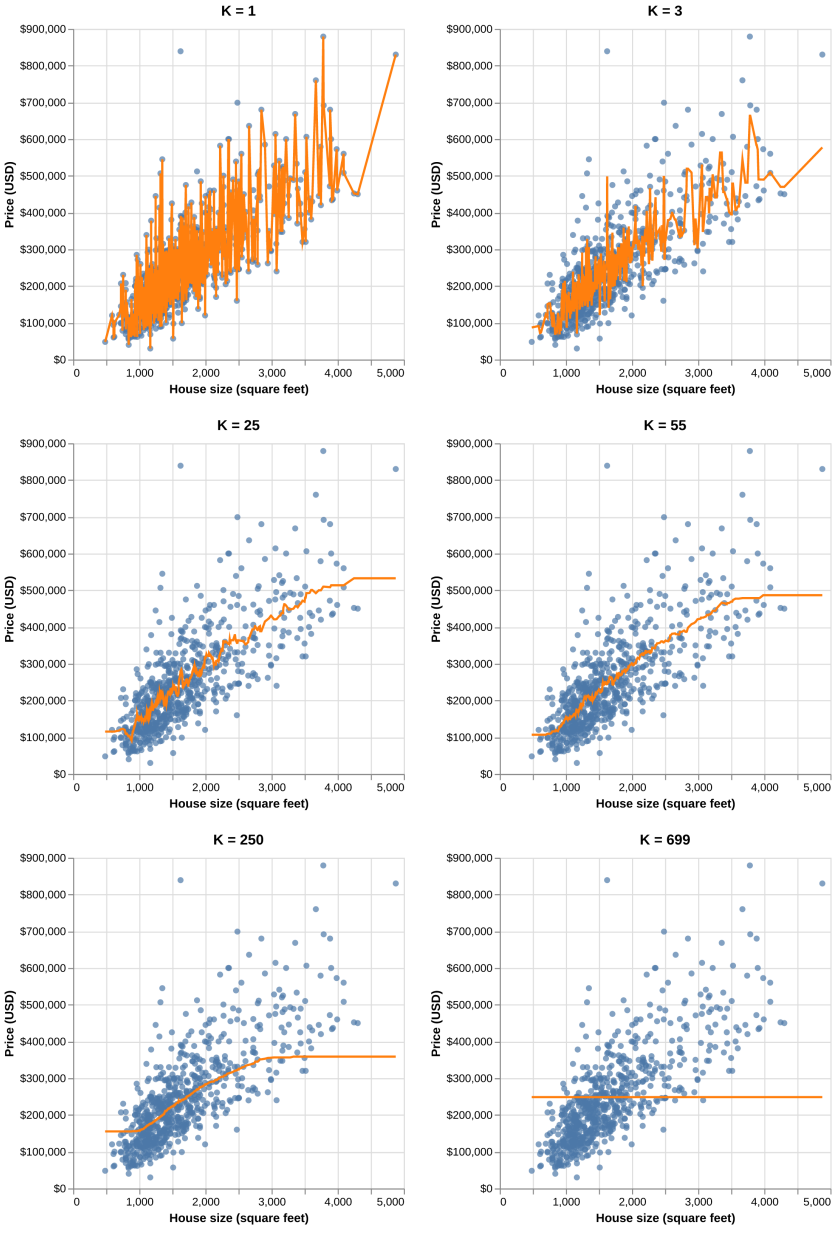


FIGURE 7.7 Predicted values for house price (represented as an orange line) from K-NN regression models for six different values for K.

values for a given region (typically) depend on just a single observation. In general, when K is too small, the line follows the training data quite closely, even if it does not match it perfectly. If we used a different training data set of house prices and sizes from the Sacramento real estate market, we would end up with completely different predictions. In other words, the model is *influenced too much* by the data. Because the model follows the training data so closely, it will not make accurate predictions on new observations which, generally, will not have the same fluctuations as the original training data. Recall from the classification chapters that this behavior—where the model is influenced too much by the noisy data—is called *overfitting*; we use this same term in the context of regression.

What about the plots in Fig. 7.7 where K is quite large, say, $K = 250$ or 699? In this case the orange line becomes extremely smooth, and actually becomes flat once K is equal to the number of datapoints in the entire data set. This happens because our predicted values for a given x value (here, home size), depend on many neighboring observations; in the case where K is equal to the size of the data set, the prediction is just the mean of the house prices in the data set (completely ignoring the house size). In contrast to the $K = 1$ example, the smooth, inflexible orange line does not follow the training observations very closely. In other words, the model is *not influenced enough* by the training data. Recall from the classification chapters that this behavior is called *underfitting*; we again use this same term in the context of regression.

Ideally, what we want is neither of the two situations discussed above. Instead, we would like a model that (1) follows the overall "trend" in the training data, so the model actually uses the training data to learn something useful, and (2) does not follow the noisy fluctuations, so that we can be confident that our model will transfer/generalize well to other new data. If we explore the other values for K, in particular $K = 55$ (as suggested by cross-validation), we can see it achieves this goal: it follows the increasing trend of house price versus house size, but is not influenced too much by the idiosyncratic variations in price. All of this is similar to how the choice of K affects K-NN classification, as discussed in the previous chapter.

7.8 Evaluating on the test set

To assess how well our model might do at predicting on unseen data, we will assess its RMSPE on the test data. To do this, we first need to retrain the K-NN regression model on the entire training data set using $K = 55$ neighbors. As we saw in Chapter 6 we do not have to do this ourselves manually;

`scikit-learn` does it for us automatically. To make predictions with the best model on the test data, we can use the `predict` method of the fit `GridSearchCV` object. We then use the `mean_squared_error` function (with the `y_true` and `y_pred` arguments) to compute the mean squared prediction error, and finally take the square root to get the RMSPE. The reason that we do not just use the `score` method—as in Chapter 6—is that the `KNeighborsRegressor` model uses a different default scoring metric than the RMSPE.

```
from sklearn.metrics import mean_squared_error

sacramento_test["predicted"] = sacr_gridsearch.predict(sacramento_test)
RMSPE = mean_squared_error(
    y_true=sacramento_test["price"],
    y_pred=sacramento_test["predicted"]
)**(1/2)
RMSPE
```

```
87498.86808211416
```

Our final model's test error as assessed by RMSPE is $87,499. Note that RMSPE is measured in the same units as the response variable. In other words, on new observations, we expect the error in our prediction to be *roughly* $87,499. From one perspective, this is good news: this is about the same as the cross-validation RMSPE estimate of our tuned model (which was $85,578, so we can say that the model appears to generalize well to new data that it has never seen before. However, much like in the case of K-NN classification, whether this value for RMSPE is *good*—i.e., whether an error of around $87,499 is acceptable—depends entirely on the application. In this application, this error is not prohibitively large, but it is not negligible either; $87,499 might represent a substantial fraction of a home buyer's budget, and could make or break whether or not they could afford put an offer on a house.

Finally, Fig. 7.8 shows the predictions that our final model makes across the range of house sizes we might encounter in the Sacramento area. Note that instead of predicting the house price only for those house sizes that happen to appear in our data, we predict it for evenly spaced values between the minimum and maximum in the data set (roughly 500 to 5000 square feet). We superimpose this prediction line on a scatter plot of the original housing price data, so that we can qualitatively assess if the model seems to fit the data well. You have already seen a few plots like this in this chapter, but here we also provide the code that generated it as a learning opportunity.

```
# Create a grid of evenly spaced values along the range of the sqft data
sqft_prediction_grid = pd.DataFrame({
    "sqft": np.arange(sacramento["sqft"].min(), sacramento["sqft"].max(), 10)
})
# Predict the price for each of the sqft values in the grid
```

(continues on next page)

(continued from previous page)

```
sqft_prediction_grid["predicted"] = sacr_gridsearch.predict(sqft_prediction_grid)

# Plot all the houses
base_plot = alt.Chart(sacramento).mark_circle(opacity=0.4).encode(
    x=alt.X("sqft")
        .scale(zero=False)
        .title("House size (square feet)"),
    y=alt.Y("price")
        .axis(format="$,.0f")
        .title("Price (USD)")
)

# Add the predictions as a line
sacr_preds_plot = base_plot + alt.Chart(
    sqft_prediction_grid,
    title=f"K = {best_k_sacr}"
).mark_line(
    color="#ff7f0e"
).encode(
    x="sqft",
    y="predicted"
)

sacr_preds_plot
```

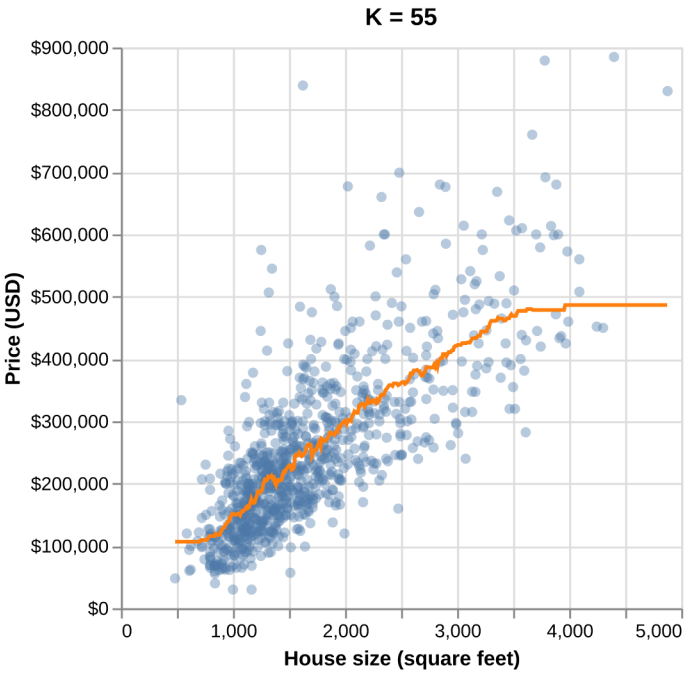


FIGURE 7.8 Predicted values of house price (orange line) for the final K-NN regression model.

7.9 Multivariable K-NN regression

As in K-NN classification, we can use multiple predictors in K-NN regression. In this setting, we have the same concerns regarding the scale of the predictors. Once again, predictions are made by identifying the K observations that are nearest to the new point we want to predict; any variables that are on a large scale will have a much larger effect than variables on a small scale. Hence, we should re-define the preprocessor in the pipeline to incorporate all predictor variables.

Note that we also have the same concern regarding the selection of predictors in K-NN regression as in K-NN classification: having more predictors is **not** always better, and the choice of which predictors to use has a potentially large influence on the quality of predictions. Fortunately, we can use the predictor selection algorithm from Chapter 6 in K-NN regression as well. As the algorithm is the same, we will not cover it again in this chapter.

We will now demonstrate a multivariable K-NN regression analysis of the Sacramento real estate data using `scikit-learn`. This time we will use house size (measured in square feet) as well as number of bedrooms as our predictors, and continue to use house sale price as our response variable that we are trying to predict. It is always a good practice to do exploratory data analysis, such as visualizing the data, before we start modeling the data. Fig. 7.9 shows that the number of bedrooms might provide useful information to help predict the sale price of a house.

```
plot_beds = alt.Chart(sacramento).mark_circle().encode(
    x=alt.X("beds").title("Number of Bedrooms"),
    y=alt.Y("price").title("Price (USD)").axis(format="$,.0f"),
)

plot_beds
```

Fig. 7.9 shows that as the number of bedrooms increases, the house sale price tends to increase as well, but that the relationship is quite weak. Does adding the number of bedrooms to our model improve our ability to predict price? To answer that question, we will have to create a new K-NN regression model using house size and number of bedrooms, and then we can compare it to the model we previously came up with that only used house size. Let's do that now.

First, we'll build a new model object and preprocessor for the analysis. Note that we pass the list `["sqft", "beds"]` into the `make_column_transformer` function to denote that we have two predictors.

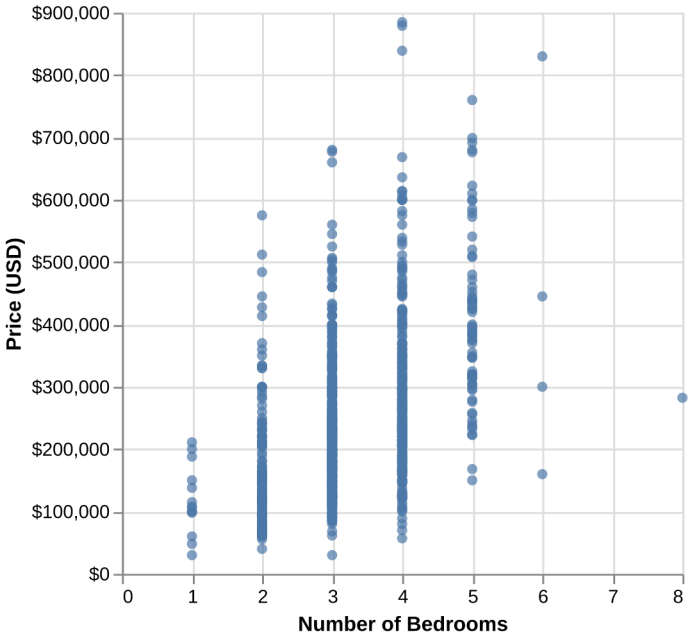


FIGURE 7.9 Scatter plot of the sale price of houses versus the number of bedrooms.

Moreover, we do not specify `n_neighbors` in `KNeighborsRegressor`, indicating that we want this parameter to be tuned by `GridSearchCV`.

```
sacr_preprocessor = make_column_transformer((StandardScaler(), ["sqft", "beds"]))
sacr_pipeline = make_pipeline(sacr_preprocessor, KNeighborsRegressor())
```

Next, we'll use 5-fold cross-validation with a `GridSearchCV` object to choose the number of neighbors via the minimum RMSPE:

```
# create the 5-fold GridSearchCV object
param_grid = {
    "kneighborsregressor__n_neighbors": range(1, 50),
}

sacr_gridsearch = GridSearchCV(
    estimator=sacr_pipeline,
    param_grid=param_grid,
    cv=5,
    scoring="neg_root_mean_squared_error"
)

sacr_gridsearch.fit(
  sacramento_train[["sqft", "beds"]],
  sacramento_train["price"]
)

# retrieve the CV scores
sacr_results = pd.DataFrame(sacr_gridsearch.cv_results_)
```

(continues on next page)

(continued from previous page)

```
sacr_results["sem_test_score"] = sacr_results["std_test_score"] / 5**(1/2)
sacr_results["mean_test_score"] = -sacr_results["mean_test_score"]
sacr_results = (
    sacr_results[[
        "param_kneighborsregressor__n_neighbors",
        "mean_test_score",
        "sem_test_score"
    ]]
    .rename(columns={"param_kneighborsregressor__n_neighbors" : "n_neighbors"})
)

# show only the row of minimum RMSPE
sacr_results.nsmallest(1, "mean_test_score")
```

```
    n_neighbors  mean_test_score  sem_test_score
28           29     85156.027067     3376.143313
```

Here we see that the smallest estimated RMSPE from cross-validation occurs when $K = 29$. If we want to compare this multivariable K-NN regression model to the model with only a single predictor *as part of the model tuning process* (e.g., if we are running forward selection as described in the chapter on evaluating and tuning classification models), then we must compare the RMSPE estimated using only the training data via cross-validation. Looking back, the estimated cross-validation RMSPE for the single-predictor model was \$85,578. The estimated cross-validation RMSPE for the multivariable model is \$85,156. Thus in this case, we did not improve the model by a large amount by adding this additional predictor.

Regardless, let's continue the analysis to see how we can make predictions with a multivariable K-NN regression model and evaluate its performance on test data. As previously, we will use the best model to make predictions on the test data via the `predict` method of the fit `GridSearchCV` object. Finally, we will use the `mean_squared_error` function to compute the RMSPE.

```
sacramento_test["predicted"] = sacr_gridsearch.predict(sacramento_test)
RMSPE_mult = mean_squared_error(
    y_true=sacramento_test["price"],
    y_pred=sacramento_test["predicted"]
)**(1/2)
RMSPE_mult
```

```
85083.2902421959
```

This time, when we performed K-NN regression on the same data set, but also included number of bedrooms as a predictor, we obtained a RMSPE test error of \$85,083. Fig. 7.10 visualizes the model's predictions overlaid on top of the data. This time the predictions are a surface in 3D space, instead of a line in 2D space, as we have 2 predictors instead of 1.

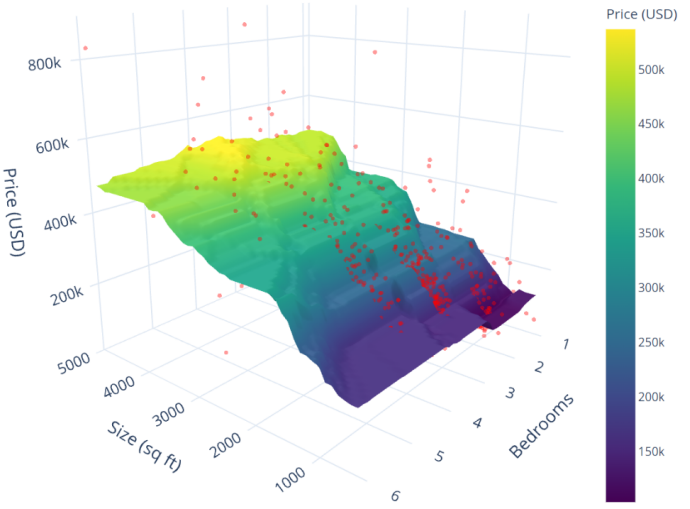


FIGURE 7.10 K-NN regression model's predictions represented as a surface in 3D space overlaid on top of the data using three predictors (price, house size, and the number of bedrooms). Note that in general we recommend against using 3D visualizations; here we use a 3D visualization only to illustrate what the surface of predictions looks like for learning purposes.

We can see that the predictions in this case, where we have 2 predictors, form a surface instead of a line. Because the newly added predictor (number of bedrooms) is related to price (as price changes, so does number of bedrooms) and is not totally determined by house size (our other predictor), we get additional and useful information for making our predictions. For example, in this model we would predict that the cost of a house with a size of 2,500 square feet generally increases slightly as the number of bedrooms increases. Without having the additional predictor of number of bedrooms, we would predict the same price for these two houses.

7.10 Strengths and limitations of K-NN regression

As with K-NN classification (or any prediction algorithm for that matter), K-NN regression has both strengths and weaknesses. Some are listed here:

Strengths: K-NN regression

1. is a simple, intuitive algorithm,
2. requires few assumptions about what the data must look like, and

3. works well with non-linear relationships (i.e., if the relationship is not a straight line).

Weaknesses: K-NN regression

1. becomes very slow as the training data gets larger,
2. may not perform well with a large number of predictors, and
3. may not predict well beyond the range of values input in your training data.

7.11 Exercises

Practice exercises for the material covered in this chapter can be found in the accompanying worksheets repository[2] in the "Regression I: K-nearest neighbors" row. You can launch an interactive version of the worksheet in your browser by clicking the "launch binder" button. You can also preview a non-interactive version of the worksheet by clicking "view worksheet". If you instead decide to download the worksheet and run it on your own machine, make sure to follow the instructions for computer setup found in Chapter 13. This will ensure that the automated feedback and guidance that the worksheets provide will function as intended.

[2] https://worksheets.python.datasciencebook.ca

8

Regression II: linear regression

8.1 Overview

Up to this point, we have solved all of our predictive problems—both classification and regression—using K-nearest neighbors (K-NN)-based approaches. In the context of regression, there is another commonly used method known as *linear regression*. This chapter provides an introduction to the basic concept of linear regression, shows how to use `scikit-learn` to perform linear regression in Python, and characterizes its strengths and weaknesses compared to K-NN regression. The focus is, as usual, on the case where there is a single predictor and single response variable of interest; but the chapter concludes with an example using *multivariable linear regression* when there is more than one predictor.

8.2 Chapter learning objectives

By the end of the chapter, readers will be able to do the following:

- Use Python to fit simple and multivariable linear regression models on training data.
- Evaluate the linear regression model on test data.
- Compare and contrast predictions obtained from K-NN regression to those obtained using linear regression from the same data set.
- Describe how linear regression is affected by outliers and multicollinearity.

8.3 Simple linear regression

At the end of the previous chapter, we noted some limitations of K-NN regression. While the method is simple and easy to understand, K-NN regression

DOI: 10.1201/9781003438397-8

does not predict well beyond the range of the predictors in the training data, and the method gets significantly slower as the training data set grows. Fortunately, there is an alternative to K-NN regression—*linear regression*—that addresses both of these limitations. Linear regression is also very commonly used in practice because it provides an interpretable mathematical equation that describes the relationship between the predictor and response variables. In this first part of the chapter, we will focus on *simple* linear regression, which involves only one predictor variable and one response variable; later on, we will consider *multivariable* linear regression, which involves multiple predictor variables. Like K-NN regression, simple linear regression involves predicting a numerical response variable (like race time, house price, or height); but *how* it makes those predictions for a new observation is quite different from K-NN regression. Instead of looking at the K-NN and averaging over their values for a prediction, in simple linear regression, we create a straight line of best fit through the training data and then "look up" the prediction using the line.

Note: Although we did not cover it in earlier chapters, there is another popular method for classification called *logistic regression* (it is used for classification even though the name, somewhat confusingly, has the word "regression" in it). In logistic regression—similar to linear regression—you "fit" the model to the training data and then "look up" the prediction for each new observation. Logistic regression and K-NN classification have an advantage/disadvantage comparison similar to that of linear regression and K-NN regression. It is useful to have a good understanding of linear regression before learning about logistic regression. After reading this chapter, see the "Additional Resources" section at the end of the classification chapters to learn more about logistic regression.

Let's return to the Sacramento housing data from Chapter 7 to learn how to apply linear regression and compare it to K-NN regression. For now, we will consider a smaller version of the housing data to help make our visualizations clear. Recall our predictive question: can we use the size of a house in the Sacramento, CA area to predict its sale price? In particular, recall that we have come across a new 2,000 square-foot house we are interested in purchasing with an advertised list price of $350,000. Should we offer the list price, or is that over/undervalued? To answer this question using simple linear regression, we use the data we have to draw the straight line of best fit through our existing data points. The small subset of data as well as the line of best fit are shown in Fig. 8.1.

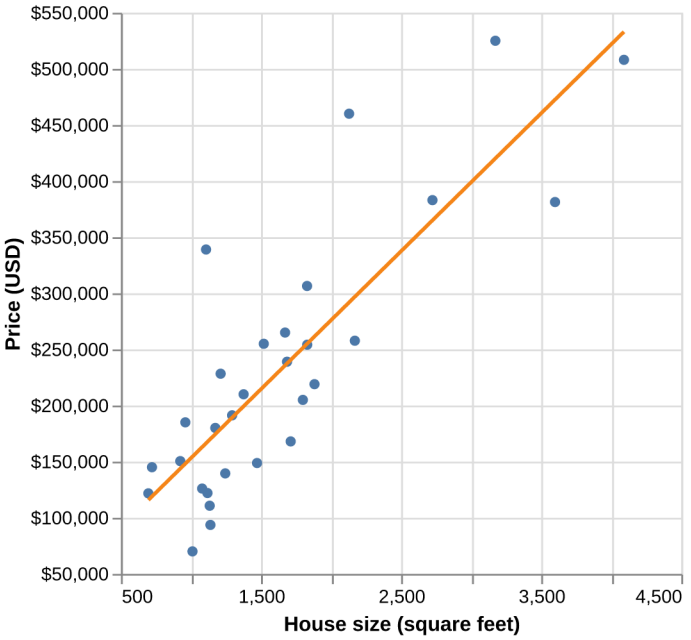


FIGURE 8.1 Scatter plot of sale price versus size with line of best fit for subset of the Sacramento housing data.

The equation for the straight line is:

$$\text{house sale price} = \beta_0 + \beta_1 \cdot (\text{house size}),$$

where

- β_0 is the *vertical intercept* of the line (the price when house size is 0)
- β_1 is the *slope* of the line (how quickly the price increases as you increase house size)

Therefore using the data to find the line of best fit is equivalent to finding coefficients β_0 and β_1 that *parametrize* (correspond to) the line of best fit. Now of course, in this particular problem, the idea of a 0 square-foot house is a bit silly; but you can think of β_0 here as the "base price", and β_1 as the increase in price for each square foot of space. Let's push this thought even further: what would happen in the equation for the line if you tried to evaluate the price of a house with size 6 *million* square feet? Or what about *negative* 2,000 square feet? As it turns out, nothing in the formula breaks; linear regression will happily make predictions for crazy predictor values if you ask it to. But even though you *can* make these wild predictions, you shouldn't. You should only make predictions roughly within the range of your original data, and perhaps a bit beyond it only if it makes sense. For example,

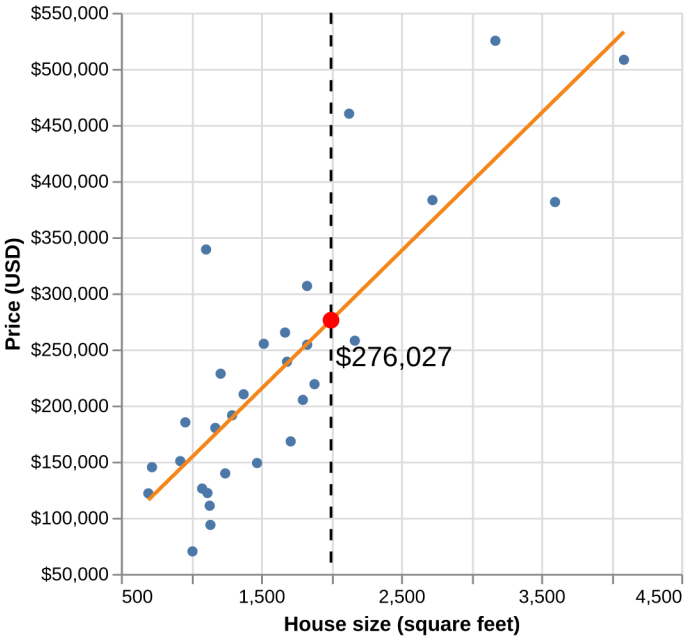


FIGURE 8.2 Scatter plot of sale price versus size with line of best fit and a red dot at the predicted sale price for a 2,000 square-foot home.

the data in Fig. 8.1 only reaches around 600 square feet on the low end, but it would probably be reasonable to use the linear regression model to make a prediction at 500 square feet, say.

Back to the example. Once we have the coefficients β_0 and β_1, we can use the equation above to evaluate the predicted sale price given the value we have for the predictor variable—here 2,000 square feet. Fig. 8.2 demonstrates this process.

By using simple linear regression on this small data set to predict the sale price for a 2,000 square-foot house, we get a predicted value of $276,027. But wait a minute … how exactly does simple linear regression choose the line of best fit? Many different lines could be drawn through the data points. Some plausible examples are shown in Fig. 8.3.

Simple linear regression chooses the straight line of best fit by choosing the line that minimizes the **average squared vertical distance** between itself and each of the observed data points in the training data (equivalent to minimizing the RMSE). Fig. 8.4 illustrates these vertical distances as lines. Finally, to assess the predictive accuracy of a simple linear regression model, we use RMSPE—the same measure of predictive performance we used with K-NN regression.

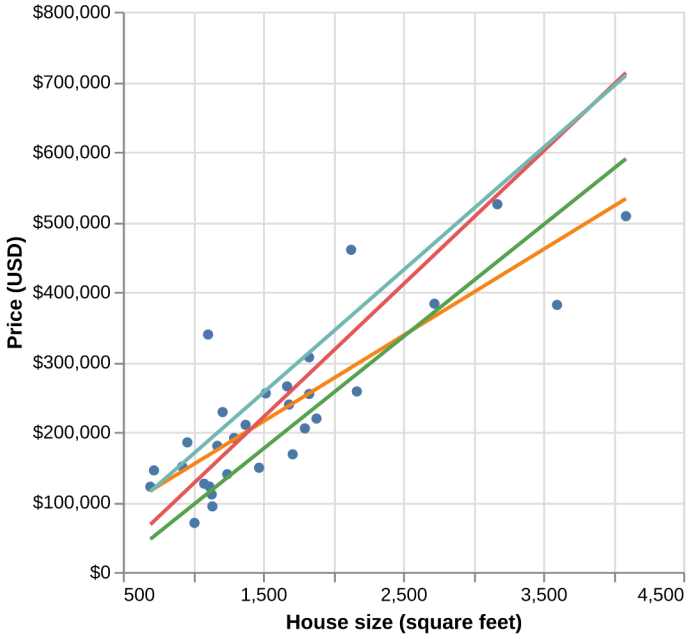


FIGURE 8.3 Scatter plot of sale price versus size with many possible lines that could be drawn through the data points.

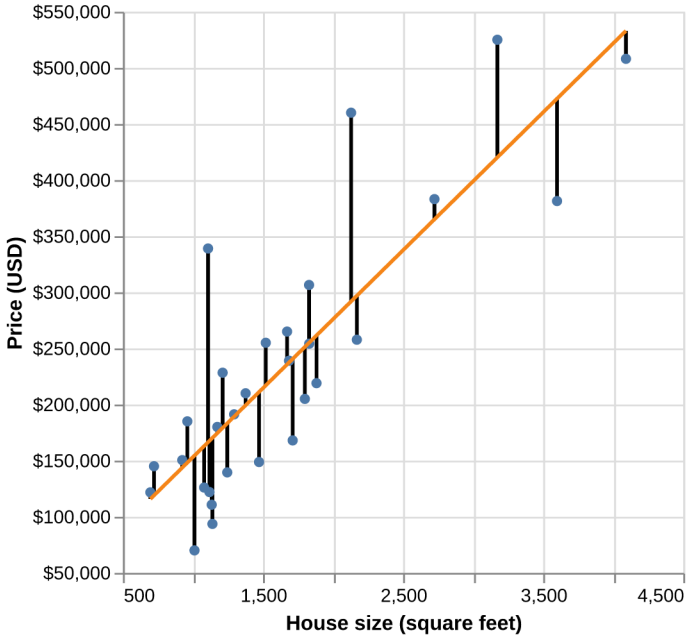


FIGURE 8.4 Scatter plot of sale price versus size with lines denoting the vertical distances between the predicted values and the observed data points.

8.4 Linear regression in Python

We can perform simple linear regression in Python using `scikit-learn` in a very similar manner to how we performed K-NN regression. To do this, instead of creating a `KNeighborsRegressor` model object, we use a `LinearRegression` model object; and as usual, we first have to import it from `sklearn`. Another difference is that we do not need to choose K in the context of linear regression, and so we do not need to perform cross-validation. Below we illustrate how we can use the usual `scikit-learn` workflow to predict house sale price given house size. We use a simple linear regression approach on the full Sacramento real estate data set.

As usual, we start by loading packages, setting the seed, loading data, and putting some test data away in a lock box that we can come back to after we choose our final model. Let's take care of that now.

```
import numpy as np
import altair as alt
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn import set_config

# Output dataframes instead of arrays
set_config(transform_output="pandas")

np.random.seed(1)

sacramento = pd.read_csv("data/sacramento.csv")

sacramento_train, sacramento_test = train_test_split(
    sacramento, train_size=0.75
)
```

Now that we have our training data, we will create and fit the linear regression model object. We will also extract the slope of the line via the `coef_[0]` property, as well as the intercept of the line via the `intercept_` property.

```
# fit the linear regression model
lm = LinearRegression()
lm.fit(
   sacramento_train[["sqft"]],  # A single-column data frame
   sacramento_train["price"]  # A series
)

# make a dataframe containing slope and intercept coefficients
pd.DataFrame({"slope": [lm.coef_[0]], "intercept": [lm.intercept_]})
```

```
        slope     intercept
0  137.285652  15642.309105
```

Note: An additional difference that you will notice here is that we do not standardize (i.e., scale and center) our predictors. In K-NN models, recall that the model fit changes depending on whether we standardize first or not. In linear regression, standardization does not affect the fit (it *does* affect the coefficients in the equation, though!). So you can standardize if you want—it won't hurt anything—but if you leave the predictors in their original form, the best fit coefficients are usually easier to interpret afterward.

Our coefficients are (intercept) $\beta_0 = 15642$ and (slope) $\beta_1 = 137$. This means that the equation of the line of best fit is

$$\text{house sale price} = 15642 + 137 \cdot (\text{house size}).$$

In other words, the model predicts that houses start at $15,642 for 0 square feet, and that every extra square foot increases the cost of the house by $137. Finally, we predict on the test data set to assess how well our model does.

```
# make predictions
sacramento_test["predicted"] = lm.predict(sacramento_test[["sqft"]])

# calculate RMSPE
RMSPE = mean_squared_error(
    y_true=sacramento_test["price"],
    y_pred=sacramento_test["predicted"]
)**(1/2)

RMSPE
```

```
85376.59691629931
```

Our final model's test error as assessed by RMSPE is $85,377. Remember that this is in units of the response variable, and here that is US Dollars (USD). Does this mean our model is "good" at predicting house sale price based off of the predictor of home size? Again, answering this is tricky and requires knowledge of how you intend to use the prediction.

To visualize the simple linear regression model, we can plot the predicted house sale price across all possible house sizes we might encounter. Since our model is linear, we only need to compute the predicted price of the minimum and maximum house size, and then connect them with a straight line. We superimpose this prediction line on a scatter plot of the original housing price data, so that we can qualitatively assess if the model seems to fit the data well. Fig. 8.5 displays the result.

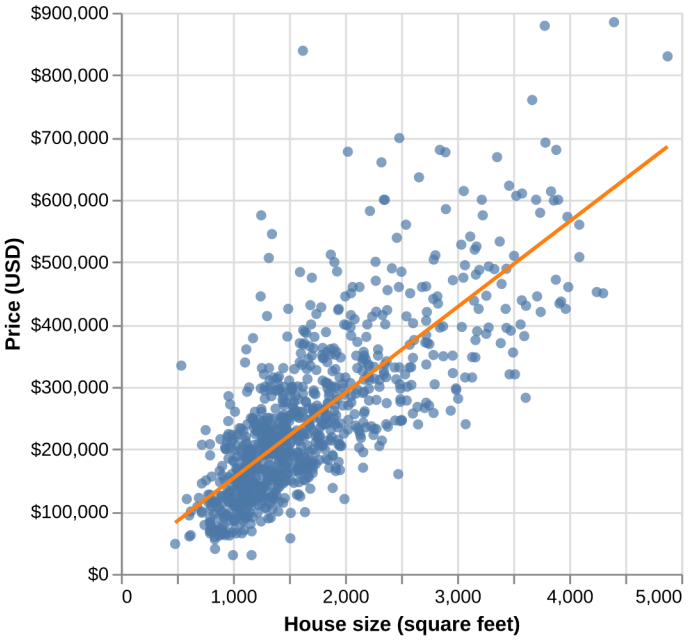


FIGURE 8.5 Scatter plot of sale price versus size with line of best fit for the full Sacramento housing data.

```
sqft_prediction_grid = sacramento[["sqft"]].agg(["min", "max"])
sqft_prediction_grid["predicted"] = lm.predict(sqft_prediction_grid)

all_points = alt.Chart(sacramento).mark_circle().encode(
    x=alt.X("sqft")
        .scale(zero=False)
        .title("House size (square feet)"),
    y=alt.Y("price")
        .axis(format="$,.0f")
        .scale(zero=False)
        .title("Price (USD)")
)

sacr_preds_plot = all_points + alt.Chart(sqft_prediction_grid).mark_line(
    color="#ff7f0e"
).encode(
    x="sqft",
    y="predicted"
)

sacr_preds_plot
```

8.5 Comparing simple linear and K-NN regression

Now that we have a general understanding of both simple linear and K-NN regression, we can start to compare and contrast these methods as well as

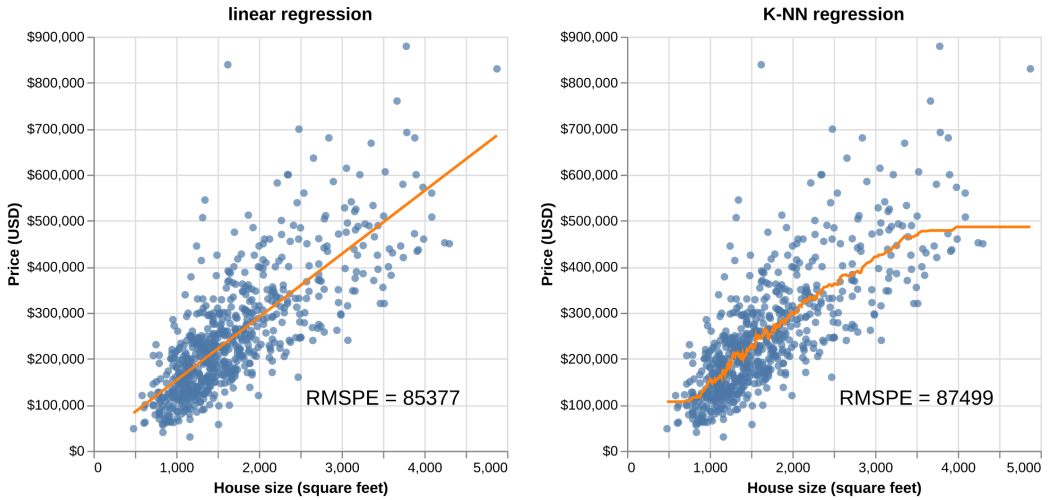


FIGURE 8.6 Comparison of simple linear regression and K-NN regression.

the predictions made by them. To start, let's look at the visualization of the simple linear regression model predictions for the Sacramento real estate data (predicting price from house size) and the "best" K-NN regression model obtained from the same problem, shown in Fig. 8.6.

What differences do we observe in Fig. 8.6? One obvious difference is the shape of the orange lines. In simple linear regression we are restricted to a straight line, whereas in K-NN regression our line is much more flexible and can be quite wiggly. But there is a major interpretability advantage in limiting the model to a straight line. A straight line can be defined by two numbers, the vertical intercept and the slope. The intercept tells us what the prediction is when all of the predictors are equal to 0; and the slope tells us what unit increase in the response variable we predict given a unit increase in the predictor variable. K-NN regression, as simple as it is to implement and understand, has no such interpretability from its wiggly line.

There can, however, also be a disadvantage to using a simple linear regression model in some cases, particularly when the relationship between the response variable and the predictor is not linear, but instead some other shape (e.g., curved or oscillating). In these cases the prediction model from a simple linear regression will underfit, meaning that model/predicted values do not match the actual observed values very well. Such a model would probably have a quite high RMSE when assessing model goodness of fit on the training data and a quite high RMSPE when assessing model prediction quality on a test data set. On such a data set, K-NN regression may fare better. Additionally, there are other types of regression you can learn about in future books that may do even better at predicting with such data.

How do these two models compare on the Sacramento house prices data set? In Fig. 8.6, we also printed the RMSPE as calculated from predicting on the test data set that was not used to train/fit the models. The RMSPE for the simple linear regression model is slightly lower than the RMSPE for the K-NN regression model. Considering that the simple linear regression model is also more interpretable, if we were comparing these in practice we would likely choose to use the simple linear regression model.

Finally, note that the K-NN regression model becomes "flat" at the left and right boundaries of the data, while the linear model predicts a constant slope. Predicting outside the range of the observed data is known as *extrapolation*; K-NN and linear models behave quite differently when extrapolating. Depending on the application, the flat or constant slope trend may make more sense. For example, if our housing data were slightly different, the linear model may have actually predicted a *negative* price for a small house (if the intercept β_0 was negative), which obviously does not match reality. On the other hand, the trend of increasing house size corresponding to increasing house price probably continues for large houses, so the "flat" extrapolation of K-NN likely does not match reality.

8.6 Multivariable linear regression

As in K-NN classification and K-NN regression, we can move beyond the simple case of only one predictor to the case with multiple predictors, known as *multivariable linear regression*. To do this, we follow a very similar approach to what we did for K-NN regression: we just specify the training data by adding more predictors. But recall that we do not need to use cross-validation to choose any parameters, nor do we need to standardize (i.e., center and scale) the data for linear regression. Note once again that we have the same concerns regarding multiple predictors as in the settings of multivariable K-NN regression and classification: having more predictors is **not** always better. But because the same predictor selection algorithm from Chapter 6 extends to the setting of linear regression, it will not be covered again in this chapter.

We will demonstrate multivariable linear regression using the Sacramento real estate data with both house size (measured in square feet) as well as number of bedrooms as our predictors, and continue to use house sale price as our response variable. The `scikit-learn` framework makes this easy to do: we just need to set both the `sqft` and `beds` variables as predictors, and then use the `fit` method as usual.

```
mlm = LinearRegression()
mlm.fit(
    sacramento_train[["sqft", "beds"]],
    sacramento_train["price"]
)
```

```
LinearRegression()
```

Finally, we make predictions on the test data set to assess the quality of our model.

```
sacramento_test["predicted"] = mlm.predict(sacramento_test[["sqft","beds"]])

lm_mult_test_RMSPE = mean_squared_error(
    y_true=sacramento_test["price"],
    y_pred=sacramento_test["predicted"]
)**(1/2)
lm_mult_test_RMSPE
```

```
82331.04630202598
```

Our model's test error as assessed by RMSPE is $82,331. In the case of two predictors, we can plot the predictions made by our linear regression creates a *plane* of best fit, as shown in Fig. 8.7.

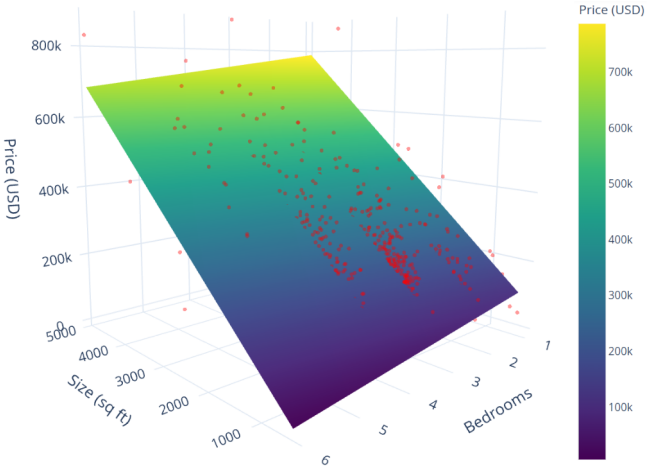


FIGURE 8.7 Linear regression plane of best fit overlaid on top of the data (using price, house size, and number of bedrooms as predictors). Note that in general we recommend against using 3D visualizations; here we use a 3D visualization only to illustrate what the regression plane looks like for learning purposes.

We see that the predictions from linear regression with two predictors form a flat plane. This is the hallmark of linear regression, and differs from the wiggly, flexible surface we get from other methods such as K-NN regression. As discussed, this can be advantageous in one aspect, which is that for each predictor, we can get slopes/intercept from linear regression, and thus describe the plane mathematically. We can extract those slope values from the `coef_` property of our model object, and the intercept from the `intercept_` property, as shown below.

```
mlm.coef_
```

```
array([   154.59235377, -20333.43213798])
```

```
mlm.intercept_
```

```
53180.26906624224
```

When we have multiple predictor variables, it is not easy to know which variable goes with which coefficient in `mlm.coef_`. In particular, you will see that `mlm.coef_` above is just an array of values without any variable names. Unfortunately you have to do this mapping yourself: the coefficients in `mlm.coef_` appear in the *same order* as the columns of the predictor data frame you used when training. So since we used `sacramento_train[["sqft", "beds"]]` when training, we have that `mlm.coef_[0]` corresponds to `sqft`, and `mlm.coef_[1]` corresponds to `beds`. Once you sort out the correspondence, you can then use those slopes to write a mathematical equation to describe the prediction plane:

$$\text{house sale price} = \beta_0 + \beta_1 \cdot (\text{house size}) + \beta_2 \cdot (\text{number of bedrooms}),$$

where:

- β_0 is the *vertical intercept* of the hyperplane (the price when both house size and number of bedrooms are 0)
- β_1 is the *slope* for the first predictor (how quickly the price increases as you increase house size)
- β_2 is the *slope* for the second predictor (how quickly the price increases as you increase the number of bedrooms)

Finally, we can fill in the values for β_0, β_1, and β_2 from the model output above to create the equation of the plane of best fit to the data:

$$\text{house sale price} = 53{,}180 + 155 \cdot (\text{house size}) - 20{,}333 \cdot (\text{number of bedrooms})$$

This model is more interpretable than the multivariable K-NN regression model; we can write a mathematical equation that explains how each predictor is affecting the predictions. But as always, we should question how well multivariable linear regression is doing compared to the other tools we have, such as simple linear regression and multivariable K-NN regression. If this comparison is part of the model tuning process—for example, if we are trying out many different sets of predictors for multivariable linear and K-NN regression—we must perform this comparison using cross-validation on only our training data. But if we have already decided on a small number (e.g., 2 or 3) of tuned candidate models and we want to make a final comparison, we can do so by comparing the prediction error of the methods on the test data.

```
lm_mult_test_RMSPE
```

```
82331.04630202598
```

We obtain an RMSPE for the multivariable linear regression model of $82,331. This prediction error is less than the prediction error for the multivariable K-NN regression model, indicating that we should likely choose linear regression for predictions of house sale price on this data set. Revisiting the simple linear regression model with only a single predictor from earlier in this chapter, we see that the RMSPE for that model was $85,377, which is slightly higher than that of our more complex model. Our model with two predictors provided a slightly better fit on test data than our model with just one. As mentioned earlier, this is not always the case: sometimes including more predictors can negatively impact the prediction performance on unseen test data.

8.7 Multicollinearity and outliers

What can go wrong when performing (possibly multivariable) linear regression? This section will introduce two common issues—*outliers* and *collinear predictors*—and illustrate their impact on predictions.

8.7.1 Outliers

Outliers are data points that do not follow the usual pattern of the rest of the data. In the setting of linear regression, these are points that have a vertical distance to the line of best fit that is either much higher or much lower than you might expect based on the rest of the data. The problem with outliers is that they can have *too much influence* on the line of best fit. In general, it

is very difficult to judge accurately which data are outliers without advanced techniques that are beyond the scope of this book.

But to illustrate what can happen when you have outliers, Fig. 8.8 shows a small subset of the Sacramento housing data again, except we have added a *single* data point (highlighted in red). This house is 5,000 square feet in size, and sold for only $50,000. Unbeknownst to the data analyst, this house was sold by a parent to their child for an absurdly low price. Of course, this is not representative of the real housing market values that the other data points follow; the data point is an *outlier*. In orange we plot the original line of best fit, and in red we plot the new line of best fit including the outlier. You can see how different the red line is from the orange line, which is entirely caused by that one extra outlier data point.

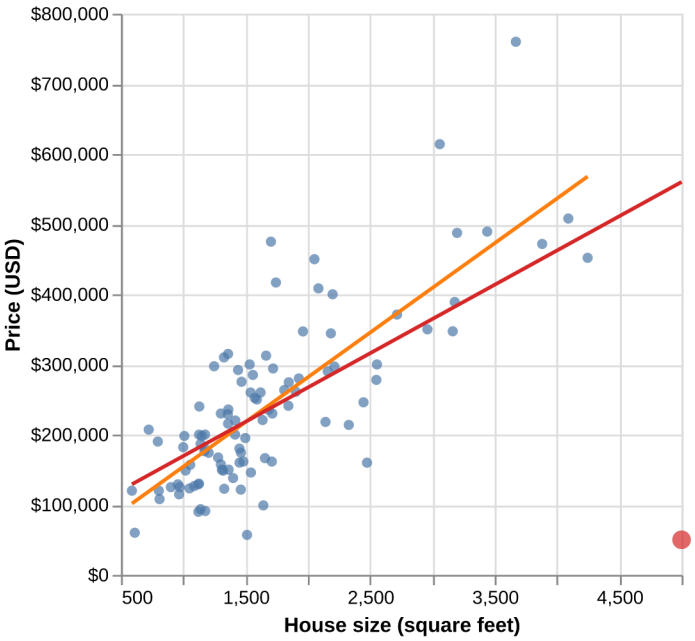


FIGURE 8.8 Scatter plot of a subset of the data, with outlier highlighted in red.

Fortunately, if you have enough data, the inclusion of one or two outliers—as long as their values are not *too* wild—will typically not have a large effect on the line of best fit. Fig. 8.9 shows how that same outlier data point from earlier influences the line of best fit when we are working with the entire original Sacramento training data. You can see that with this larger data set, the line changes much less when adding the outlier. Nevertheless, it is still important when working with linear regression to critically think about how much any individual data point is influencing the model.

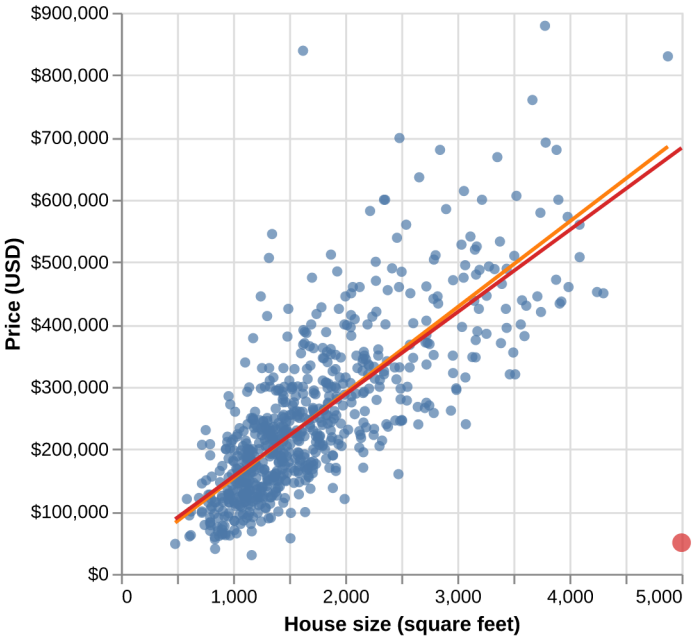


FIGURE 8.9 Scatter plot of the full data, with outlier highlighted in red.

8.7.2 Multicollinearity

The second, and much more subtle, issue can occur when performing multivariable linear regression. In particular, if you include multiple predictors that are strongly linearly related to one another, the coefficients that describe the plane of best fit can be very unreliable—small changes to the data can result in large changes in the coefficients. Consider an extreme example using the Sacramento housing data where the house was measured twice by two people. Since the two people are each slightly inaccurate, the two measurements might not agree exactly, but they are very strongly linearly related to each other, as shown in Fig. 8.10.

If we again fit the multivariable linear regression model on this data, then the plane of best fit has regression coefficients that are very sensitive to the exact values in the data. For example, if we change the data ever so slightly—e.g., by running cross-validation, which splits up the data randomly into different chunks—the coefficients vary by large amounts:

Best Fit 1: house sale price $= 17{,}238 + 169 \cdot(\text{house size 1 } (\text{ft}^2)) + -32 \cdot(\text{house size 2 } (\text{ft}^2))$.

Best Fit 2: house sale price $= 7{,}041 + -28 \cdot(\text{house size 1 } (\text{ft}^2)) + 166 \cdot(\text{house size 2 } (\text{ft}^2))$.

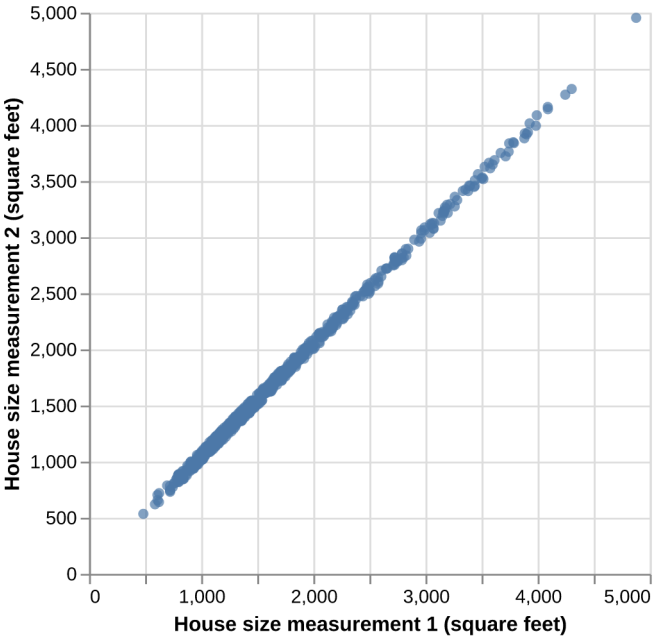


FIGURE 8.10 Scatter plot of house size (in square feet) measured by person 1 versus house size (in square feet) measured by person 2.

Best Fit 3: house sale price = 15,539 + 135 ·(house size 1 (ft^2))+ 2 ·(house size 2 (ft^2)).

Therefore, when performing multivariable linear regression, it is important to avoid including very linearly related predictors. However, techniques for doing so are beyond the scope of this book; see the list of additional resources at the end of this chapter to find out where you can learn more.

8.8 Designing new predictors

We were quite fortunate in our initial exploration to find a predictor variable (house size) that seems to have a meaningful and nearly linear relationship with our response variable (sale price). But what should we do if we cannot immediately find such a nice variable? Well, sometimes it is just a fact that the variables in the data do not have enough of a relationship with the response variable to provide useful predictions. For example, if the only available predictor was "the current house owner's favorite ice cream flavor", we likely would have little hope of using that variable to predict the house's sale price (barring any future remarkable scientific discoveries about the

relationship between the housing market and homeowner ice cream preferences). In cases like these, the only option is to obtain measurements of more useful variables.

There are, however, a wide variety of cases where the predictor variables do have a meaningful relationship with the response variable, but that relationship does not fit the assumptions of the regression method you have chosen. For example, a data frame `df` with two variables—`x` and `y`—with a nonlinear relationship between the two variables will not be fully captured by simple linear regression, as shown in Fig. 8.11.

```
df
```

```
          x         y
0    0.5994  0.288853
1    0.1688  0.092090
2    0.9859  1.021194
3    0.9160  0.812375
4    0.6400  0.212624
..      ...       ...
95   0.7341  0.333609
96   0.8434  0.656970
97   0.3329  0.106273
98   0.7170  0.311442
99   0.7895  0.567003

[100 rows x 2 columns]
```

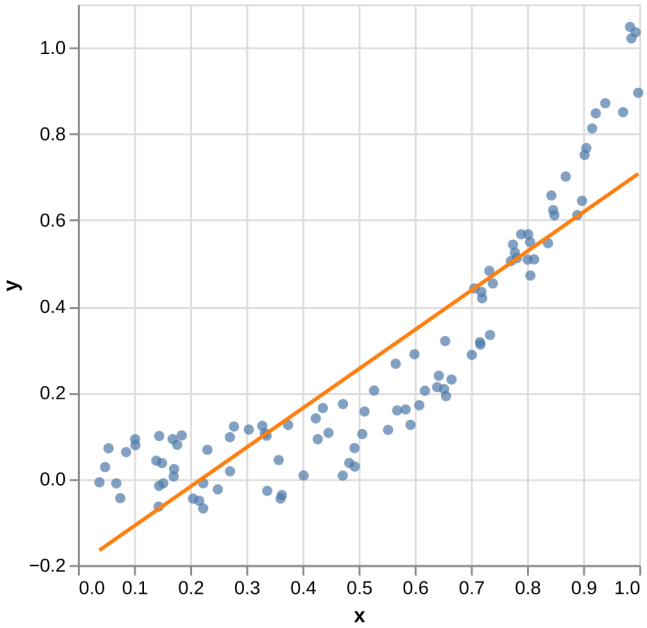


FIGURE 8.11 Example of a data set with a nonlinear relationship between the predictor and the response.

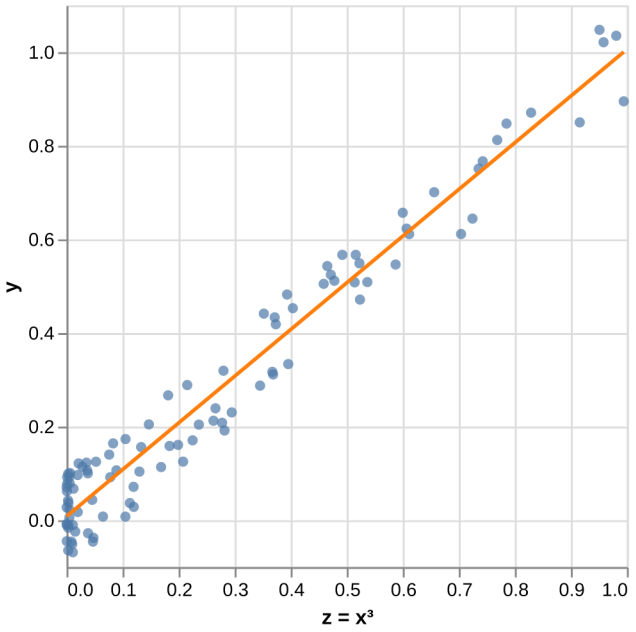


FIGURE 8.12 Relationship between the transformed predictor and the response.

Instead of trying to predict the response `y` using a linear regression on `x`, we might have some scientific background about our problem to suggest that `y` should be a cubic function of `x`. So before performing regression, we might *create a new predictor variable* `z`:

```
df["z"] = df["x"] ** 3
```

Then we can perform linear regression for `y` using the predictor variable `z`, as shown in Fig. 8.12. Here you can see that the transformed predictor `z` helps the linear regression model make more accurate predictions. Note that none of the `y` response values have changed between Figs. 8.11 and 8.12; the only change is that the `x` values have been replaced by `z` values.

The process of transforming predictors (and potentially combining multiple predictors in the process) is known as *feature engineering*. In real data analysis problems, you will need to rely on a deep understanding of the problem—as well as the wrangling tools from previous chapters—to engineer useful new features that improve predictive performance.

Note: Feature engineering is *part of tuning your model*, and as such you must not use your test data to evaluate the quality of the features you produce. You are free to use cross-validation, though.

8.9 The other sides of regression

So far in this textbook we have used regression only in the context of prediction. However, regression can also be seen as a method to understand and quantify the effects of individual variables on a response variable of interest. In the housing example from this chapter, beyond just using past data to predict future sale prices, we might also be interested in describing the individual relationships of house size and the number of bedrooms with house price, quantifying how strong each of these relationships are, and assessing how accurately we can estimate their magnitudes. And even beyond that, we may be interested in understanding whether the predictors *cause* changes in the price. These sides of regression are well beyond the scope of this book; but the material you have learned here should give you a foundation of knowledge that will serve you well when moving to more advanced books on the topic.

8.10 Exercises

Practice exercises for the material covered in this chapter can be found in the accompanying worksheets repository[1] in the "Regression II: linear regression" row. You can launch an interactive version of the worksheet in your browser by clicking the "launch binder" button. You can also preview a non-interactive version of the worksheet by clicking "view worksheet". If you instead decide to download the worksheet and run it on your own machine, make sure to follow the instructions for computer setup found in Chapter 13. This will ensure that the automated feedback and guidance that the worksheets provide will function as intended.

8.11 Additional resources

- The `scikit-learn` website[2] is an excellent reference for more details on, and advanced usage of, the functions and packages in the past two chapters. Aside from that, it also offers many useful tutorials[3] and an extensive list of

[1] https://worksheets.python.datasciencebook.ca
[2] https://scikit-learn.org/stable/
[3] https://scikit-learn.org/stable/tutorial/index.html

more advanced examples[4] that you can use to continue learning beyond the scope of this book.

- *An Introduction to Statistical Learning* [James *et al.*, 2013] provides a great next stop in the process of learning about regression. Chapter 3 covers linear regression at a slightly more mathematical level than we do here, but it is not too large a leap and so should provide a good stepping stone. Chapter 6 discusses how to pick a subset of "informative" predictors when you have a data set with many predictors, and you expect only a few of them to be relevant. Chapter 7 covers regression models that are more flexible than linear regression models but still enjoy the computational efficiency of linear regression. In contrast, the K-NN methods we covered earlier are indeed more flexible but become very slow when given lots of data.

[4] https://scikit-learn.org/stable/auto_examples/index.html#general-examples

9

Clustering

9.1 Overview

As part of exploratory data analysis, it is often helpful to see if there are meaningful subgroups (or *clusters*) in the data. This grouping can be used for many purposes, such as generating new questions or improving predictive analyses. This chapter provides an introduction to clustering using the K-means algorithm, including techniques to choose the number of clusters.

9.2 Chapter learning objectives

By the end of the chapter, readers will be able to do the following:

- Describe a situation in which clustering is an appropriate technique to use, and what insight it might extract from the data.
- Explain the K-means clustering algorithm.
- Interpret the output of a K-means analysis.
- Differentiate between clustering, classification, and regression.
- Identify when it is necessary to scale variables before clustering, and do this using Python.
- Perform K-means clustering in Python using `scikit-learn`.
- Use the elbow method to choose the number of clusters for K-means.
- Visualize the output of K-means clustering in Python using a colored scatter plot.
- Describe advantages, limitations and assumptions of the K-means clustering algorithm.

DOI: 10.1201/9781003438397-9

9.3 Clustering

Clustering is a data analysis task involving separating a data set into subgroups of related data. For example, we might use clustering to separate a data set of documents into groups that correspond to topics, a data set of human genetic information into groups that correspond to ancestral subpopulations, or a data set of online customers into groups that correspond to purchasing behaviors. Once the data are separated, we can, for example, use the subgroups to generate new questions about the data and follow up with a predictive modeling exercise. In this course, clustering will be used only for exploratory analysis, i.e., uncovering patterns in the data.

Note that clustering is a fundamentally different kind of task than classification or regression. In particular, both classification and regression are *supervised tasks* where there is a *response variable* (a category label or value), and we have examples of past data with labels/values that help us predict those of future data. By contrast, clustering is an *unsupervised task*, as we are trying to understand and examine the structure of data without any response variable labels or values to help us. This approach has both advantages and disadvantages. Clustering requires no additional annotation or input on the data. For example, while it would be nearly impossible to annotate all the articles on Wikipedia with human-made topic labels, we can cluster the articles without this information to find groupings corresponding to topics automatically. However, given that there is no response variable, it is not as easy to evaluate the "quality" of a clustering. With classification, we can use a test data set to assess prediction performance. In clustering, there is not a single good choice for evaluation. In this book, we will use visualization to ascertain the quality of a clustering, and leave rigorous evaluation for more advanced courses.

Given that there is no response variable, it is not as easy to evaluate the "quality" of a clustering. With classification, we can use a test data set to assess prediction performance. In clustering, there is not a single good choice for evaluation. In this book, we will use visualization to ascertain the quality of a clustering, and leave rigorous evaluation for more advanced courses.

As in the case of classification, there are many possible methods that we could use to cluster our observations to look for subgroups. In this book, we will focus on the widely used K-means algorithm [Lloyd, 1982]. In your future studies, you might encounter hierarchical clustering, principal component analysis, multidimensional scaling, and more; see the additional resources section at the end of this chapter for where to begin learning more about these other methods.

Note: There are also so-called *semisupervised* tasks, where only some of the data come with response variable labels/values, but the vast majority don't. The goal is to try to uncover underlying structure in the data that allows one to guess the missing labels. This sort of task is beneficial, for example, when one has an unlabeled data set that is too large to manually label, but one is willing to provide a few informative example labels as a "seed" to guess the labels for all the data.

9.4 An illustrative example

In this chapter, we will focus on a data set from the `palmerpenguins` R package[1] [Horst *et al.*, 2020]. This data set was collected by Dr. Kristen Gorman and the Palmer Station, Antarctica Long Term Ecological Research Site, and includes measurements for adult penguins (Fig. 9.1) found near there [Gorman *et al.*, 2014]. Our goal will be to use two variables—penguin bill and flipper length, both in millimeters—to determine whether there are distinct types of penguins in our data. Understanding this might help us with species discovery and classification in a data-driven way. Note that we have reduced the size of the data set to 18 observations and 2 variables; this will help us make clear visualizations that illustrate how clustering works for learning purposes.

Before we get started, we will set a random seed. This will ensure that our analysis will be reproducible. As we will learn in more detail later in the chapter, setting the seed here is important because the K-means clustering algorithm uses randomness when choosing a starting position for each cluster.

```
import numpy as np

np.random.seed(6)
```

Now we can load and preview the `penguins` data.

```
import pandas as pd

penguins = pd.read_csv("data/penguins.csv")
penguins
```

[1] https://allisonhorst.github.io/palmerpenguins/

FIGURE 9.1 A Gentoo penguin.

	bill_length_mm	flipper_length_mm
0	39.2	196
1	36.5	182
2	34.5	187
3	36.7	187
4	38.1	181
5	39.2	190
6	36.0	195
7	37.8	193
8	46.5	213
9	46.1	215
10	47.8	215
11	45.0	220
12	49.1	212
13	43.3	208
14	46.0	195
15	46.7	195
16	52.2	197
17	46.8	189

We will begin by using a version of the data that we have standardized, `penguins_standardized`, to illustrate how K-means clustering works (recall standardization from Chapter 5). Later in this chapter, we will return to the original `penguins` data to see how to include standardization automatically in the clustering pipeline.

```
penguins_standardized
```

	bill_length_standardized	flipper_length_standardized
0	-0.641361	-0.189773
1	-1.144917	-1.328412
2	-1.517922	-0.921755
3	-1.107617	-0.921755
4	-0.846513	-1.409743
5	-0.641361	-0.677761
6	-1.238168	-0.271104
7	-0.902464	-0.433767
8	0.720106	1.192860
9	0.645505	1.355522
10	0.962559	1.355522
11	0.440353	1.762179
12	1.205012	1.111528
13	0.123299	0.786203
14	0.626855	-0.271104
15	0.757407	-0.271104
16	1.783170	-0.108442
17	0.776057	-0.759092

Next, we can create a scatter plot using this data set to see if we can detect subtypes or groups in our data set.

```
import altair as alt

scatter_plot = alt.Chart(penguins_standardized).mark_circle().encode(
    x=alt.X("flipper_length_standardized").title("Flipper Length (standardized)
↪"),
    y=alt.Y("bill_length_standardized").title("Bill Length (standardized)")
)
```

Based on the visualization in Fig. 9.2, we might suspect there are a few subtypes of penguins within our data set. We can see roughly 3 groups of observations in Fig. 9.2, including:

1. a small flipper and bill length group,
2. a small flipper length, but large bill length group, and
3. a large flipper and bill length group.

Data visualization is a great tool to give us a rough sense of such patterns when we have a small number of variables. But if we are to group data—and select the number of groups—as part of a reproducible analysis, we need something a bit more automated. Additionally, finding groups via visualization becomes more difficult as we increase the number of variables we consider when clustering. The way to rigorously separate the data into groups is to use a clustering algorithm. In this chapter, we will focus on the *K-means* algorithm, a widely used and often very effective clustering method, combined with the *elbow method* for selecting the number of clusters. This procedure will separate the data into groups; Fig. 9.3 shows these groups denoted by colored scatter points.

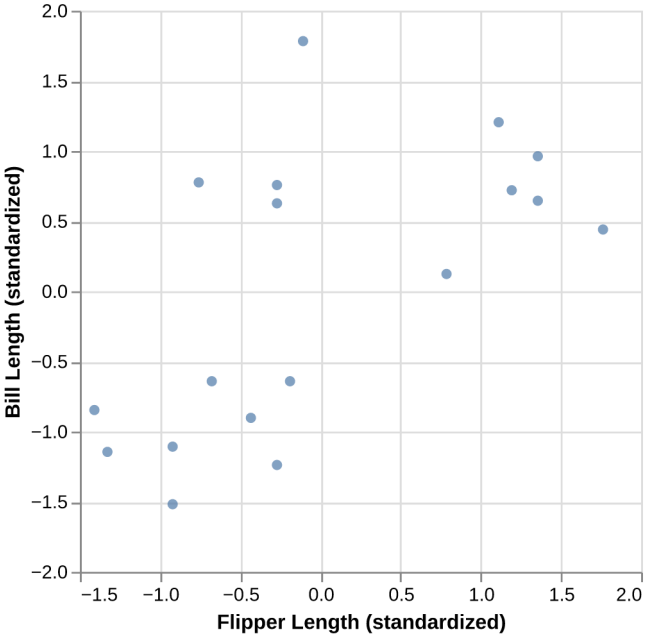


FIGURE 9.2 Scatter plot of standardized bill length versus standardized flipper length.

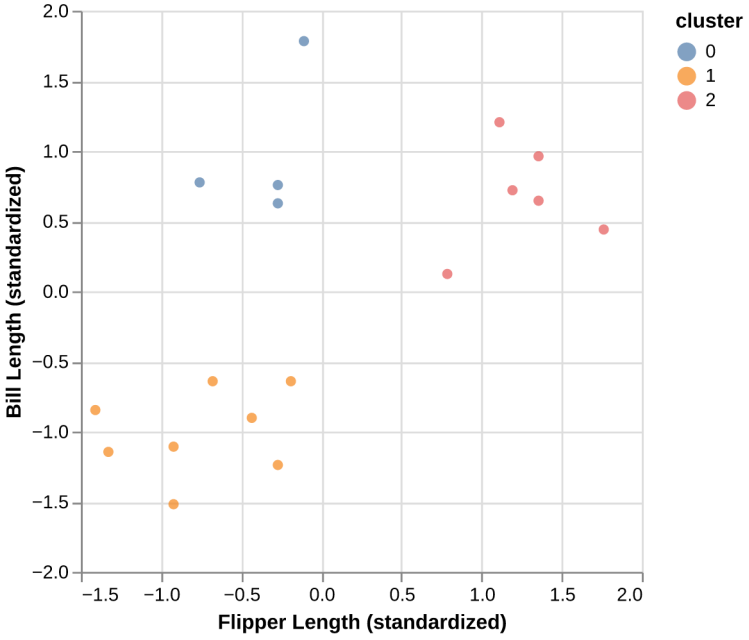


FIGURE 9.3 Scatter plot of standardized bill length versus standardized flipper length with colored groups.

What are the labels for these groups? Unfortunately, we don't have any. K-means, like almost all clustering algorithms, just outputs meaningless "cluster labels" that are typically whole numbers: 0, 1, 2, 3, etc. But in a simple case like this, where we can easily visualize the clusters on a scatter plot, we can give human-made labels to the groups using their positions on the plot:

- small flipper length and small bill length (orange cluster),
- small flipper length and large bill length (blue cluster).
- and large flipper length and large bill length (red cluster).

Once we have made these determinations, we can use them to inform our species classifications or ask further questions about our data. For example, we might be interested in understanding the relationship between flipper length and bill length, and that relationship may differ depending on the type of penguin we have.

9.5 K-means

9.5.1 Measuring cluster quality

The K-means algorithm is a procedure that groups data into K clusters. It starts with an initial clustering of the data, and then iteratively improves it by making adjustments to the assignment of data to clusters until it cannot improve any further. But how do we measure the "quality" of a clustering, and what does it mean to improve it? In K-means clustering, we measure the quality of a cluster by its *within-cluster sum-of-squared-distances* (WSSD), also called *inertia*. Computing this involves two steps. First, we find the cluster centers by computing the mean of each variable over data points in the cluster. For example, suppose we have a cluster containing four observations, and we are using two variables, x and y, to cluster the data. Then we would compute the coordinates, μ_x and μ_y, of the cluster center via

$$\mu_x = \frac{1}{4}(x_1 + x_2 + x_3 + x_4) \quad \mu_y = \frac{1}{4}(y_1 + y_2 + y_3 + y_4)$$

In the first cluster from the example, there are 4 data points. These are shown with their cluster center (standardized flipper length -0.35, standardized bill length 0.99) highlighted in Fig. 9.4

The second step in computing the WSSD is to add up the squared distance between each point in the cluster and the cluster center. We use the straight-line / Euclidean distance formula that we learned about in Chapter 5. In the

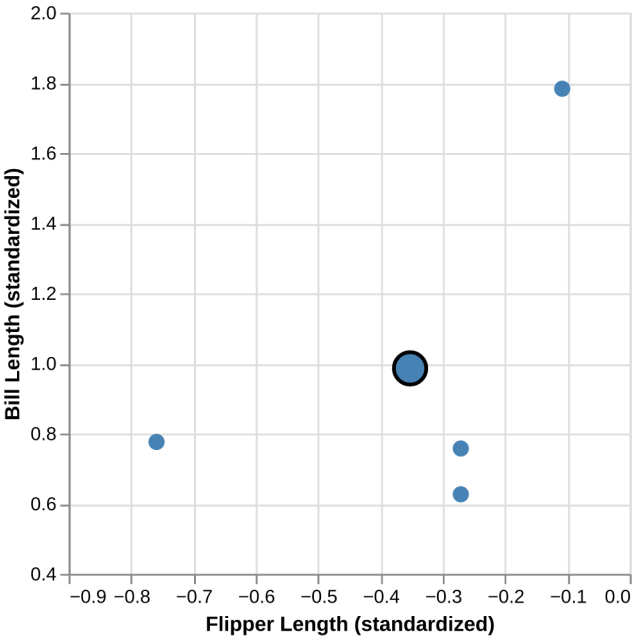


FIGURE 9.4 Cluster 0 from the `penguins_standardized` data set example. Observations are small blue points, with the cluster center highlighted as a large blue point with a black outline.

4-observation cluster example above, we would compute the WSSD S^2 via

$$S^2 = \left((x_1 - \mu_x)^2 + (y_1 - \mu_y)^2\right) + \left((x_2 - \mu_x)^2 + (y_2 - \mu_y)^2\right) + \left((x_3 - \mu_x)^2 + (y_3 - \mu_y)^2\right) + \left((x_4 - \mu_x)^2 + (y_4 - \mu_y)^2\right)$$

These distances are denoted by lines in Fig. 9.5 for the first cluster of the penguin data example.

The larger the value of S^2, the more spread out the cluster is, since large S^2 means that points are far from the cluster center. Note, however, that "large" is relative to *both* the scale of the variables for clustering *and* the number of points in the cluster. A cluster where points are very close to the center might still have a large S^2 if there are many data points in the cluster.

After we have calculated the WSSD for all the clusters, we sum them together to get the *total WSSD*. For our example, this means adding up all the squared distances for the 18 observations. These distances are denoted by black lines in Fig. 9.6.

Since K-means uses the straight-line distance to measure the quality of a clustering, it is limited to clustering based on quantitative variables. However, note that there are variants of the K-means algorithm, as well as other

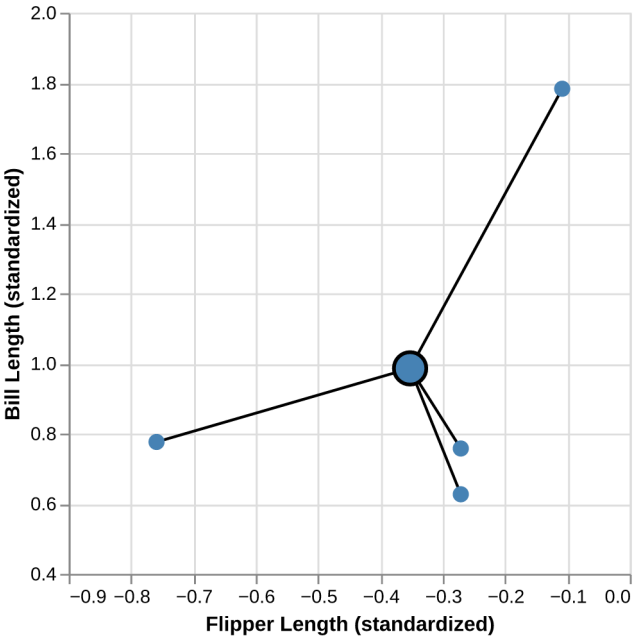


FIGURE 9.5 Cluster 0 from the `penguins_standardized` data set example. Observations are small blue points, with the cluster center highlighted as a large blue point with a black outline. The distances from the observations to the cluster center are represented as black lines.

clustering algorithms entirely, that use other distance metrics to allow for non-quantitative data to be clustered. These are beyond the scope of this book.

9.5.2 The clustering algorithm

We begin the K-means algorithm by picking K, and randomly assigning a roughly equal number of observations to each of the K clusters. An example random initialization is shown in Fig. 9.7

Then K-means consists of two major steps that attempt to minimize the sum of WSSDs over all the clusters, i.e., the *total WSSD*:

1. **Center update:** Compute the center of each cluster.
2. **Label update:** Reassign each data point to the cluster with the nearest center.

These two steps are repeated until the cluster assignments no longer change. We show what the first three iterations of K-means would look like in Fig. 9.8. Each row corresponds to an iteration, where the left column depicts the center

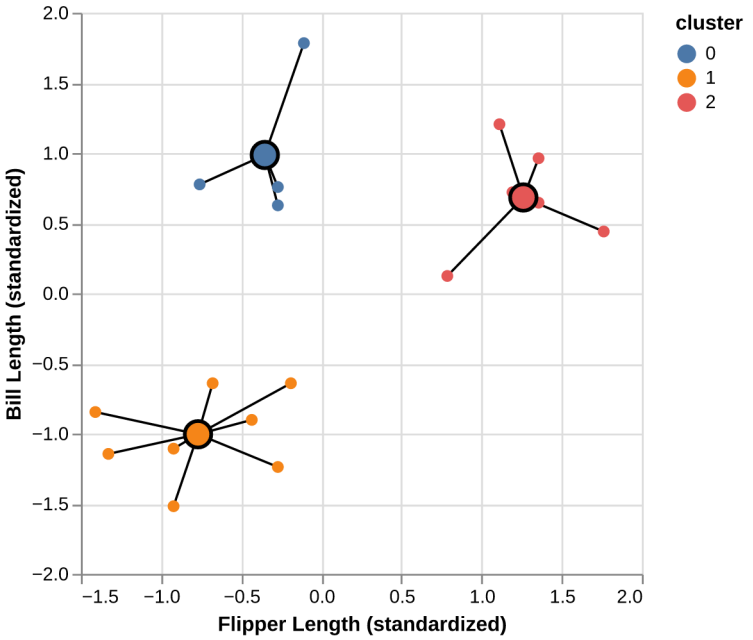


FIGURE 9.6 All clusters from the `penguins_standardized` data set example. Observations are small orange, blue, and yellow points with cluster centers denoted by larger points with a black outline. The distances from the observations to each of the respective cluster centers are represented as black lines.

update, and the right column depicts the label update (i.e., the reassignment of data to clusters).

Note that at this point, we can terminate the algorithm since none of the assignments changed in the third iteration; both the centers and labels will remain the same from this point onward.

Note: Is K-means *guaranteed* to stop at some point, or could it iterate forever? As it turns out, thankfully, the answer is that K-means is guaranteed to stop after *some* number of iterations. For the interested reader, the logic for this has three steps: (1) both the label update and the center update decrease total WSSD in each iteration, (2) the total WSSD is always greater than or equal to 0, and (3) there are only a finite number of possible ways to assign the data to clusters. So at some point, the total WSSD must stop decreasing, which means none of the assignments are changing, and the algorithm terminates.

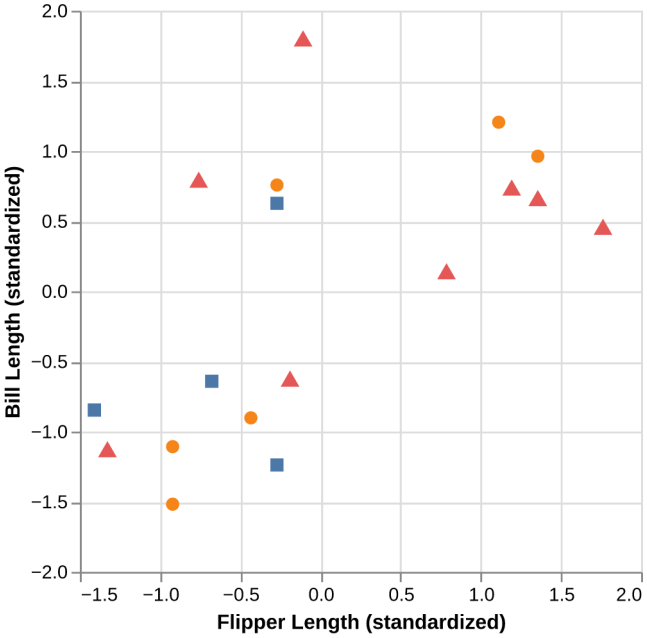


FIGURE 9.7 Random initialization of labels. Each cluster is depicted as a different color and shape.

9.5.3 Random restarts

Unlike the classification and regression models we studied in previous chapters, K-means can get "stuck" in a bad solution. For example, Fig. 9.9 illustrates an unlucky random initialization by K-means.

Fig. 9.10 shows what the iterations of K-means would look like with the unlucky random initialization shown in Fig. 9.9

This looks like a relatively bad clustering of the data, but K-means cannot improve it. To solve this problem when clustering data using K-means, we should randomly re-initialize the labels a few times, run K-means for each initialization, and pick the clustering that has the lowest final total WSSD.

9.5.4 Choosing K

In order to cluster data using K-means, we also have to pick the number of clusters, K. But unlike in classification, we have no response variable and cannot perform cross-validation with some measure of model prediction error. Further, if K is chosen too small, then multiple clusters get grouped together; if K is too large, then clusters get subdivided. In both cases, we will potentially miss interesting structure in the data. Fig. 9.11 illustrates the impact of K on

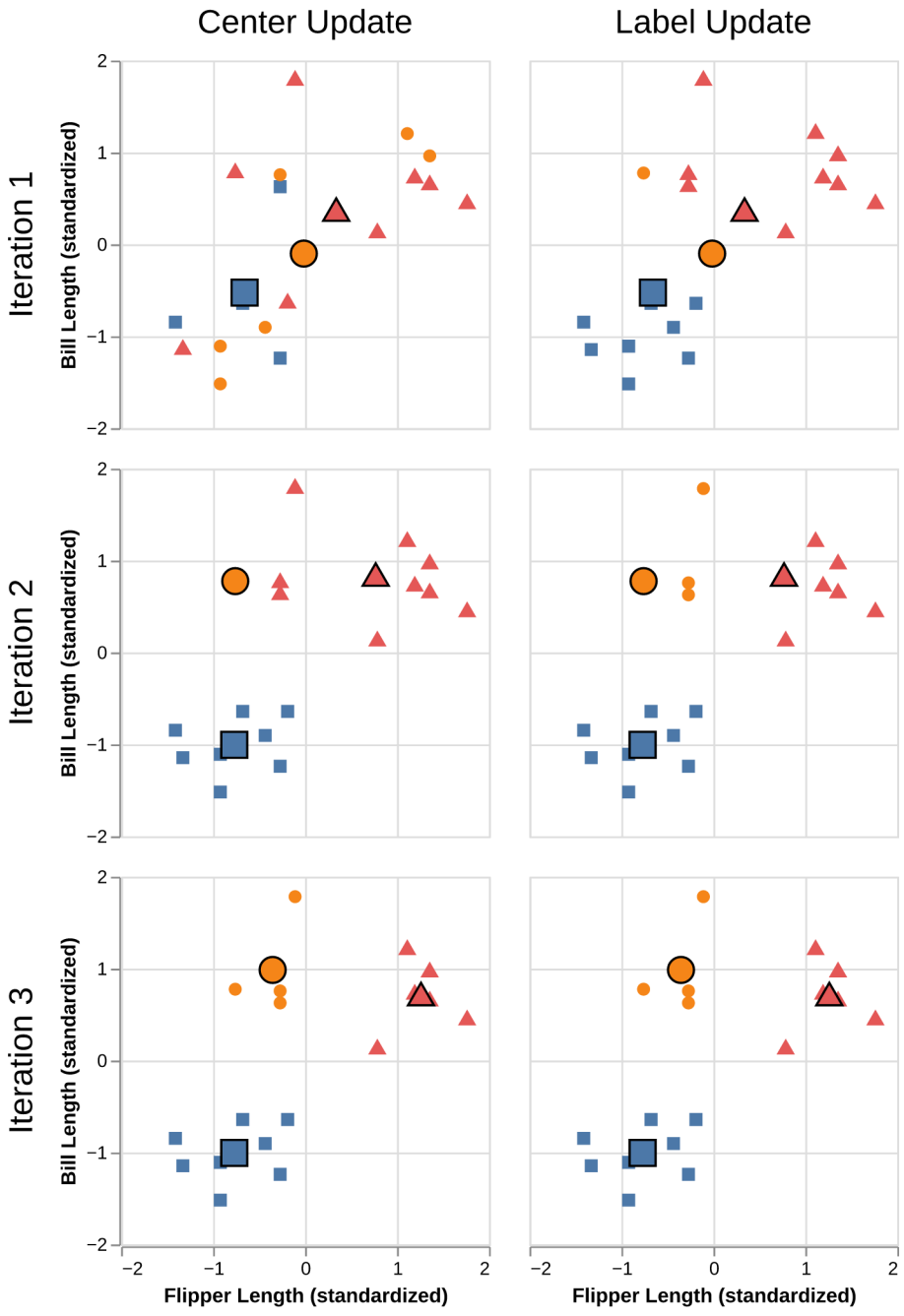


FIGURE 9.8 First three iterations of K-means clustering on the `penguins_standardized` example data set. Each pair of plots corresponds to an iteration. Within the pair, the first plot depicts the center update, and the second plot depicts the reassignment of data to clusters. Cluster centers are indicated by larger points that are outlined in black.

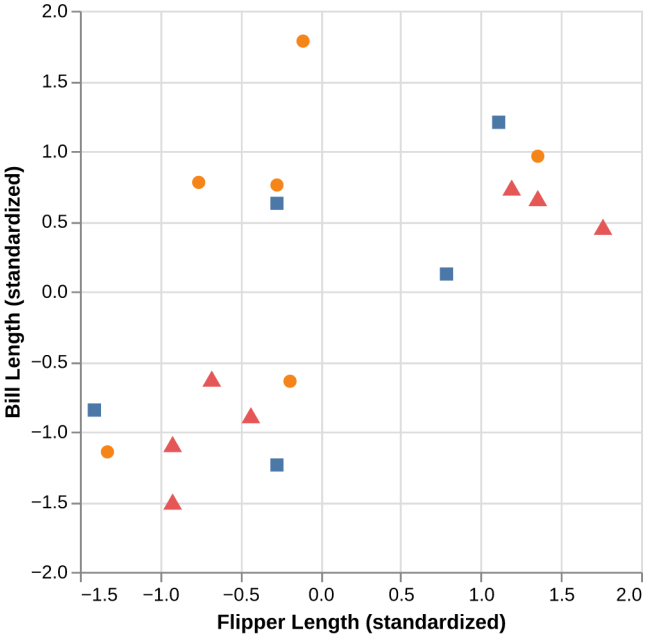


FIGURE 9.9 Random initialization of labels.

K-means clustering of our penguin flipper and bill length data by showing the different clusterings for K's ranging from 1 to 9.

If we set K less than 3, then the clustering merges separate groups of data; this causes a large total WSSD, since the cluster center (denoted by large shapes with black outlines) is not close to any of the data in the cluster. On the other hand, if we set K greater than 3, the clustering subdivides subgroups of data; this does indeed still decrease the total WSSD, but by only a *diminishing amount*. If we plot the total WSSD versus the number of clusters, we see that the decrease in total WSSD levels off (or forms an "elbow shape") when we reach roughly the right number of clusters (Fig. 9.12).

9.6 K-means in Python

We can perform K-means in Python using a workflow similar to those in the earlier classification and regression chapters. Returning to the original (unstandardized) `penguins` data, recall that K-means clustering uses straight-line distance to decide which points are similar to each other. Therefore, the *scale* of each of the variables in the data will influence which cluster data points end up being assigned. Variables with a large scale will have a much

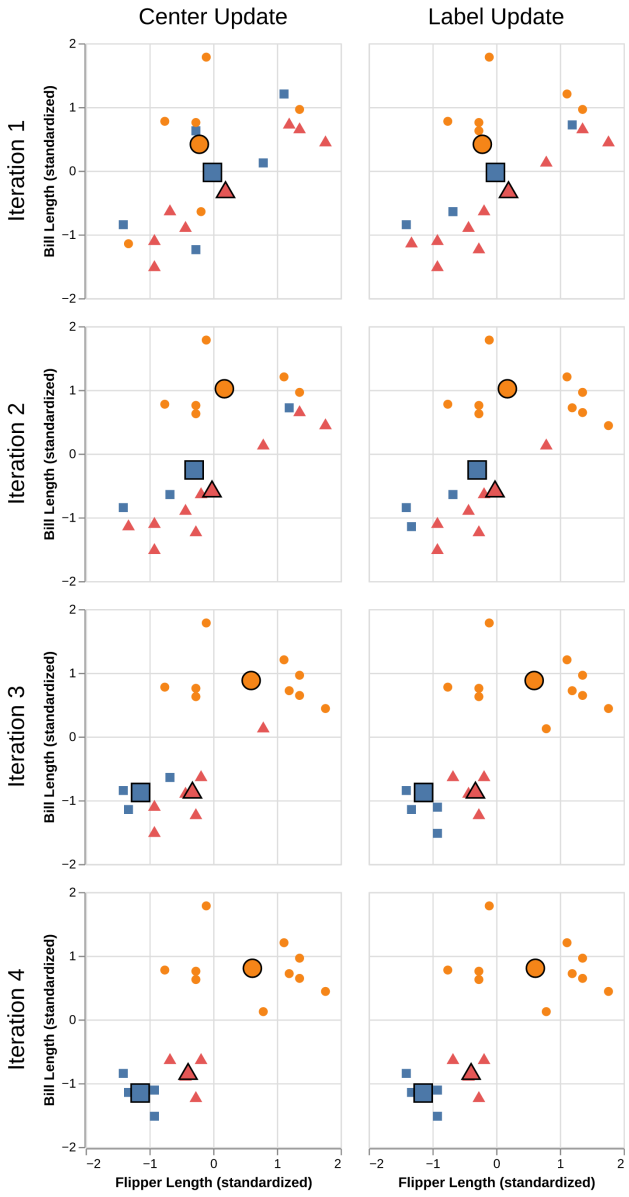


FIGURE 9.10 First four iterations of K-means clustering on the `penguins_standardized` example data set with a poor random initialization. Each pair of plots corresponds to an iteration. Within the pair, the first plot depicts the center update, and the second plot depicts the reassignment of data to clusters. Cluster centers are indicated by larger points that are outlined in black.

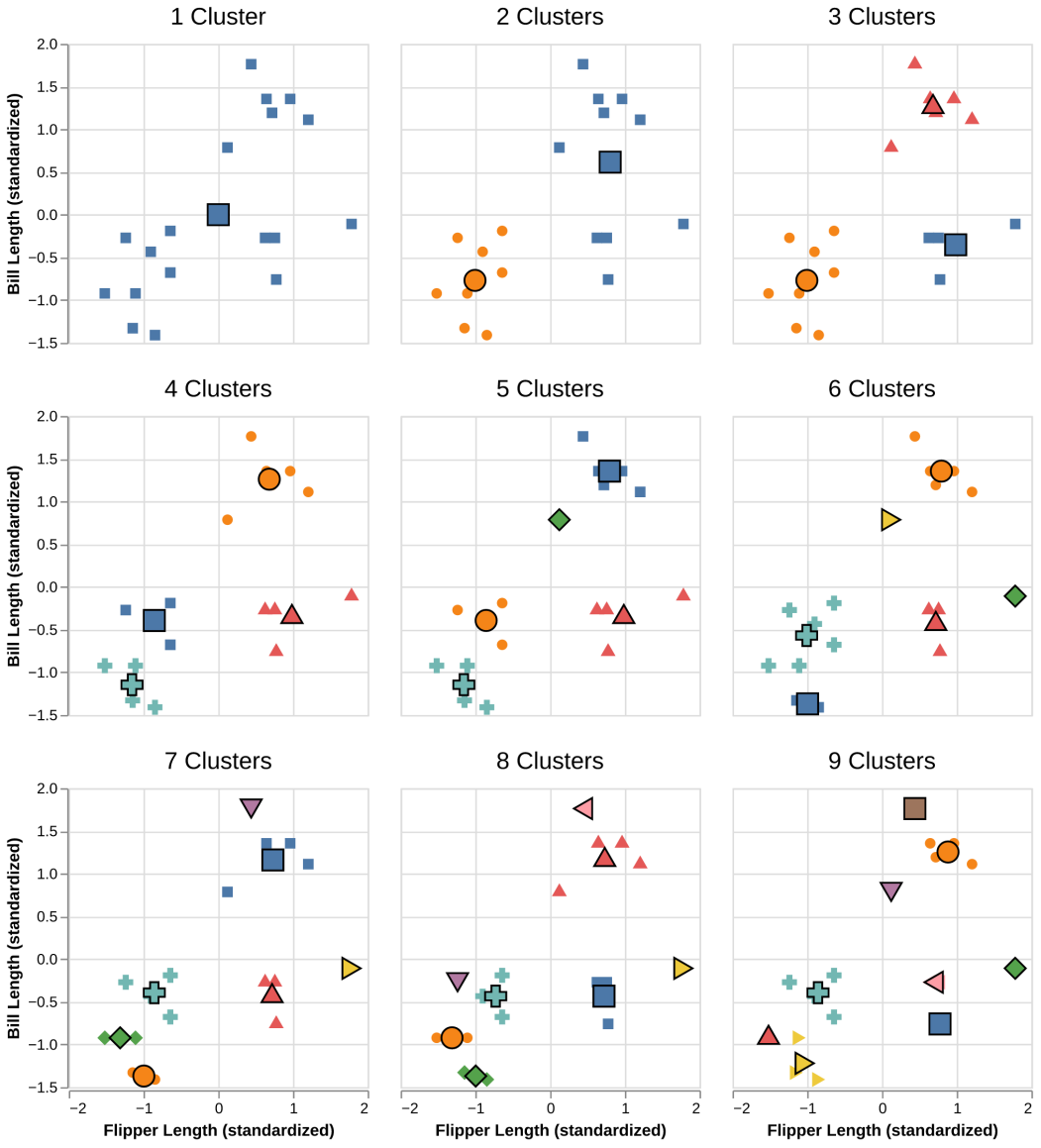


FIGURE 9.11 Clustering of the penguin data for K clusters ranging from 1 to 9. Cluster centers are indicated by larger points that are outlined in black.

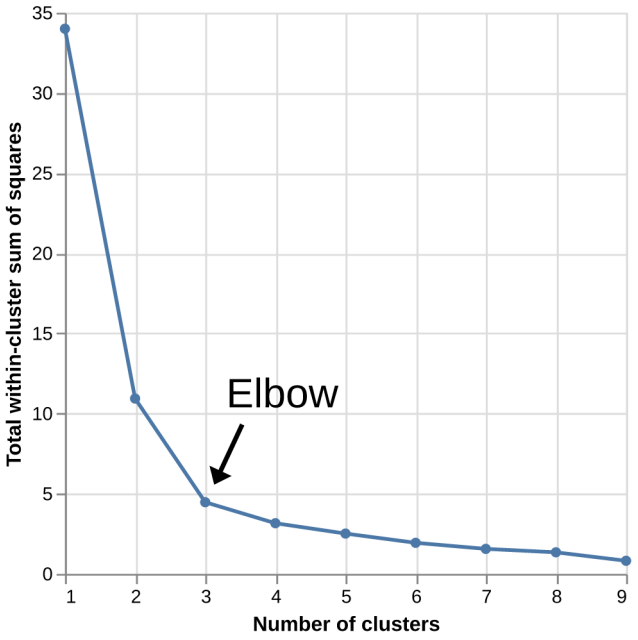


FIGURE 9.12 Total WSSD for K clusters ranging from 1 to 9.

larger effect on deciding cluster assignment than variables with a small scale. To address this problem, we typically standardize our data before clustering, which ensures that each variable has a mean of 0 and standard deviation of 1. The `StandardScaler` function in `scikit-learn` can be used to do this.

```
from sklearn.preprocessing import StandardScaler
from sklearn.compose import make_column_transformer
from sklearn import set_config

# Output dataframes instead of arrays
set_config(transform_output="pandas")

preprocessor = make_column_transformer(
    (StandardScaler(), ["bill_length_mm", "flipper_length_mm"]),
    verbose_feature_names_out=False,
)
preprocessor
```

```
ColumnTransformer(transformers=[('standardscaler', StandardScaler(),
                                 ['bill_length_mm', 'flipper_length_mm'])],
                  verbose_feature_names_out=False)
```

To indicate that we are performing K-means clustering, we will create a `KMeans` model object. It takes at least one argument: the number of clusters `n_clusters`, which we set to 3.

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=3)
kmeans
```

```
KMeans(n_clusters=3)
```

To actually run the K-means clustering, we combine the preprocessor and model object in a `Pipeline`, and use the `fit` function. Note that the K-means algorithm uses a random initialization of assignments, but since we set the random seed in the beginning of this chapter, the clustering will be reproducible.

```
from sklearn.pipeline import make_pipeline

penguin_clust = make_pipeline(preprocessor, kmeans)
penguin_clust.fit(penguins)
penguin_clust
```

```
Pipeline(steps=[('columntransformer',
                 ColumnTransformer(transformers=[('standardscaler',
                                                  StandardScaler(),
                                                  ['bill_length_mm',
                                                   'flipper_length_mm'])],
                                   verbose_feature_names_out=False)),
                ('kmeans', KMeans(n_clusters=3))])
```

The fit `KMeans` object—which is the second item in the pipeline, and can be accessed as `penguin_clust[1]`—has a lot of information that can be used to visualize the clusters, pick K, and evaluate the total WSSD. Let's start by visualizing the clusters as a colored scatter plot. In order to do that, we first need to augment our original `penguins` data frame with the cluster assignments. We can access these using the `labels_` attribute of the clustering object ("labels" is a common alternative term to "assignments" in clustering), and add them to the data frame.

```
penguins["cluster"] = penguin_clust[1].labels_
penguins
```

```
    bill_length_mm  flipper_length_mm  cluster
0             39.2                196        1
1             36.5                182        1
2             34.5                187        1
3             36.7                187        1
4             38.1                181        1
5             39.2                190        1
6             36.0                195        1
7             37.8                193        1
8             46.5                213        2
9             46.1                215        2
10            47.8                215        2
```

(continues on next page)

(continued from previous page)

```
11              45.0              220         2
12              49.1              212         2
13              43.3              208         2
14              46.0              195         0
15              46.7              195         0
16              52.2              197         0
17              46.8              189         0
```

Now that we have the cluster assignments included in the `penguins` data frame, we can visualize them as shown in Fig. 9.13. Note that we are plotting the *un-standardized* data here; if we for some reason wanted to visualize the *standardized* data, we would need to use the `fit` and `transform` functions on the `StandardScaler` preprocessor directly to obtain that first. As in Chapter 4, adding the `:N` suffix ensures that `altair` will treat the `cluster` variable as a nominal/categorical variable, and hence use a discrete color map for the visualization.

```
cluster_plot=alt.Chart(penguins).mark_circle().encode(
    x=alt.X("flipper_length_mm").title("Flipper Length").scale(zero=False),
    y=alt.Y("bill_length_mm").title("Bill Length").scale(zero=False),
    color=alt.Color("cluster:N").title("Cluster"),
)
```

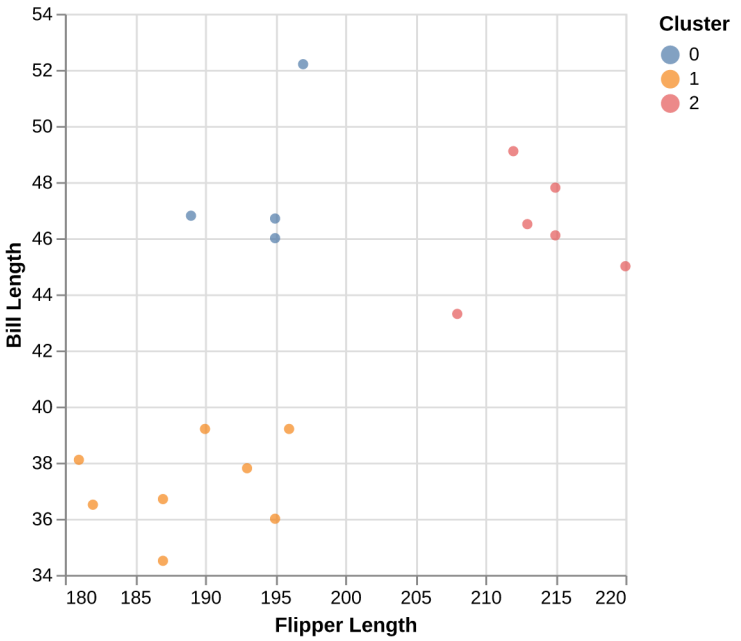


FIGURE 9.13 The data colored by the cluster assignments returned by K-means.

As mentioned above, we also need to select K by finding where the "elbow" occurs in the plot of total WSSD versus the number of clusters. The total WSSD is stored in the `.inertia_` attribute of the clustering object ("inertia" is the term `scikit-learn` uses to denote WSSD).

```
penguin_clust[1].inertia_
```

```
4.730719092276117
```

To calculate the total WSSD for a variety of Ks, we will create a data frame that contains different values of `k` and the WSSD of running K-means with each values of k. To create this data frame, we will use what is called a "list comprehension" in Python, where we repeat an operation multiple times and return a list with the result. Here is an examples of a list comprehension that stores the numbers 0–2 in a list:

```
[n for n in range(3)]
```

```
[0, 1, 2]
```

We can change the variable `n` to be called whatever we prefer and we can also perform any operation we want as part of the list comprehension. For example, we could square all the numbers from 1 to 4 and store them in a list:

```
[number**2 for number in range(1, 5)]
```

```
[1, 4, 9, 16]
```

Next, we will use this approach to compute the WSSD for the K-values 1 through 9. For each value of K, we create a new `KMeans` model and wrap it in a `scikit-learn` pipeline with the preprocessor we created earlier. We store the WSSD values in a list that we will use to create a data frame of both the K-values and their corresponding WSSDs.

Note: We are creating the variable `ks` to store the range of possible k-values, so that we only need to change this range in one place if we decide to change which values of k we want to explore. Otherwise it would be easy to forget to update it in either the list comprehension or in the data frame assignment. If you are using a value multiple times, it is always the safest to assign it to a variable name for reuse.

```
ks = range(1, 10)
wssds = [
    make_pipeline(
```

(continues on next page)

(continued from previous page)

```
            preprocessor,
            KMeans(n_clusters=k)  # Create a new KMeans model with `k` clusters
    ).fit(penguins)[1].inertia_
    for k in ks
]

penguin_clust_ks = pd.DataFrame({
    "k": ks,
    "wssd": wssds,
})

penguin_clust_ks
```

	k	wssd
0	1	36.000000
1	2	11.576264
2	3	4.730719
3	4	3.343613
4	5	2.362131
5	6	1.678383
6	7	1.293320
7	8	0.975016
8	9	0.785232

Now that we have `wssd` and `k` as columns in a data frame, we can make a line plot (Fig. 9.14) and search for the "elbow" to find which value of K to use.

```
elbow_plot = alt.Chart(penguin_clust_ks).mark_line(point=True).encode(
    x=alt.X("k").title("Number of clusters"),
    y=alt.Y("wssd").title("Total within-cluster sum of squares"),
)
```

It looks like three clusters is the right choice for this data, since that is where the "elbow" of the line is the most distinct. In the plot, you can also see that the WSSD is always decreasing, as we would expect when we add more clusters. However, it is possible to have an elbow plot where the WSSD increases at one of the steps, causing a small bump in the line. This is because K-means can get "stuck" in a bad solution due to an unlucky initialization of the initial center positions as we mentioned earlier in the chapter.

Note: It is rare that the implementation of K-means from `scikit-learn` gets stuck in a bad solution, because `scikit-learn` tries to choose the initial centers carefully to prevent this from happening. If you still find yourself in a situation where you have a bump in the elbow plot, you can increase the `n_init` parameter when creating the `KMeans` object, e.g., `KMeans(n_clusters=k, n_init=10)`, to try more different random center initializations. The larger the value the better from an analysis perspective, but there is a trade-off that doing many clusterings could take a long time.

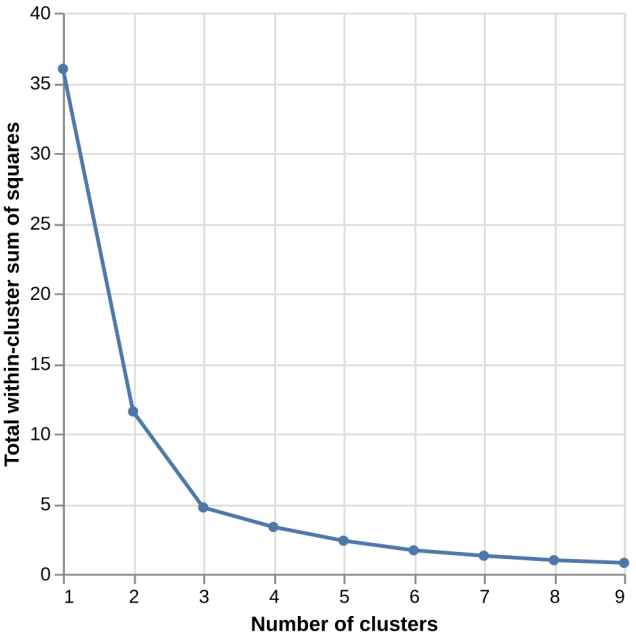


FIGURE 9.14 A plot showing the total WSSD versus the number of clusters.

9.7 Exercises

Practice exercises for the material covered in this chapter can be found in the accompanying worksheets repository[2] in the "Clustering" row. You can launch an interactive version of the worksheet in your browser by clicking the "launch binder" button. You can also preview a non-interactive version of the worksheet by clicking "view worksheet". If you instead decide to download the worksheet and run it on your own machine, make sure to follow the instructions for computer setup found in Chapter 13. This will ensure that the automated feedback and guidance that the worksheets provide will function as intended.

9.8 Additional resources

- Chapter 10 of *An Introduction to Statistical Learning* [James *et al.*, 2013] provides a great next stop in the process of learning about clustering and unsupervised learning in general. In the realm of clustering specifically, it provides a great companion introduction to K-means, but also covers

[2] https://worksheets.python.datasciencebook.ca

hierarchical clustering for when you expect there to be subgroups, and then subgroups within subgroups, etc., in your data. In the realm of more general unsupervised learning, it covers *principal components analysis (PCA)*, which is a very popular technique for reducing the number of predictors in a data set.

10

Statistical inference

10.1 Overview

A typical data analysis task in practice is to draw conclusions about some unknown aspect of a population of interest based on observed data sampled from that population; we typically do not get data on the *entire* population. Data analysis questions regarding how summaries, patterns, trends, or relationships in a data set extend to the wider population are called *inferential questions*. This chapter will start with the fundamental ideas of sampling from populations and then introduce two common techniques in statistical inference: *point estimation* and *interval estimation*.

10.2 Chapter learning objectives

By the end of the chapter, readers will be able to do the following:

- Describe real-world examples of questions that can be answered with statistical inference.
- Define common population parameters (e.g., mean, proportion, standard deviation) that are often estimated using sampled data, and estimate these from a sample.
- Define the following statistical sampling terms: population, sample, population parameter, point estimate, and sampling distribution.
- Explain the difference between a population parameter and a sample point estimate.
- Use Python to draw random samples from a finite population.
- Use Python to create a sampling distribution from a finite population.
- Describe how sample size influences the sampling distribution.
- Define bootstrapping.

DOI: 10.1201/9781003438397-10

- Use Python to create a bootstrap distribution to approximate a sampling distribution.
- Contrast the bootstrap and sampling distributions.

10.3 Why do we need sampling?

We often need to understand how quantities we observe in a subset of data relate to the same quantities in the broader population. For example, suppose a retailer is considering selling iPhone accessories, and they want to estimate how big the market might be. Additionally, they want to strategize how they can market their products on North American college and university campuses. This retailer might formulate the following question:

What proportion of all undergraduate students in North America own an iPhone?

In the above question, we are interested in making a conclusion about *all* undergraduate students in North America; this is referred to as the **population**. In general, the population is the complete collection of individuals or cases we are interested in studying. Further, in the above question, we are interested in computing a quantity—the proportion of iPhone owners—based on the entire population. This proportion is referred to as a **population parameter**. In general, a population parameter is a numerical characteristic of the entire population. To compute this number in the example above, we would need to ask every single undergraduate in North America whether they own an iPhone. In practice, directly computing population parameters is often time-consuming and costly, and sometimes impossible.

A more practical approach would be to make measurements for a **sample**, i.e., a subset of individuals collected from the population. We can then compute a **sample estimate**—a numerical characteristic of the sample—that estimates the population parameter. For example, suppose we randomly selected ten undergraduate students across North America (the sample) and computed the proportion of those students who own an iPhone (the sample estimate). In that case, we might suspect that proportion is a reasonable estimate of the proportion of students who own an iPhone in the entire population. Fig. 10.1 illustrates this process. In general, the process of using a sample to make a conclusion about the broader population from which it is taken is referred to as **statistical inference**.

Note that proportions are not the *only* kind of population parameter we might be interested in. For example, suppose an undergraduate student studying at

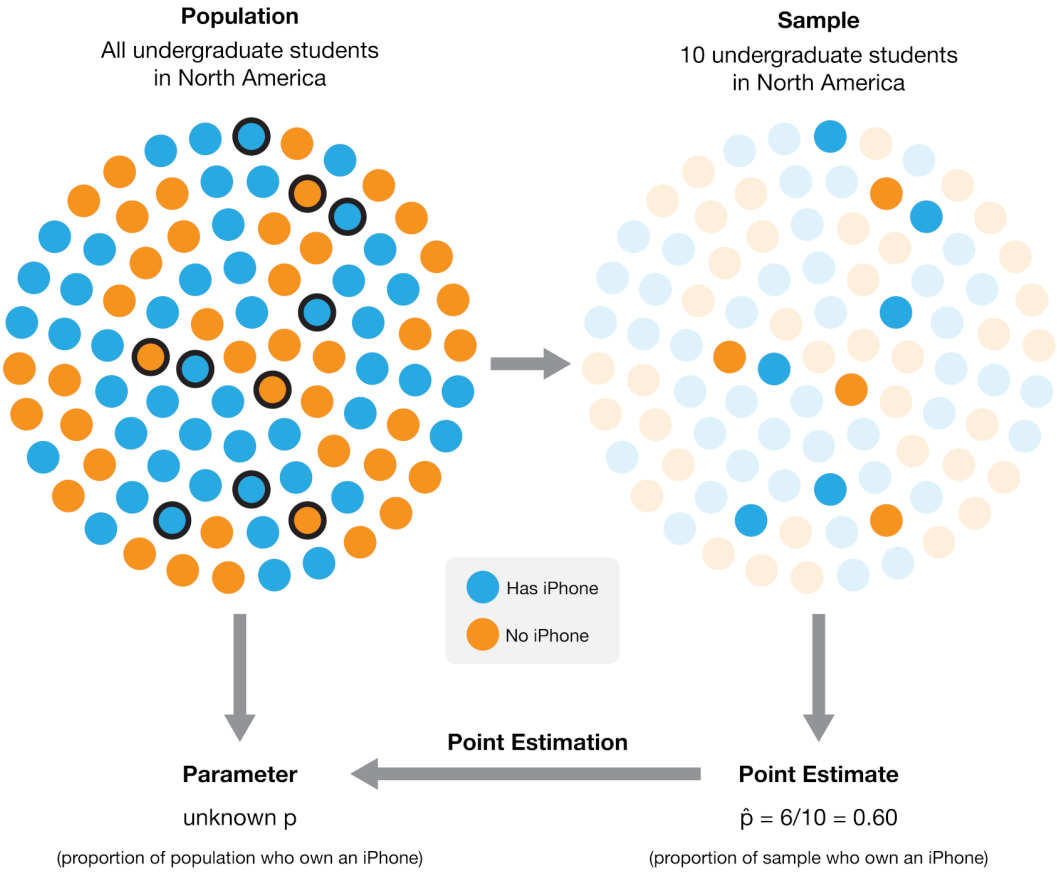


FIGURE 10.1 The process of using a sample from a broader population to obtain a point estimate of a population parameter. In this case, a sample of 10 individuals yielded 6 who own an iPhone, resulting in an estimated population proportion of 60% iPhone owners. The actual population proportion in this example illustration is 53.8%.

the University of British Columbia in Canada is looking for an apartment to rent. They need to create a budget, so they want to know something about studio apartment rental prices in Vancouver, BC. This student might formulate the following question:

What is the average price-per-month of studio apartment rentals in Vancouver, Canada?

In this case, the population consists of all studio apartment rentals in Vancouver, and the population parameter is the *average price-per-month*. Here we used the average as a measure of the center to describe the "typical value" of studio apartment rental prices. But even within this one example, we could also be interested in many other population parameters. For instance, we

know that not every studio apartment rental in Vancouver will have the same price per month. The student might be interested in how much monthly prices vary and want to find a measure of the rentals' spread (or variability), such as the standard deviation. Or perhaps the student might be interested in the fraction of studio apartment rentals that cost more than $1000 per month. The question we want to answer will help us determine the parameter we want to estimate. If we were somehow able to observe the whole population of studio apartment rental offerings in Vancouver, we could compute each of these numbers exactly; therefore, these are all population parameters. There are many kinds of observations and population parameters that you will run into in practice, but in this chapter, we will focus on two settings:

1. Using categorical observations to estimate the proportion of a category
2. Using quantitative observations to estimate the average (or mean)

10.4 Sampling distributions

10.4.1 Sampling distributions for proportions

We will look at an example using data from Inside Airbnb[1] [Cox, n.d.]. Airbnb is an online marketplace for arranging vacation rentals and places to stay. The data set contains listings for Vancouver, Canada, in September 2020. Our data includes an ID number, neighborhood, type of room, the number of people the rental accommodates, number of bathrooms, bedrooms, beds, and the price per night.

```
import pandas as pd

airbnb = pd.read_csv("data/listings.csv")
airbnb
```

```
        id              neighbourhood        room_type  accommodates  \
0        1                   Downtown  Entire home/apt             5
1        2          Downtown Eastside  Entire home/apt             4
2        3                   West End  Entire home/apt             2
3        4  Kensington-Cedar Cottage  Entire home/apt             2
4        5  Kensington-Cedar Cottage  Entire home/apt             4
...    ...                        ...              ...           ...
4589  4590          Downtown Eastside  Entire home/apt             5
4590  4591                   Oakridge     Private room             2
4591  4592          Dunbar Southlands     Private room             2
4592  4593                   West End  Entire home/apt             4
```

(continues on next page)

[1] http://insideairbnb.com/

(continued from previous page)

```
4593  4594               Shaughnessy  Entire home/apt             6

              bathrooms  bedrooms  beds   price
0               2 baths         2     2  150.00
1               2 baths         2     2  132.00
2                1 bath         1     1   85.00
3                1 bath         1     0  146.00
4                1 bath         1     2  110.00
...                 ...       ...   ...     ...
4589             1 bath         1     1   99.00
4590          1.5 baths         1     1   42.51
4591  1.5 shared baths         1     1   53.29
4592             1 bath         2     2  145.00
4593             1 bath         3     4  135.00

[4594 rows x 8 columns]
```

Suppose the city of Vancouver wants information about Airbnb rentals to help plan city bylaws, and they want to know how many Airbnb places are listed as entire homes and apartments (rather than as private or shared rooms). Therefore they may want to estimate the true proportion of all Airbnb listings where the room type is listed as "entire home or apartment". Of course, we usually do not have access to the true population, but here let's imagine (for learning purposes) that our data set represents the population of all Airbnb rental listings in Vancouver, Canada. We can find the proportion of listings for each room type by using the `value_counts` function with the `normalize` parameter as we did in previous chapters.

```
airbnb["room_type"].value_counts(normalize=True)
```

```
room_type
Entire home/apt    0.747497
Private room       0.246408
Shared room        0.005224
Hotel room         0.000871
Name: proportion, dtype: float64
```

We can see that the proportion of `Entire home/apt` listings in the data set is 0.747. This value, 0.747, is the population parameter. Remember, this parameter value is usually unknown in real data analysis problems, as it is typically not possible to make measurements for an entire population.

Instead, perhaps we can approximate it with a small subset of data. To investigate this idea, let's try randomly selecting 40 listings (i.e., taking a random sample of size 40 from our population), and computing the proportion for that sample. We will use the `sample` method of the `DataFrame` object to take the sample. The argument `n` of `sample` is the size of the sample to take and since we are starting to use randomness here, we are also setting the random seed via numpy to make the results reproducible.

```
import numpy as np


np.random.seed(155)

airbnb.sample(n=40)["room_type"].value_counts(normalize=True)
```

```
room_type
Entire home/apt    0.725
Private room       0.250
Shared room        0.025
Name: proportion, dtype: float64
```

Here we see that the proportion of entire home/apartment listings in this random sample is 0.725. Wow—that's close to our true population value! But remember, we computed the proportion using a random sample of size 40. This has two consequences. First, this value is only an *estimate*, i.e., our best guess of our population parameter using this sample. Given that we are estimating a single value here, we often refer to it as a **point estimate**. Second, since the sample was random, if we were to take *another* random sample of size 40 and compute the proportion for that sample, we would not get the same answer:

```
airbnb.sample(n=40)["room_type"].value_counts(normalize=True)
```

```
room_type
Entire home/apt    0.625
Private room       0.350
Shared room        0.025
Name: proportion, dtype: float64
```

Confirmed! We get a different value for our estimate this time. That means that our point estimate might be unreliable. Indeed, estimates vary from sample to sample due to **sampling variability**. But just how much should we expect the estimates of our random samples to vary? Or in other words, how much can we really trust our point estimate based on a single sample?

To understand this, we will simulate many samples (much more than just two) of size 40 from our population of listings and calculate the proportion of entire home/apartment listings in each sample. This simulation will create many sample proportions, which we can visualize using a histogram. The distribution of the estimate for all possible samples of a given size (which we commonly refer to as n) from a population is called a **sampling distribution**. The sampling distribution will help us see how much we would expect our sample proportions from this population to vary for samples of size 40.

We again use the `sample` to take samples of size 40 from our population of Airbnb listings. But this time we use a list comprehension to repeat the

operation multiple times (as we did previously in Chapter 9). In this case we repeat the operation 20,000 times to obtain 20,000 samples of size 40. To make it clear which rows in the data frame come which of the 20,000 samples, we also add a column called `replicate` with this information using the `assign` function, introduced previously in Chapter 3. The call to `concat` concatenates all the 20,000 data frames returned from the list comprehension into a single big data frame.

```
samples = pd.concat([
    airbnb.sample(40).assign(replicate=n)
    for n in range(20_000)
])
samples
```

```
        id     neighbourhood        room_type  accommodates          bathrooms␣
↪ \
605    606           Marpole  Entire home/apt             3              1 bath
4579  4580          Downtown  Entire home/apt             3              1 bath
1739  1740          West End  Entire home/apt             2              1 bath
3904  3905          Oakridge     Private room             6    1 private bath
1596  1597         Kitsilano  Entire home/apt             1              1 bath
...    ...               ...              ...           ...               ...
3060  3061  Hastings-Sunrise     Private room             2  3.5 shared baths
527    528         Kitsilano     Private room             4    1 private bath
1587  1588          Downtown  Entire home/apt             6              1 bath
3860  3861          Downtown  Entire home/apt             3              1 bath
2747  2748          Downtown  Entire home/apt             3              1 bath

      bedrooms  beds  price  replicate
605          1     1   91.0          0
4579         1     2  160.0          0
1739         1     2  151.0          0
3904         3     3  185.0          0
1596         1     1   99.0          0
...        ...   ...    ...        ...
3060         1     1   78.0      19999
527          1     1   99.0      19999
1587         1     3  169.0      19999
3860         1     1  100.0      19999
2747         1     1  285.0      19999

[800000 rows x 9 columns]
```

Since the column `replicate` indicates the replicate/sample number, we can verify that we indeed seem to have 20,0000 samples starting at sample 0 and ending at sample 19,999.

Now that we have obtained the samples, we need to compute the proportion of entire home/apartment listings in each sample. We first group the data by the `replicate` variable—to group the set of listings in each sample together—and then use `value_counts` with `normalize=True` to compute the proportion in each sample. Both the first and last few entries of the resulting data frame are printed below to show that we end up with 20,000 point estimates, one for each of the 20,000 samples.

```
(
    samples
    .groupby("replicate")
    ["room_type"]
    .value_counts(normalize=True)
)
```

```
replicate  room_type
0          Entire home/apt    0.750
           Private room       0.250
1          Entire home/apt    0.775
           Private room       0.225
2          Entire home/apt    0.750
                              ...
19998      Entire home/apt    0.700
           Private room       0.275
           Shared room        0.025
19999      Entire home/apt    0.750
           Private room       0.250
Name: proportion, Length: 44552, dtype: float64
```

The returned object is a series, and as we have previously learned we can use `reset_index` to change it to a data frame. However, there is one caveat here: when we use the `value_counts` function on a grouped series and try to `reset_index` we will end up with two columns with the same name and therefore get an error (in this case, `room_type` will occur twice). Fortunately, there is a simple solution: when we call `reset_index`, we can specify the name of the new column with the `name` parameter:

```
(
    samples
    .groupby("replicate")
    ["room_type"]
    .value_counts(normalize=True)
    .reset_index(name="sample_proportion")
)
```

```
       replicate        room_type  sample_proportion
0              0  Entire home/apt              0.750
1              0     Private room              0.250
2              1  Entire home/apt              0.775
3              1     Private room              0.225
4              2  Entire home/apt              0.750
...          ...              ...                ...
44547      19998  Entire home/apt              0.700
44548      19998     Private room              0.275
44549      19998      Shared room              0.025
44550      19999  Entire home/apt              0.750
44551      19999     Private room              0.250

[44552 rows x 3 columns]
```

Below we put everything together and also filter the data frame to keep only the room types that we are interested in.

```
sample_estimates = (
    samples
    .groupby("replicate")
    ["room_type"]
    .value_counts(normalize=True)
    .reset_index(name="sample_proportion")
)

sample_estimates = sample_estimates[sample_estimates["room_type"] == "Entire
↪home/apt"]
sample_estimates
```

```
       replicate        room_type  sample_proportion
0              0  Entire home/apt              0.750
2              1  Entire home/apt              0.775
4              2  Entire home/apt              0.750
6              3  Entire home/apt              0.675
8              4  Entire home/apt              0.725
...          ...              ...                ...
44541      19995  Entire home/apt              0.775
44543      19996  Entire home/apt              0.750
44545      19997  Entire home/apt              0.725
44547      19998  Entire home/apt              0.700
44550      19999  Entire home/apt              0.750

[20000 rows x 3 columns]
```

We can now visualize the sampling distribution of sample proportions for samples of size 40 using a histogram in Fig. 10.2. Keep in mind: in the real world, we don't have access to the full population. So we can't take many samples and can't actually construct or visualize the sampling distribution. We have created this particular example such that we *do* have access to the full population, which lets us visualize the sampling distribution directly for learning purposes.

```
sampling_distribution = alt.Chart(sample_estimates).mark_bar().encode(
    x=alt.X("sample_proportion")
        .bin(maxbins=20)
        .title("Sample proportions"),
    y=alt.Y("count()").title("Count"),
)

sampling_distribution
```

The sampling distribution in Fig. 10.2 appears to be bell-shaped, is roughly symmetric, and has one peak. It is centered around 0.75 and the sample proportions range from about 0.55 to about 0.95. In fact, we can calculate the mean of the sample proportions.

```
sample_estimates["sample_proportion"].mean()
```

```
0.74848375
```

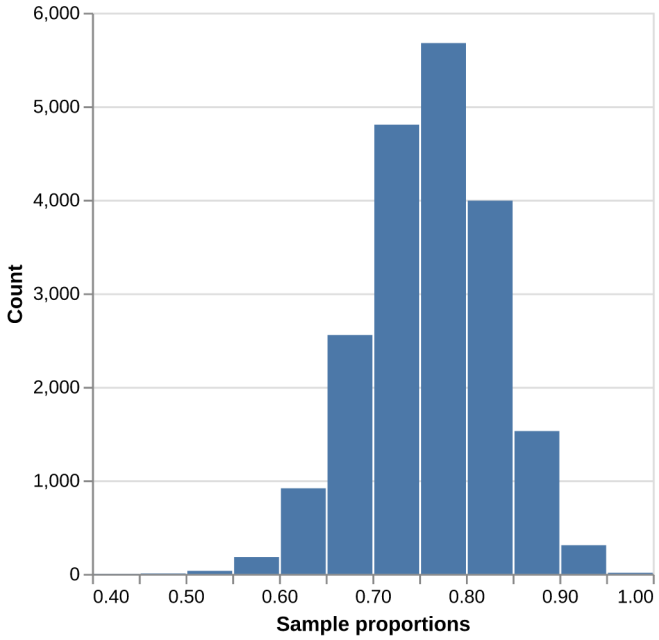


FIGURE 10.2 Sampling distribution of the sample proportion for sample size 40.

We notice that the sample proportions are centered around the population proportion value, 0.748. In general, the mean of the sampling distribution should be equal to the population proportion. This is great news because it means that the sample proportion is neither an overestimate nor an underestimate of the population proportion. In other words, if you were to take many samples as we did above, there is no tendency toward over or underestimating the population proportion. In a real data analysis setting where you just have access to your single sample, this implies that you would suspect that your sample point estimate is roughly equally likely to be above or below the true population proportion.

10.4.2 Sampling distributions for means

In the previous section, our variable of interest—`room_type`—was *categorical*, and the population parameter was a proportion. As mentioned in the chapter introduction, there are many choices of the population parameter for each type of variable. What if we wanted to infer something about a population of *quantitative* variables instead? For instance, a traveler visiting Vancouver, Canada may wish to estimate the population *mean* (or average) price per night of Airbnb listings. Knowing the average could help them tell whether a

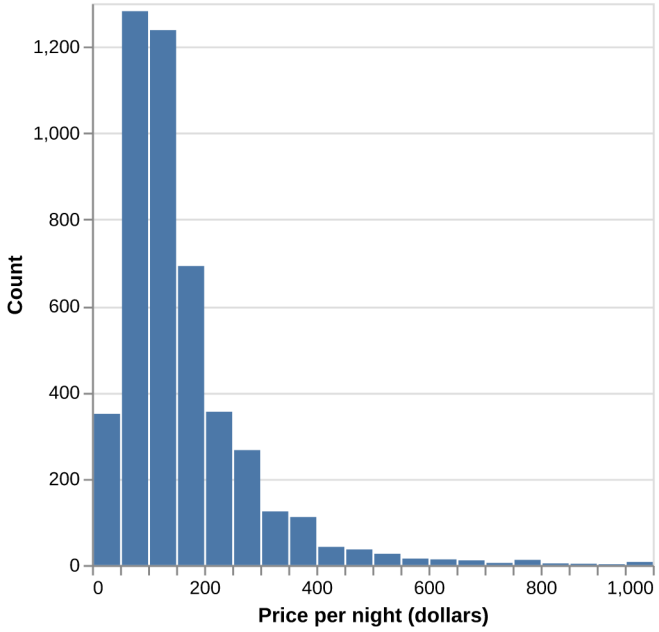


FIGURE 10.3 Population distribution of price per night (dollars) for all Airbnb listings in Vancouver, Canada.

particular listing is overpriced. We can visualize the population distribution of the price per night with a histogram.

```
population_distribution = alt.Chart(airbnb).mark_bar().encode(
    x=alt.X("price")
        .bin(maxbins=30)
        .title("Price per night (dollars)"),
    y=alt.Y("count()", title="Count"),
)

population_distribution
```

In Fig. 10.3, we see that the population distribution has one peak. It is also skewed (i.e., is not symmetric): most of the listings are less than $250 per night, but a small number of listings cost much more, creating a long tail on the histogram's right side. Along with visualizing the population, we can calculate the population mean, the average price per night for all the Airbnb listings.

```
airbnb["price"].mean()
```

```
154.5109773617762
```

The price per night of all Airbnb rentals in Vancouver, BC is $154.51, on average. This value is our population parameter since we are calculating it using the population data.

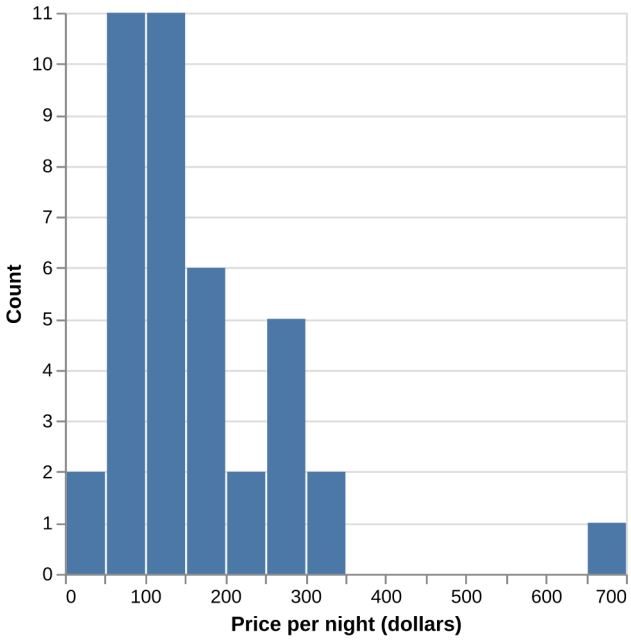


FIGURE 10.4 Distribution of price per night (dollars) for sample of 40 Airbnb listings.

Now suppose we did not have access to the population data (which is usually the case!), yet we wanted to estimate the mean price per night. We could answer this question by taking a random sample of as many Airbnb listings as our time and resources allow. Let's say we could do this for 40 listings. What would such a sample look like? Let's take advantage of the fact that we do have access to the population data and simulate taking one random sample of 40 listings in Python, again using `sample`.

```
one_sample = airbnb.sample(n=40)
```

We can create a histogram to visualize the distribution of observations in the sample (Fig. 10.4), and calculate the mean of our sample.

```
sample_distribution = alt.Chart(one_sample).mark_bar().encode(
    x=alt.X("price")
        .bin(maxbins=30)
        .title("Price per night (dollars)"),
    y=alt.Y("count()").title("Count"),
)

sample_distribution
```

```
one_sample["price"].mean()
```

```
153.48225
```

The average value of the sample of size 40 is $153.48. This number is a point estimate for the mean of the full population. Recall that the population mean was $154.51. So our estimate was fairly close to the population parameter: the mean was about 0.7% off. Note that we usually cannot compute the estimate's accuracy in practice since we do not have access to the population parameter; if we did, we wouldn't need to estimate it.

Also, recall from the previous section that the point estimate can vary; if we took another random sample from the population, our estimate's value might change. So then, did we just get lucky with our point estimate above? How much does our estimate vary across different samples of size 40 in this example? Again, since we have access to the population, we can take many samples and plot the sampling distribution of sample means to get a sense for this variation. In this case, we'll use the 20,000 samples of size 40 that we already stored in the `samples` variable. First, we will calculate the sample mean for each replicate and then plot the sampling distribution of sample means for samples of size 40.

```
sample_estimates = (
    samples
    .groupby("replicate")
    ["price"]
    .mean()
    .reset_index()
    .rename(columns={"price": "mean_price"})
)
sample_estimates
```

```
       replicate  mean_price
0              0   187.00000
1              1   148.56075
2              2   165.50500
3              3   140.93925
4              4   139.14650
...          ...         ...
19995      19995   198.50000
19996      19996   192.66425
19997      19997   144.88600
19998      19998   146.08800
19999      19999   156.25000

[20000 rows x 2 columns]
```

```
sampling_distribution = alt.Chart(sample_estimates).mark_bar().encode(
    x=alt.X("mean_price")
        .bin(maxbins=30)
        .title("Sample mean price per night (dollars)"),
    y=alt.Y("count()").title("Count")
)

sampling_distribution
```

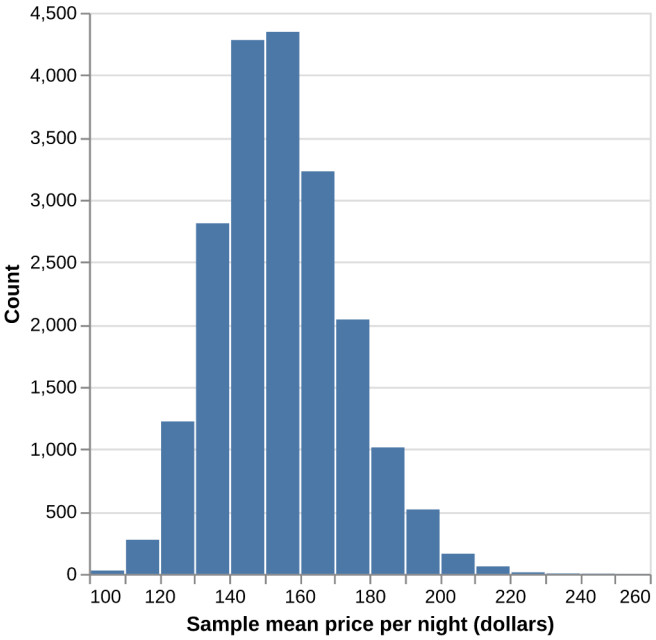


FIGURE 10.5 Sampling distribution of the sample means for sample size of 40.

In Fig. 10.5, the sampling distribution of the mean has one peak and is bell-shaped. Most of the estimates are between about \$140 and \$170; but there is a good fraction of cases outside this range (i.e., where the point estimate was not close to the population parameter). So it does indeed look like we were quite lucky when we estimated the population mean with only 0.7% error.

Let's visualize the population distribution, distribution of the sample, and the sampling distribution on one plot to compare them in Fig. 10.6. Comparing these three distributions, the centers of the distributions are all around the same price (around \$150). The original population distribution has a long right tail, and the sample distribution has a similar shape to that of the population distribution. However, the sampling distribution is not shaped like the population or sample distribution. Instead, it has a bell shape, and it has a lower spread than the population or sample distributions. The sample means vary less than the individual observations because there will be some high values and some small values in any random sample, which will keep the average from being too extreme.

Given that there is quite a bit of variation in the sampling distribution of the sample mean—i.e., the point estimate that we obtain is not very reliable—is there any way to improve the estimate? One way to improve a point estimate is to take a *larger* sample. To illustrate what effect this has, we will take

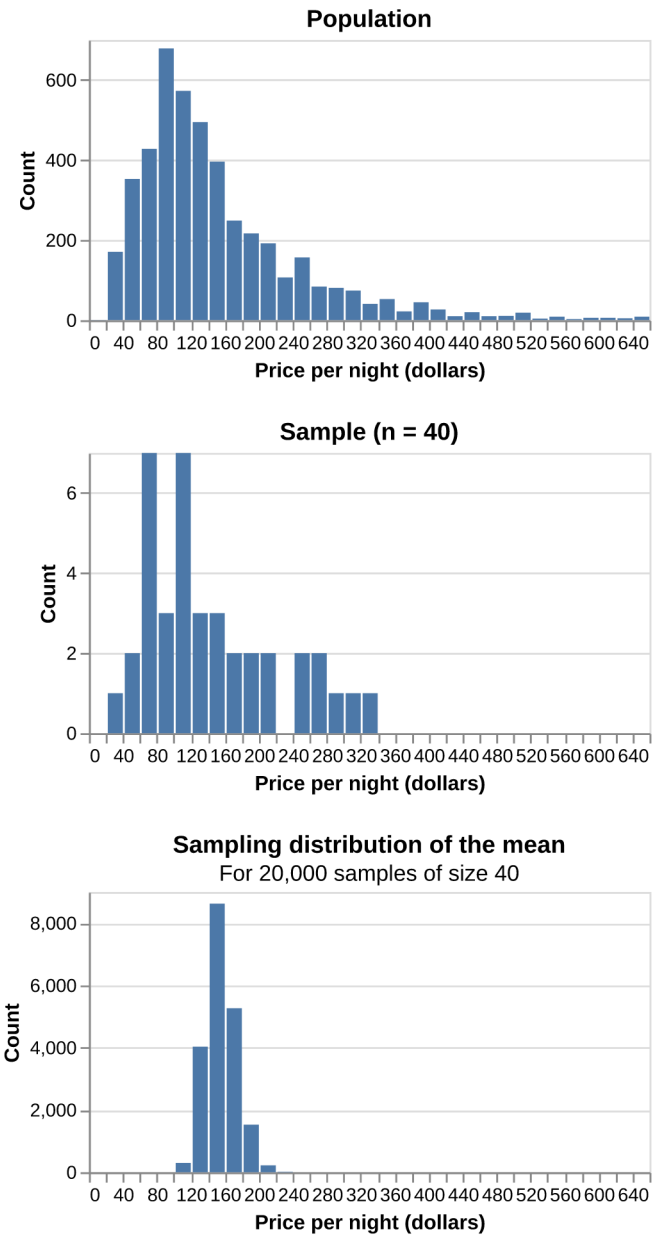


FIGURE 10.6 Comparison of population distribution, sample distribution, and sampling distribution.

many samples of size 20, 50, 100, and 500, and plot the sampling distribution of the sample mean. We indicate the mean of the sampling distribution with a vertical line.

Based on the visualization in Fig. 10.7, three points about the sample mean become clear:

1. The mean of the sample mean (across samples) is equal to the population mean. In other words, the sampling distribution is centered at the population mean.
2. Increasing the size of the sample decreases the spread (i.e., the variability) of the sampling distribution. Therefore, a larger sample size results in a more reliable point estimate of the population parameter.
3. The distribution of the sample mean is roughly bell-shaped.

Note: You might notice that in the `n = 20` case in Fig. 10.7, the distribution is not *quite* bell-shaped. There is a bit of skew toward the right. You might also notice that in the `n = 50` case and larger, that skew seems to disappear. In general, the sampling distribution—for both means and proportions—only becomes bell-shaped *once the sample size is large enough*. How large is "large enough?" Unfortunately, it depends entirely on the problem at hand. But as a rule of thumb, often a sample size of at least 20 will suffice.

10.4.3 Summary

1. A point estimate is a single value computed using a sample from a population (e.g., a mean or proportion).
2. The sampling distribution of an estimate is the distribution of the estimate for all possible samples of a fixed size from the same population.
3. The shape of the sampling distribution is usually bell-shaped with one peak and centered at the population mean or proportion.
4. The spread of the sampling distribution is related to the sample size. As the sample size increases, the spread of the sampling distribution decreases.

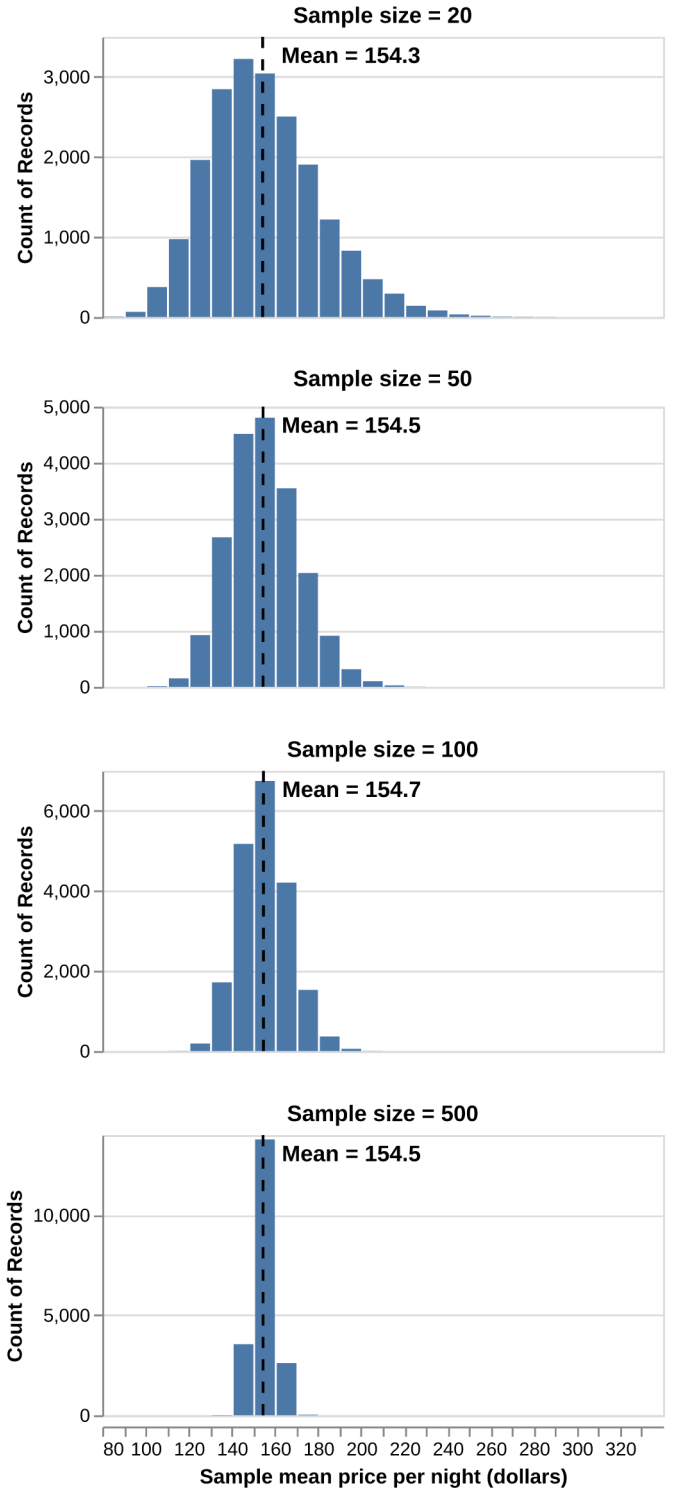


FIGURE 10.7 Comparison of sampling distributions, with mean highlighted as a vertical line.

10.5 Bootstrapping

10.5.1 Overview

Why all this emphasis on sampling distributions?

We saw in the previous section that we could compute a **point estimate** of a population parameter using a sample of observations from the population. And since we constructed examples where we had access to the population, we could evaluate how accurate the estimate was, and even get a sense of how much the estimate would vary for different samples from the population. But in real data analysis settings, we usually have *just one sample* from our population and do not have access to the population itself. Therefore we cannot construct the sampling distribution as we did in the previous section. And as we saw, our sample estimate's value can vary significantly from the population parameter. So reporting the point estimate from a single sample alone may not be enough. We also need to report some notion of *uncertainty* in the value of the point estimate.

Unfortunately, we cannot construct the exact sampling distribution without full access to the population. However, if we could somehow *approximate* what the sampling distribution would look like for a sample, we could use that approximation to then report how uncertain our sample point estimate is (as we did above with the *exact* sampling distribution). There are several methods to accomplish this; in this book, we will use the *bootstrap*. We will discuss **interval estimation** and construct **confidence intervals** using just a single sample from a population. A confidence interval is a range of plausible values for our population parameter.

Here is the key idea. First, if you take a big enough sample, it *looks like* the population. Notice the histograms' shapes for samples of different sizes taken from the population in Fig. 10.8. We see that the sample's distribution looks like that of the population for a large enough sample.

In the previous section, we took many samples of the same size *from our population* to get a sense of the variability of a sample estimate. But if our sample is big enough that it looks like our population, we can pretend that our sample *is* the population, and take more samples (with replacement) of the same size from it instead. This very clever technique is called **the bootstrap**. Note that by taking many samples from our single, observed sample, we do not obtain the true sampling distribution, but rather an approximation that we call **the bootstrap distribution**.

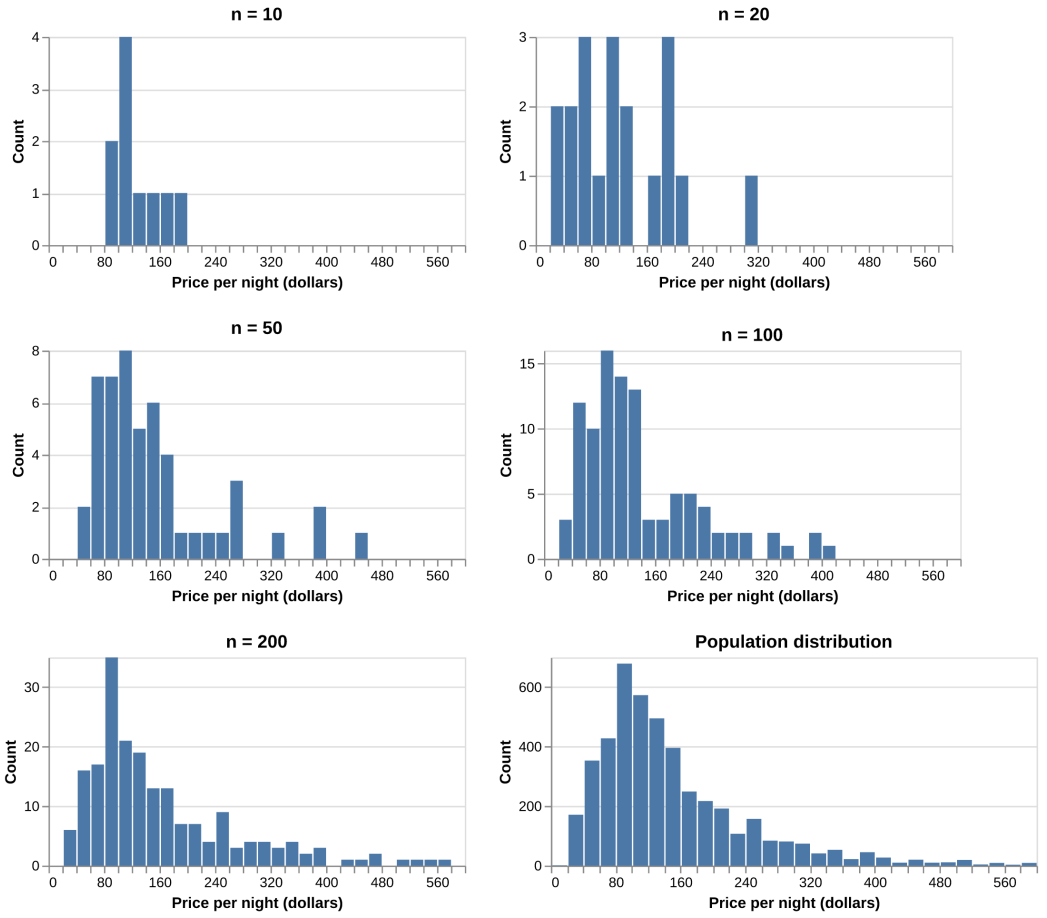


FIGURE 10.8 Comparison of samples of different sizes from the population.

Note: We must sample *with* replacement when using the bootstrap. Otherwise, if we had a sample of size n, and obtained a sample from it of size n *without* replacement, it would just return our original sample.

This section will explore how to create a bootstrap distribution from a single sample using Python. The process is visualized in Fig. 10.9. For a sample of size n, you would do the following:

1. Randomly select an observation from the original sample, which was drawn from the population.
2. Record the observation's value.
3. Replace that observation.

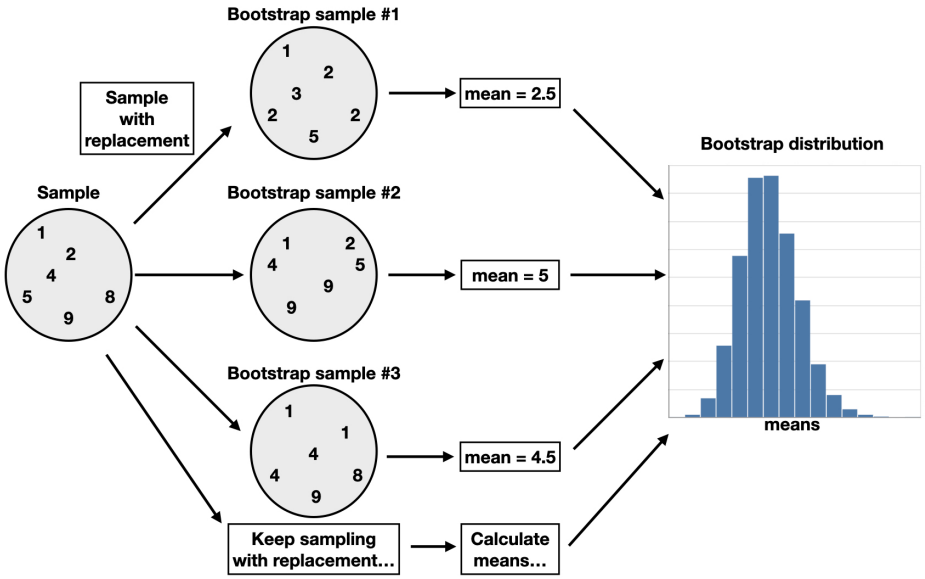


FIGURE 10.9 Overview of the bootstrap process.

4. Repeat steps 1–3 (sampling *with* replacement) until you have n observations, which form a bootstrap sample.
5. Calculate the bootstrap point estimate (e.g., mean, median, proportion, slope, etc.) of the n observations in your bootstrap sample.
6. Repeat steps 1–5 many times to create a distribution of point estimates (the bootstrap distribution).
7. Calculate the plausible range of values around our observed point estimate.

10.5.2 Bootstrapping in Python

Let's continue working with our Airbnb example to illustrate how we might create and use a bootstrap distribution using just a single sample from the population. Once again, suppose we are interested in estimating the population mean price per night of all Airbnb listings in Vancouver, Canada, using a single sample size of 40. Recall our point estimate was \$153.48. The histogram of prices in the sample is displayed in Fig. 10.10.

```
one_sample
```

```
        id              neighbourhood        room_type  accommodates  \
4025  4026       Renfrew-Collingwood     Private room             1
1977  1978                  Fairview     Private room             1
4008  4009                  Downtown  Entire home/apt             4
1543  1544  Kensington-Cedar Cottage  Entire home/apt             6
```

(continues on next page)

(continued from previous page)

```
3350  3351                  Downtown  Entire home/apt             2
804    805            Mount Pleasant     Private room             2
2286  2287                   Marpole  Entire home/apt             4
1010  1011                Strathcona  Entire home/apt             2
1878  1879                  Fairview     Private room             2
1644  1645                  Downtown  Entire home/apt             4
4579  4580                  Downtown  Entire home/apt             3
2771  2772         Dunbar Southlands  Entire home/apt             5
4151  4152                 Kitsilano  Entire home/apt             2
4495  4496               Riley Park  Entire home/apt             2
1308  1309               Riley Park  Entire home/apt             2
2246  2247            Mount Pleasant  Entire home/apt             4
2335  2336  Kensington-Cedar Cottage  Entire home/apt             3
4059  4060                  Downtown  Entire home/apt             5
1280  1281          Hastings-Sunrise  Entire home/apt             6
4324  4325       Renfrew-Collingwood     Private room             1
3403  3404             Arbutus Ridge  Entire home/apt            15
1729  1730       Renfrew-Collingwood  Entire home/apt             6
3722  3723                  Downtown  Entire home/apt             4
241    242          Hastings-Sunrise     Private room             2
3955  3956         Dunbar Southlands     Private room             2
1042  1043                 Kitsilano     Private room             1
649    650                    Sunset  Entire home/apt             5
1995  1996               Riley Park     Private room             2
363    364  Kensington-Cedar Cottage  Entire home/apt             3
1783  1784         Downtown Eastside     Private room             2
805    806            Mount Pleasant  Entire home/apt             4
254    255                  Downtown  Entire home/apt             4
3365  3366                    Sunset     Private room             1
4562  4563                  Downtown     Private room             1
2124  2125                  Downtown  Entire home/apt             4
18      19                  Downtown  Entire home/apt             4
1997  1998                  Downtown  Entire home/apt             6
4329  4330                  West End  Entire home/apt             2
3408  3409                  Downtown  Entire home/apt             2
635    636        Grandview-Woodland  Entire home/apt             6

             bathrooms  bedrooms  beds   price
4025    1 shared bath         1     1   40.00
1977          2 baths         3     1   70.00
4008           1 bath         1     1  269.00
1543          2 baths         3     5  320.00
3350           1 bath         1     1  140.00
804    1 private bath         1     1   77.00
2286           1 bath         2     2  105.00
1010           1 bath         1     1  120.00
1878   1 private bath         1     1  175.00
1644          2 baths         2     1  150.00
4579           1 bath         1     2  160.00
2771           1 bath         2     2  130.00
4151           1 bath         1     1  289.00
4495           1 bath         1     1  115.00
1308           1 bath         1     1  105.00
2246           1 bath         1     2  105.00
2335           1 bath         2     2   85.00
4059          2 baths         2     3  250.00
1280           1 bath         2     3  100.00
4324    1 shared bath         1     0   25.00
3403          4 baths         6     8  664.00
1729           1 bath         2     3   93.00
3722           1 bath         3     6  275.00
```

(continues on next page)

(continued from previous page)

```
241      1 shared bath         1     1   50.00
3955  1.5 shared baths         1     1   60.00
1042     1 shared bath         1     1   66.00
649              1 bath        2     2  196.00
1995     1 shared bath         1     1   70.00
363              1 bath        1     2  120.00
1783    1 private bath         1     1   60.00
805              1 bath        2     2  150.00
254           1.5 baths        2     2  300.00
3365    1 private bath         1     1  100.00
4562     1 shared bath         1     1   64.29
2124             1 bath        2     3  200.00
18              2 baths        2     2  200.00
1997            2 baths        3     3  257.00
4329             1 bath        1     0   92.00
3408            2 baths        2     2  189.00
635             2 baths        1     2  103.00
```

```
one_sample_dist = alt.Chart(one_sample).mark_bar().encode(
    x=alt.X("price")
        .bin(maxbins=30)
        .title("Price per night (dollars)"),
    y=alt.Y("count()").title("Count"),
)

one_sample_dist
```

The histogram for the sample is skewed, with a few observations out to the right. The mean of the sample is \$153.48. Remember, in practice, we usually

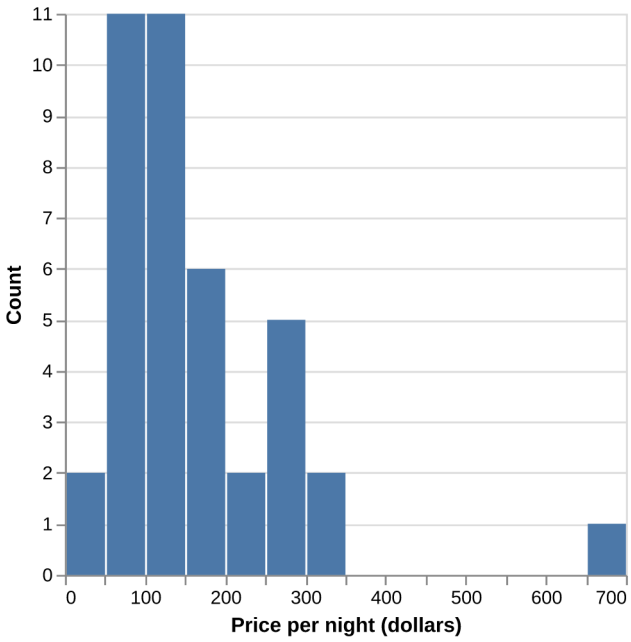


FIGURE 10.10 Histogram of price per night (dollars) for one sample of size 40.

only have this one sample from the population. So this sample and estimate are the only data we can work with.

We now perform steps 1–5 listed above to generate a single bootstrap sample in Python and calculate a point estimate from that bootstrap sample. We will continue using the `sample` function of our data frame. Critically, note that we now set `frac=1` ("fraction") to indicate that we want to draw as many samples as there are rows in the data frame (we could also have set `n=40` but then we would need to manually keep track of how many rows there are). Since we need to sample with replacement when bootstrapping, we change the `replace` parameter to `True`.

```
boot1 = one_sample.sample(frac=1, replace=True)
boot1_dist = alt.Chart(boot1).mark_bar().encode(
    x=alt.X("price")
        .bin(maxbins=30)
        .title("Price per night (dollars)"),
    y=alt.Y("count()", title="Count"),
)

boot1_dist
```

```
boot1["price"].mean()
```

```
132.65
```

Notice in Fig. 10.11 that the histogram of our bootstrap sample has a similar shape to the original sample histogram. Though the shapes of the distribu-

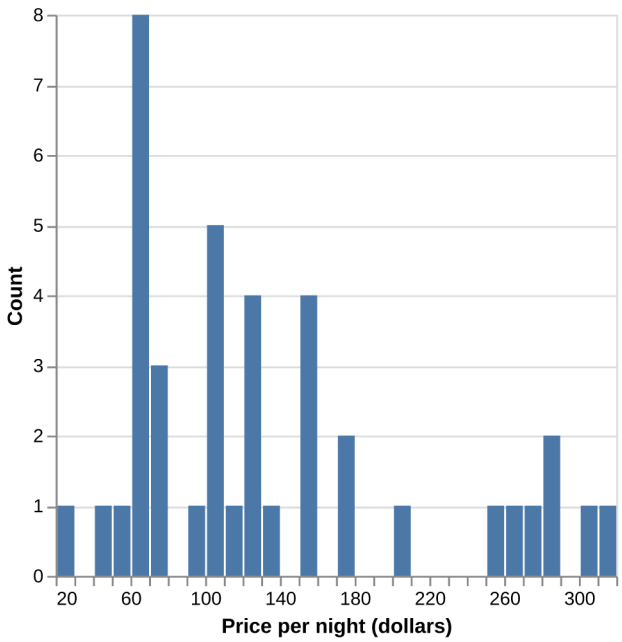


FIGURE 10.11 Bootstrap distribution.

tions are similar, they are not identical. You'll also notice that the original sample mean and the bootstrap sample mean differ. How might that happen? Remember that we are sampling with replacement from the original sample, so we don't end up with the same sample values again. We are *pretending* that our single sample is close to the population, and we are trying to mimic drawing another sample from the population by drawing one from our original sample.

Let's now take 20,000 bootstrap samples from the original sample (`one_sample`) and calculate the means for each of those replicates. Recall that this assumes that `one_sample` *looks like* our original population; but since we do not have access to the population itself, this is often the best we can do. Note that here we break the list comprehension over multiple lines so that it is easier to read.

```
boot20000 = pd.concat([
    one_sample.sample(frac=1, replace=True).assign(replicate=n)
    for n in range(20_000)
])
boot20000
```

```
        id      neighbourhood        room_type  accommodates       bathrooms ␣
↪\
2286  2287            Marpole  Entire home/apt             4          1 bath
254    255          Downtown  Entire home/apt             4       1.5 baths
2246  2247    Mount Pleasant  Entire home/apt             4          1 bath
4579  4580          Downtown  Entire home/apt             3          1 bath
4495  4496       Riley Park  Entire home/apt             2          1 bath
...    ...               ...              ...           ...            ...
1997  1998          Downtown  Entire home/apt             6         2 baths
241    242  Hastings-Sunrise     Private room             2   1 shared bath
1878  1879          Fairview     Private room             2  1 private bath
1783  1784  Downtown Eastside     Private room             2  1 private bath
4495  4496       Riley Park  Entire home/apt             2          1 bath

      bedrooms  beds  price  replicate
2286         2     2  105.0          0
254          2     2  300.0          0
2246         1     2  105.0          0
4579         1     2  160.0          0
4495         1     1  115.0          0
...        ...   ...    ...        ...
1997         3     3  257.0      19999
241          1     1   50.0      19999
1878         1     1  175.0      19999
1783         1     1   60.0      19999
4495         1     1  115.0      19999

[800000 rows x 9 columns]
```

Let's take a look at the histograms of the first six replicates of our bootstrap samples.

```
six_bootstrap_samples = boot20000.query("replicate < 6")
six_bootstrap_fig = alt.Chart(six_bootstrap_samples, height=150).mark_bar().
↪encode(
    x=alt.X("price")
        .bin(maxbins=20)
        .title("Price per night (dollars)"),
    y=alt.Y("count()").title("Count")
).facet(
    "replicate:N",  # Recall that `:N` converts the variable to a categorical␣
↪type
    columns=2
)
six_bootstrap_fig
```

We see in Fig. 10.12 how the distributions of the bootstrap samples differ. If we calculate the sample mean for each of these six samples, we can see

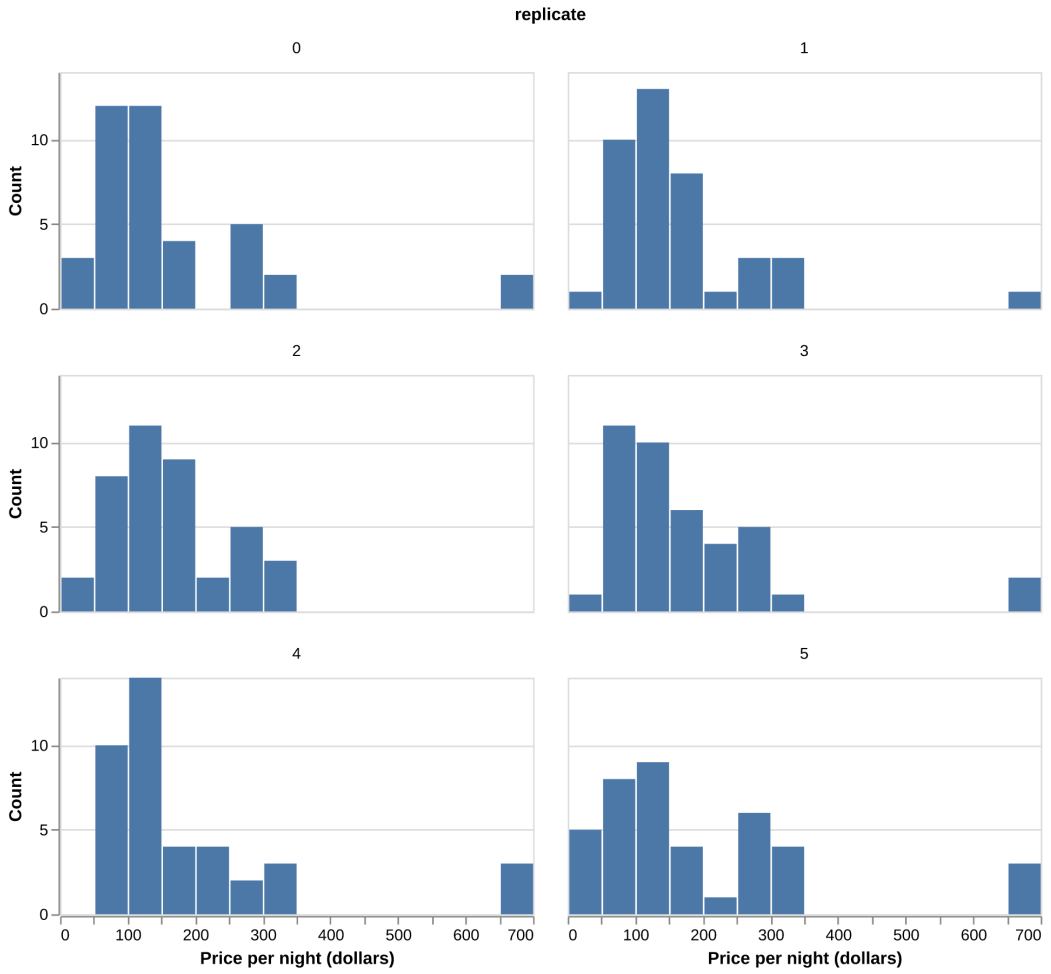


FIGURE 10.12 Histograms of the first six replicates of the bootstrap samples.

that these are also different between samples. To compute the mean for each sample, we first group by the "replicate" which is the column containing the sample/replicate number. Then we compute the mean of the `price` column and rename it to `mean_price` for it to be more descriptive. Finally, we use `reset_index` to get the `replicate` values back as a column in the data frame.

```
(
    six_bootstrap_samples
    .groupby("replicate")
    ["price"]
    .mean()
    .reset_index()
    .rename(columns={"price": "mean_price"})
)
```

```
   replicate  mean_price
0          0   155.67175
1          1   154.42500
2          2   149.35000
3          3   169.13225
4          4   179.79675
5          5   188.28225
```

The distributions and the means differ between the bootstrapped samples because we are sampling *with replacement*. If we instead would have sampled *without replacement*, we would end up with the exact same values in the sample each time.

We will now calculate point estimates of the mean for our 20,000 bootstrap samples and generate a bootstrap distribution of these point estimates. The bootstrap distribution (Fig. 10.13) suggests how we might expect our point estimate to behave if we take multiple samples.

```
boot20000_means = (
    boot20000
    .groupby("replicate")
    ["price"]
    .mean()
    .reset_index()
    .rename(columns={"price": "mean_price"})
)

boot20000_means
```

```
       replicate  mean_price
0              0   155.67175
1              1   154.42500
2              2   149.35000
3              3   169.13225
4              4   179.79675
...          ...         ...
19995      19995   159.29675
```

(continues on next page)

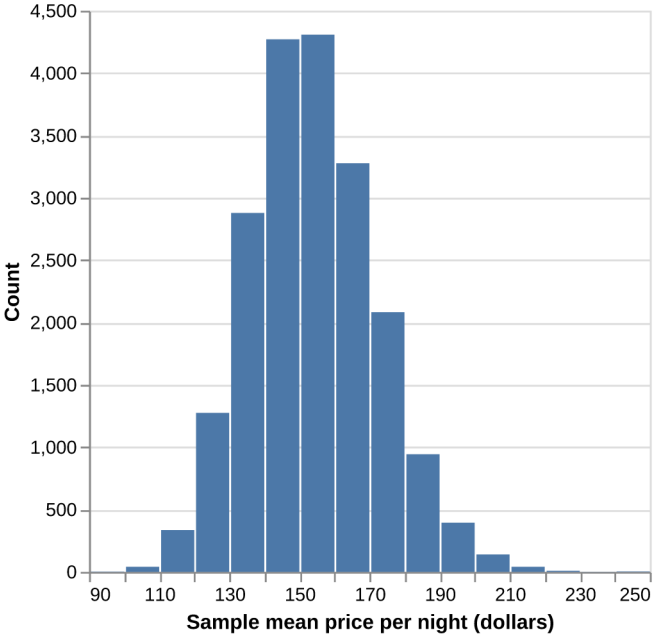


FIGURE 10.13 Distribution of the bootstrap sample means.

(continued from previous page)

```
19996      19996   137.20000
19997      19997   136.55725
19998      19998   161.93950
19999      19999   170.22500

[20000 rows x 2 columns]
```

```
boot_est_dist = alt.Chart(boot20000_means).mark_bar().encode(
    x=alt.X("mean_price")
        .bin(maxbins=20)
        .title("Sample mean price per night (dollars)"),
    y=alt.Y("count()").title("Count"),
)

boot_est_dist
```

Let's compare the bootstrap distribution—which we construct by taking many samples from our original sample of size 40—with the true sampling distribution—which corresponds to taking many samples from the population.

There are two essential points that we can take away from Fig. 10.14. First, the shape and spread of the true sampling distribution and the bootstrap distribution are similar; the bootstrap distribution lets us get a sense of the point estimate's variability. The second important point is that the means of these two distributions are slightly different. The sampling distribution is centered at $154.51, the population mean value. However, the bootstrap

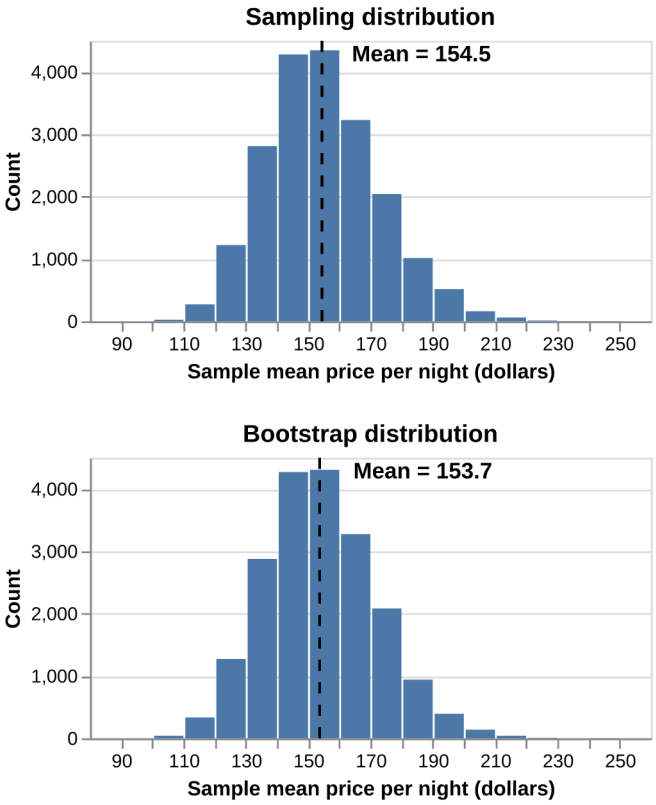


FIGURE 10.14 Comparison of the distribution of the bootstrap sample means and sampling distribution.

distribution is centered at the original sample's mean price per night, $153.48. Because we are resampling from the original sample repeatedly, we see that the bootstrap distribution is centered at the original sample's mean value (unlike the sampling distribution of the sample mean, which is centered at the population parameter value).

Fig. 10.15 summarizes the bootstrapping process. The idea here is that we can use this distribution of bootstrap sample means to approximate the sampling distribution of the sample means when we only have one sample. Since the bootstrap distribution pretty well approximates the sampling distribution spread, we can use the bootstrap spread to help us develop a plausible range for our population parameter along with our estimate.

10.5.3 Using the bootstrap to calculate a plausible range

Now that we have constructed our bootstrap distribution, let's use it to create an approximate 95% percentile bootstrap confidence interval. A **confidence interval** is a range of plausible values for the population parameter. We will

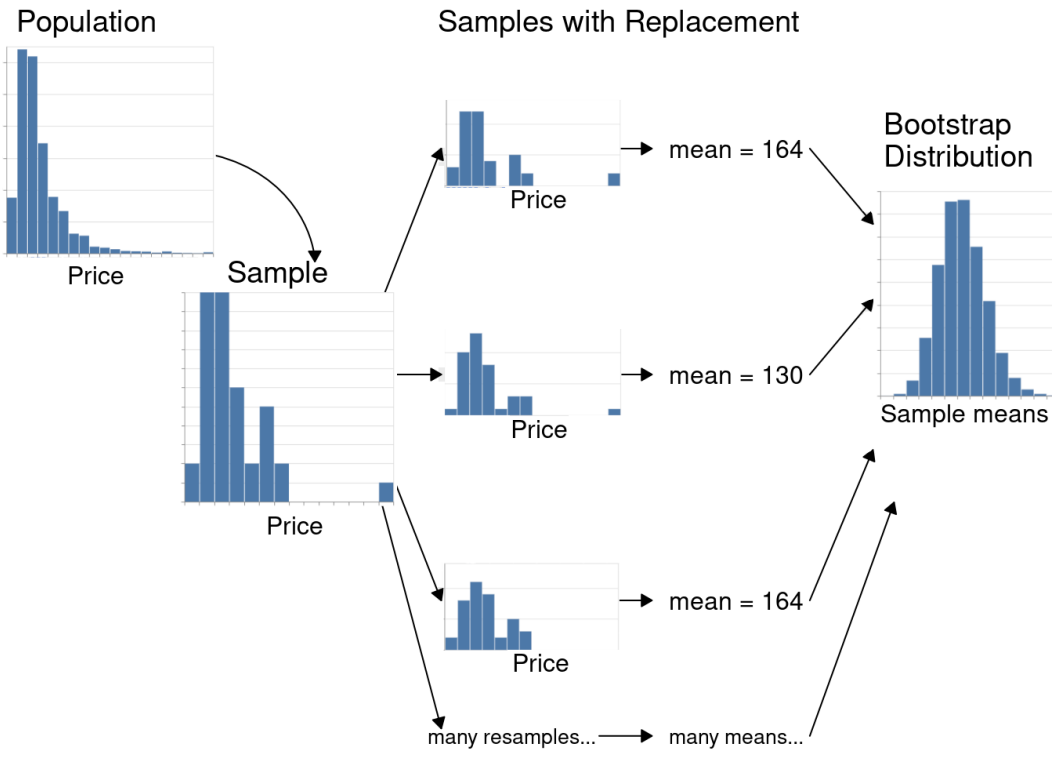


FIGURE 10.15 Summary of bootstrapping process.

find the range of values covering the middle 95% of the bootstrap distribution, giving us a 95% confidence interval. You may be wondering, what does "95% confidence" mean? If we took 100 random samples and calculated 100 95% confidence intervals, then about 95% of the ranges would capture the population parameter's value. Note there's nothing special about 95%. We could have used other levels, such as 90% or 99%. There is a balance between our level of confidence and precision. A higher confidence level corresponds to a wider range of the interval, and a lower confidence level corresponds to a narrower range. Therefore the level we choose is based on what chance we are willing to take of being wrong based on the implications of being wrong for our application. In general, we choose confidence levels to be comfortable with our level of uncertainty but not so strict that the interval is unhelpful. For instance, if our decision impacts human life and the implications of being wrong are deadly, we may want to be very confident and choose a higher confidence level.

To calculate a 95% percentile bootstrap confidence interval, we will do the following:

1. Arrange the observations in the bootstrap distribution in ascending order.
2. Find the value such that 2.5% of observations fall below it (the 2.5% percentile). Use that value as the lower bound of the interval.
3. Find the value such that 97.5% of observations fall below it (the 97.5% percentile). Use that value as the upper bound of the interval.

To do this in Python, we can use the `quantile` function of our DataFrame. Quantiles are expressed in proportions rather than percentages, so the 2.5th and 97.5th percentiles would be the 0.025 and 0.975 quantiles, respectively.

```
ci_bounds = boot20000_means["mean_price"].quantile([0.025, 0.975])
ci_bounds
```

```
0.025    121.607069
0.975    191.525362
Name: mean_price, dtype: float64
```

Our interval, $121.61 to $191.53, captures the middle 95% of the sample mean prices in the bootstrap distribution. We can visualize the interval on our distribution in Fig. 10.16.

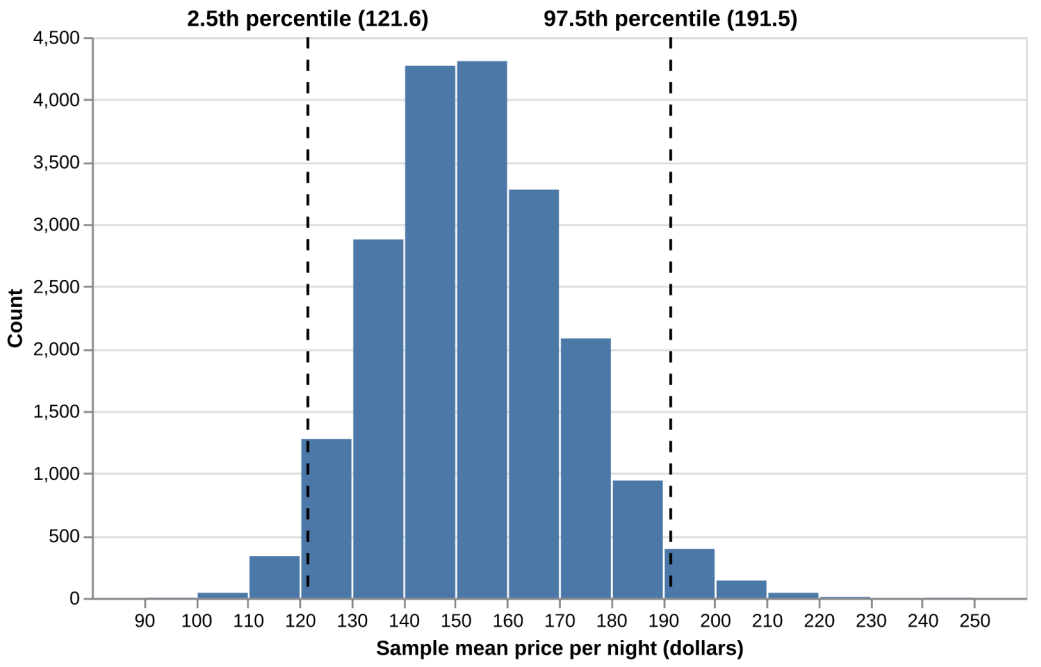


FIGURE 10.16 Distribution of the bootstrap sample means with percentile lower and upper bounds.

To finish our estimation of the population parameter, we would report the point estimate and our confidence interval's lower and upper bounds. Here the sample mean price-per-night of 40 Airbnb listings was $153.48, and we are 95% "confident" that the true population mean price-per-night for all Airbnb listings in Vancouver is between $121.61 and $191.53. Notice that our interval does indeed contain the true population mean value, $154.51. However, in practice, we would not know whether our interval captured the population parameter or not because we usually only have a single sample, not the entire population. This is the best we can do when we only have one sample.

This chapter is only the beginning of the journey into statistical inference. We can extend the concepts learned here to do much more than report point estimates and confidence intervals, such as testing for real differences between populations, tests for associations between variables, and so much more. We have just scratched the surface of statistical inference; however, the material presented here will serve as the foundation for more advanced statistical techniques you may learn about in the future.

10.6 Exercises

Practice exercises for the material covered in this chapter can be found in the accompanying worksheets repository[2] in the two "Statistical inference" rows. You can launch an interactive version of each worksheet in your browser by clicking the "launch binder" button. You can also preview a non-interactive version of each worksheet by clicking "view worksheet". If you instead decide to download the worksheets and run them on your own machine, make sure to follow the instructions for computer setup found in Chapter 13. This will ensure that the automated feedback and guidance that the worksheets provide will function as intended.

10.7 Additional resources

- Chapters 4 to 7 of *OpenIntro Statistics* [Diez *et al.*, 2019] provide a good next step in learning about inference. Although it is still certainly an introductory text, things get a bit more mathematical here. Depending on your background, you may actually want to start going through Chapters 1 to 3 first, where you will learn some fundamental concepts in probability theory.

[2] https://worksheets.python.datasciencebook.ca

Although it may seem like a diversion, probability theory is *the language of statistics*; if you have a solid grasp of probability, more advanced statistics will come naturally to you.

11

Combining code and text with Jupyter

11.1 Overview

A typical data analysis involves not only writing and executing code, but also writing text and displaying images that help tell the story of the analysis. In fact, ideally, we would like to *interleave* these three media, with the text and images serving as narration for the code and its output. In this chapter, we will show you how to accomplish this using Jupyter notebooks, a common coding platform in data science. Jupyter notebooks do precisely what we need: they let you combine text, images, and (executable!) code in a single document. In this chapter, we will focus on the *use* of Jupyter notebooks to program in Python and write text via a web interface. These skills are essential to getting your analysis running; think of it like getting dressed in the morning. Note that we assume that you already have Jupyter set up and ready to use. If that is not the case, please first read Chapter 13 to learn how to install and configure Jupyter on your own computer.

11.2 Chapter learning objectives

By the end of the chapter, readers will be able to do the following:

- Create new Jupyter notebooks.
- Write, edit, and execute Python code in a Jupyter notebook.
- Write, edit, and view text in a Jupyter notebook.
- Open and view plain text data files in Jupyter.
- Export Jupyter notebooks to other standard file types (e.g., `.html`, `.pdf`).

DOI: 10.1201/9781003438397-11

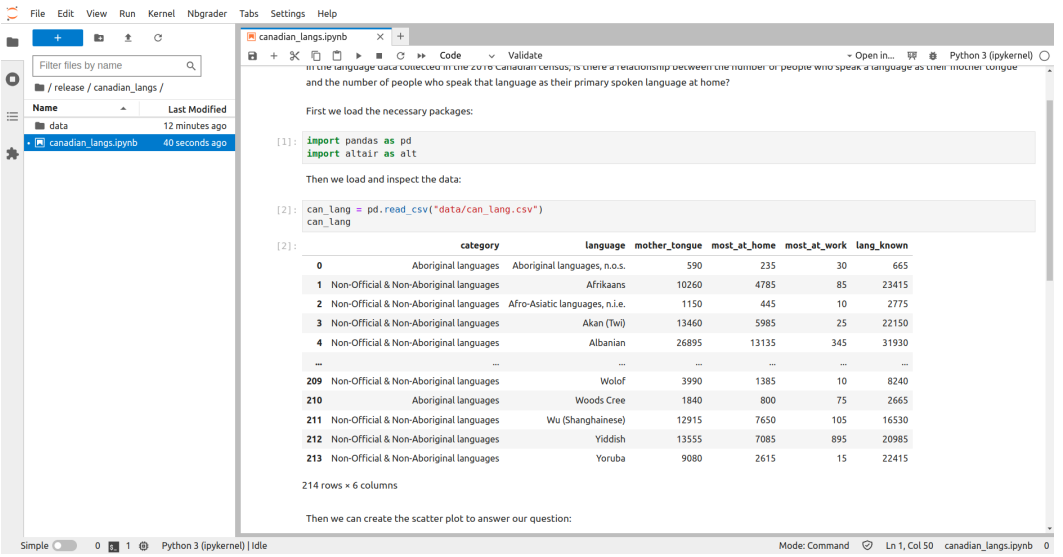


FIGURE 11.1 A screenshot of a Jupyter Notebook.

11.3 Jupyter

Jupyter [Kluyver *et al.*, 2016] is a web-based interactive development environment for creating, editing, and executing documents called Jupyter notebooks. Jupyter notebooks are documents that contain a mix of computer code (and its output) and formattable text. Given that they combine these two analysis artifacts in a single document—code is not separate from the output or written report—notebooks are one of the leading tools to create reproducible data analyses. Reproducible data analysis is one where you can reliably and easily re-create the same results when analyzing the same data. Although this sounds like something that should always be true of any data analysis, in reality, this is not often the case; one needs to make a conscious effort to perform data analysis in a reproducible manner. An example of what a Jupyter notebook looks like is shown in Fig. 11.1.

11.3.1 Accessing Jupyter

One of the easiest ways to start working with Jupyter is to use a web-based platform called JupyterHub. JupyterHubs often have Jupyter, Python, a number of Python packages, and collaboration tools installed, configured and ready to use. JupyterHubs are usually created and provisioned by organizations, and require authentication to gain access. For example, if you are reading this book as part of a course, your instructor may have a JupyterHub already set up for

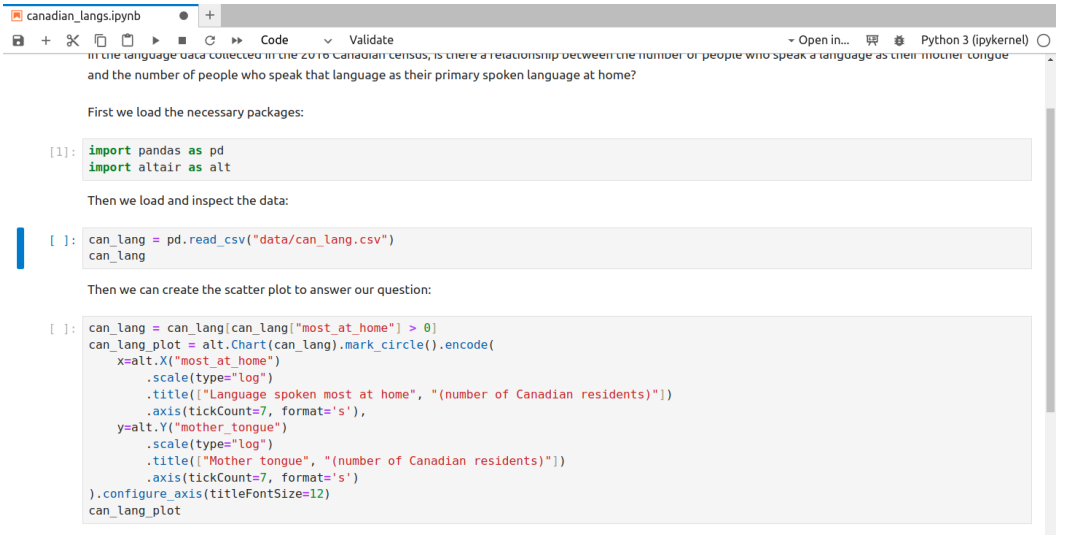


FIGURE 11.2 A code cell in Jupyter that has not yet been executed.

you to use. Jupyter can also be installed on your own computer; see Chapter 13 for instructions.

11.4 Code cells

The sections of a Jupyter notebook that contain code are referred to as code cells. A code cell that has not yet been executed has no number inside the square brackets to the left of the cell (Fig. 11.2). Running a code cell will execute all of the code it contains, and the output (if any exists) will be displayed directly underneath the code that generated it. Outputs may include printed text or numbers, data frames and data visualizations. Cells that have been executed also have a number inside the square brackets to the left of the cell. This number indicates the order in which the cells were run (Fig. 11.3).

11.4.1 Executing code cells

Code cells can be run independently or as part of executing the entire notebook using one of the "**Run all**" commands found in the **Run** or **Kernel** menus in Jupyter. Running a single code cell independently is a workflow typically used when editing or writing your own Python code. Executing an entire notebook is a workflow typically used to ensure that your analysis runs in its entirety before sharing it with others, and when using a notebook as part of an automated process.

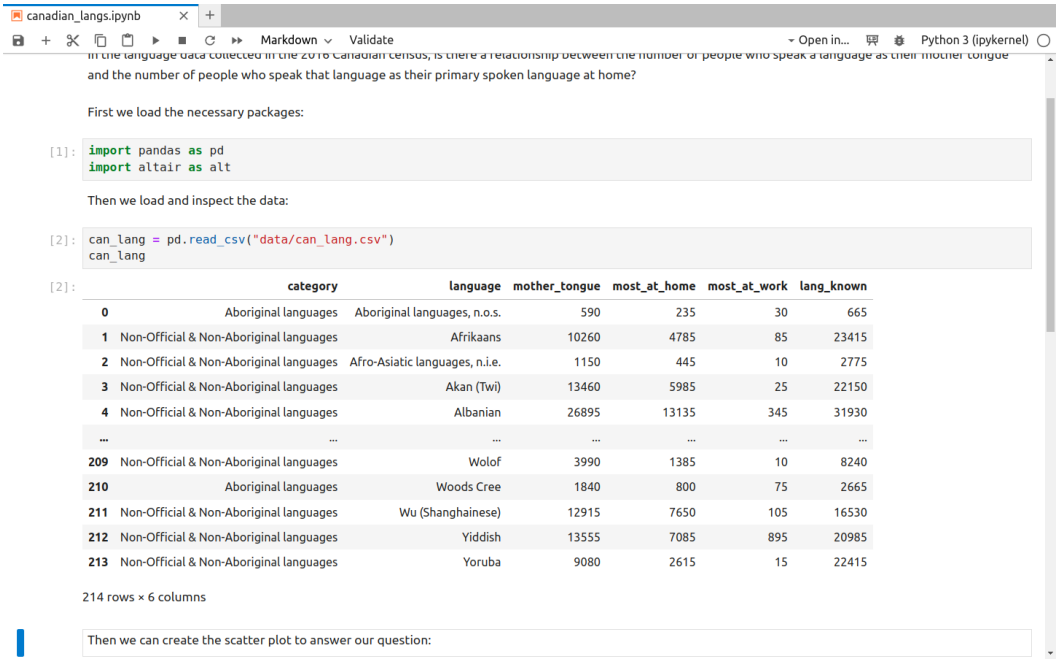


FIGURE 11.3 A code cell in Jupyter that has been executed.

To run a code cell independently, the cell needs to first be activated. This is done by clicking on it with the cursor. Jupyter will indicate a cell has been activated by highlighting it with a blue rectangle to its left. After the cell has been activated (Fig. 11.4), the cell can be run by either pressing the **Run** (▶) button in the toolbar, or by using a keyboard shortcut of `Shift + Enter`.

To execute all of the code cells in an entire notebook, you have three options:

1. Select **Run** » **Run All Cells** from the menu.
2. Select **Kernel** » **Restart Kernel and Run All Cells…** from the menu (Fig. 11.5).
3. Click the (⏭) button in the tool bar.

All of these commands result in all of the code cells in a notebook being run. However, there is a slight difference between them. In particular, only options 2 and 3 above will restart the Python session before running all of the cells; option 1 will not restart the session. Restarting the Python session means that all previous objects that were created from running cells before this command was run will be deleted. In other words, restarting the session and then running all cells (options 2 or 3) emulates how your notebook code would run if you completely restarted Jupyter before executing your entire notebook.

FIGURE 11.4 An activated cell that is ready to be run. The red arrow points to the blue rectangle to the cell's left. The blue rectangle indicates that it is ready to be run. This can be done by clicking the run button (circled in red).

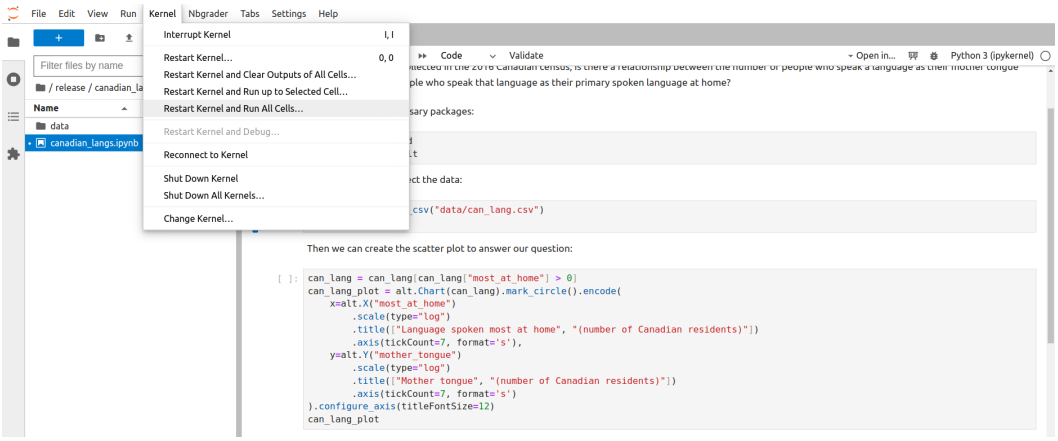


FIGURE 11.5 Restarting the Python session can be accomplished by clicking Restart Kernel and Run All Cells.

11.4.2 The Kernel

The kernel is a program that executes the code inside your notebook and outputs the results. Kernels for many different programming languages have been created for Jupyter, which means that Jupyter can interpret and execute the code of many different programming languages. To run Python code, your notebook will need a Python kernel. In the top right of your window, you can see a circle that indicates the status of your kernel. If the circle is empty ( ), the kernel is idle and ready to execute code. If the circle is filled in ( ), the kernel is busy running some code.

You may run into problems where your kernel is stuck for an excessive amount of time, your notebook is very slow and unresponsive, or your kernel loses its connection. If this happens, try the following steps:

1. At the top of your screen, click **Kernel**, then **Interrupt Kernel**.
2. If that doesn't help, click **Kernel**, then **Restart Kernel ...** If you do this, you will have to run your code cells from the start of your notebook up until where you paused your work.
3. If that still doesn't help, restart Jupyter. First, save your work by clicking **File** at the top left of your screen, then **Save Notebook**. Next, if you are accessing Jupyter using a JupyterHub server, from the **File** menu click **Hub Control Panel**. Choose **Stop My Server** to shut it down, then the **My Server** button to start it back up. If you are running Jupyter on your own computer, from the **File** menu click **Shut Down**, then start Jupyter again. Finally, navigate back to the notebook you were working on.

11.4.3 Creating new code cells

To create a new code cell in Jupyter (Fig. 11.6), click the + button in the toolbar. By default, all new cells in Jupyter start out as code cells, so after this, all you have to do is write Python code within the new cell you just created.

11.5 Markdown cells

Text cells inside a Jupyter notebook are called Markdown cells. Markdown cells are rich formatted text cells, which means you can **bold** and *italicize* text,

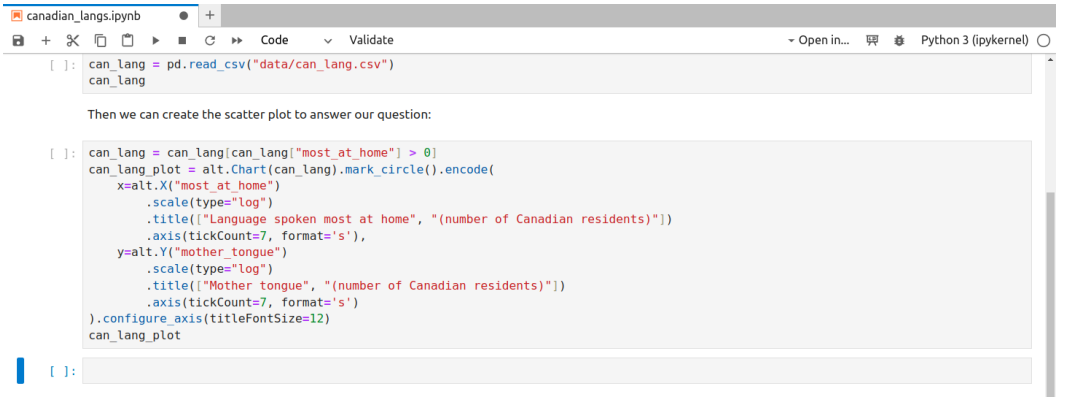


FIGURE 11.6 New cells can be created by clicking the + button, and are by default code cells.

create subject headers, create bullet and numbered lists, and more. These cells are given the name "Markdown" because they use *Markdown language* to specify the rich text formatting. You do not need to learn Markdown to write text in the Markdown cells in Jupyter; plain text will work just fine. However, you might want to learn a bit about Markdown eventually to enable you to create nicely formatted analyses. See the additional resources at the end of this chapter to find out where you can start learning Markdown.

11.5.1 Editing Markdown cells

To edit a Markdown cell in Jupyter, you need to double click on the cell. Once you do this, the unformatted (or *unrendered*) version of the text will be shown (Fig. 11.7). You can then use your keyboard to edit the text. To view the formatted (or *rendered*) text (Fig. 11.8), click the **Run** (▶) button in the toolbar, or use the `Shift + Enter` keyboard shortcut.

11.5.2 Creating new Markdown cells

To create a new Markdown cell in Jupyter, click the + button in the toolbar. By default, all new cells in Jupyter start as code cells, so the cell format needs to be changed to be recognized and rendered as a Markdown cell. To do this, click on the cell with your cursor to ensure it is activated. Then click on the drop-down box on the toolbar that says "Code" (it is next to the ⏭ button), and change it from "**Code**" to "**Markdown**" (Fig. 11.9).

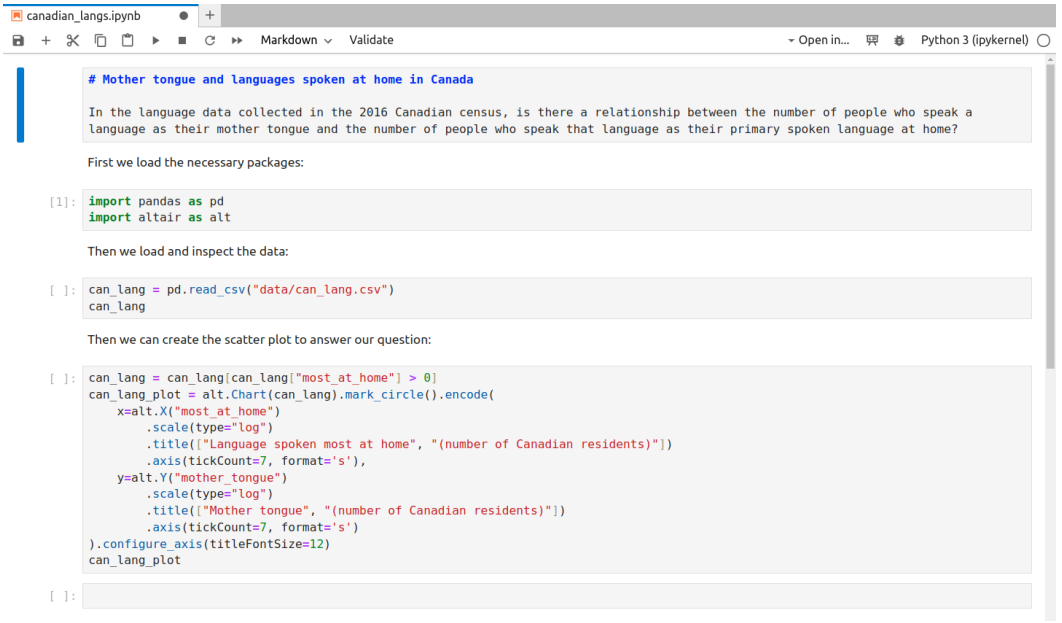


FIGURE 11.7 A Markdown cell in Jupyter that has not yet been rendered and can be edited.

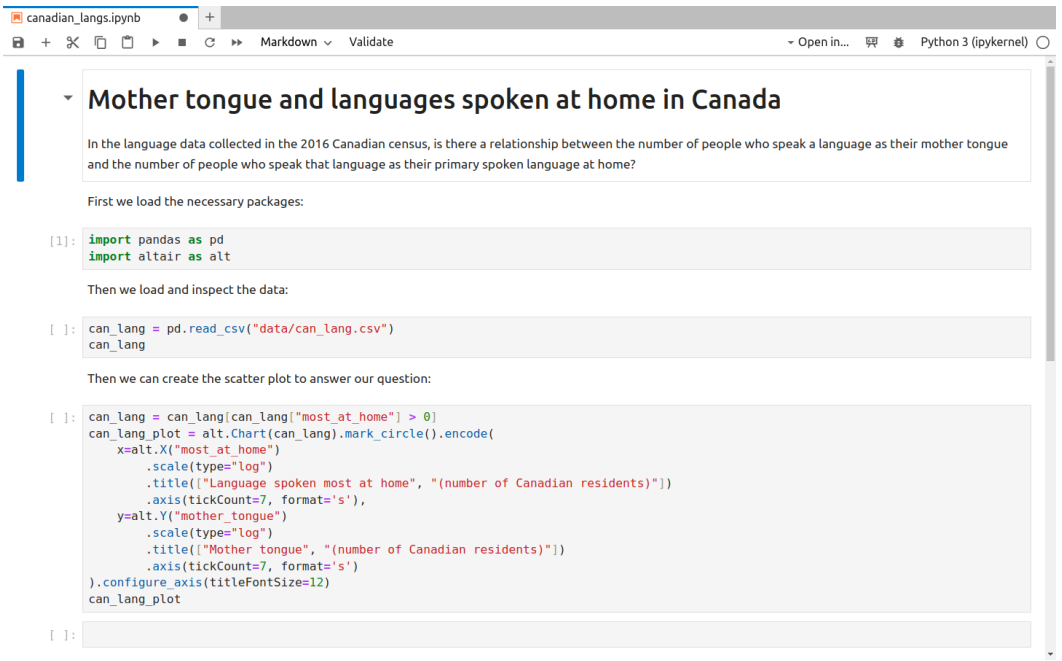


FIGURE 11.8 A Markdown cell in Jupyter that has been rendered and exhibits rich text formatting.

FIGURE 11.9 New cells are by default code cells. To create Markdown cells, the cell format must be changed.

11.6 Saving your work

As with any file you work on, it is critical to save your work often so you don't lose your progress. Jupyter has an autosave feature, where open files are saved periodically. The default for this is every two minutes. You can also manually save a Jupyter notebook by selecting **Save Notebook** from the **File** menu, by clicking the disk icon on the toolbar, or by using a keyboard shortcut (`Control + S` for Windows, or `Command + S` for Mac OS).

11.7 Best practices for running a notebook

11.7.1 Best practices for executing code cells

As you might know (or at least imagine) by now, Jupyter notebooks are great for interactively editing, writing, and running Python code; this is what they were designed for. Consequently, Jupyter notebooks are flexible in regard to code cell execution order. This flexibility means that code cells can be run in any arbitrary order using the **Run** (▶) button. But this flexibility has a downside: it can lead to Jupyter notebooks whose code cannot be executed in a linear order (from top to bottom of the notebook). A nonlinear notebook is problematic because a linear order is the conventional way code documents are run, and others will have this expectation when running your notebook. Finally, if the code is used in some automated process, it will need to run in a linear order, from top to bottom of the notebook.

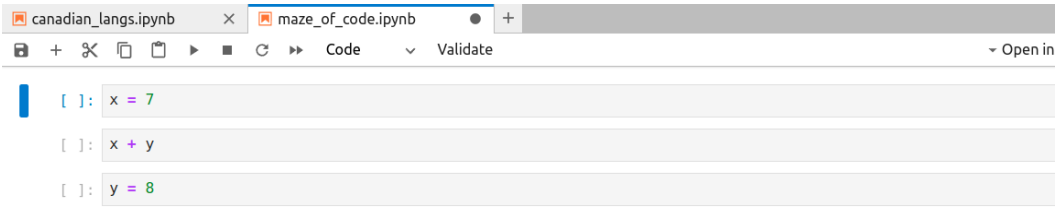


FIGURE 11.10 Code that was written out of order, but not yet executed.

The most common way to inadvertently create a nonlinear notebook is to rely solely on using the (▶) button to execute cells. For example, suppose you write some Python code that creates a Python object, say a variable named `y`. When you execute that cell and create `y`, it will continue to exist until it is deliberately deleted with Python code, or when the Jupyter notebook Python session (i.e., kernel) is stopped or restarted. It can also be referenced in another distinct code cell (Fig. 11.10). Together, this means that you could then write a code cell further above in the notebook that references `y` and execute it without error in the current session (Fig. 11.11). This could also be done successfully in future sessions if, and only if, you run the cells in the same unconventional order. However, it is difficult to remember this unconventional order, and it is not the order that others would expect your code to be executed in. Thus, in the future, this would lead to errors when the notebook is run in the conventional linear order (Fig. 11.12).

You can also accidentally create a nonfunctioning notebook by creating an object in a cell that later gets deleted. In such a scenario, that object only exists for that one particular Python session and will not exist once the notebook is restarted and run again. If that object was referenced in another cell in that notebook, an error would occur when the notebook was run again in a new session.

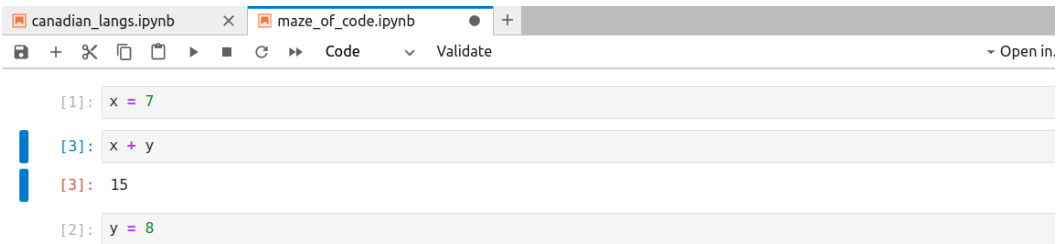


FIGURE 11.11 Code that was written out of order, and was executed using the run button in a nonlinear order without error. The order of execution can be traced by following the numbers to the left of the code cells; their order indicates the order in which the cells were executed.

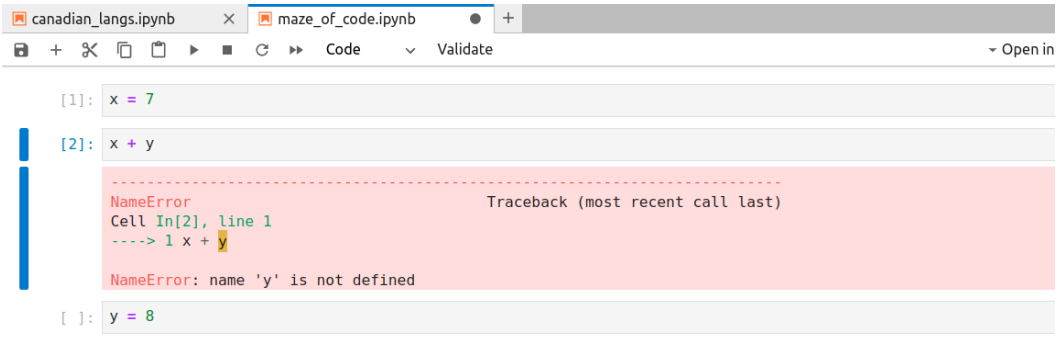


FIGURE 11.12 Code that was written out of order, and was executed in a linear order using "Restart Kernel and Run All Cells…" This resulted in an error at the execution of the second code cell and it failed to run all code cells in the notebook.

These events may not negatively affect the current Python session when the code is being written; but as you might now see, they will likely lead to errors when that notebook is run in a future session. Regularly executing the entire notebook in a fresh Python session will help guard against this. If you restart your session and new errors seem to pop up when you run all of your cells in linear order, you can at least be aware that there is an issue. Knowing this sooner rather than later will allow you to fix the issue and ensure your notebook can be run linearly from start to finish.

We recommend as a best practice to run the entire notebook in a fresh Python session at least 2–3 times within any period of work. Note that, critically, you *must do this in a fresh Python session* by restarting your kernel. We recommend using either the **Kernel** » **Restart Kernel and Run All Cells…** command from the menu or the ⏭ button in the toolbar. Note that the **Run** » **Run All Cells** menu item will not restart the kernel, and so it is not sufficient to guard against these errors.

11.7.2 Best practices for including Python packages in notebooks

Most data analyses these days depend on functions from external Python packages that are not built into Python. One example is the `pandas` package that we heavily rely on in this book. This package provides us access to functions like `read_csv` for reading data, and `loc[]` for subsetting rows and columns. We also use the `altair` package for creating high-quality graphics.

As mentioned earlier in the book, external Python packages need to be loaded before the functions they contain can be used. Our recommended way to do this is via `import package_name`, and perhaps also to give it a shorter alias

like `import package_name as pn`. But where should this line of code be written in a Jupyter notebook? One idea could be to load the library right before the function is used in the notebook. However, although this technically works, this causes hidden, or at least non-obvious, Python package dependencies when others view or try to run the notebook. These hidden dependencies can lead to errors when the notebook is executed on another computer if the needed Python packages are not installed. Additionally, if the data analysis code takes a long time to run, uncovering the hidden dependencies that need to be installed so that the analysis can run without error can take a great deal of time to uncover.

Therefore, we recommend you load all Python packages in a code cell near the top of the Jupyter notebook. Loading all your packages at the start ensures that all packages are loaded before their functions are called, assuming the notebook is run in a linear order from top to bottom as recommended above. It also makes it easy for others viewing or running the notebook to see what external Python packages are used in the analysis, and hence, what packages they should install on their computer to run the analysis successfully.

11.7.3 Summary of best practices for running a notebook

1. Write code so that it can be executed in a linear order.
2. As you write code in a Jupyter notebook, run the notebook in a linear order and in its entirety often (2–3 times every work session) via the **Kernel** » **Restart Kernel and Run All Cells ...** command from the Jupyter menu or the ⏭ button in the toolbar.
3. Write the code that loads external Python packages near the top of the Jupyter notebook.

11.8 Exploring data files

It is essential to preview data files before you try to read them into Python to see whether or not there are column names, what the separators are, and if there are lines you need to skip. In Jupyter, you preview data files stored as plain text files (e.g., comma- and tab-separated files) in their plain text format (Fig. 11.14) by right-clicking on the file's name in the Jupyter file explorer, selecting **Open with**, and then selecting **Editor** (Fig. 11.13). Suppose you

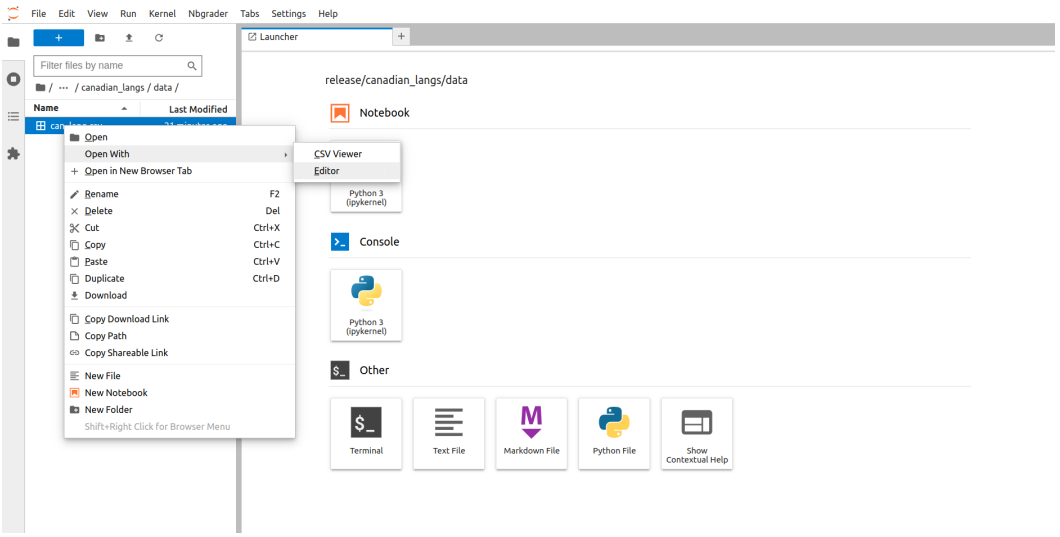


FIGURE 11.13 Opening data files with an editor in Jupyter.

do not specify to open the data file with an editor. In that case, Jupyter will render a nice table for you, and you will not be able to see the column separators, and therefore you will not know which function to use, nor which arguments to use and values to specify for them.

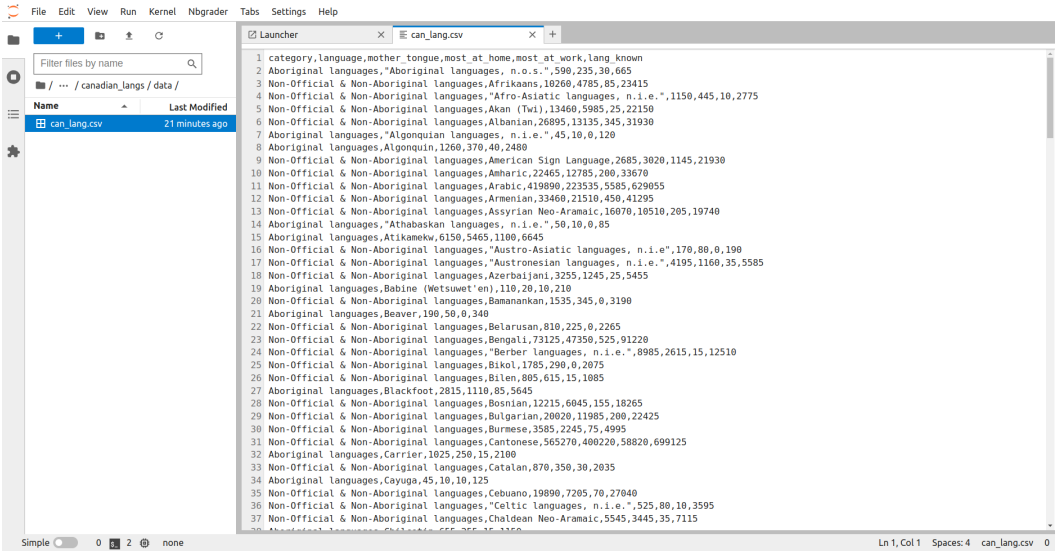


FIGURE 11.14 A data file as viewed in an editor in Jupyter.

11.9 Exporting to a different file format

In Jupyter, viewing, editing and running Python code is done in the Jupyter notebook file format with file extension `.ipynb`. This file format is not easy to open and view outside of Jupyter. Thus, to share your analysis with people who do not commonly use Jupyter, it is recommended that you export your executed analysis as a more common file type, such as an `.html` file, or a `.pdf`. We recommend exporting the Jupyter notebook after executing the analysis so that you can also share the outputs of your code. Note, however, that your audience will not be able to *run* your analysis using a `.html` or `.pdf` file. If you want your audience to be able to reproduce the analysis, you must provide them with the `.ipynb` Jupyter notebook file.

11.9.1 Exporting to HTML

Exporting to `.html` will result in a shareable file that anyone can open using a web browser (e.g., Firefox, Safari, Chrome, or Edge). The `.html` output will produce a document that is visually similar to what the Jupyter notebook looked like inside Jupyter. One point of caution here is that if there are images in your Jupyter notebook, you will need to share the image files and the `.html` file to see them.

11.9.2 Exporting to PDF

Exporting to `.pdf` will result in a shareable file that anyone can open using many programs, including Adobe Acrobat, Preview, web browsers, and many more. The benefit of exporting to PDF is that it is a standalone document, even if the Jupyter notebook included references to image files. Unfortunately, the default settings will result in a document that visually looks quite different from what the Jupyter notebook looked like. The font, page margins, and other details will appear different in the `.pdf` output.

11.10 Creating a new Jupyter notebook

At some point, you will want to create a new, fresh Jupyter notebook for your own project instead of viewing, running or editing a notebook that was started by someone else. To do this, navigate to the **Launcher** tab, and click on the Python icon under the **Notebook** heading. If no **Launcher** tab is visible,

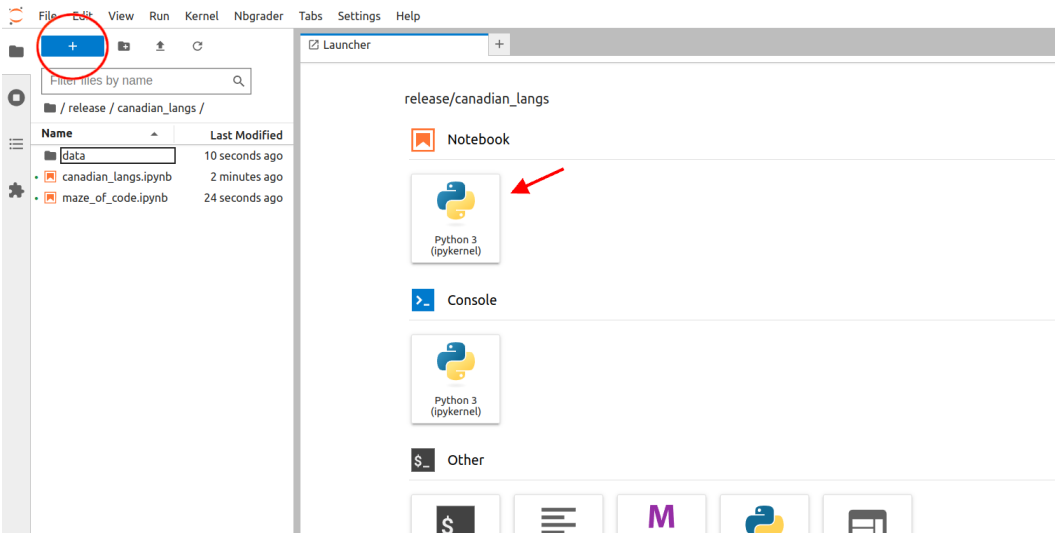


FIGURE 11.15 Clicking on the Python icon under the Notebook heading will create a new Jupyter notebook with a Python kernel.

you can get a new one via clicking the + button at the top of the Jupyter file explorer (Fig. 11.15).

Once you have created a new Jupyter notebook, be sure to give it a descriptive name, as the default file name is `Untitled.ipynb`. You can rename files by first right-clicking on the file name of the notebook you just created, and then clicking **Rename**. This will make the file name editable. Use your keyboard to change the name. Pressing `Enter` or clicking anywhere else in the Jupyter interface will save the changed file name.

We recommend not using white space or non-standard characters in file names. Doing so will not prevent you from using that file in Jupyter. However, these sorts of things become troublesome as you start to do more advanced data science projects that involve repetition and automation. We recommend naming files using lower case characters and separating words by a dash (-) or an underscore (_).

11.11 Additional resources

- The JupyterLab Documentation[1] is a good next place to look for more information about working in Jupyter notebooks. This documentation goes into

[1] https://jupyterlab.readthedocs.io/en/latest/

significantly more detail about all of the topics we covered in this chapter, and covers more advanced topics as well.

- If you are keen to learn about the Markdown language for rich text formatting, two good places to start are CommonMark's Markdown cheatsheet[2] and Markdown tutorial[3].

[2] https://commonmark.org/help/
[3] https://commonmark.org/help/tutorial/

12

Collaboration with version control

You mostly collaborate with yourself, and me-from-two-months-ago never responds to email.

–Mark T. Holder

12.1 Overview

This chapter will introduce the concept of using version control systems to track changes to a project over its lifespan, to share and edit code in a collaborative team, and to distribute the finished project to its intended audience. This chapter will also introduce how to use the two most common version control tools: Git for local version control, and GitHub for remote version control. We will focus on the most common version control operations used day-to-day in a standard data science project. There are many user interfaces for Git; in this chapter we will cover the Jupyter Git interface.

12.2 Chapter learning objectives

By the end of the chapter, readers will be able to do the following:

- Describe what version control is and why data analysis projects can benefit from it.
- Create a remote version control repository on GitHub.
- Use Jupyter's Git version control tools for project versioning and collaboration:

DOI: 10.1201/9781003438397-12

  - Clone a remote version control repository to create a local repository.
  - Commit changes to a local version control repository.
  - Push local changes to a remote version control repository.
  - Pull changes from a remote version control repository to a local version control repository.
  - Resolve merge conflicts.
- Give collaborators access to a remote GitHub repository.
- Communicate with collaborators using GitHub issues.
- Use best practices when collaborating on a project with others.

12.3 What is version control, and why should I use it?

Data analysis projects often require iteration and revision to move from an initial idea to a finished product ready for the intended audience. Without deliberate and conscious effort toward tracking changes made to the analysis, projects tend to become messy. This mess can have serious, negative repercussions on an analysis project, including results that your code cannot reproduce, temporary files with snippets of ideas that are forgotten or not easy to find, mind-boggling file names that make it unclear which is the current working version of the file (e.g., `document_final_draft_final.txt`, `to_hand_in_final_v2.txt`, etc.), and more.

Additionally, the iterative nature of data analysis projects means that most of the time, the final version of the analysis that is shared with the audience is only a fraction of what was explored during the development of that analysis. Changes in data visualizations and modeling approaches, as well as some negative results, are often not observable from reviewing only the final, polished analysis. The lack of observability of these parts of the analysis development can lead to others repeating things that did not work well, instead of seeing what did not work well, and using that as a springboard to new, more fruitful approaches.

Finally, data analyses are typically completed by a team of people rather than a single person. This means that files need to be shared across multiple computers, and multiple people often end up editing the project simultaneously.

In such a situation, determining who has the latest version of the project—and how to resolve conflicting edits—can be a real challenge.

Version control helps solve these challenges. Version control is the process of keeping a record of changes to documents, including when the changes were made and who made them, throughout the history of their development. It also provides the means both to view earlier versions of the project and to revert changes. Version control is most commonly used in software development, but can be used for any electronic files for any type of project, including data analyses. Being able to record and view the history of a data analysis project is important for understanding how and why decisions to use one method or another were made, among other things. Version control also facilitates collaboration via tools to share edits with others and resolve conflicting edits. But even if you're working on a project alone, you should still use version control. It helps you keep track of what you've done, when you did it, and what you're planning to do next.

To version control a project, you generally need two things: a *version control system* and a *repository hosting service*. The version control system is the software responsible for tracking changes, sharing changes you make with others, obtaining changes from others, and resolving conflicting edits. The repository hosting service is responsible for storing a copy of the version-controlled project online (a *repository*), where you and your collaborators can access it remotely, discuss issues and bugs, and distribute your final product. For both of these items, there is a wide variety of choices. In this textbook we'll use Git for version control, and GitHub for repository hosting, because both are currently the most widely used platforms. In the additional resources section at the end of the chapter, we list many of the common version control systems and repository hosting services in use today.

Note: Technically you don't *have to* use a repository hosting service. You can, for example, version control a project that is stored only in a folder on your computer—never sharing it on a repository hosting service. But using a repository hosting service provides a few big benefits, including managing collaborator access permissions, tools to discuss and track bugs, and the ability to have external collaborators contribute work, not to mention the safety of having your work backed up in the cloud. Since most repository hosting services now offer free accounts, there are not many situations in which you wouldn't want to use one for your project.

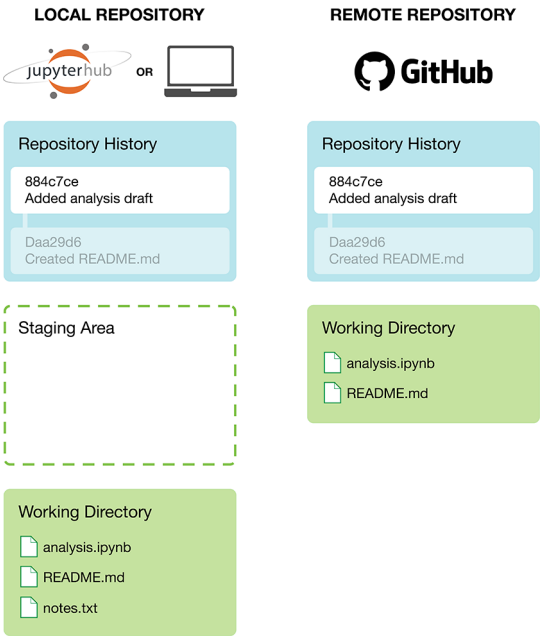


FIGURE 12.1 Schematic of local and remote version control repositories.

12.4 Version control repositories

Typically, when we put a data analysis project under version control, we create two copies of the repository (Fig. 12.1). One copy we use as our primary workspace where we create, edit, and delete files. This copy is commonly referred to as the **local repository**. The local repository most commonly exists on our computer or laptop, but can also exist within a workspace on a server (e.g., JupyterHub). The other copy is typically stored in a repository hosting service (e.g., GitHub), where we can easily share it with our collaborators. This copy is commonly referred to as the **remote repository**.

Both copies of the repository have a **working directory** where you can create, store, edit, and delete files (e.g., `analysis.ipynb` in Fig. 12.1). Both copies of the repository also maintain a full project history (Fig. 12.1). This history is a record of all versions of the project files that have been created. The repository history is not automatically generated; Git must be explicitly told when to record a version of the project. These records are called **commits**. They are a snapshot of the file contents as well metadata about the repository at that time the record was created (who made the commit, when it was made, etc.). In the local and remote repositories shown in Fig. 12.1, there are two commits represented as rectangles inside the "Repository History"

sections. The white rectangle represents the most recent commit, while faded rectangles represent previous commits. Each commit can be identified by a human-readable **message**, which you write when you make a commit, and a **commit hash** that Git automatically adds for you.

The purpose of the message is to contain a brief, rich description of what work was done since the last commit. Messages act as a very useful narrative of the changes to a project over its lifespan. If you ever want to view or revert to an earlier version of the project, the message can help you identify which commit to view or revert to. In Fig. 12.1, you can see two such messages, one for each commit: `Created README.md` and `Added analysis draft.`

The hash is a string of characters consisting of about 40 letters and numbers. The purpose of the hash is to serve as a unique identifier for the commit, and is used by Git to index project history. Although hashes are quite long—imagine having to type out 40 precise characters to view an old project version!—Git is able to work with shorter versions of hashes. In Fig. 12.1, you can see two of these shortened hashes, one for each commit: `Daa29d6` and `884c7ce`.

12.5 Version control workflows

When you work in a local version-controlled repository, there are generally three additional steps you must take as part of your regular workflow. In addition to just working on files—creating, editing, and deleting files as you normally would—you must:

1. Tell Git when to make a commit of your own changes in the local repository.
2. Tell Git when to send your new commits to the remote GitHub repository.
3. Tell Git when to retrieve any new changes (that others made) from the remote GitHub repository.

In this section we will discuss all three of these steps in detail.

12.5.1 Committing changes to a local repository

When working on files in your local version control repository (e.g., using Jupyter) and saving your work, these changes will only initially exist in the working directory of the local repository (Fig. 12.2).

FIGURE 12.2 Local repository with changes to files.

Once you reach a point that you want Git to keep a record of the current version of your work, you need to **commit** (i.e., snapshot) your changes. A prerequisite to this is telling Git which files should be included in that snapshot. We call this step **adding** the files to the **staging area**. Note that the staging area is not a real physical location on your computer; it is instead a conceptual placeholder for these files until they are committed. The benefit of the Git version control system using a staging area is that you can choose to commit changes in only certain files. For example, in Fig. 12.3, we add only the two files that are important to the analysis project (`analysis.ipynb` and `README.md`) and not our personal scratch notes for the project (`notes.txt`).

Once the files we wish to commit have been added to the staging area, we can then commit those files to the repository history (Fig. 12.4). When we do this, we are required to include a helpful *commit message* to tell collaborators (which often includes future you!) about the changes that were made. In Fig. 12.4, the message is `Message about changes...`; in your work you should make sure to replace this with an informative message about what changed. It is also important to note here that these changes are only being committed to the local repository's history. The remote repository on GitHub has not changed, and collaborators would not yet be able to see your new changes.

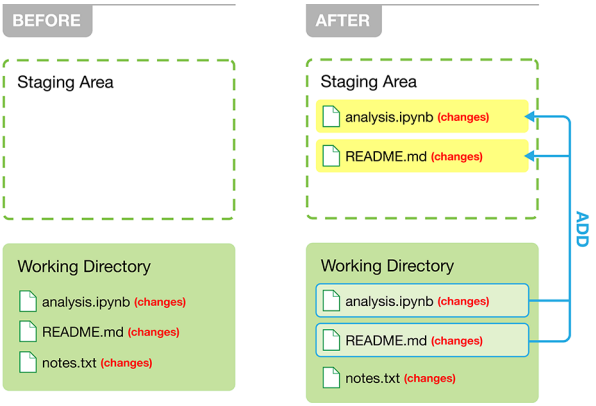


FIGURE 12.3 Adding modified files to the staging area in the local repository.

12.5.2 Pushing changes to a remote repository

Once you have made one or more commits that you want to share with your collaborators, you need to **push** (i.e., send) those commits back to GitHub (Fig. 12.5). This updates the history in the remote repository (i.e., GitHub) to match what you have in your local repository. Now when collaborators interact with the remote repository, they will be able to see the changes you made. And you can also take comfort in the fact that your work is now backed up in the cloud.

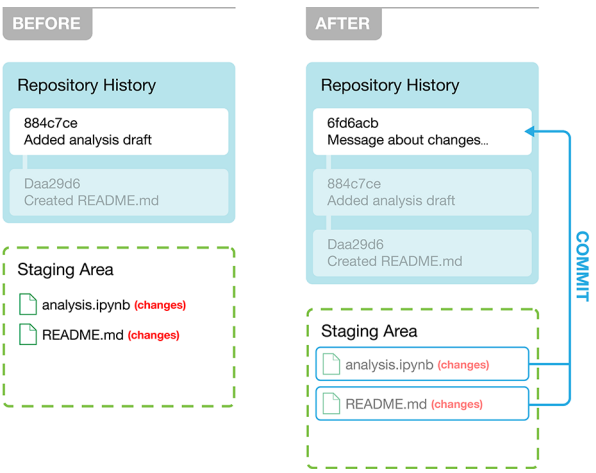


FIGURE 12.4 Committing the modified files in the staging area to the local repository history, with an informative message about what changed.

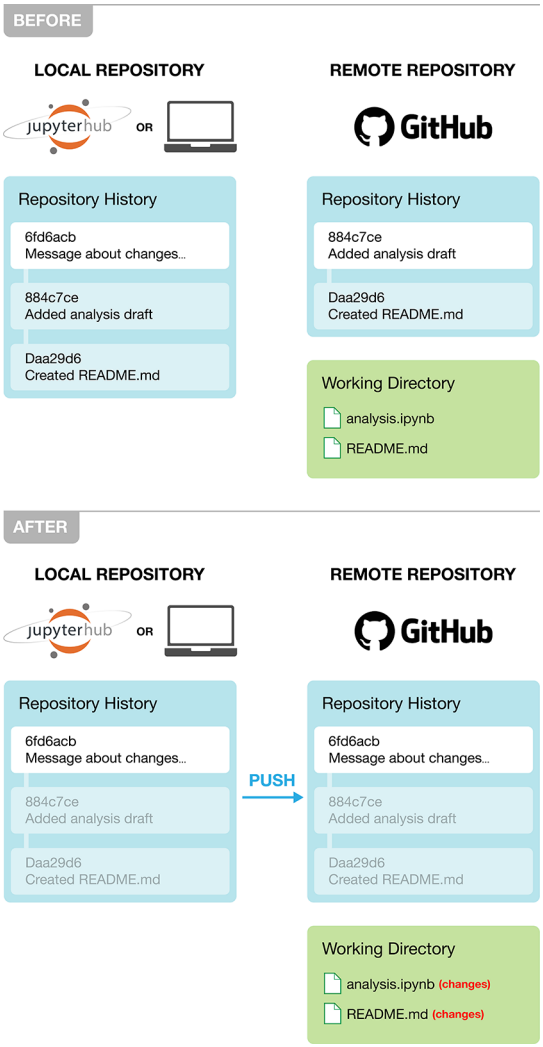


FIGURE 12.5 Pushing the commit to send the changes to the remote repository on GitHub.

12.5.3 Pulling changes from a remote repository

If you are working on a project with collaborators, they will also be making changes to files (e.g., to the analysis code in a Jupyter notebook and the project's README file), committing them to their own local repository, and pushing their commits to the remote GitHub repository to share them with you. When they push their changes, those changes will only initially exist in the remote GitHub repository and not in your local repository (Fig. 12.6).

To obtain the new changes from the remote repository on GitHub, you will need to **pull** those changes to your own local repository. By pulling changes, you synchronize your local repository to what is present on GitHub (Fig. 12.7).

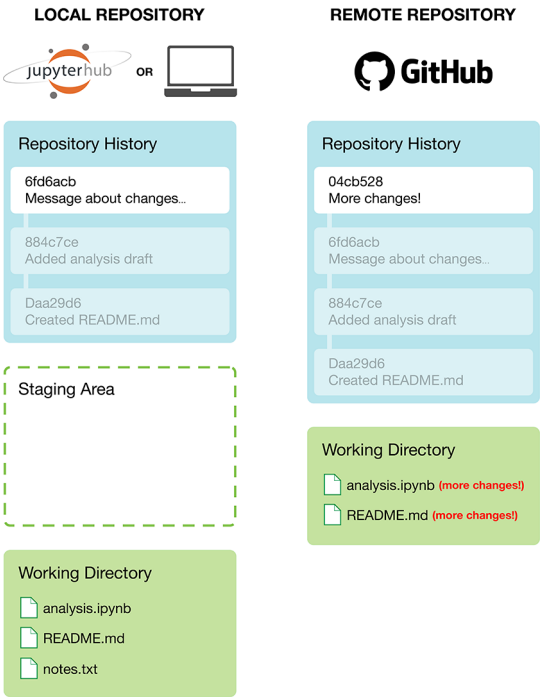


FIGURE 12.6 Changes pushed by collaborators, or created directly on GitHub will not be automatically sent to your local repository.

Additionally, until you pull changes from the remote repository, you will not be able to push any more changes yourself (though you will still be able to work and make commits in your own local repository).

12.6 Working with remote repositories using GitHub

Now that you have been introduced to some of the key general concepts and workflows of Git version control, we will walk through the practical steps. There are several different ways to start using version control with a new project. For simplicity and ease of setup, we recommend creating a remote repository first. This section covers how to both create and edit a remote repository on GitHub. Once you have a remote repository set up, we recommend **cloning** (or copying) that repository to create a local repository in which you primarily work. You can clone the repository either on your own computer or in a workspace on a server (e.g., a JupyterHub server). Section 12.7 below will cover this second step in detail.

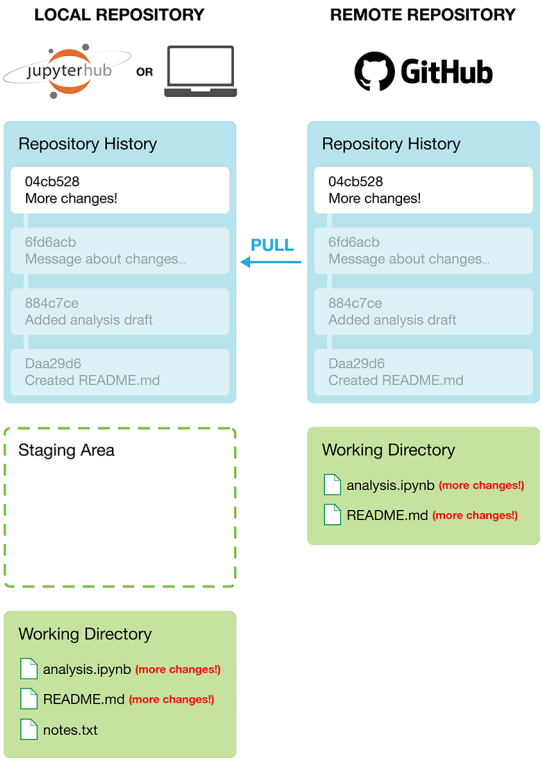


FIGURE 12.7 Pulling changes from the remote GitHub repository to synchronize your local repository.

12.6.1 Creating a remote repository on GitHub

Before you can create remote repositories on GitHub, you will need a GitHub account; you can sign up for a free account at github.com[1]. Once you have logged into your account, you can create a new repository to host your project by clicking on the "+" icon in the upper right-hand corner, and then on "New Repository", as shown in Fig. 12.8.

Repositories can be set up with a variety of configurations, including a name, optional description, and the inclusion (or not) of several template files. One of the most important configuration items to choose is the visibility to the outside world, either public or private. *Public* repositories can be viewed by anyone. *Private* repositories can be viewed by only you. Both public and private repositories are only editable by you, but you can change that by giving access to other collaborators.

To get started with a *public* repository having a template `README.md` file, take the following steps shown in Fig. 12.9:

[1] https://github.com/

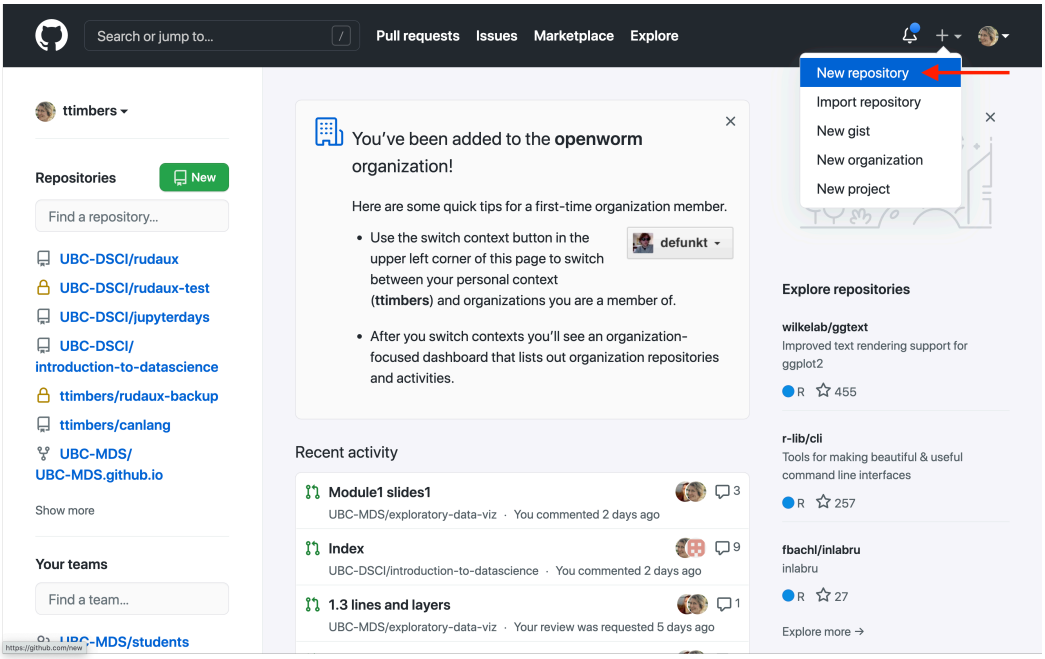


FIGURE 12.8 New repositories on GitHub can be created by clicking on "New Repository" from the + menu.

1. Enter the name of your project repository. In the example below, we use `canadian_languages`. Most repositories follow a similar naming convention involving only lowercase letter words separated by either underscores or hyphens.
2. Choose an option for the privacy of your repository.
3. Select "Add a README file". This creates a template `README.md` file in your repository's root folder.
4. When you are happy with your repository name and configuration, click on the green "Create Repository" button.

A newly created public repository with a `README.md` template file should look something like what is shown in Fig. 12.10.

12.6.2 Editing files on GitHub with the pen tool

The pen tool can be used to edit existing plain text files. When you click on the pen tool, the file will be opened in a text box where you can use your keyboard to make changes (Figs. 12.11 and 12.12).

After you are done with your edits, they can be "saved" by *committing* your changes. When you *commit a file* in a repository, the version control system

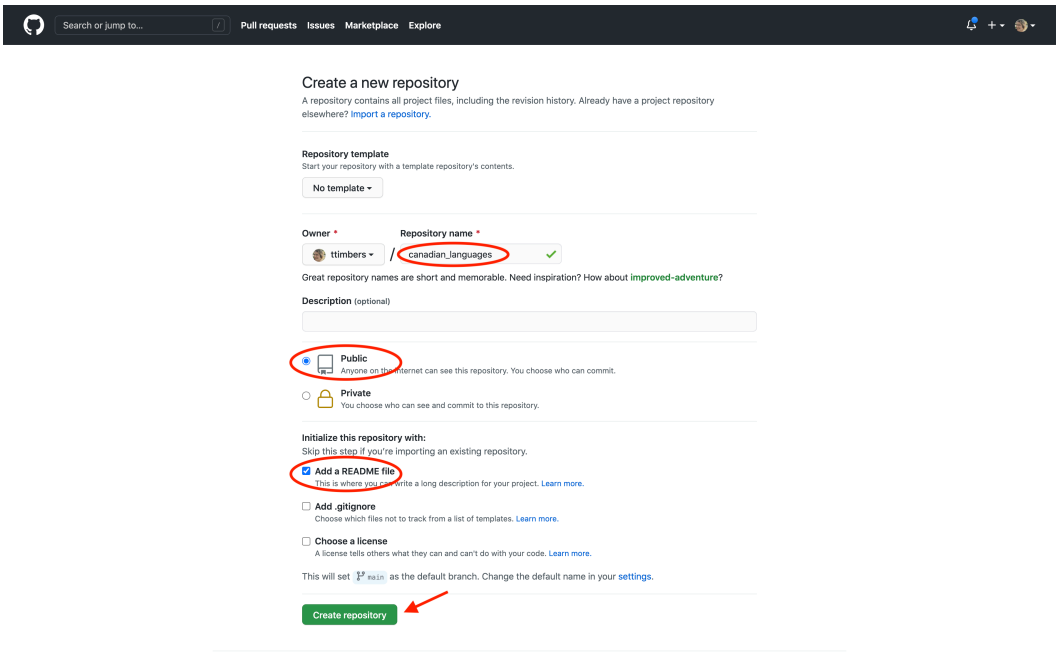


FIGURE 12.9 Repository configuration for a project that is public and initialized with a README.md[2] template file.

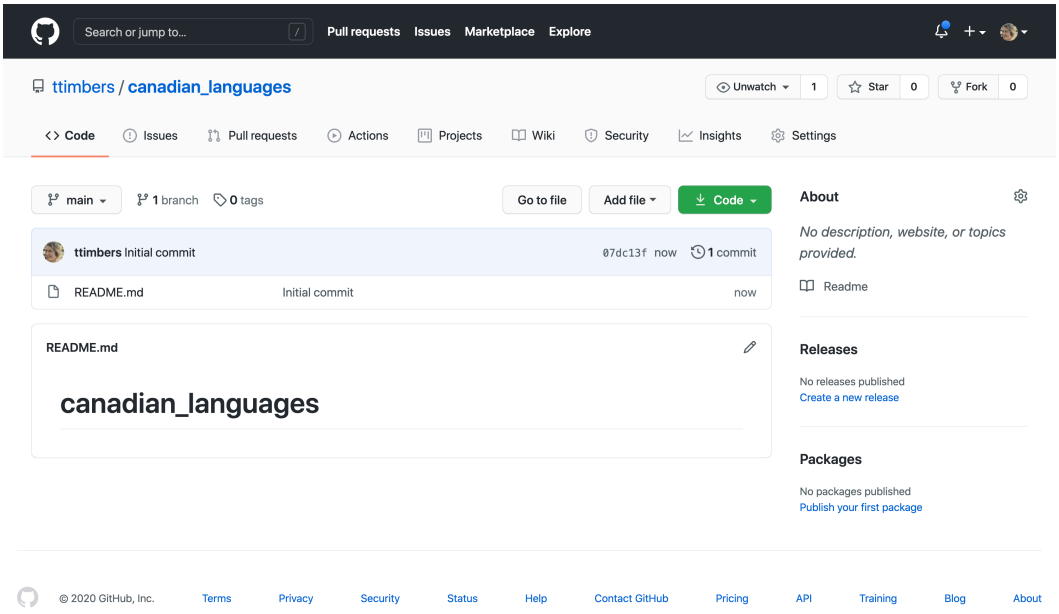


FIGURE 12.10 Repository configuration for a project that is public and initialized with a README.md[3] template file.

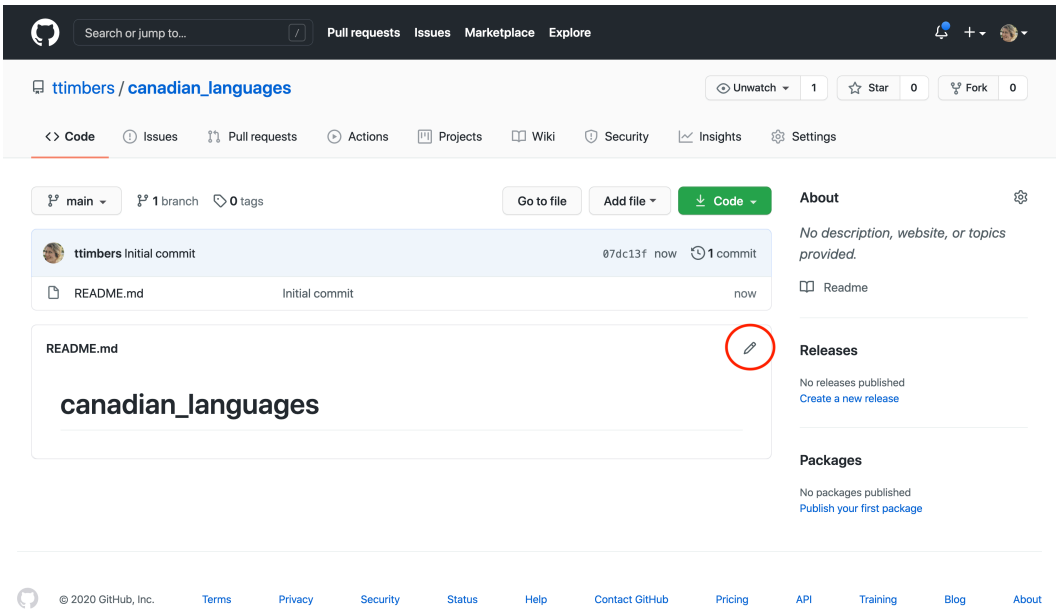


FIGURE 12.11 Clicking on the pen tool opens a text box for editing plain text files.

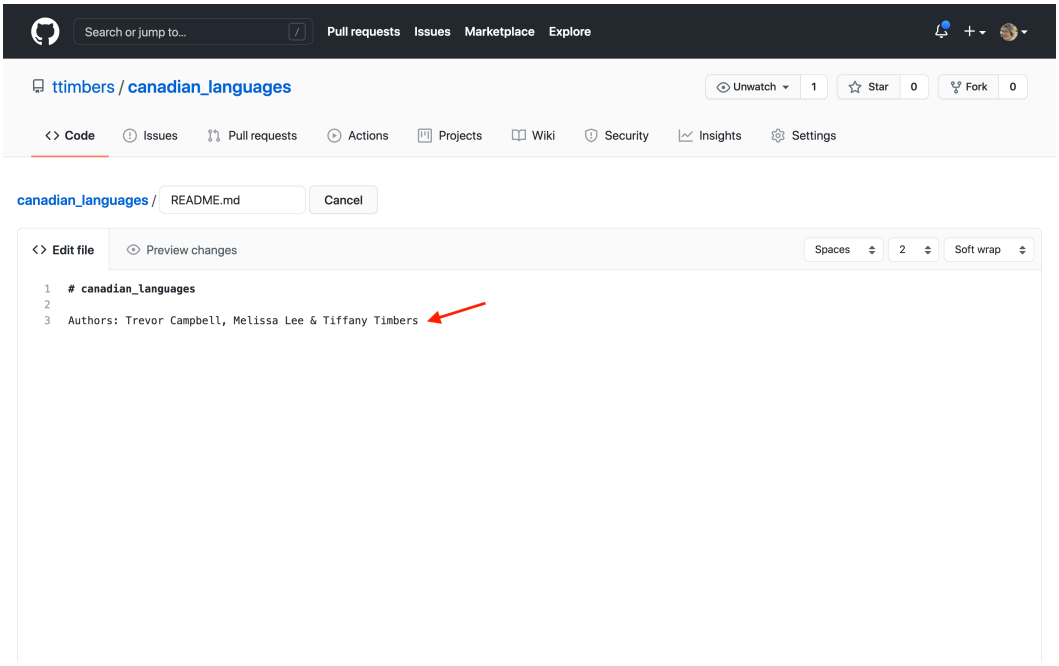


FIGURE 12.12 The text box where edits can be made after clicking on the pen tool.

takes a snapshot of what the file looks like. As you continue working on the project, over time you will possibly make many commits to a single file; this generates a useful version history for that file. On GitHub, if you click the green "Commit changes" button, it will save the file and then make a commit (Fig. 12.13).

Recall from Section 12.5.1 that you normally have to add files to the staging area before committing them. Why don't we have to do that when we work directly on GitHub? Behind the scenes, when you click the green "Commit changes" button, GitHub *is* adding that one file to the staging area prior to committing it. But note that on GitHub you are limited to committing changes to only one file at a time. When you work in your own local repository, you can commit changes to multiple files simultaneously. This is especially useful when one "improvement" to the project involves modifying multiple files. You can also do things like run code when working in a local repository, which you cannot do on GitHub. In general, editing on GitHub is reserved for small edits to plain text files.

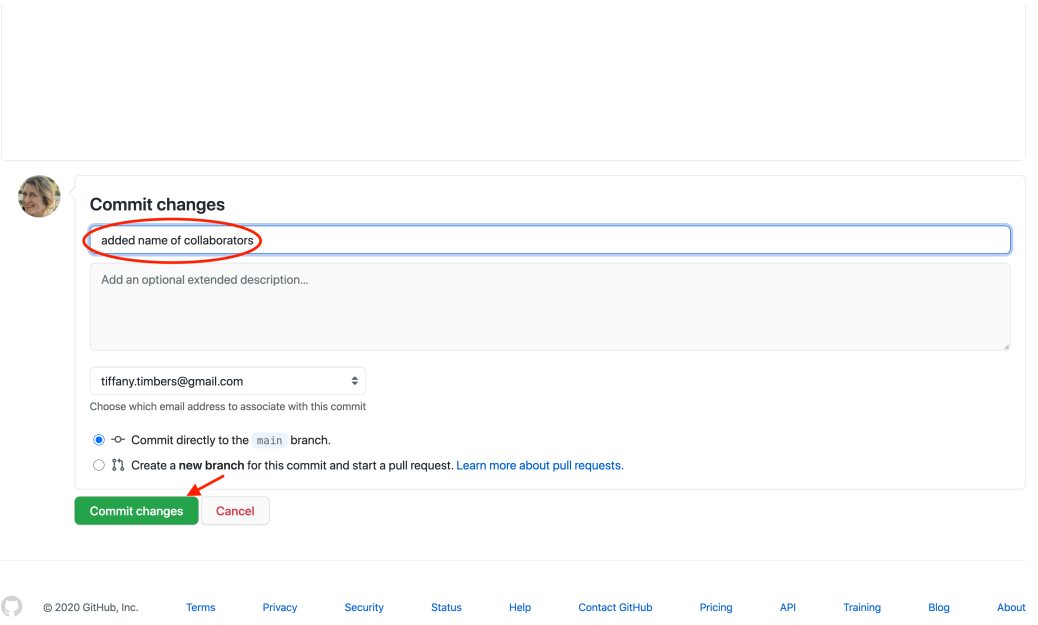


FIGURE 12.13 Saving changes using the pen tool requires committing those changes, and an associated commit message.

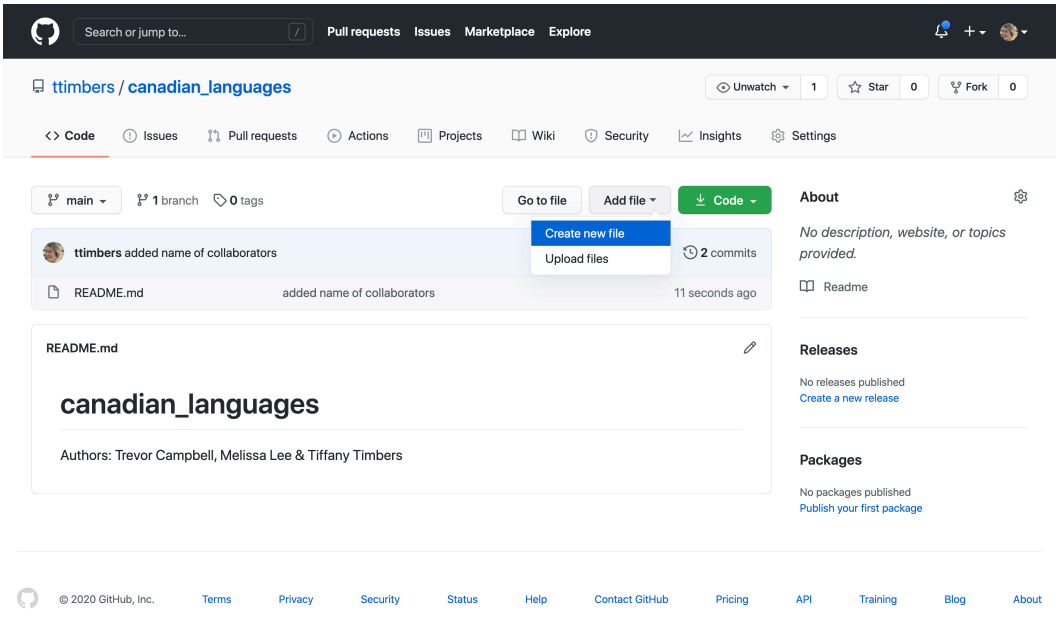


FIGURE 12.14 New plain text files can be created directly on GitHub.

12.6.3 Creating files on GitHub with the "Add file" menu

The "Add file" menu can be used to create new plain text files and upload files from your computer. To create a new plain text file, click the "Add file" drop-down menu and select the "Create new file" option (Fig. 12.14).

A page will open with a small text box for the file name to be entered, and a larger text box where the desired file content text can be entered. Note the two tabs, "Edit new file" and "Preview". Toggling between them lets you enter and edit text and view what the text will look like when rendered, respectively (Fig. 12.15). Note that GitHub understands and renders `.md` files using a markdown syntax very similar to Jupyter notebooks, so the "Preview" tab is especially helpful for checking markdown code correctness.

Save and commit your changes by clicking the green "Commit changes" button at the bottom of the page (Fig. 12.16).

You can also upload files that you have created on your local machine by using the "Add file" drop-down menu and selecting "Upload files" (Fig. 12.17). To select the files from your local computer to upload, you can either drag and drop them into the gray box area shown in Fig. 12.18, or click the "choose your files" link to access a file browser dialog. Once the files you want to upload have been selected, click the green "Commit changes" button at the bottom of the page (Fig. 12.18).

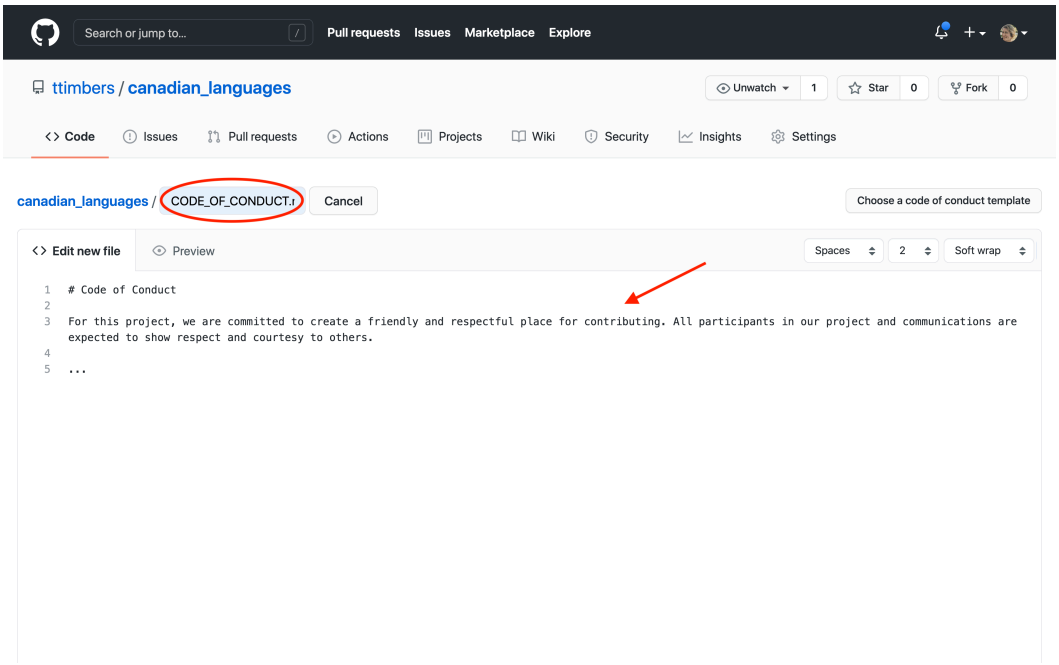


FIGURE 12.15 New plain text files require a file name in the text box circled in red, and file content entered in the larger text box (red arrow).

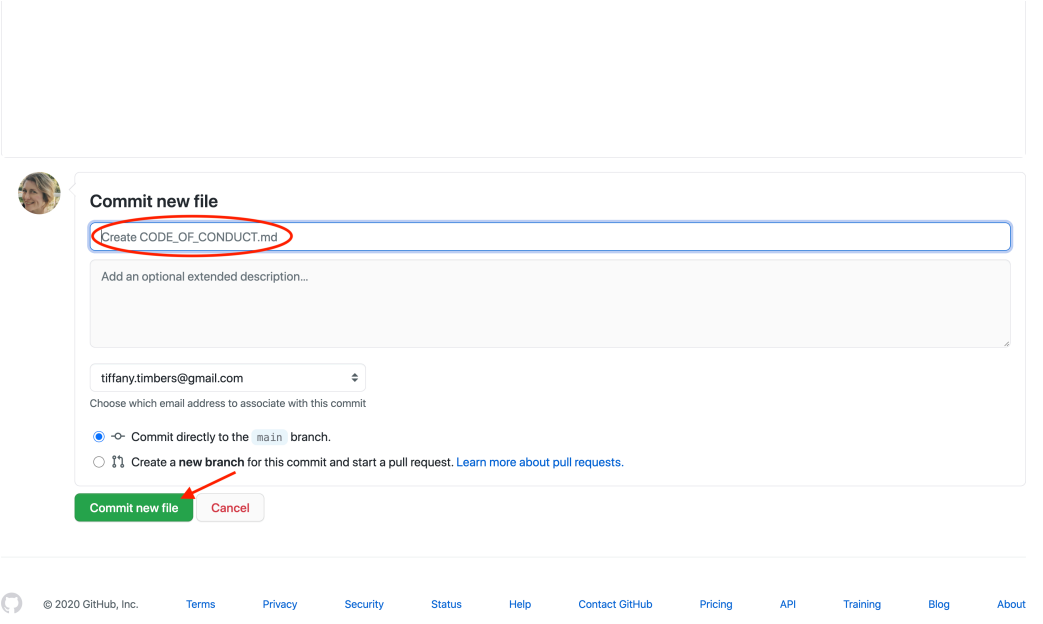


FIGURE 12.16 To be saved, newly created files are required to be committed along with an associated commit message.

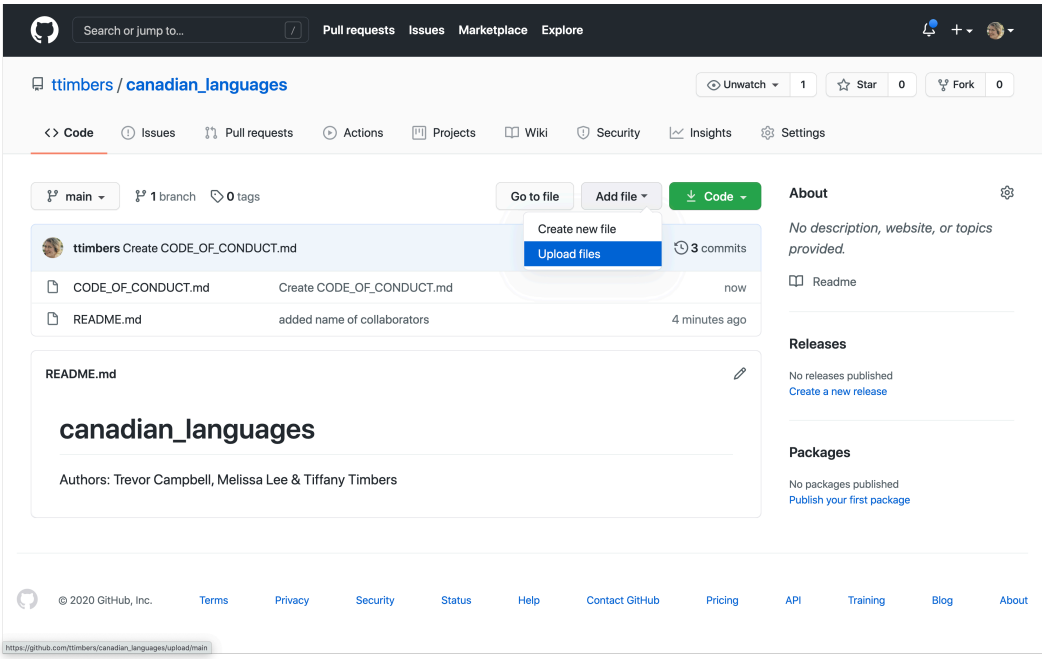


FIGURE 12.17 New files of any type can be uploaded to GitHub.

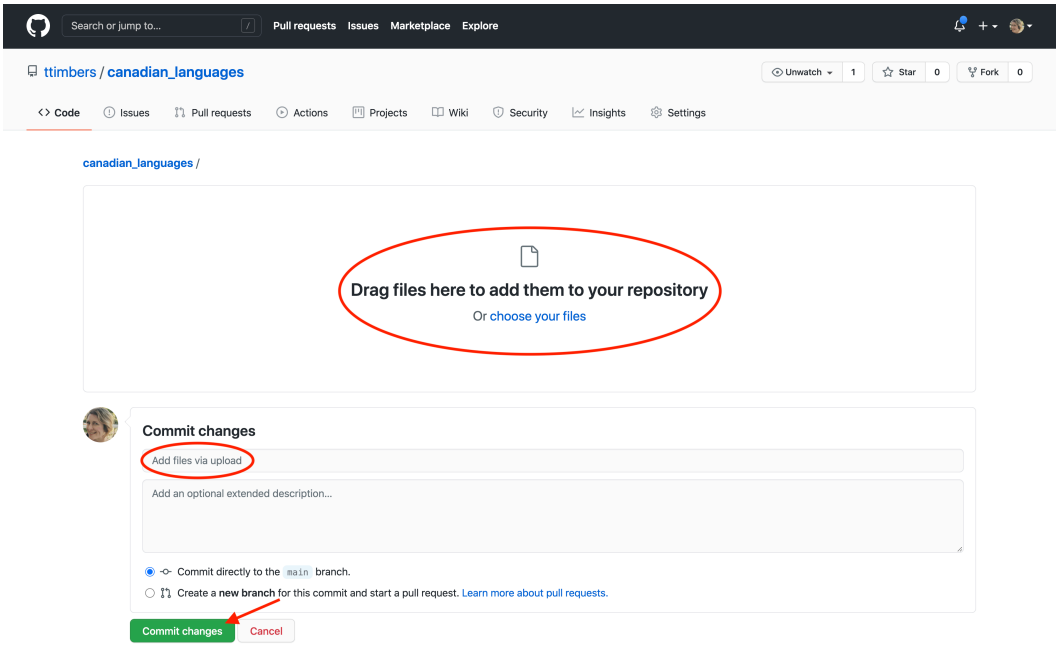


FIGURE 12.18 Specify files to upload by dragging them into the GitHub website (red circle) or by clicking on "choose your files". Uploaded files are also required to be committed along with an associated commit message.

Note that Git and GitHub are designed to track changes in individual files. **Do not** upload your whole project in an archive file (e.g., `.zip`). If you do, then Git can only keep track of changes to the entire `.zip` file, which will not be human-readable. Committing one big archive defeats the whole purpose of using version control: you won't be able to see, interpret, or find changes in the history of any of the actual content of your project.

12.7 Working with local repositories using Jupyter

Although there are several ways to create and edit files on GitHub, they are not quite powerful enough for efficiently creating and editing complex files, or files that need to be executed to assess whether they work (e.g., files containing code). For example, you wouldn't be able to run an analysis written with Python code directly on GitHub. Thus, it is useful to be able to connect the remote repository that was created on GitHub to a local coding environment. This can be done by creating and working in a local copy of the repository. In this chapter, we focus on interacting with Git via Jupyter using the Jupyter Git extension. The Jupyter Git extension can be run by Jupyter on your local computer, or on a JupyterHub server. We recommend reading Chapter 11 to learn how to use Jupyter before reading this chapter.

12.7.1 Generating a GitHub personal access token

To send and retrieve work between your local repository and the remote repository on GitHub, you will frequently need to authenticate with GitHub to prove you have the required permission. There are several methods to do this, but for beginners we recommend using the HTTPS method because it is easier and requires less setup. In order to use the HTTPS method, GitHub requires you to provide a *personal access token*. A personal access token is like a password—so keep it a secret!—but it gives you more fine-grained control over what parts of your account the token can be used to access, and lets you set an expiry date for the authentication. To generate a personal access token, you must first visit https://github.com/settings/tokens, which will take you to the "Personal access tokens" page in your account settings. Once there, click "Generate new token" (Fig. 12.19). Note that you may be asked to re-authenticate with your username and password to proceed.

You will be asked to add a note to describe the purpose for your personal access token. Next, you need to select permissions for the token; this is where you can control what parts of your account the token can be used to access. Make sure to choose only those permissions that you absolutely require. In

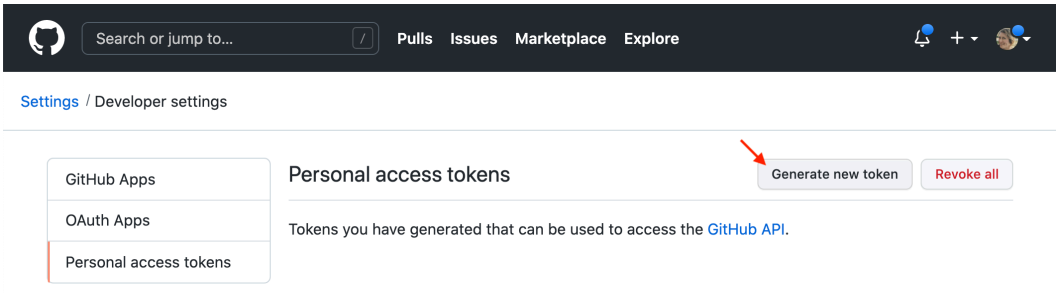


FIGURE 12.19 The "Generate new token" button used to initiate the creation of a new personal access token. It is found in the "Personal access tokens" section of the "Developer settings" page in your account settings.

Fig. 12.20, we tick only the "repo" box, which gives the token access to our repositories (so that we can push and pull) but none of our other GitHub account features. Finally, to generate the token, scroll to the bottom of that page and click the green "Generate token" button (Fig. 12.20).

Finally, you will be taken to a page where you will be able to see and copy the personal access token you just generated (Fig. 12.21). Since it provides access to certain parts of your account, you should treat this token like a password; for example, you should consider securely storing it (and your other passwords and tokens, too!) using a password manager. Note that this page will only display the token to you once, so make sure you store it in a safe place right away. If you accidentally forget to store it, though, do not fret—you can delete that token by clicking the "Delete" button next to your token, and generate a new one from scratch. To learn more about GitHub authentication, see the additional resources section at the end of this chapter.

12.7.2 Cloning a repository using Jupyter

Cloning a remote repository from GitHub to create a local repository results in a copy that knows where it was obtained from so that it knows where to send/receive new committed edits. In order to do this, first copy the URL from the HTTPS tab of the Code drop-down menu on GitHub (Fig. 12.22).

Open Jupyter, and click the Git+ icon on the file browser tab (Fig. 12.23).

Paste the URL of the GitHub project repository you created and click the blue "CLONE" button (Fig. 12.24).

On the file browser tab, you will now see a folder for the repository. Inside this folder will be all the files that existed on GitHub (Fig. 12.25).

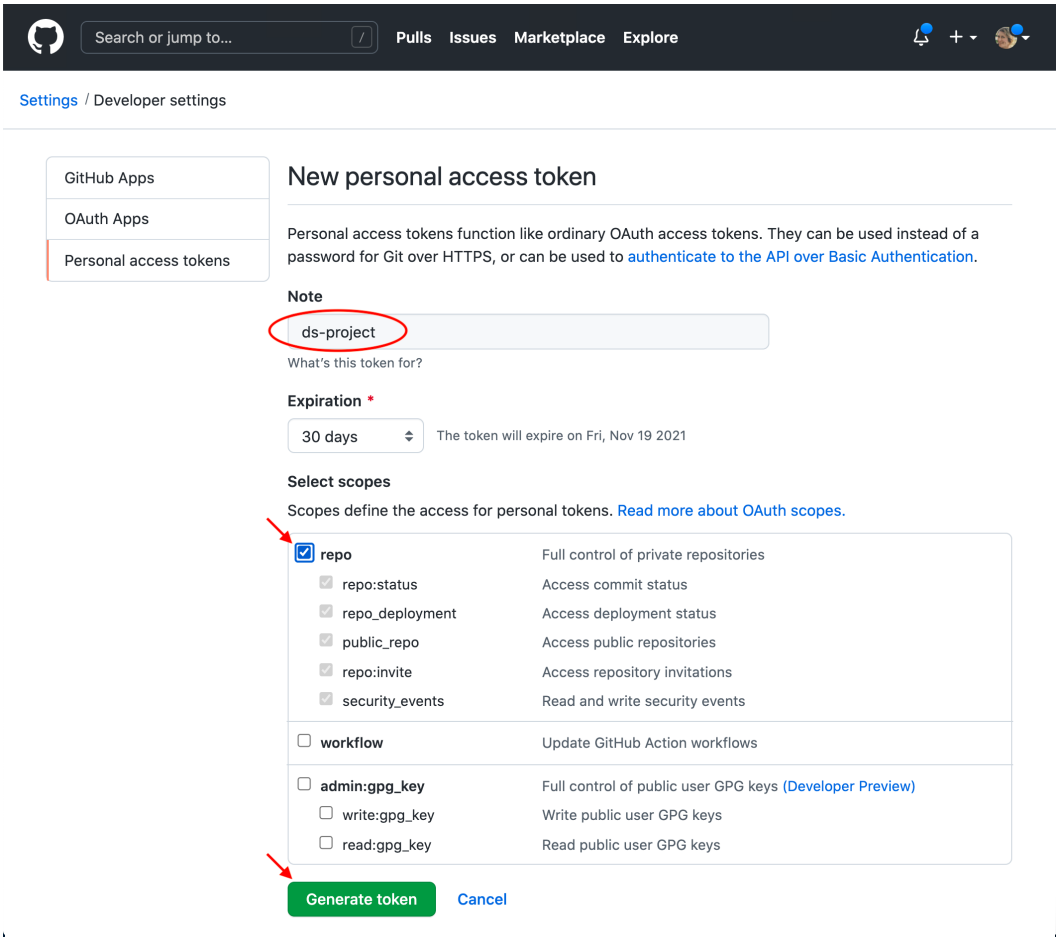


FIGURE 12.20 Webpage for creating a new personal access token.

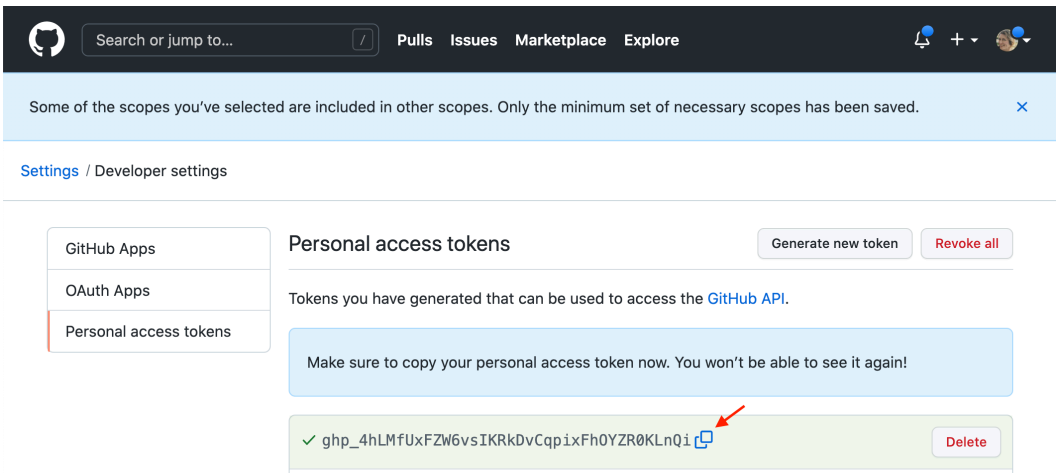


FIGURE 12.21 Display of the newly generated personal access token.

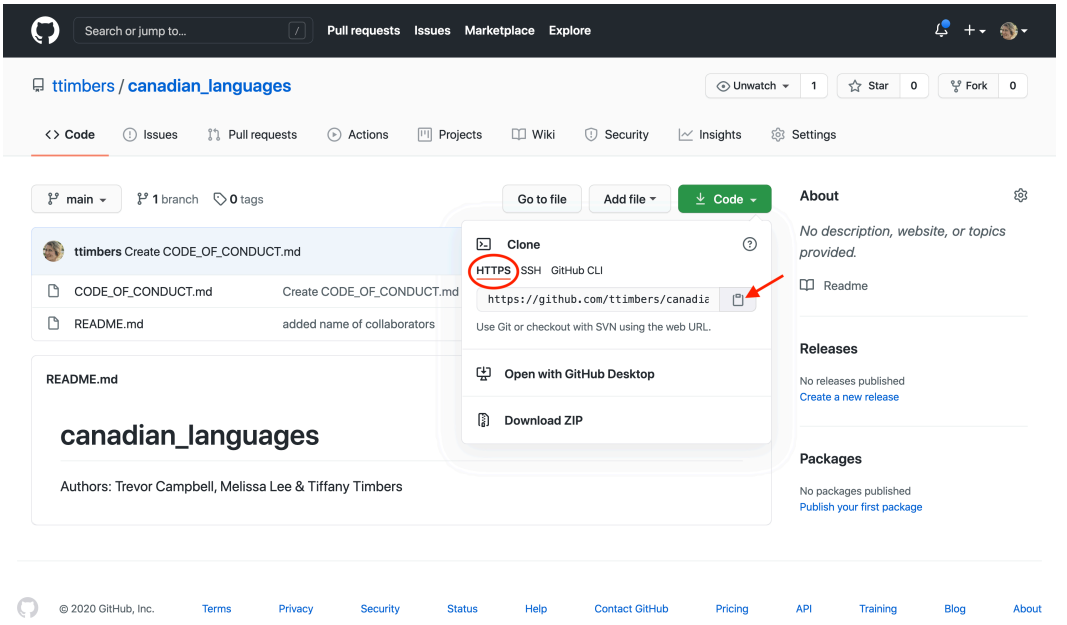


FIGURE 12.22 The green "Code" drop-down menu contains the remote address (URL) corresponding to the location of the remote GitHub repository.

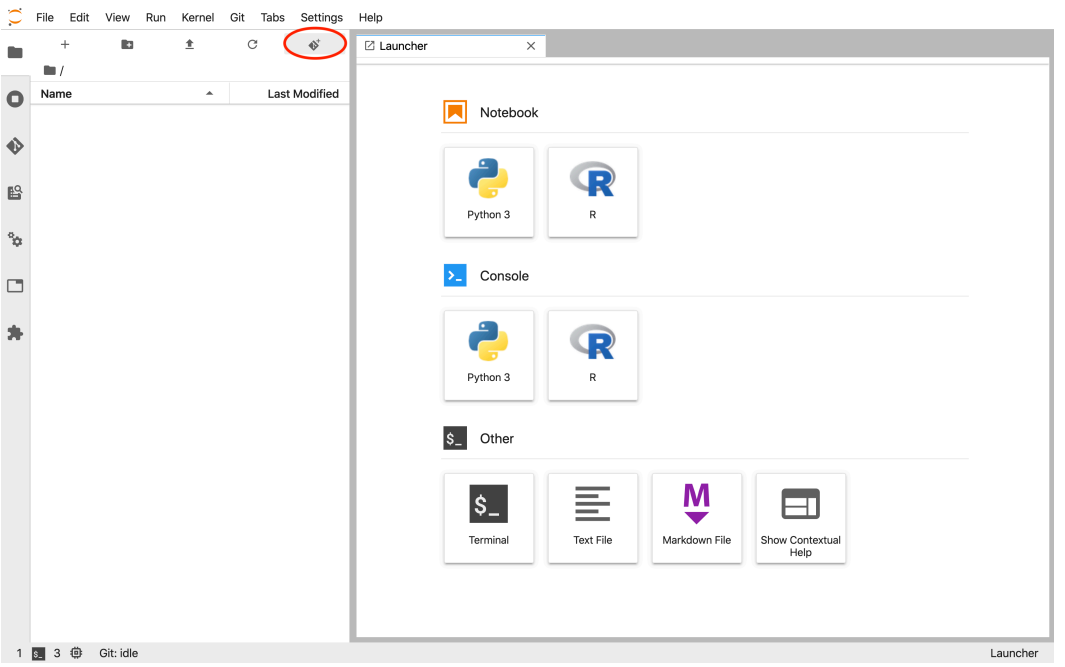


FIGURE 12.23 The Jupyter Git Clone icon (red circle).

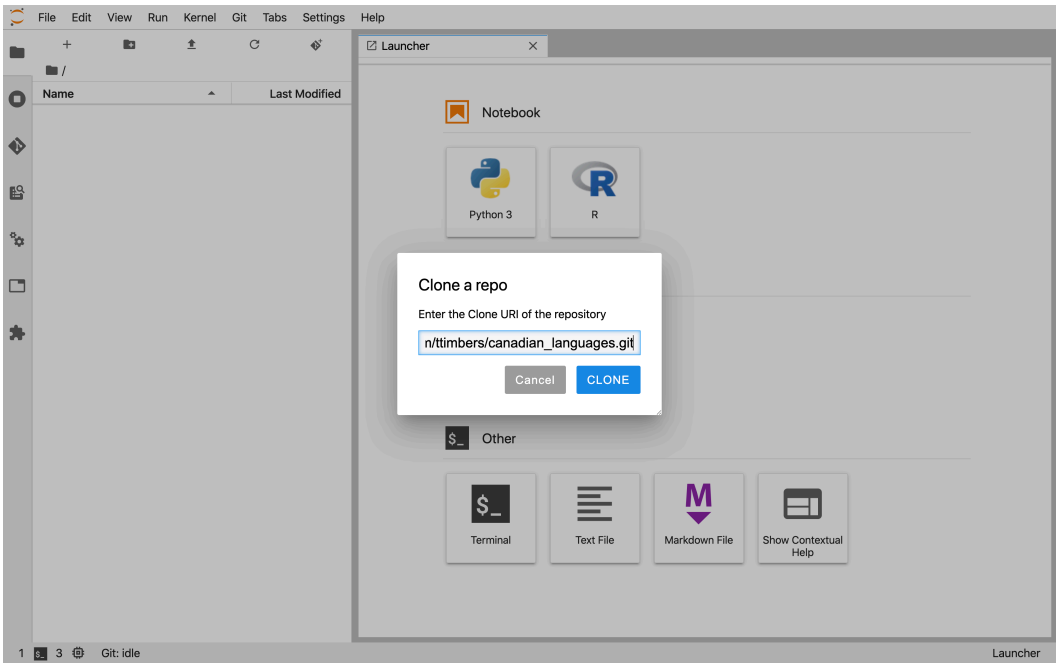


FIGURE 12.24 Prompt where the remote address (URL) corresponding to the location of the GitHub repository needs to be input in Jupyter.

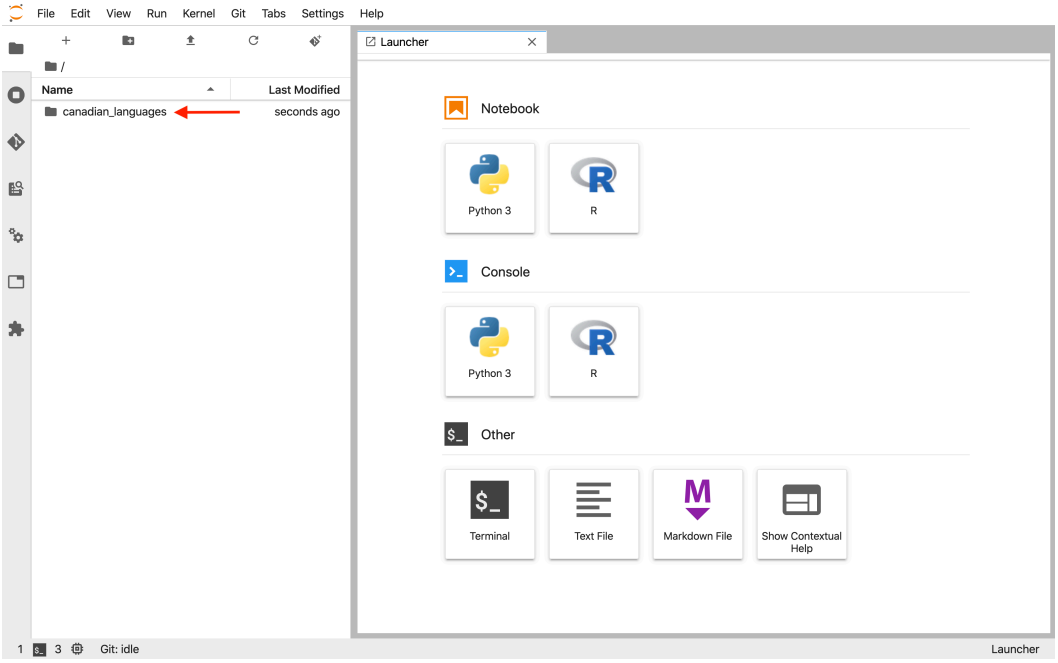


FIGURE 12.25 Cloned GitHub repositories can been seen and accessed via the Jupyter file browser.

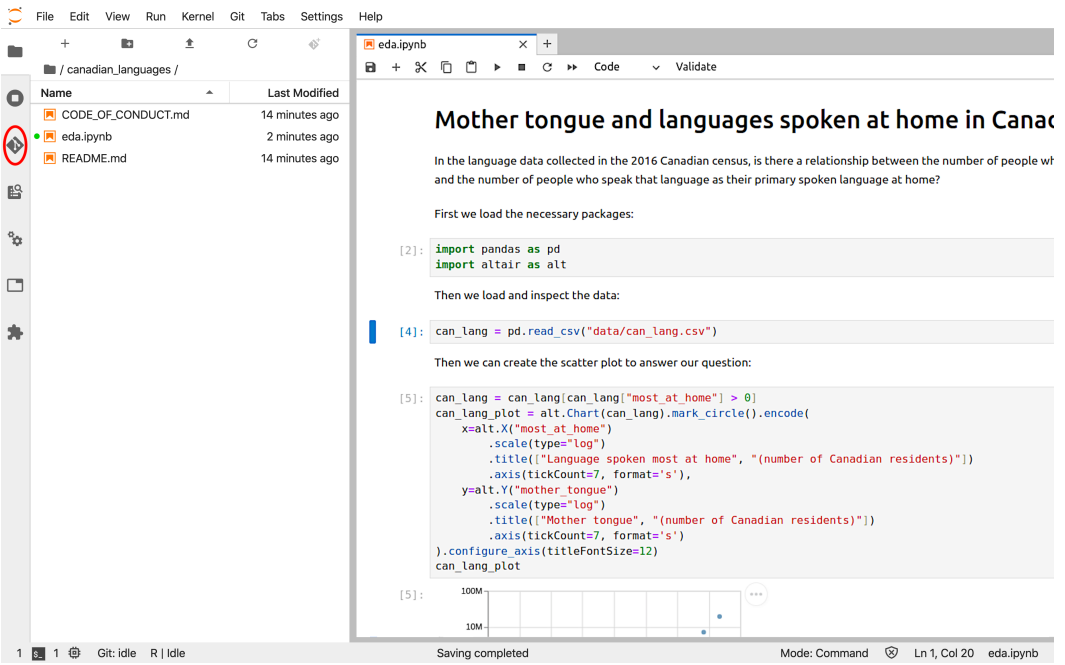


FIGURE 12.26 Jupyter Git extension icon (circled in red).

12.7.3 Specifying files to commit

Now that you have cloned the remote repository from GitHub to create a local repository, you can get to work editing, creating, and deleting files. For example, suppose you created and saved a new file (named `eda.ipynb`) that you would like to send back to the project repository on GitHub (Fig. 12.26). To "add" this modified file to the staging area (i.e., flag that this is a file whose changes we would like to commit), click the Jupyter Git extension icon on the far left-hand side of Jupyter (Fig. 12.26).

This opens the Jupyter Git graphical user interface pane. Next, click the plus sign (+) beside the file(s) that you want to "add" (Fig. 12.27). Note that because this is the first change for this file, it falls under the "Untracked" heading. However, next time you edit this file and want to add the changes, you will find it under the "Changed" heading.

You will also see an `eda-checkpoint.ipynb` file under the "Untracked" heading. This is a temporary "checkpoint file" created by Jupyter when you work on `eda.ipynb`. You generally do not want to add auto-generated files to Git repositories; only add the files you directly create and edit.

Clicking the plus sign (+) moves the file from the "Untracked" heading to the "Staged" heading, so that Git knows you want a snapshot of its current state as a commit (Fig. 12.28). Now you are ready to "commit" the changes. Make

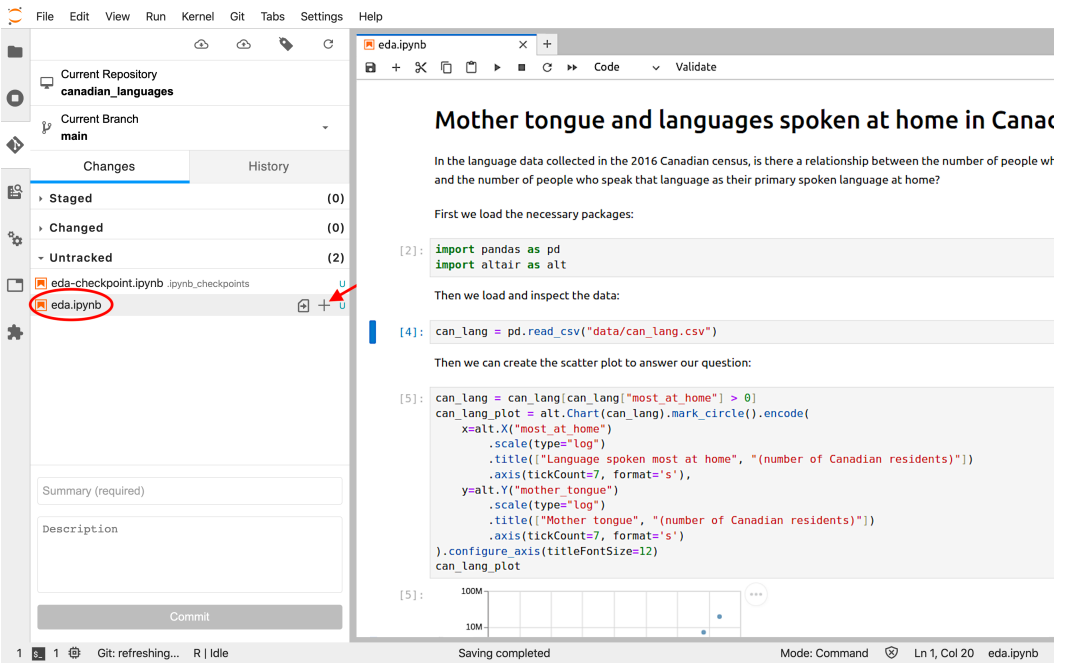


FIGURE 12.27 `eda.ipynb` is added to the staging area via the plus sign (+).

sure to include a (clear and helpful!) message about what was changed so that your collaborators (and future you) know what happened in this commit.

12.7.4 Making the commit

To snapshot the changes with an associated commit message, you must put a message in the text box at the bottom of the Git pane and click on the blue "Commit" button (Fig. 12.29). It is highly recommended to write useful and meaningful messages about what was changed. These commit messages, and the datetime stamp for a given commit, are the primary means to navigate through the project's history in the event that you need to view or retrieve a past version of a file, or revert your project to an earlier state. When you click the "Commit" button for the first time, you will be prompted to enter your name and email. This only needs to be done once for each machine you use Git on.

After "committing" the file(s), you will see there are 0 "Staged" files. You are now ready to push your changes to the remote repository on GitHub (Fig. 12.30).

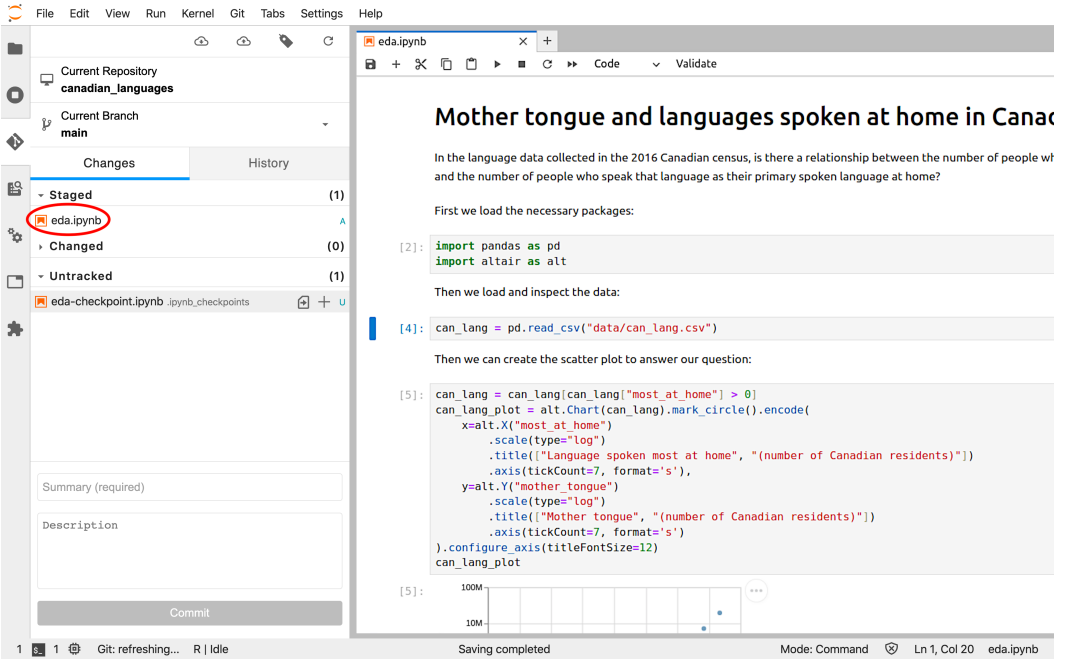


FIGURE 12.28 Adding `eda.ipynb` makes it visible in the staging area.

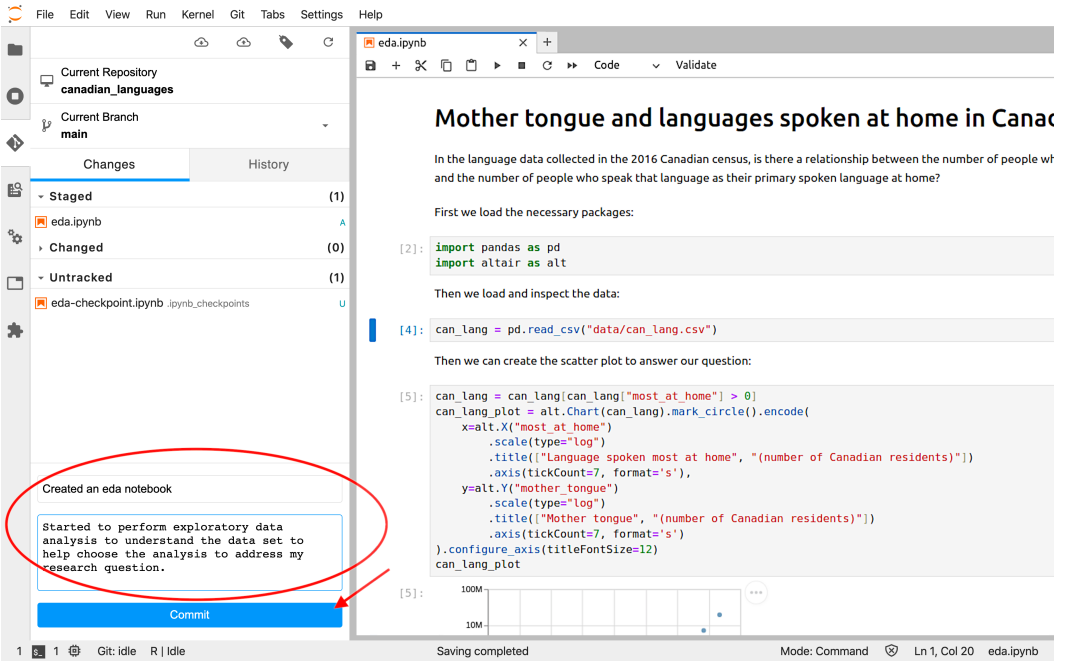


FIGURE 12.29 A commit message must be added into the Jupyter Git extension commit text box before the blue Commit button can be used to record the commit.

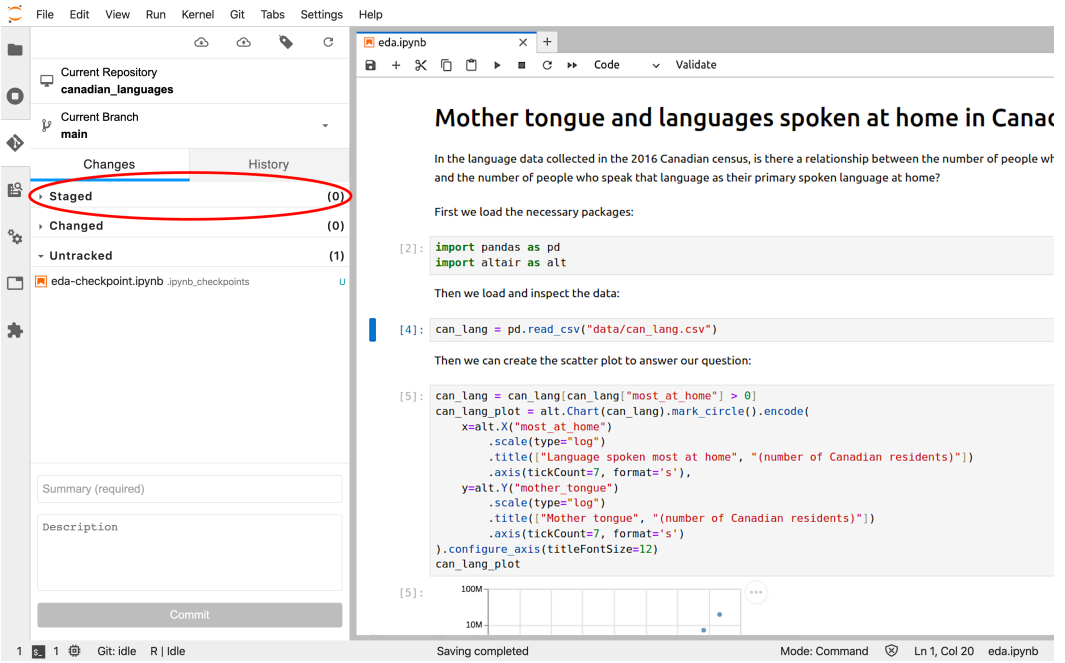


FIGURE 12.30 After recording a commit, the staging area should be empty.

12.7.5 Pushing the commits to GitHub

To send the committed changes back to the remote repository on GitHub, you need to *push* them. To do this, click on the cloud icon with the up arrow on the Jupyter Git tab (Fig. 12.31).

You will then be prompted to enter your GitHub username and the personal access token that you generated earlier (not your account password!). Click the blue "OK" button to initiate the push (Fig. 12.32).

If the files were successfully pushed to the project repository on GitHub, you will be shown a success message (Fig. 12.33). Click "Dismiss" to continue working in Jupyter.

If you visit the remote repository on GitHub, you will see that the changes now exist there too (Fig. 12.34).

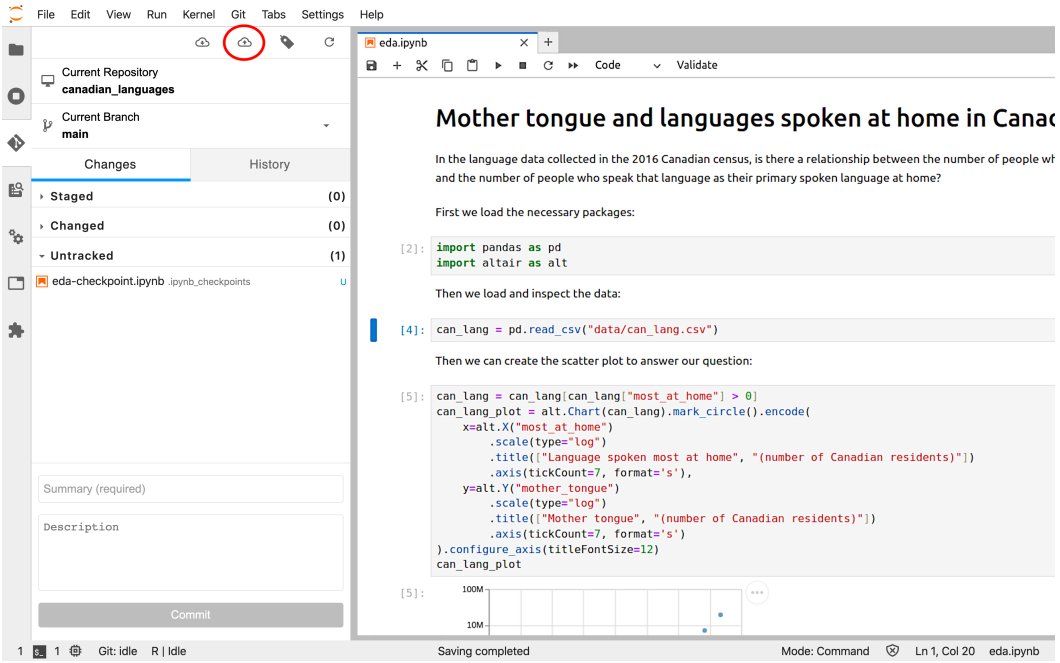


FIGURE 12.31 The Jupyter Git extension "push" button (circled in red).

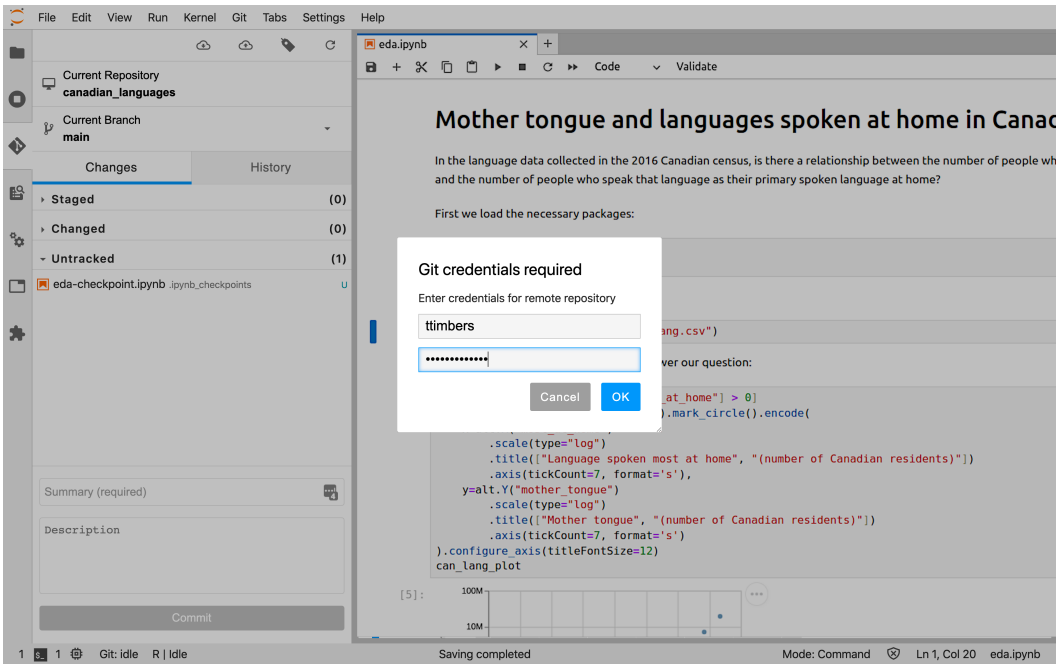


FIGURE 12.32 Enter your Git credentials to authorize the push to the remote repository.

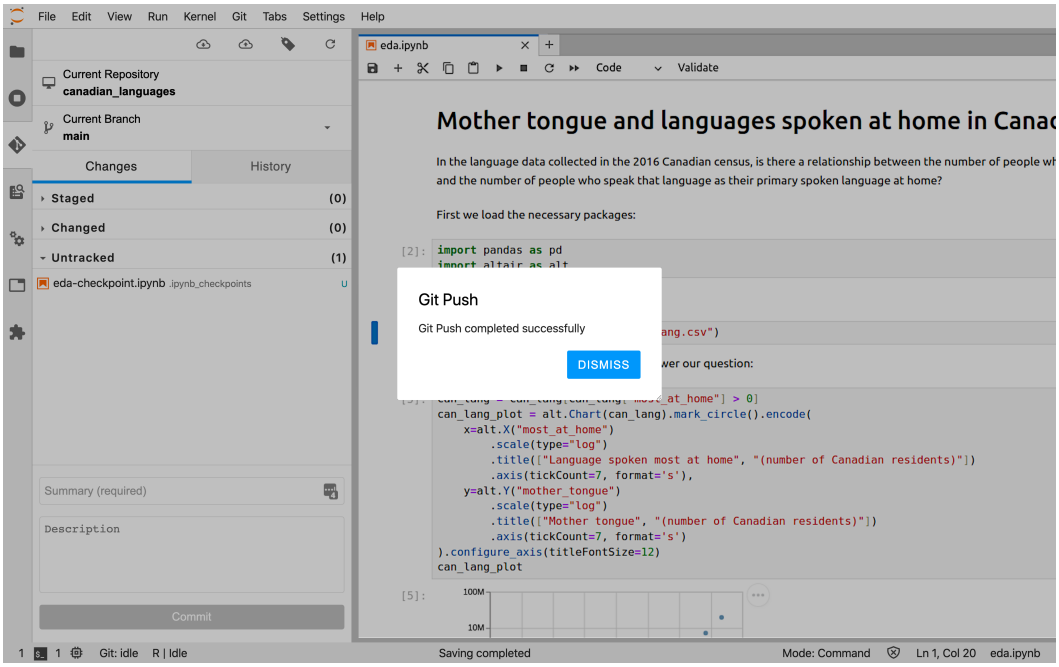


FIGURE 12.33 The prompt that the push was successful.

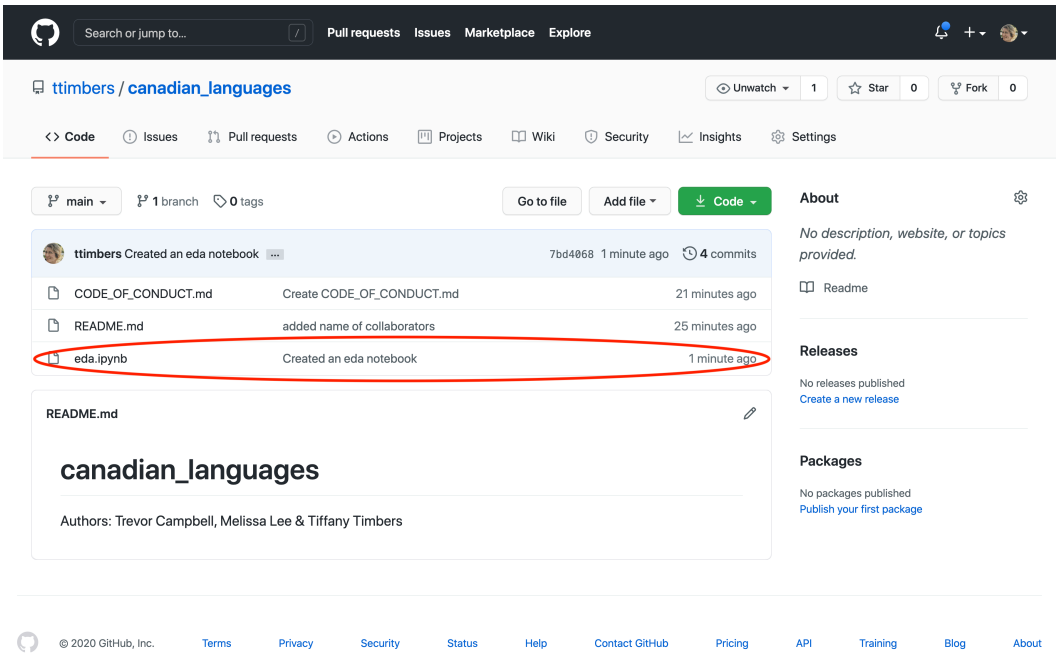


FIGURE 12.34 The GitHub web interface shows a preview of the commit message, and the time of the most recently pushed commit for each file.

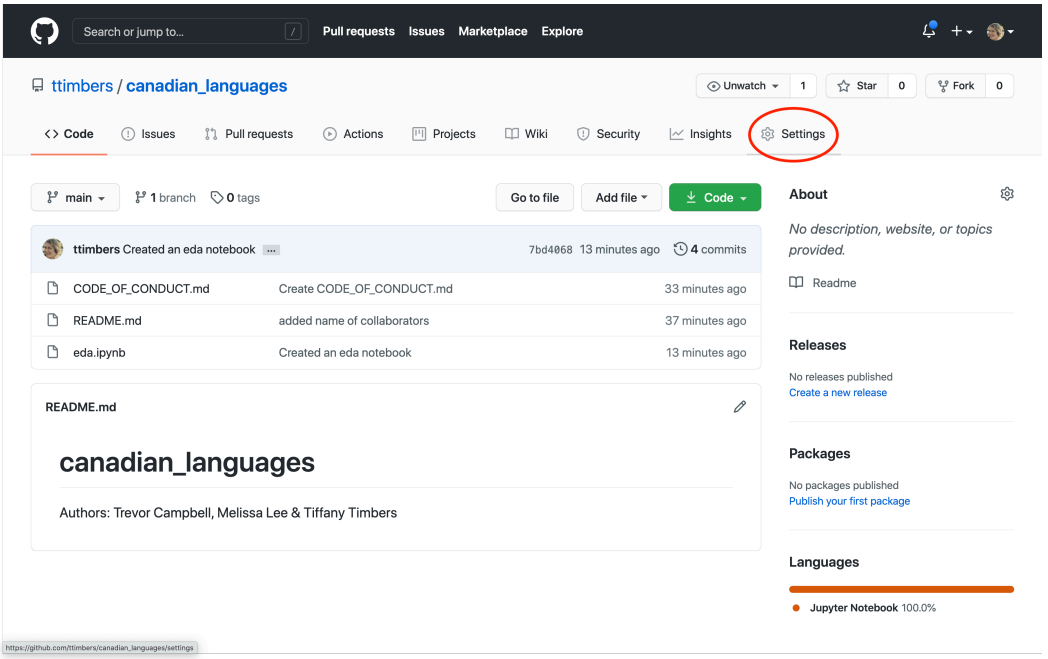


FIGURE 12.35 The "Settings" tab on the GitHub web interface.

12.8 Collaboration

12.8.1 Giving collaborators access to your project

As mentioned earlier, GitHub allows you to control who has access to your project. The default of both public and private projects are that only the person who created the GitHub repository has permissions to create, edit and delete files (*write access*). To give your collaborators write access to the projects, navigate to the "Settings" tab (Fig. 12.35).

Then click "Manage access" (Fig. 12.36).

Then click the green "Invite a collaborator" button (Fig. 12.37).

Type in the collaborator's GitHub username or email, and select their name when it appears (Fig. 12.38).

Finally, click the green "Add <COLLABORATORS_GITHUB_USER_NAME> to this repository" button (Fig. 12.39).

After this, you should see your newly added collaborator listed under the "Manage access" tab. They should receive an email invitation to join the

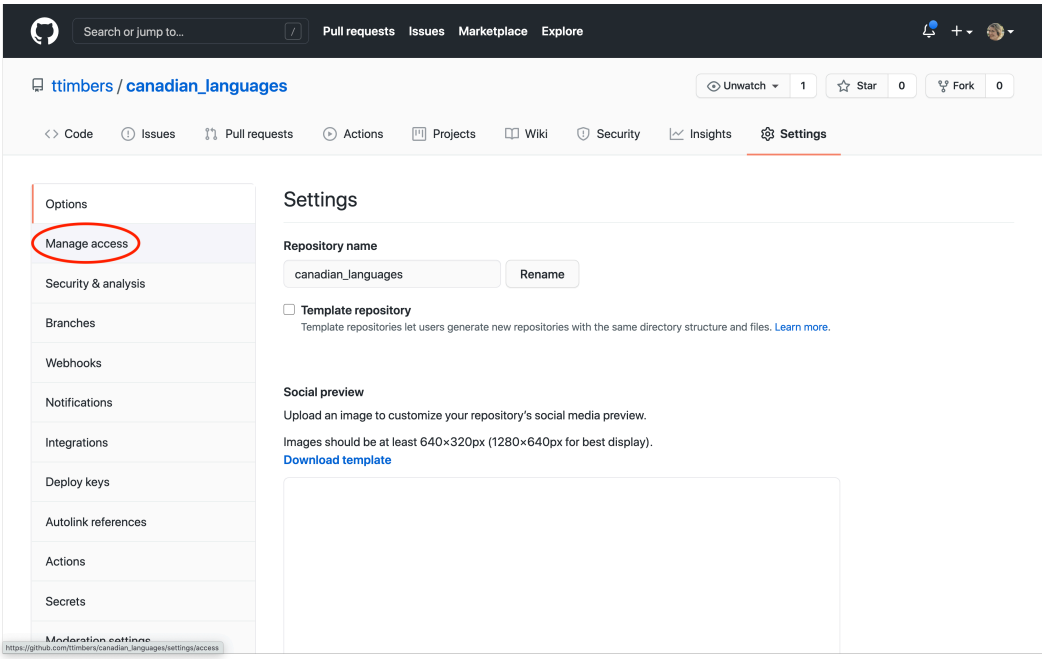


FIGURE 12.36 The "Manage access" tab on the GitHub web interface.

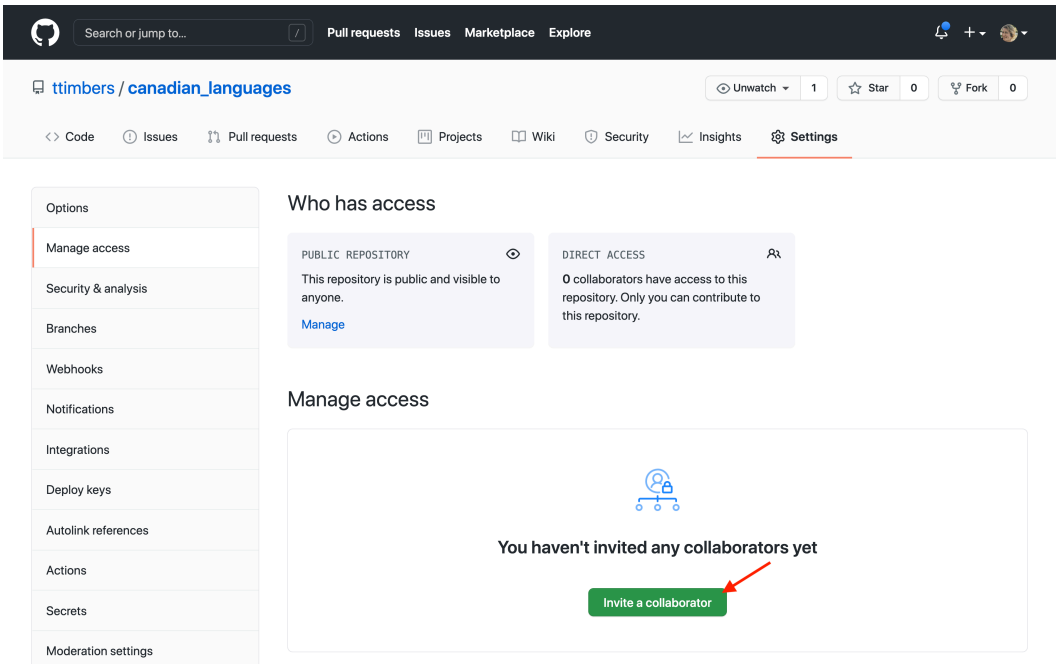


FIGURE 12.37 The "Invite a collaborator" button on the GitHub web interface.

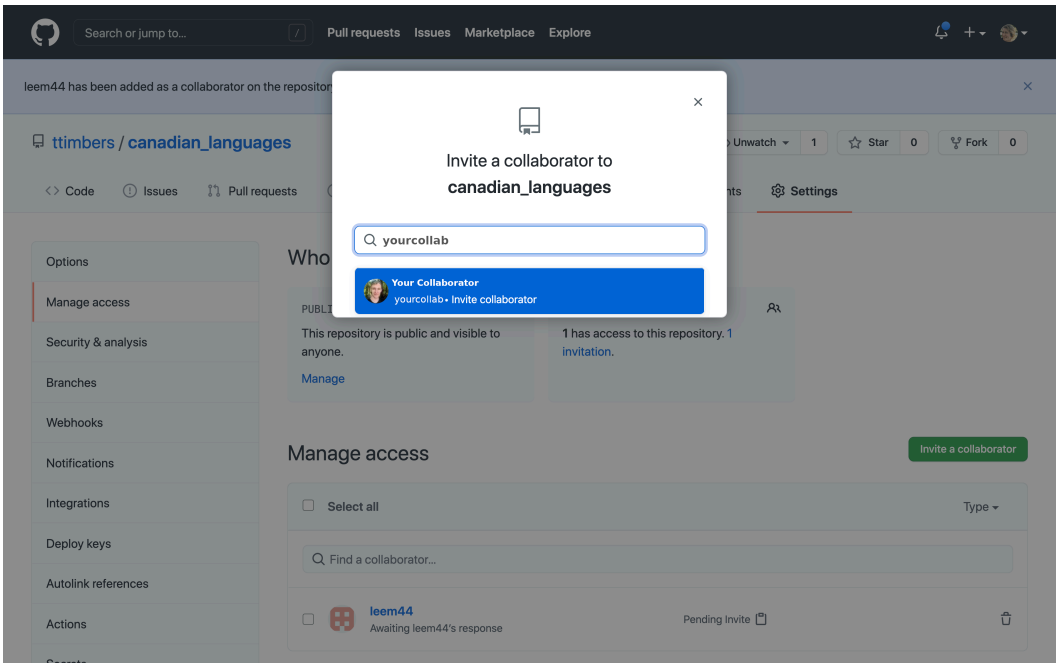


FIGURE 12.38 The text box where a collaborator's GitHub username or email can be entered.

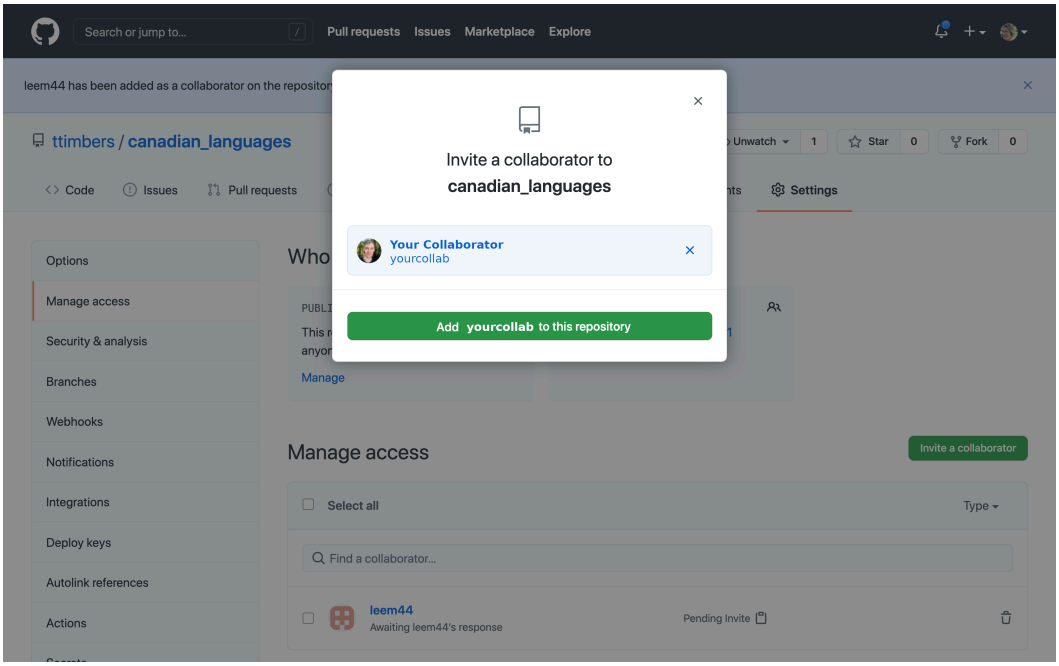


FIGURE 12.39 The confirmation button for adding a collaborator to a repository on the GitHub web interface.

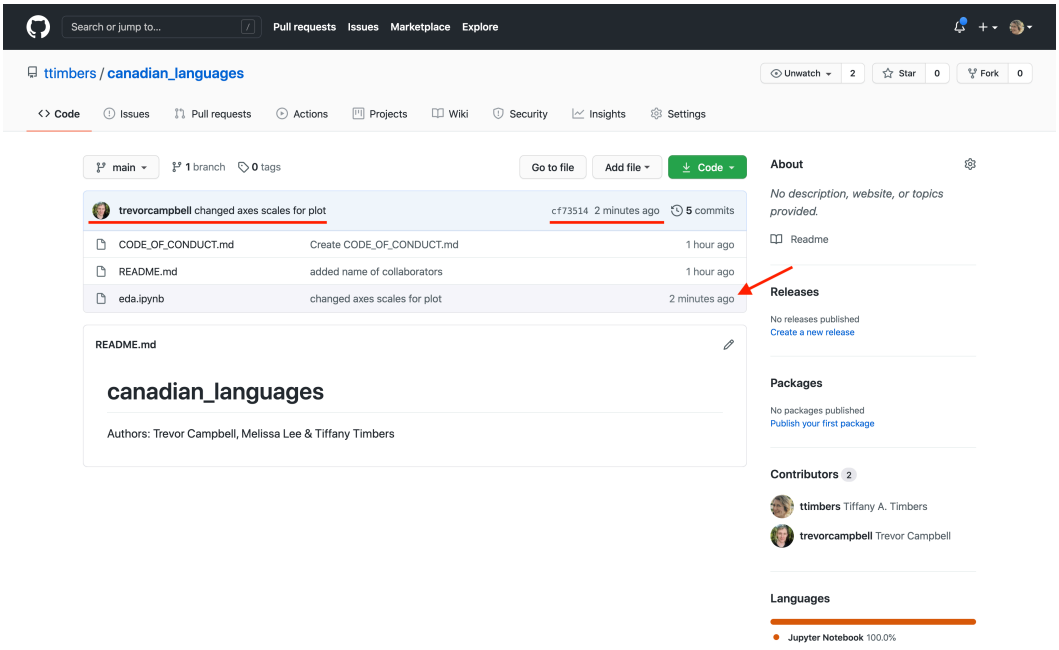


FIGURE 12.40 The GitHub interface indicates the name of the last person to push a commit to the remote repository, a preview of the associated commit message, the unique commit identifier, and how long ago the commit was snapshotted.

GitHub repository as a collaborator. They need to accept this invitation to enable write access.

12.8.2 Pulling changes from GitHub using Jupyter

We will now walk through how to use the Jupyter Git extension tool to pull changes to our `eda.ipynb` analysis file that were made by a collaborator (Fig. 12.40).

You can tell Git to "pull" by clicking on the cloud icon with the down arrow in Jupyter (Fig. 12.41).

Once the files are successfully pulled from GitHub, you need to click "Dismiss" to keep working (Fig. 12.42).

And then when you open (or refresh) the files whose changes you just pulled, you should be able to see them (Fig. 12.43).

It can be very useful to review the history of the changes to your project. You can do this directly in Jupyter by clicking "History" in the Git tab (Fig. 12.44).

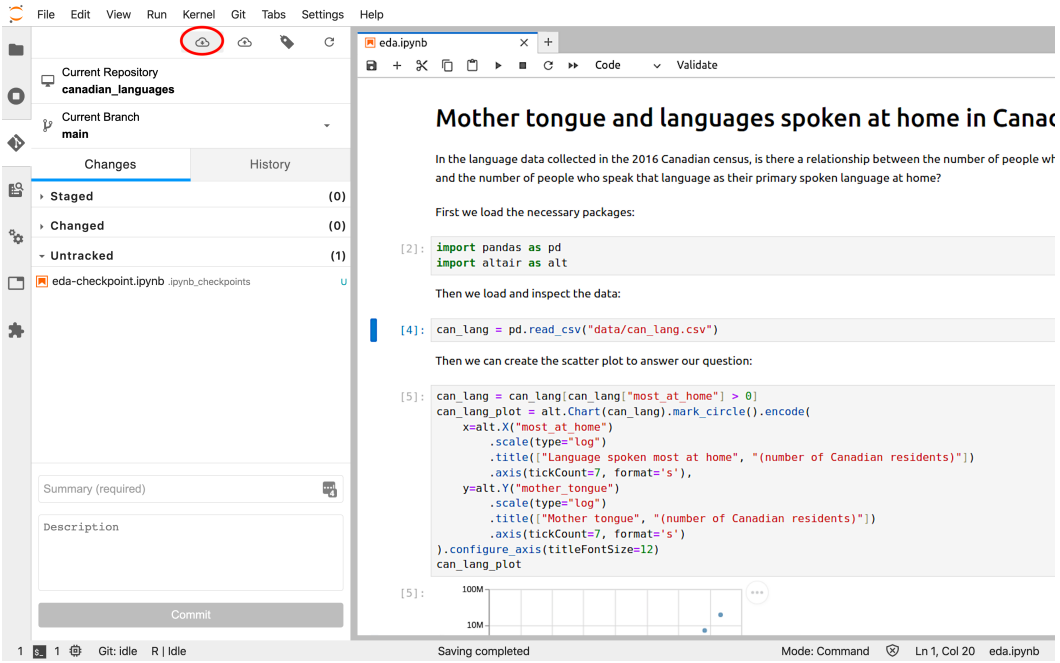


FIGURE 12.41 The Jupyter Git extension clone button.

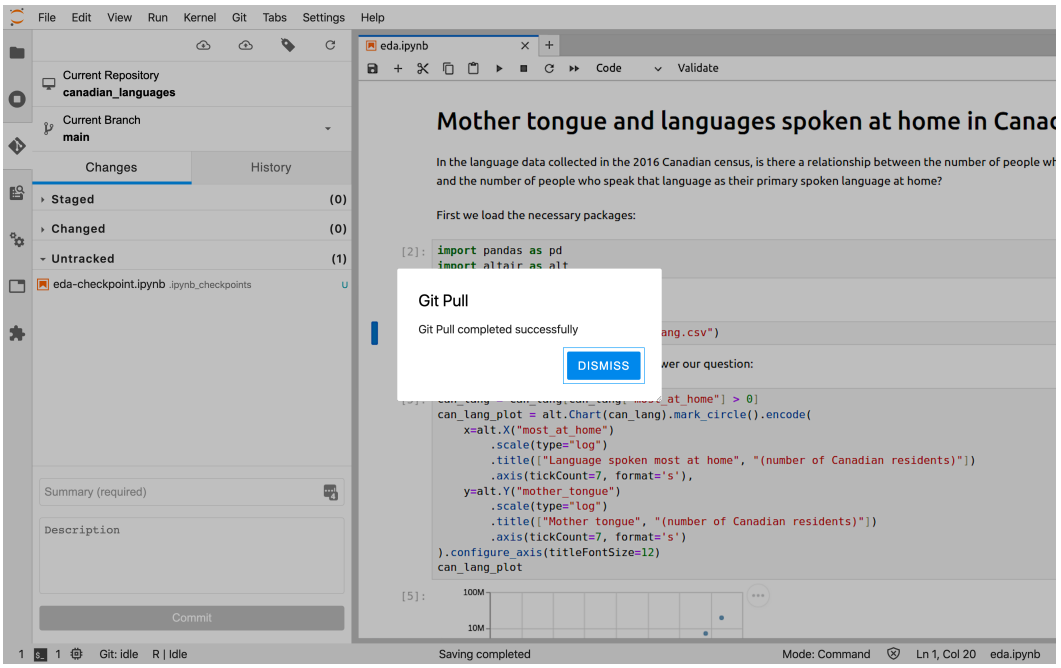


FIGURE 12.42 The prompt after changes have been successfully pulled from a remote repository.

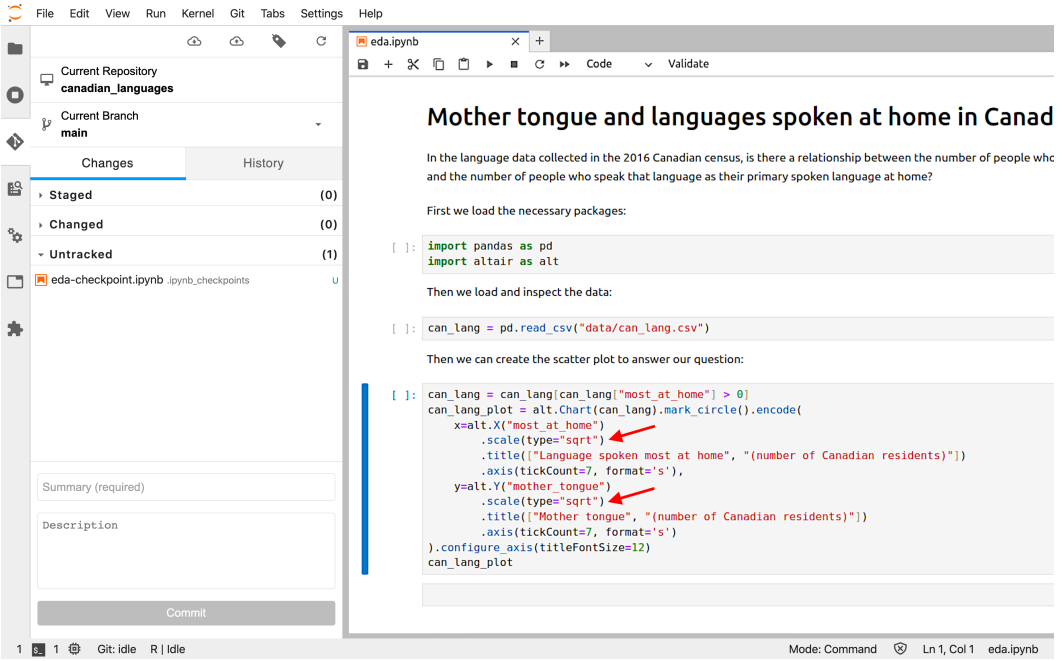


FIGURE 12.43 Changes made by the collaborator to `eda.ipynb` (code highlighted by red arrows).

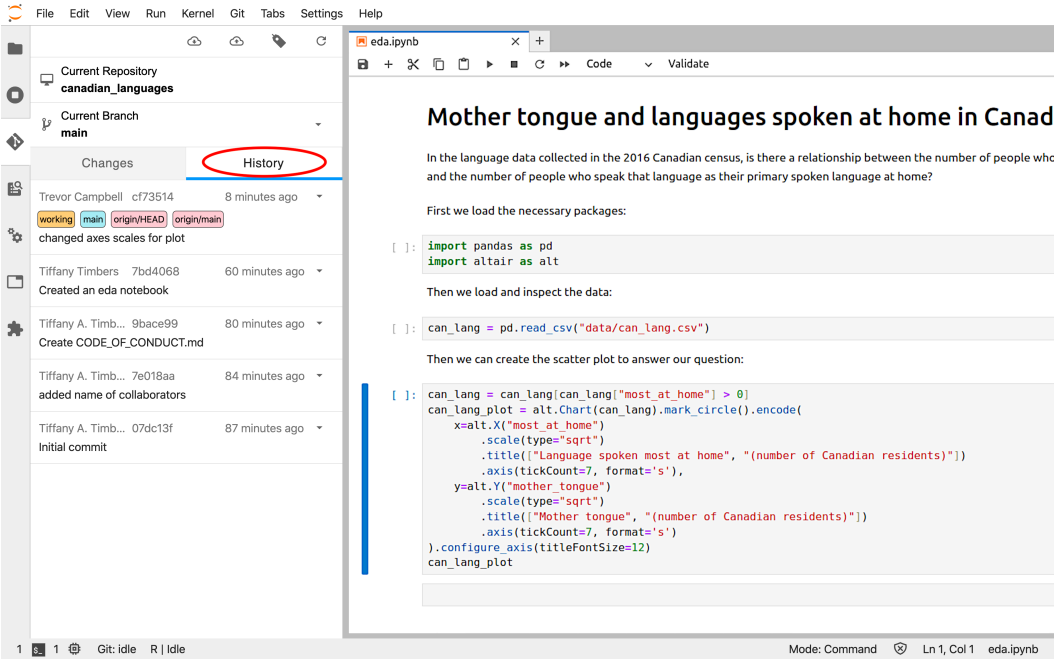


FIGURE 12.44 Version control repository history viewed using the Jupyter Git extension.

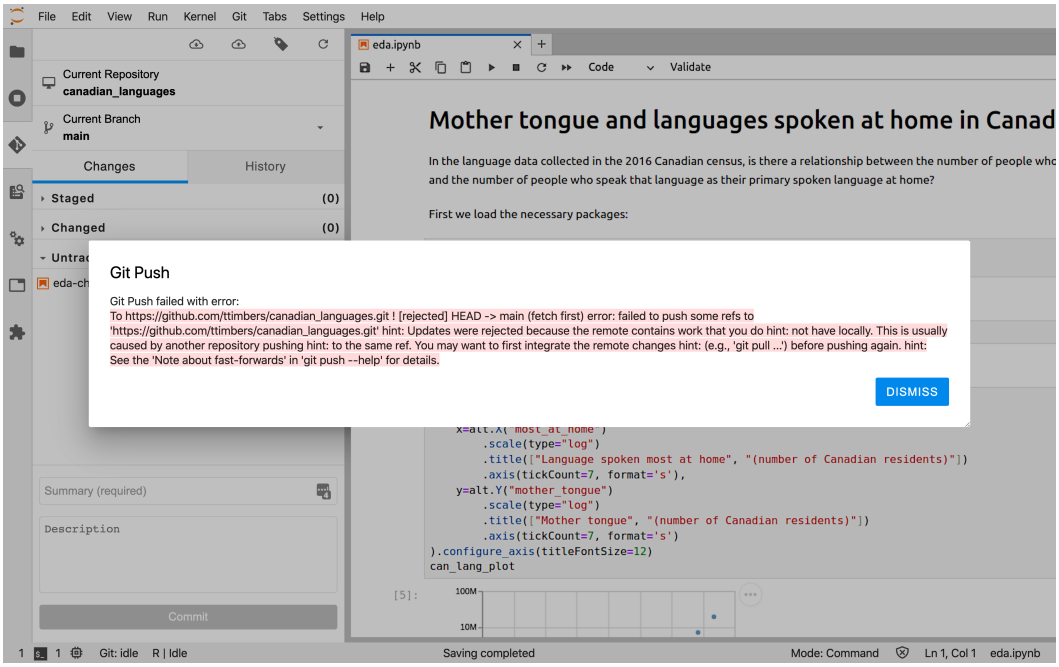


FIGURE 12.45 Error message that indicates that there are changes on the remote repository that you do not have locally.

It is good practice to pull any changes at the start of *every* work session before you start working on your local copy. If you do not do this, and your collaborators have pushed some changes to the project to GitHub, then you will be unable to push your changes to GitHub until you pull. This situation can be recognized by the error message shown in Fig. 12.45.

Usually, getting out of this situation is not too troublesome. First you need to pull the changes that exist on GitHub that you do not yet have in the local repository. Usually when this happens, Git can automatically merge the changes for you, even if you and your collaborators were working on different parts of the same file.

If, however, you and your collaborators made changes to the same line of the same file, Git will not be able to automatically merge the changes—it will not know whether to keep your version of the line(s), your collaborators version of the line(s), or some blend of the two. When this happens, Git will tell you that you have a merge conflict in certain file(s) (Fig. 12.46).

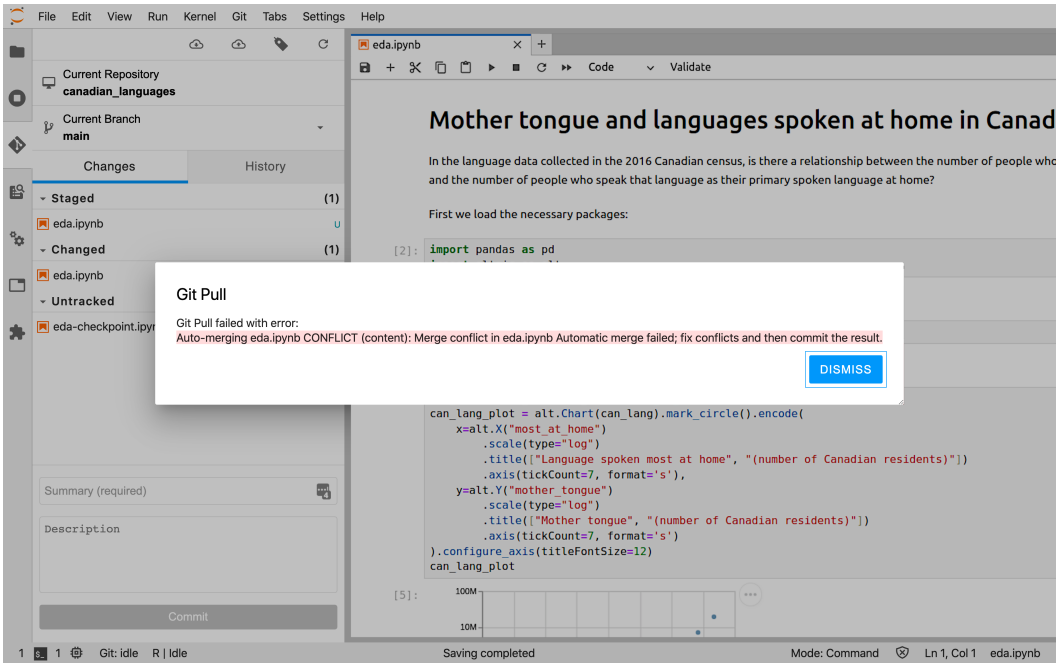


FIGURE 12.46 Error message that indicates you and your collaborators made changes to the same line of the same file and that Git will not be able to automatically merge the changes.

12.8.3 Handling merge conflicts

To fix the merge conflict, you need to open the offending file in a plain text editor and look for special marks that Git puts in the file to tell you where the merge conflict occurred (Fig. 12.47).

The beginning of the merge conflict is preceded by `<<<<<<< HEAD` and the end of the merge conflict is marked by `>>>>>>>`. Between these markings, Git also inserts a separator (`=======`). The version of the change before the separator is your change, and the version that follows the separator was the change that existed on GitHub. In Fig. 12.48, you can see that in your local repository there is a line of code that sets the axis scaling to `"sqrt"`. It looks like your collaborator made an edit to that line too, except with axis scaling `"log"`.

Once you have decided which version of the change (or what combination!) to keep, you need to use the plain text editor to remove the special marks that Git added (Fig. 12.49).

The file must be saved, added to the staging area, and then committed before you will be able to push your changes to GitHub.

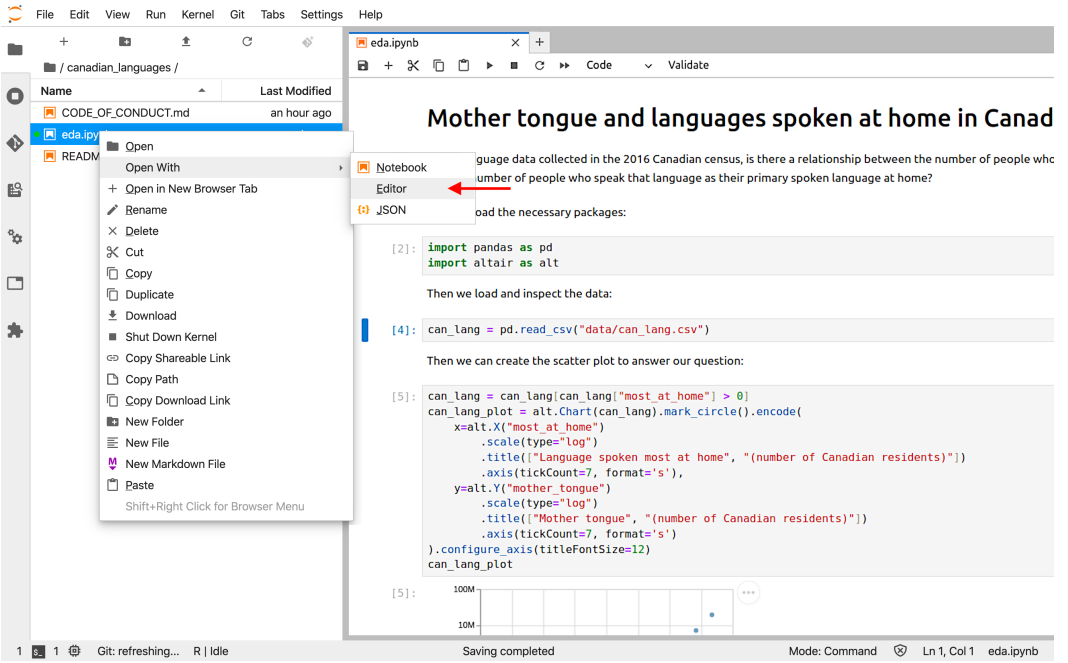


FIGURE 12.47 How to open a Jupyter notebook as a plain text file view in Jupyter.

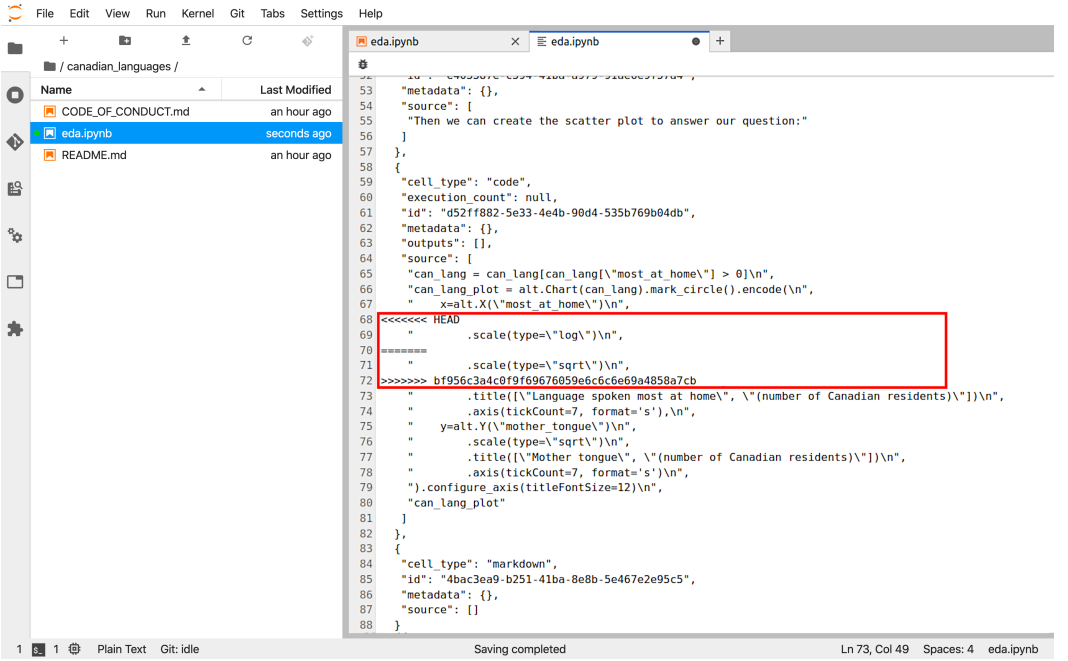


FIGURE 12.48 Merge conflict identifiers (highlighted in red).

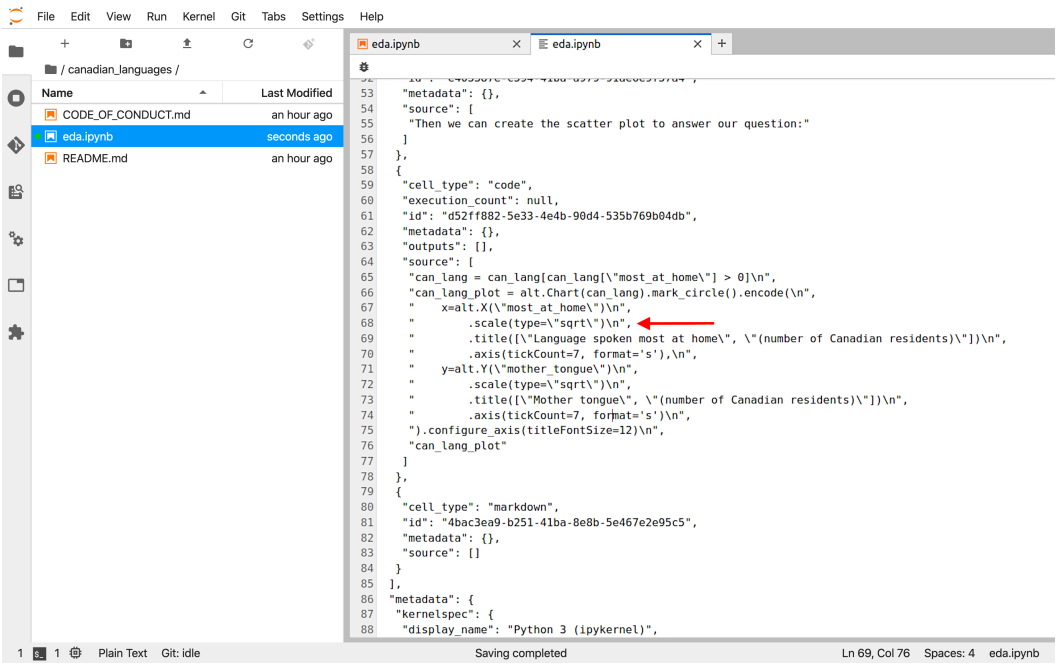


FIGURE 12.49 File where a merge conflict has been resolved.

12.8.4 Communicating using GitHub issues

When working on a project in a team, you don't just want a historical record of who changed what file and when in the project—you also want a record of decisions that were made, ideas that were floated, problems that were identified and addressed, and all other communication surrounding the project. Email and messaging apps are both very popular for general communication, but are not designed for project-specific communication: they both generally do not have facilities for organizing conversations by project subtopics, searching for conversations related to particular bugs or software versions, etc.

GitHub *issues* are an alternative written communication medium to email and messaging apps, and were designed specifically to facilitate project-specific communication. Issues are *opened* from the "Issues" tab on the project's GitHub page, and they persist there even after the conversation is over and the issue is *closed* (in contrast to email, issues are not usually deleted). One issue thread is usually created per topic, and they are easily searchable using GitHub's search tools. All issues are accessible to all project collaborators, so no one is left out of the conversation. Finally, issues can be set up so that team members get email notifications when a new issue is created or a new post is made in an issue thread. Replying to issues from email is also possible. Given all of these advantages, we highly recommend the use of issues for project-related communication.

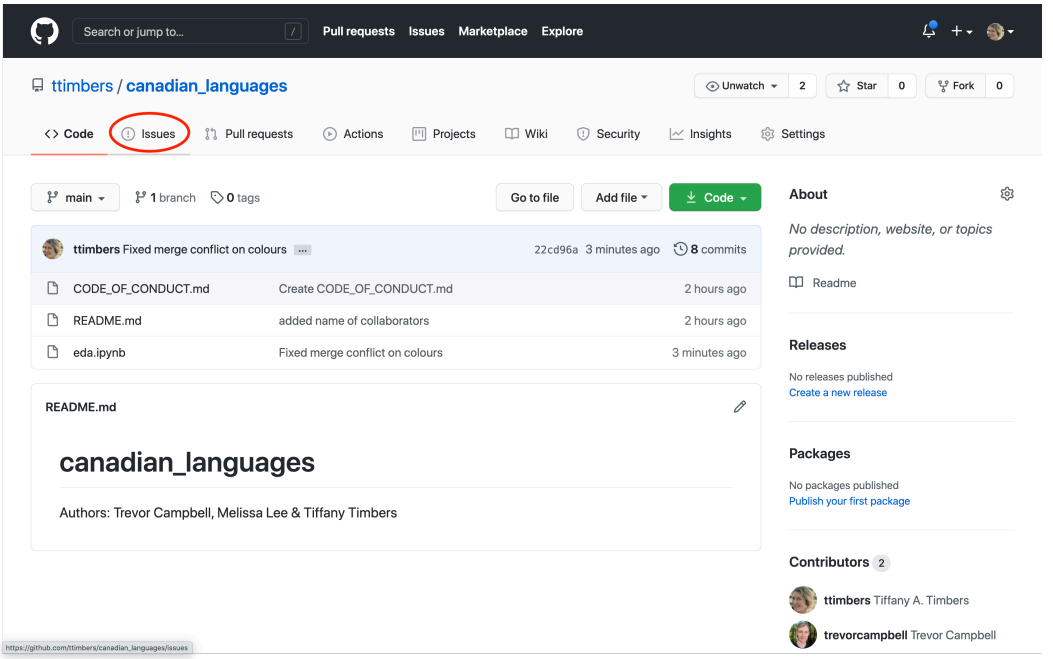


FIGURE 12.50 The "Issues" tab on the GitHub web interface.

To open a GitHub issue, first click on the "Issues" tab (Fig. 12.50).

Next click the "New issue" button (Fig. 12.51).

Add an issue title (which acts like an email subject line), and then put the body of the message in the larger text box. Finally, click "Submit new issue" to post the issue to share with others (Fig. 12.52).

You can reply to an issue that someone opened by adding your written response to the large text box and clicking comment (Fig. 12.53).

When a conversation is resolved, you can click "Close issue". The closed issue can be later viewed by clicking the "Closed" header link in the "Issue" tab (Fig. 12.54).

12.9 Exercises

Practice exercises for the material covered in this chapter can be found in the accompanying worksheets repository[4] in the "Collaboration with version control" row. You can launch an interactive version of the worksheet in your

[4] https://worksheets.python.datasciencebook.ca

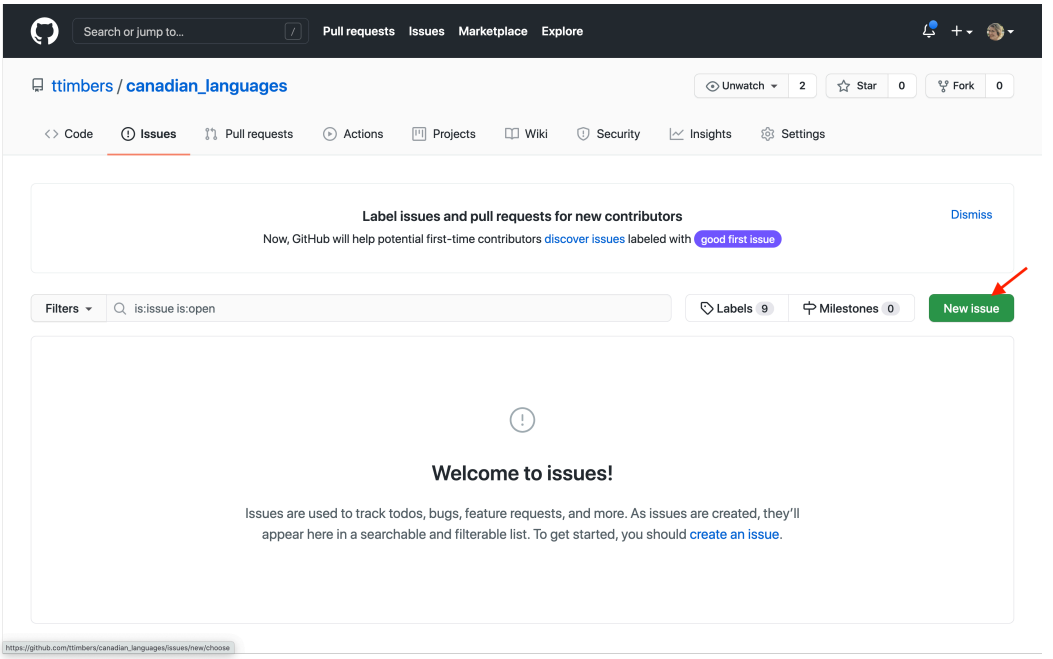


FIGURE 12.51 The "New issues" button on the GitHub web interface.

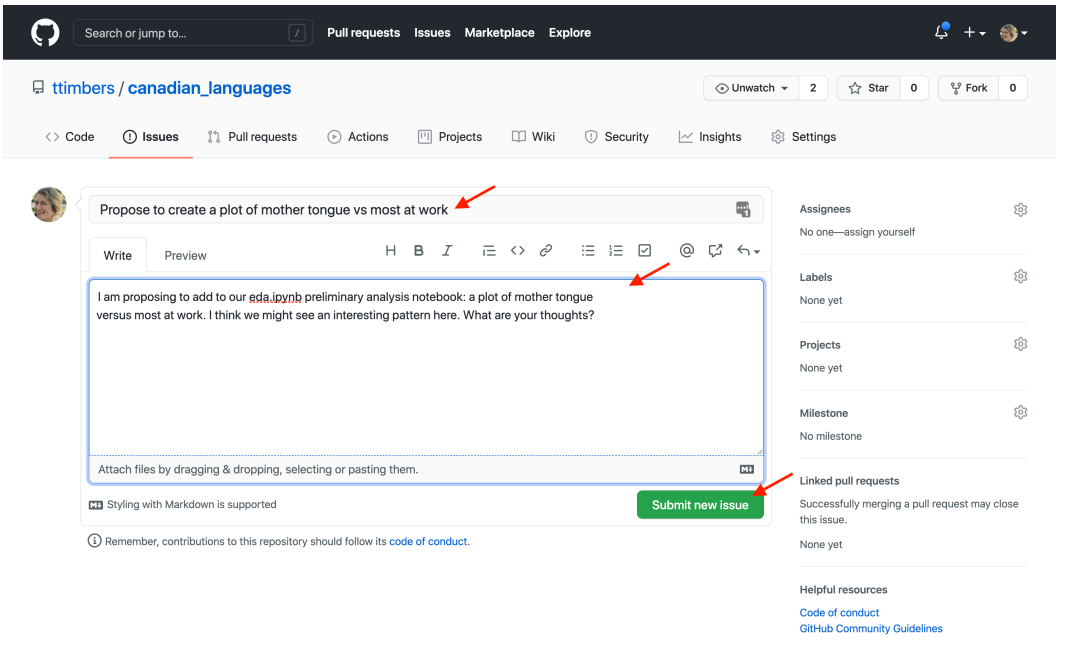


FIGURE 12.52 Dialog boxes and submission button for creating new GitHub issues.

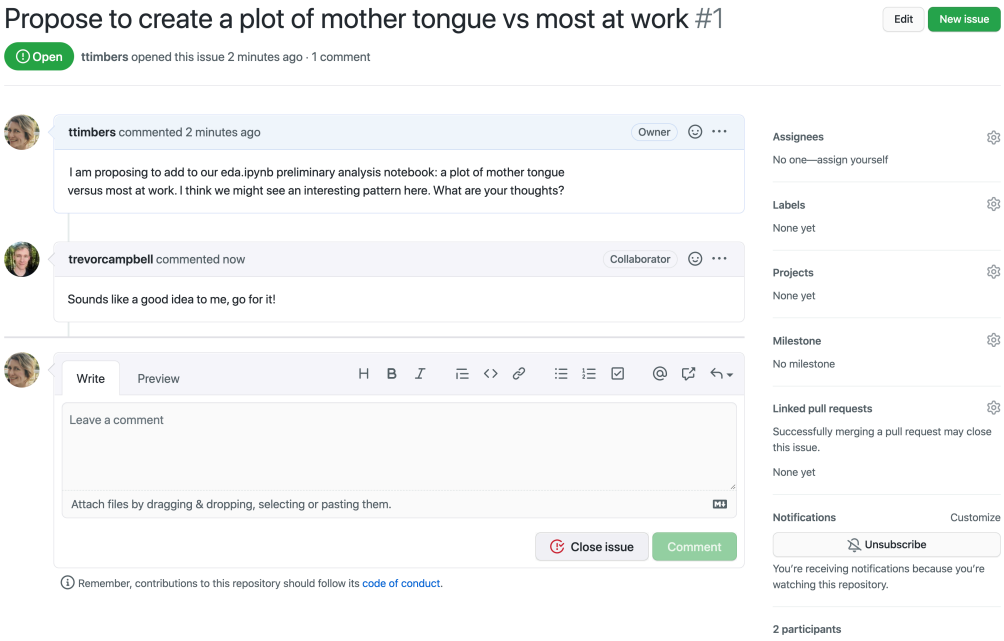


FIGURE 12.53 Dialog box for replying to GitHub issues.

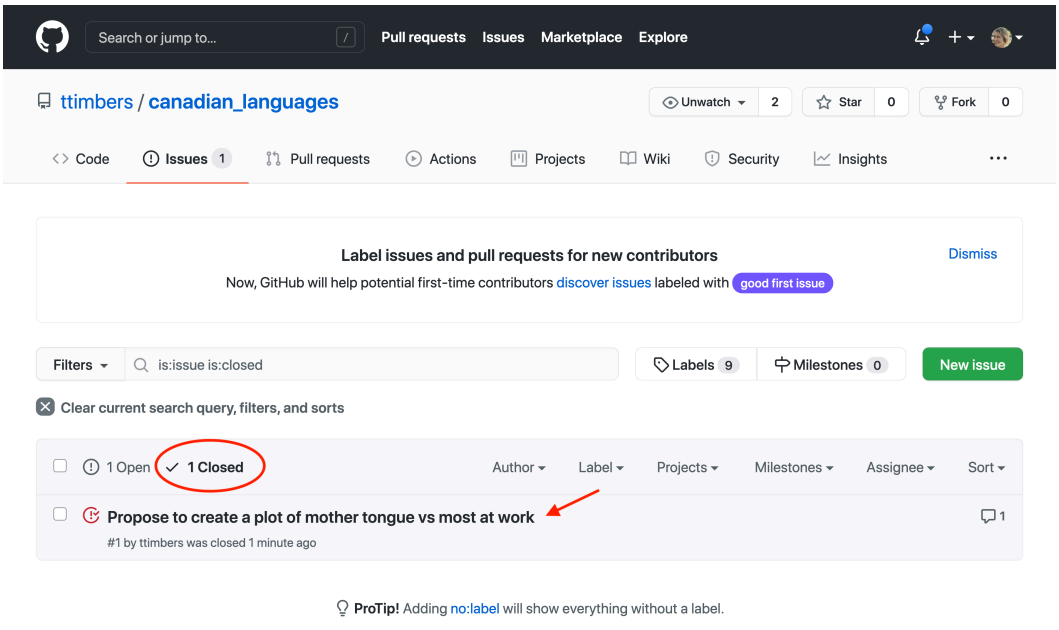


FIGURE 12.54 The "Closed" issues tab on the GitHub web interface.

browser by clicking the "launch binder" button. You can also preview a non-interactive version of the worksheet by clicking "view worksheet". If you instead decide to download the worksheet and run it on your own machine, make sure to follow the instructions for computer setup found in Chapter 13. This will ensure that the automated feedback and guidance that the worksheets provide will function as intended.

12.10 Additional resources

Now that you've picked up the basics of version control with Git and GitHub, you can expand your knowledge through the resources listed below:

- GitHub's guides website[5] is a great source to take the next steps in learning about Git and GitHub.
- Good enough practices in scientific computing[6] [Wilson *et al.*, 2017] provides more advice on useful workflows and "good enough" practices in data analysis projects.
- In addition to GitHub[7], there are other popular Git repository hosting services such as GitLab[8] and BitBucket[9]. Comparing all of these options is beyond the scope of this book, and until you become a more advanced user, you are perfectly fine to just stick with GitHub. Just be aware that you have options!
- GitHub's documentation on creating a personal access token[10] is an excellent additional resource to consult if you need help generating and using personal access tokens.

[5] https://guides.github.com/

[6] https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005510#sec014

[7] https://github.com

[8] https://gitlab.com

[9] https://bitbucket.org

[10] https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token

13

Setting up your computer

13.1 Overview

In this chapter, you'll learn how to set up the software needed to follow along with this book on your own computer. Given that installation instructions can vary based on computer setup, we provide instructions for multiple operating systems (Ubuntu Linux, MacOS, and Windows). Although the instructions in this chapter will likely work on many systems, we have specifically verified that they work on a computer that:

- runs Windows 10 Home, MacOS 13 Ventura, or Ubuntu 22.04,
- uses a 64-bit CPU,
- has a connection to the internet,
- uses English as the default language.

13.2 Chapter learning objectives

By the end of the chapter, readers will be able to do the following:

- Download the worksheets that accompany this book.
- Install the Docker virtualization engine.
- Edit and run the worksheets using JupyterLab running inside a Docker container.
- Install Git, JupyterLab Desktop, and Python packages.
- Edit and run the worksheets using JupyterLab Desktop.

DOI: 10.1201/9781003438397-13

13.3 Obtaining the worksheets for this book

The worksheets containing exercises for this book are online at https://worksheets.python.datasciencebook.ca. The worksheets can be launched directly from that page using the Binder links in the rightmost column of the table. This is the easiest way to access the worksheets, but note that you will not be able to save your work and return to it again later. In order to save your progress, you will need to download the worksheets to your own computer and work on them locally. You can download the worksheets as a compressed zip file using the link at the top of the page[1]. Once you unzip the downloaded file, you will have a folder containing all of the Jupyter notebook worksheets accompanying this book. See Chapter 11 for instructions on working with Jupyter notebooks.

13.4 Working with Docker

Once you have downloaded the worksheets, you will next need to install and run the software required to work on Jupyter notebooks on your own computer. Doing this setup manually can be quite tricky, as it involves quite a few different software packages, not to mention getting the right versions of everything—the worksheets and autograder tests may not work unless all the versions are exactly right. To keep things simple, we instead recommend that you install Docker[2]. Docker lets you run your Jupyter notebooks inside a pre-built *container* that comes with precisely the right versions of all software packages needed run the worksheets that come with this book.

Note: A *container* is a virtual user space within your computer. Within the container, you can run software in isolation without interfering with the other software that already exists on your machine. In this book, we use a container to run a specific version of the Python programming language, as well as other necessary packages. The container ensures that the worksheets function correctly, even if you have a different version of Python installed on your computer—or even if you haven't installed Python at all.

[1] https://github.com/UBC-DSCI/data-science-a-first-intro-python-worksheets/archive/refs/heads/main.zip

[2] https://docker.com

13.4.1 Windows

Installation To install Docker on Windows, visit the online Docker documentation[3], and download the `Docker Desktop Installer.exe` file. Double-click the file to open the installer and follow the instructions on the installation wizard, choosing **WSL-2** instead of **Hyper-V** when prompted.

Note: Occasionally, when you first run Docker on Windows, you will encounter an error message. Some common errors you may see:

- If you need to update WSL, you can enter `cmd.exe` in the Start menu to run the command line. Type `wsl --update` to update WSL.
- If the admin account on your computer is different to your user account, you must add the user to the "docker-users" group. Run Computer Management as an administrator and navigate to `Local Users` and `Groups -> Groups -> docker-users`. Right-click to add the user to the group. Log out and log back in for the changes to take effect.
- If you need to enable virtualization, you will need to edit your BIOS. Restart your computer, and enter the BIOS using the hotkey (usually Delete, Esc, and/or one of the F# keys). Look for an "Advanced" menu, and under your CPU settings, set the "Virtualization" option to "enabled". Then save the changes and reboot your machine. If you are not familiar with BIOS editing, you may want to find an expert to help you with this, as editing the BIOS can be dangerous. Detailed instructions for doing this are beyond the scope of this book.

Running JupyterLab Run Docker Desktop. Once it is running, you need to download and run the Docker *image* that we have made available for the worksheets (an *image* is like a "snapshot" of a computer with all the right packages pre-installed). You only need to do this step one time; the image will remain the next time you run Docker Desktop. In the Docker Desktop search bar, enter `ubcdsci/py-dsci-100`, as this is the name of the image. You will see the `ubcdsci/py-dsci-100` image in the list (Fig. 13.1), and "latest" in the Tag drop down menu. We need to change "latest" to the right image version before proceeding. To find the right tag, open the `Dockerfile` in the worksheets repository[4], and look for the line `FROM ubcdsci/py-dsci-100:` followed by the tag consisting of a sequence of numbers and letters. Back in

[3] https://docs.docker.com/desktop/install/windows-install/

[4] https://raw.githubusercontent.com/UBC-DSCI/data-science-a-first-intro-python-worksheets/main/Dockerfile

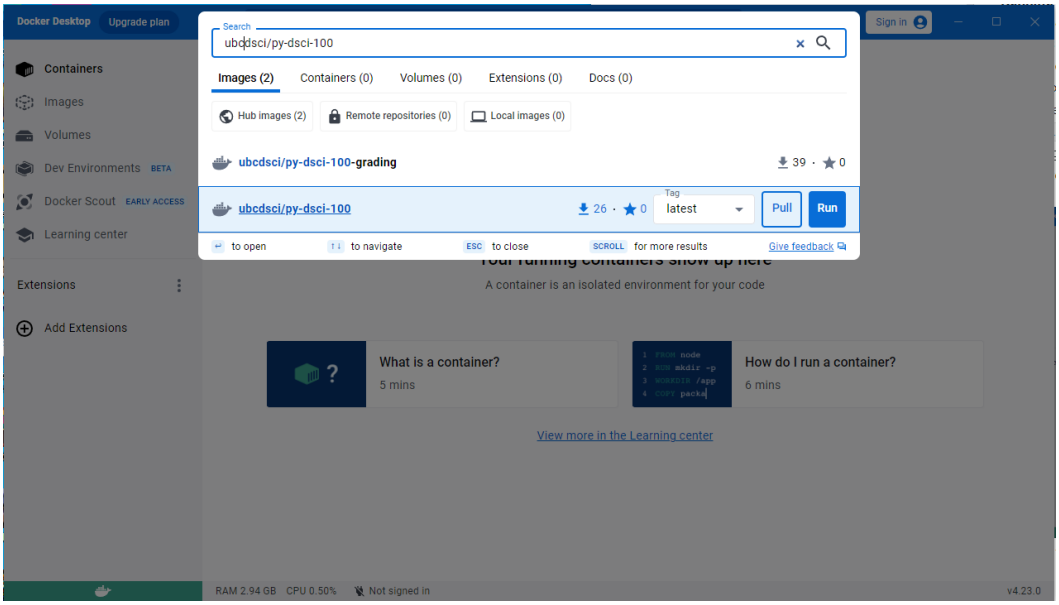


FIGURE 13.1 The Docker Desktop search window. Make sure to click the Tag drop down menu and find the right version of the image before clicking the Pull button to download it.

Docker Desktop, in the "Tag" drop down menu, click that tag to select the correct image version. Then click the "Pull" button to download the image.

Once the image is done downloading, click the "Images" button on the left side of the Docker Desktop window (Fig. 13.2). You will see the recently downloaded image listed there under the "Local" tab.

To start up a *container* using that image, click the play button beside the image. This will open the run configuration menu (Fig. 13.3). Expand the "Optional settings" drop down menu. In the "Host port" textbox, enter `8888`. In the "Volumes" section, click the "Host path" box and navigate to the folder where your Jupyter worksheets are stored. In the "Container path" text box, enter `/home/jovyan/work`. Then click the "Run" button to start the container.

After clicking the "Run" button, you will see a terminal. The terminal will then print some text as the Docker container starts. Once the text stops scrolling, find the URL in the terminal that starts with `http://127.0.0.1:8888` (highlighted by the red box in Fig. 13.4), and paste it into your browser to start JupyterLab.

When you are done working, make sure to shut down and remove the container by clicking the red trash can symbol (in the top right corner of Fig. 13.4). You

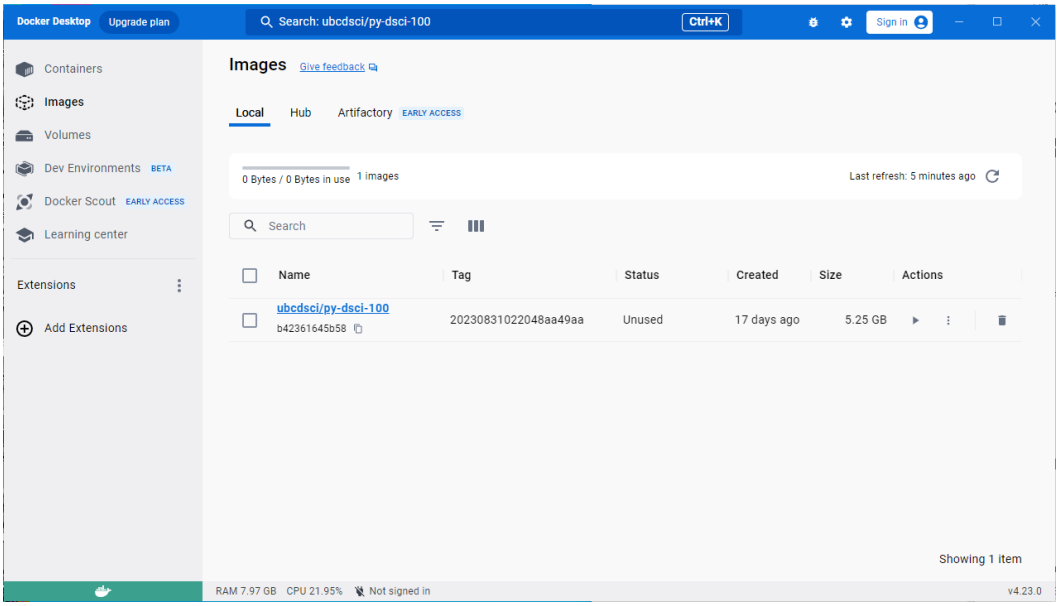


FIGURE 13.2 The Docker Desktop images tab.

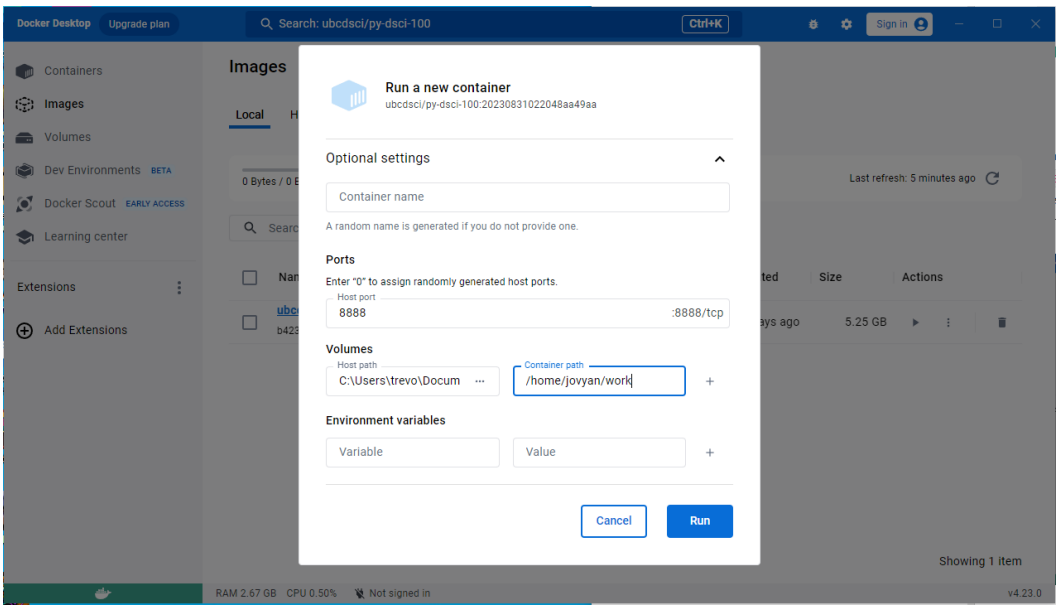


FIGURE 13.3 The Docker Desktop container run configuration menu.

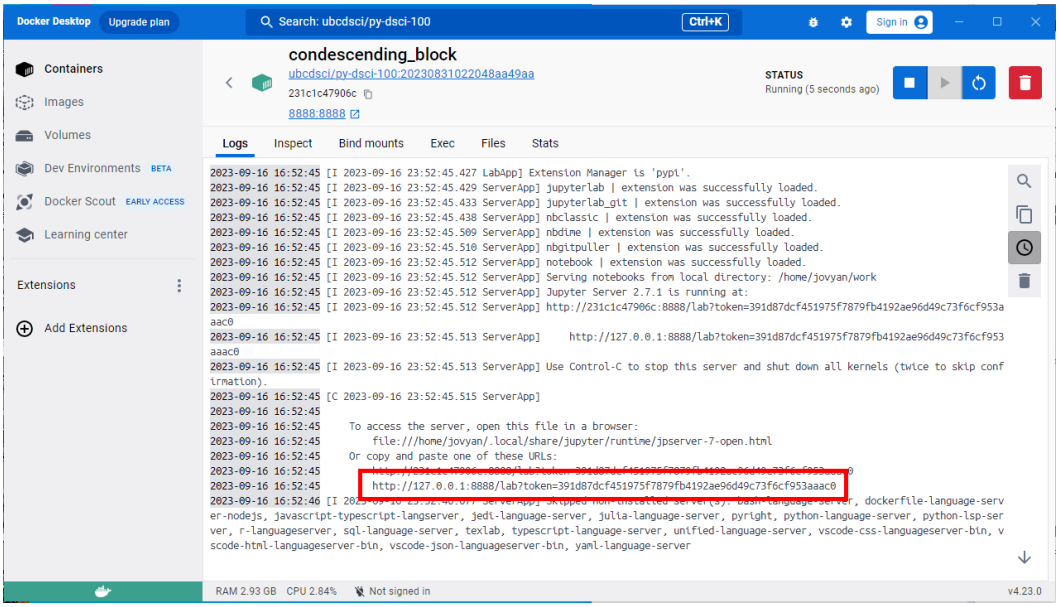


FIGURE 13.4 The terminal text after running the Docker container. The red box indicates the URL that you should paste into your browser to open JupyterLab.

will not be able to start the container again until you do so. More information on installing and running Docker on Windows, as well as troubleshooting tips, can be found in the online Docker documentation[5].

13.4.2 MacOS

Installation To install Docker on MacOS, visit the online Docker documentation[6], and download the `Docker.dmg` installation file that is appropriate for your computer. To know which installer is right for your machine, you need to know whether your computer has an Intel processor (older machines) or an Apple processor (newer machines); the Apple support page[7] has information to help you determine which processor you have. Once downloaded, double-click the file to open the installer, then drag the Docker icon to the Applications folder. Double-click the icon in the Applications folder to start Docker. In the installation window, use the recommended settings.

Running JupyterLab Run Docker Desktop. Once it is running, follow the instructions above in the Windows section on *Running JupyterLab* (the user interface is the same). More information on installing and running Docker on

[5] https://docs.docker.com/desktop/install/windows-install/
[6] https://docs.docker.com/desktop/install/mac-install/
[7] https://support.apple.com/en-ca/HT211814

MacOS, as well as troubleshooting tips, can be found in the online Docker documentation[8].

13.4.3 Ubuntu

Installation To install Docker on Ubuntu, open the terminal and enter the following five commands.

```
sudo apt update
sudo apt install ca-certificates curl gnupg
curl -fsSL https://get.docker.com -o get-docker.sh
sudo chmod u+x get-docker.sh
sudo sh get-docker.sh
```

Running JupyterLab First, open the `Dockerfile` in the worksheets repository[9], and look for the line `FROM ubcdsci/py-dsci-100:` followed by a tag consisting of a sequence of numbers and letters. Then in the terminal, navigate to the directory where you want to run JupyterLab, and run the following command, replacing `TAG` with the *tag* you found earlier.

```
docker run --rm -v $(pwd):/home/jovyan/work -p 8888:8888 ubcdsci/py-dsci-100:TAG␣
↪jupyter lab
```

The terminal will then print some text as the Docker container starts. Once the text stops scrolling, find the URL in your terminal that starts with `http://127.0.0.1:8888` (highlighted by the red box in Fig. 13.5), and paste it into your browser to start JupyterLab. More information on installing and running Docker on Ubuntu, as well as troubleshooting tips, can be found in the online Docker documentation[10].

13.5 Working with JupyterLab Desktop

You can also run the worksheets accompanying this book on your computer using JupyterLab Desktop[11]. The advantage of JupyterLab Desktop over Docker is that it can be easier to install; Docker can sometimes run into some fairly technical issues (especially on Windows computers) that require expert troubleshooting. The downside of JupyterLab Desktop is that there is a (very) small chance that you may not end up with the right versions of all the Python

[8] https://docs.docker.com/desktop/install/mac-install/

[9] https://raw.githubusercontent.com/UBC-DSCI/data-science-a-first-intro-python-worksheets/main/Dockerfile

[10] https://docs.docker.com/engine/install/ubuntu/

[11] https://github.com/jupyterlab/jupyterlab-desktop

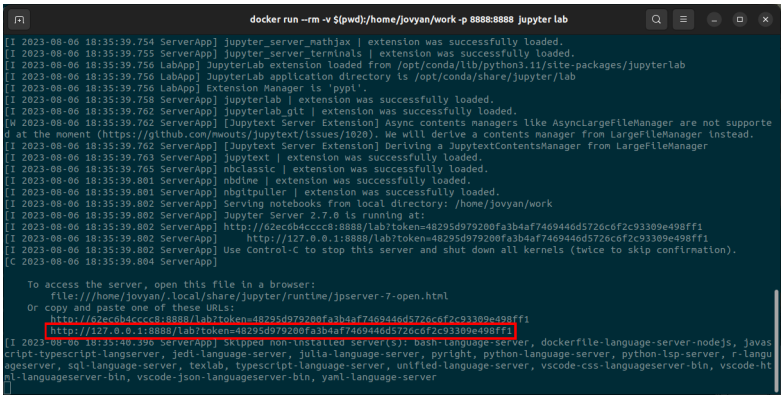


FIGURE 13.5 The terminal text after running the Docker container in Ubuntu. The red box indicates the URL that you should paste into your browser to open JupyterLab.

packages needed for the worksheets. Docker, on the other hand, *guarantees* that the worksheets will work exactly as intended.

In this section, we will cover how to install JupyterLab Desktop, Git and the JupyterLab Git extension (for version control, as discussed in Chapter 12), and all of the Python packages needed to run the code in this book.

13.5.1 Windows

Installation First, we will install Git for version control. Go to the Git download page[12] and download the Windows version of Git. Once the download has finished, run the installer and accept the default configuration for all pages. Next, visit the "Installation" section of the JupyterLab Desktop homepage[13]. Download the `JupyterLab-Setup-Windows.exe` installer file for Windows. Double-click the installer to run it, use the default settings. Run JupyterLab Desktop by clicking the icon on your desktop.

Configuring JupyterLab Desktop Next, in the JupyterLab Desktop graphical interface that appears (Fig. 13.6), you will see text at the bottom saying "Python environment not found". Click "Install using the bundled installer" to set up the environment.

Next, we need to add the JupyterLab Git extension (so that we can use version control directly from within JupyterLab Desktop), the IPython kernel (to enable the Python programming language), and various Python software packages. Click "New session…" in the JupyterLab Desktop user interface,

[12] https://git-scm.com/download/win
[13] https://github.com/jupyterlab/jupyterlab-desktop#installation

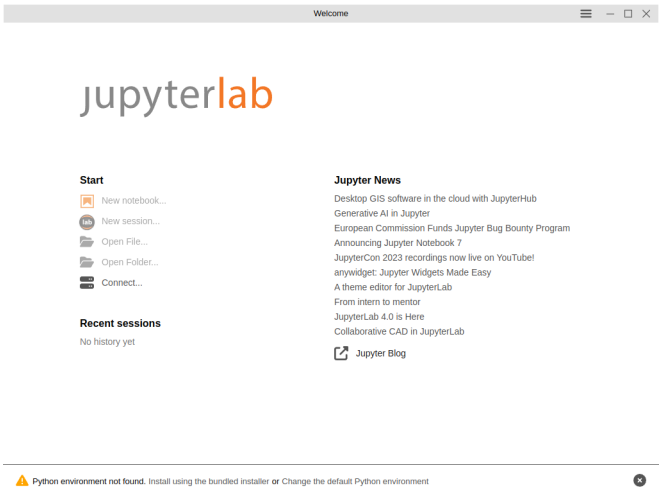


FIGURE 13.6 The JupyterLab Desktop graphical user interface.

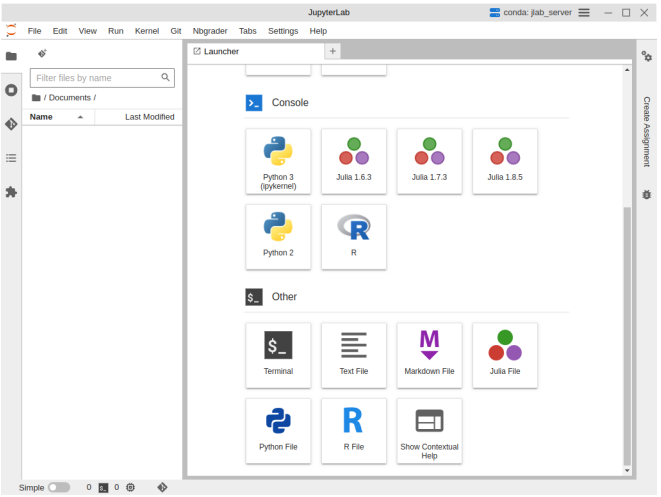


FIGURE 13.7 A JupyterLab Desktop session, showing the Terminal option at the bottom.

then scroll to the bottom, and click "Terminal" under the "Other" heading (Fig. 13.7).

In this terminal, run the following commands:

```
pip install --upgrade jupyterlab-git
conda env update --file https://raw.githubusercontent.com/UBC-DSCI/data-science-
↪a-first-intro-python-worksheets/main/environment.yml
```

The second command installs the specific Python and package versions specified in the `environment.yml` file found in the worksheets repository[14]. We will always keep the versions in the `environment.yml` file updated so that

[14] https://worksheets.python.datasciencebook.ca

they are compatible with the exercise worksheets that accompany the book. Once all of the software installation is complete, it is a good idea to restart JupyterLab Desktop entirely before you proceed to doing your data analysis. This will ensure all the software and settings you put in place are correctly set up and ready for use.

13.5.2 MacOS

Installation First, we will install Git for version control. Open the terminal (how-to video[15]) and type the following command:

```
xcode-select --install
```

Next, visit the "Installation" section of the JupyterLab Desktop homepage[16]. Download the `JupyterLab-Setup-MacOS-x64.dmg` or `JupyterLab-Setup-MacOS-arm64.dmg` installer file. To know which installer is right for your machine, you need to know whether your computer has an Intel processor (older machines) or an Apple processor (newer machines); the Apple support page[17] has information to help you determine which processor you have. Once downloaded, double-click the file to open the installer, then drag the JupyterLab Desktop icon to the Applications folder. Double-click the icon in the Applications folder to start JupyterLab Desktop.

Configuring JupyterLab Desktop From this point onward, with JupyterLab Desktop running, follow the instructions in the Windows section on *Configuring JupyterLab Desktop* to set up the environment, install the JupyterLab Git extension, and install the various Python software packages needed for the worksheets.

13.5.3 Ubuntu

Installation First, we will install Git for version control. Open the terminal and type the following commands:

```
sudo apt update
sudo apt install git
```

Next, visit the "Installation" section of the JupyterLab Desktop homepage[18]. Download the `JupyterLab-Setup-Debian.deb` installer file for Ubuntu/Debian. Open a terminal, navigate to where the installer file was downloaded, and run the command

[15] https://youtu.be/5AJbWEWwnbY
[16] https://github.com/jupyterlab/jupyterlab-desktop#installation
[17] https://support.apple.com/en-ca/HT211814
[18] https://github.com/jupyterlab/jupyterlab-desktop#installation

```
sudo dpkg -i JupyterLab-Setup-Debian.deb
```

Run JupyterLab Desktop using the command

```
jlab
```

Configuring JupyterLab Desktop From this point onward, with JupyterLab Desktop running, follow the instructions in the Windows section on *Configuring JupyterLab Desktop* to set up the environment, install the JupyterLab Git extension, and install the various Python software packages needed for the worksheets.

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

Bibliography

[GvR01] Nick Coghlan Guido van Rossum, Barry Warsaw. *PEP 8 – Style Guide for Python Code*. 2001. URL: https://peps.python.org/pep-0008/.

[LP15] Jeffrey Leek and Roger Peng. What is the question? *Science*, 347(6228):1314–1315, 2015.

[PM15] Roger D Peng and Elizabeth Matsui. *The Art of Data Science: A Guide for Anyone Who Works with Data*. Skybrude Consulting, LLC, 2015. URL: https://bookdown.org/rdpeng/artofdatascience/.

[Tim20] Tiffany Timbers. *canlang: Canadian Census language data*. 2020. R package version 0.0.9. URL: https://ttimbers.github.io/canlang/.

[Wal17] Nick Walker. Mapping indigenous languages in Canada. *Canadian Geographic*, 2017. URL: https://www.canadiangeographic.ca/article/mapping-indigenous-languages-canada (visited on 2021-05-27).

[Wil18] Kory Wilson. *Pulling Together: Foundations Guide*. BCcampus, 2018. URL: https://opentextbc.ca/indigenizationfoundations/ (visited on 2021-05-27).

[StatisticsCanada16a] Statistics Canada. Population census. 2016. URL: https://www12.statcan.gc.ca/census-recensement/2016/dp-pd/index-eng.cfm.

[StatisticsCanada16b] Statistics Canada. The Aboriginal languages of First Nations people, Métis and Inuit. 2016. URL: https://www12.statcan.gc.ca/census-recensement/2016/as-sa/98-200-x/2016022/98-200-x2016022-eng.cfm.

[StatisticsCanada18] Statistics Canada. The evolution of language populations in Canada, by mother tongue, from 1901 to 2016. 2018. URL: https://www150.statcan.gc.ca/n1/pub/11-630-x/11-630-x2018001-eng.htm (visited on 2021-05-27).

[ThePDTeam20] The Pandas Development Team. *pandas-dev/pandas: Pandas*. February 2020. URL: https://doi.org/10.5281/zenodo.3509134, doi:10.5281/zenodo.3509134[19].

[TruthaRCoCanada12] Truth and Reconciliation Commission of Canada. *They Came for the Children: Canada, Aboriginal Peoples, and the Residential Schools*. Public Works & Government Services Canada, 2012.

[TruthaRCoCanada15] Truth and Reconciliation Commission of Canada. Calls to Action. 2015. URL: https://www2.gov.bc.ca/assets/gov/british-columbians-our-governments/indigenous-people/aboriginal-peoples-documents/calls_to_action_english2.pdf.

[WesMcKinney10] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, 56 – 61. 2010. doi:10.25080/Majora-92bf1922-00a[20].

[McK12] Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* " O'Reilly Media, Inc.", 2012.

[RThePSFoundation23] Kenneth Reitz and The Python Software Foundation. Requests: http for humans. *URL: https://requests.readthedocs.io/en/latest/*, Accessed Online: 2023.

[Ric07] Leonard Richardson. Beautiful soup documentation. *April*, 2007.

[NASAESACSA+23] NASA, ESA, CSA, STScI, K. Pontoppidan (STScI), and A. Pagan (STScI). Rho ophiuchi cloud complex. *URL: https://esawebb.org/images/weic2316a/*, Accessed Online: 2023.

[RealTSProject21] Real Time Statistics Project. Internet live stats: google search statistics. 2021. URL: https://www.internetlivestats.com/google-search-statistics/.

[McK12] Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* " O'Reilly Media, Inc.", 2012.

[Wic14] Hadley Wickham. Tidy data. *Journal of Statistical Software*, 59(10):1–23, 2014.

[Dee05] Sameer Deeb. The molecular basis of variation in human color vision. *Clinical Genetics*, 67:369–377, 2005.

[19] https://doi.org/10.5281/zenodo.3509134
[20] https://doi.org/10.25080/Majora-92bf1922-00a

[Har91] Wolfgang Hardle. *Smoothing Techniques with Implementation in S*. Springer, New York, 1991.

[McK12] Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. " O'Reilly Media, Inc.", 2012.

[McN77] Donald R. McNeil. *Interactive Data Analysis: A Practical Primer*. Wiley, 1977.

[Mic82] Albert Michelson. Experimental determination of the velocity of light made at the United States Naval Academy, Annapolis. *Astronomic Papers*, 1:135–8, 1882.

[TK20] Pieter Tans and Ralph Keeling. Trends in atmospheric carbon dioxide. 2020. URL: https://gml.noaa.gov/ccgg/trends/data.html (visited on 2020-07-04).

[Tim20] Tiffany Timbers. *canlang: Canadian Census language data*. 2020. R package version 0.0.9. URL: https://ttimbers.github.io/canlang/.

[VGH+18] Jacob VanderPlas, Brian Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. Altair: interactive statistical visualizations for python. *Journal of Open Source Software*, 3(32):1057, 2018. URL: https://doi.org/10.21105/joss.01057, doi:10.21105/joss.01057[21].

[Wil19] Claus Wilke. *Fundamentals of Data Visualization*. O'Reilly Media, 2019. URL: https://clauswilke.com/dataviz/.

[BLB+13] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 108–122. 2013.

[CH67] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[FH51] Evelyn Fix and Joseph Hodges. Discriminatory analysis. nonparametric discrimination: consistency properties. Technical Report, USAF School of Aviation Medicine, Randolph Field, Texas, 1951.

[21] https://doi.org/10.21105/joss.01057

[SWM93] William Nick Street, William Wolberg, and Olvi Mangasarian. Nuclear feature extraction for breast tumor diagnosis. In *International Symposium on Electronic Imaging: Science and Technology*. 1993.

[StanfordHCare21] Stanford Health Care. What is cancer? 2021. URL: https://stanfordhealthcare.org/medical-conditions/cancer/cancer.html.

[BKM67] Evelyn Martin Lansdowne Beale, Maurice George Kendall, and David Mann. The discarding of variables in multivariate analysis. *Biometrika*, 54(3-4):357–366, 1967.

[DS66] Norman Draper and Harry Smith. *Applied Regression Analysis*. Wiley, 1966.

[Efo66] M. Eforymson. Stepwise regression—a backward and forward look. In *Eastern Regional Meetings of the Institute of Mathematical Statistics*. 1966.

[HL67] Ronald Hocking and R. N. Leslie. Selection of the best subset in regression analysis. *Technometrics*, 9(4):531–540, 1967.

[JWHT13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. Springer, 1st edition, 2013. URL: https://www.statlearning.com/.

[McK12] Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. " O'Reilly Media, Inc.", 2012.

[SWM93] William Nick Street, William Wolberg, and Olvi Mangasarian. Nuclear feature extraction for breast tumor diagnosis. In *International Symposium on Electronic Imaging: Science and Technology*. 1993.

[CH67] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[FH51] Evelyn Fix and Joseph Hodges. Discriminatory analysis. nonparametric discrimination: consistency properties. Technical Report, USAF School of Aviation Medicine, Randolph Field, Texas, 1951.

[JWHT13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. Springer, 1st edition, 2013. URL: https://www.statlearning.com/.

[GWF14] Kristen Gorman, Tony Williams, and William Fraser. Ecological sexual dimorphism and environmental variability within a community of Antarctic penguins (genus pygoscelis). *PLoS ONE*, 2014.

[HHG20] Allison Horst, Alison Hill, and Kristen Gorman. *palmerpenguins: Palmer Archipelago penguin data*. 2020. R package version 0.1.0. URL: https://allisonhorst.github.io/palmerpenguins/.

[JWHT13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. Springer, 1st edition, 2013. URL: https://www.statlearning.com/.

[Llo82] Stuart Lloyd. Least square quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982. Originally released as a Bell Telephone Laboratories Paper in 1957.

[Coxd.] Murray Cox. Inside Airbnb. n.d. URL: http://insideairbnb.com/ (visited on 2020-09-01).

[DcCetinkayaRB19] David Diez, Mine Çetinkaya-Rundel, and Christopher Barr. *OpenIntro Statistics*. OpenIntro, Inc., 2019. URL: https://openintro.org/book/os/.

[KRKPerez+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. Jupyter notebooks: a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas: Proceedings of the 20th international conference on electronic publishing*, volume 87. Amsterdam, 2016. IOS Press.

[WBC+17] Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy Teal. Good enough practices in scientific computing. *PLoS Computational Biology*, 2017.

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

Index