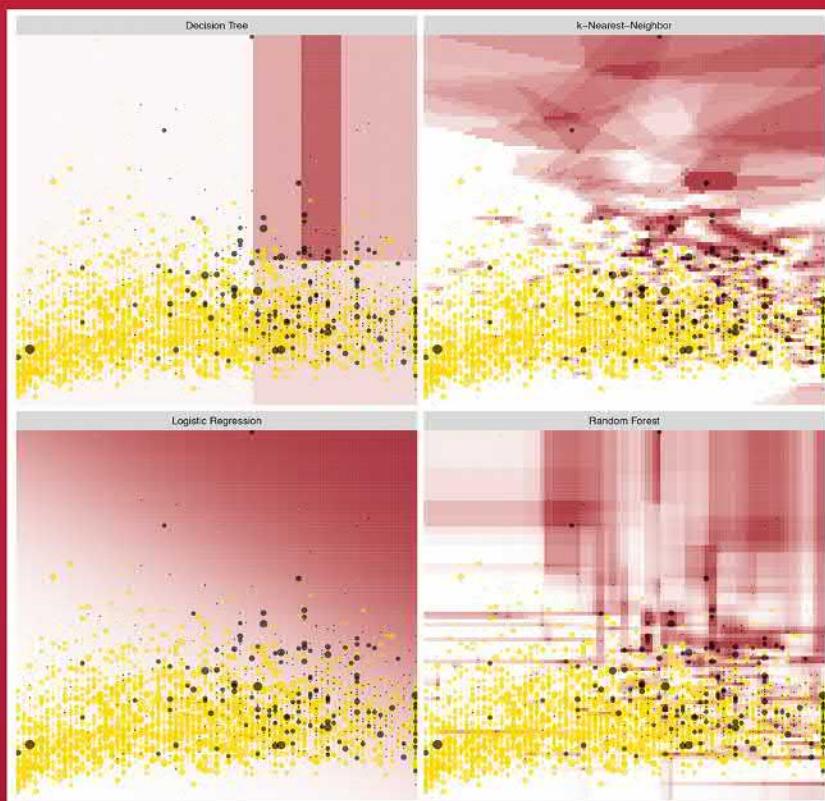


Texts in Statistical Science

Modern Data Science with R

Second Edition



Benjamin S. Baumer
Daniel T. Kaplan
Nicholas J. Horton



CRC Press
Taylor & Francis Group

A CHAPMAN & HALL BOOK

Modern Data Science with R

CHAPMAN & HALL/CRC

Texts in Statistical Science Series

Joseph K. Blitzstein, *Harvard University, USA*

Julian J. Faraway, *University of Bath, UK*

Martin Tanner, *Northwestern University, USA*

Jim Zidek, *University of British Columbia, Canada*

Recently Published Titles

Practical Multivariate Analysis, Sixth Edition

Abdelmonem Afifi, Susanne May, Robin A. Donatello, and Virginia A. Clark

Time Series: A First Course with Bootstrap Starter

Tucker S. McElroy and Dimitris N. Politis

Probability and Bayesian Modeling

Jim Albert and Jingchen Hu

Surrogates

Gaussian Process Modeling, Design, and Optimization for the Applied Sciences

Robert B. Gramacy

Statistical Analysis of Financial Data

With Examples in R

James Gentle

Statistical Rethinking

A Bayesian Course with Examples in R and STAN, Second Edition

Richard McElreath

Statistical Machine Learning

A Model-Based Approach

Richard Golden

Randomization, Bootstrap and Monte Carlo Methods in Biology

Fourth Edition

Bryan F. J. Manly, Jorge A. Navarro Alberto

Principles of Uncertainty, Second Edition

Joseph B. Kadane

Beyond Multiple Linear Regression

Applied Generalized Linear Models and Multilevel Models in R

Paul Roback, Julie Legler

Bayesian Thinking in Biostatistics

Gary L. Rosner, Purushottam W. Laud, and Wesley O. Johnson

Modern Data Science with R, Second Edition

Benjamin S. Baumer, Daniel T. Kaplan, and Nicholas J. Horton

For more information about this series, please visit: <https://www.crcpress.com/Chapman--Hall-CRC-Texts-in-Statistical-Science/book-series/CHTEXSTASCI>

Modern Data Science with R

2nd edition

Benjamin S. Baumer
Daniel T. Kaplan
Nicholas J. Horton



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

First edition published 2021
by CRC Press
6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

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

© 2021 Taylor & Francis Group, LLC

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

The right of Benjamin S. Baumer, Daniel T. Kaplan, and Nicholas J. Horton to be identified as authors of this work has been asserted by them in accordance with sections 77 and 78 of the Copyright, Designs and Patents Act 1988.

Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, access www.copyright.com or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. For works that are not available on CCC please contact mpkbookspermissions@tandf.co.uk

Trademark notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

ISBN: 9780367191498 (hbk)
ISBN: 9780367745448 (pbk)
ISBN: 9780429200717 (ebk)

Typeset in Latin Modern font
by KnowledgeWorks Global Ltd.

Contents

About the Authors	xi
Preface	xiii
I Part I: Introduction to Data Science	1
1 Prologue: Why data science?	3
1.1 What is data science?	4
1.2 Case study: The evolution of sabermetrics	6
1.3 Datasets	7
1.4 Further resources	8
2 Data visualization	9
2.1 The 2012 federal election cycle	9
2.2 Composing data graphics	16
2.3 Importance of data graphics: <i>Challenger</i>	24
2.4 Creating effective presentations	28
2.5 The wider world of data visualization	29
2.6 Further resources	31
2.7 Exercises	32
2.8 Supplementary exercises	33
3 A grammar for graphics	35
3.1 A grammar for data graphics	35
3.2 Canonical data graphics in R	43
3.3 Extended example: Historical baby names	53
3.4 Further resources	62
3.5 Exercises	62
3.6 Supplementary exercises	65
4 Data wrangling on one table	67
4.1 A grammar for data wrangling	67
4.2 Extended example: Ben's time with the Mets	76
4.3 Further resources	84
4.4 Exercises	84
4.5 Supplementary exercises	88
5 Data wrangling on multiple tables	89
5.1 <code>inner_join()</code>	89
5.2 <code>left_join()</code>	91
5.3 Extended example: Manny Ramirez	92
5.4 Further resources	99
5.5 Exercises	99

5.6	Supplementary exercises	101
6	Tidy data	103
6.1	Tidy data	103
6.2	Reshaping data	112
6.3	Naming conventions	120
6.4	Data intake	121
6.5	Further resources	135
6.6	Exercises	135
6.7	Supplementary exercises	138
7	Iteration	139
7.1	Vectorized operations	139
7.2	Using <code>across()</code> with <code>dplyr</code> functions	142
7.3	The <code>map()</code> family of functions	143
7.4	Iterating over a one-dimensional vector	144
7.5	Iteration over subgroups	146
7.6	Simulation	151
7.7	Extended example: Factors associated with BMI	153
7.8	Further resources	155
7.9	Exercises	157
7.10	Supplementary exercises	157
8	Data science ethics	159
8.1	Introduction	159
8.2	Truthful falsehoods	160
8.3	Role of data science in society	161
8.4	Some settings for professional ethics	163
8.5	Some principles to guide ethical action	167
8.6	Algorithmic bias	171
8.7	Data and disclosure	172
8.8	Reproducibility	174
8.9	Ethics, collectively	175
8.10	Professional guidelines for ethical conduct	176
8.11	Further resources	176
8.12	Exercises	177
8.13	Supplementary exercises	179
II	Part II: Statistics and Modeling	181
9	Statistical foundations	183
9.1	Samples and populations	183
9.2	Sample statistics	186
9.3	The bootstrap	190
9.4	Outliers	194
9.5	Statistical models: Explaining variation	196
9.6	Confounding and accounting for other factors	199
9.7	The perils of p-values	202
9.8	Further resources	204
9.9	Exercises	205
9.10	Supplementary exercises	206

10 Predictive modeling	207
10.1 Predictive modeling	208
10.2 Simple classification models	209
10.3 Evaluating models	216
10.4 Extended example: Who has diabetes?	223
10.5 Further resources	227
10.6 Exercises	227
10.7 Supplementary exercises	228
11 Supervised learning	229
11.1 Non-regression classifiers	229
11.2 Parameter tuning	245
11.3 Example: Evaluation of income models redux	246
11.4 Extended example: Who has diabetes this time?	250
11.5 Regularization	255
11.6 Further resources	258
11.7 Exercises	259
11.8 Supplementary exercises	261
12 Unsupervised learning	263
12.1 Clustering	263
12.2 Dimension reduction	270
12.3 Further resources	278
12.4 Exercises	278
12.5 Supplementary exercises	279
13 Simulation	281
13.1 Reasoning in reverse	281
13.2 Extended example: Grouping cancers	282
13.3 Randomizing functions	285
13.4 Simulating variability	286
13.5 Random networks	293
13.6 Key principles of simulation	293
13.7 Further resources	296
13.8 Exercises	296
13.9 Supplementary exercises	298
III Part III: Topics in Data Science	299
14 Dynamic and customized data graphics	301
14.1 Rich Web content using <code>d3.js</code> and <code>htmlwidgets</code>	301
14.2 Animation	306
14.3 Flexdashboard	306
14.4 Interactive web apps with Shiny	308
14.5 Customization of <code>ggplot2</code> graphics	313
14.6 Extended example: Hot dog eating	317
14.7 Further resources	322
14.8 Exercises	322
14.9 Supplementary exercises	324
15 Database querying using SQL	325
15.1 From <code>dplyr</code> to SQL	325

15.2	Flat-file databases	329
15.3	The SQL universe	331
15.4	The SQL data manipulation language	332
15.5	Extended example: FiveThirtyEight flights	352
15.6	SQL vs. R	360
15.7	Further resources	360
15.8	Exercises	360
15.9	Supplementary exercises	362
16	Database administration	363
16.1	Constructing efficient SQL databases	363
16.2	Changing SQL data	369
16.3	Extended example: Building a database	371
16.4	Scalability	375
16.5	Further resources	375
16.6	Exercises	375
16.7	Supplementary exercises	376
17	Working with geospatial data	377
17.1	Motivation: What's so great about geospatial data?	377
17.2	Spatial data structures	380
17.3	Making maps	382
17.4	Extended example: Congressional districts	391
17.5	Effective maps: How (not) to lie	399
17.6	Projecting polygons	401
17.7	Playing well with others	402
17.8	Further resources	403
17.9	Exercises	404
17.10	Supplementary exercises	405
18	Geospatial computations	407
18.1	Geospatial operations	407
18.2	Geospatial aggregation	416
18.3	Geospatial joins	418
18.4	Extended example: Trail elevations at MacLeish	419
18.5	Further resources	423
18.6	Exercises	423
18.7	Supplementary exercises	424
19	Text as data	425
19.1	Regular expressions using <i>Macbeth</i>	425
19.2	Extended example: Analyzing textual data from arXiv.org	431
19.3	Ingesting text	445
19.4	Further resources	448
19.5	Exercises	448
19.6	Supplementary exercises	450
20	Network science	451
20.1	Introduction to network science	451
20.2	Extended example: Six degrees of Kristen Stewart	456
20.3	PageRank	465
20.4	Extended example: 1996 men's college basketball	467

20.5	Further resources	474
20.6	Exercises	475
20.7	Supplementary exercises	475
21	Epilogue: Towards “big data”	477
21.1	Notions of big data	477
21.2	Tools for bigger data	479
21.3	Alternatives to R	489
21.4	Closing thoughts	489
21.5	Further resources	490
IV	Part IV: Appendices	491
A	Packages used in this book	493
A.1	The mdsr package	493
A.2	Other packages	493
A.3	Further resources	498
B	Introduction to R and RStudio	499
B.1	Installation	499
B.2	Learning R	500
B.3	Fundamental structures and objects	501
B.4	Add-ons: Packages	508
B.5	Further resources	514
B.6	Exercises	515
B.7	Supplementary exercises	517
C	Algorithmic thinking	519
C.1	Introduction	519
C.2	Simple example	519
C.3	Extended example: Law of large numbers	522
C.4	Non-standard evaluation	525
C.5	Debugging and defensive coding	527
C.6	Further resources	529
C.7	Exercises	529
C.8	Supplementary exercises	530
D	Reproducible analysis and workflow	531
D.1	Scriptable statistical computing	532
D.2	Reproducible analysis with R Markdown	532
D.3	Projects and version control	535
D.4	Further resources	537
D.5	Exercises	537
D.6	Supplementary exercises	540
E	Regression modeling	541
E.1	Simple linear regression	541
E.2	Multiple regression	546
E.3	Inference for regression	552
E.4	Assumptions underlying regression	553
E.5	Logistic regression	556
E.6	Further resources	559

E.7 Exercises	561
E.8 Supplementary exercises	562
F Setting up a database server	563
F.1 SQLite	563
F.2 MySQL	564
F.3 PostgreSQL	567
F.4 Connecting to SQL	568
Bibliography	573
Indices	589
Subject index	590
R index	622

About the Authors

Benjamin S. Baumer is an associate professor in the Statistical & Data Sciences program at Smith College. He has been a practicing data scientist since 2004, when he became the first full-time statistical analyst for the New York Mets. Ben is a co-author of *The Sabermetric Revolution* and *Analyzing Baseball Data with R*. He received the 2019 Waller Education Award and the 2016 Contemporary Baseball Analysis Award from the Society for American Baseball Research.

Daniel T. Kaplan is the DeWitt Wallace Professor of Mathematics and Computer Science at Macalester College. He is the author of several textbooks on statistical modeling and statistical computing. Danny received the 2006 Macalester Excellence in Teaching award and the 2017 CAUSE Lifetime Achievement Award.

Nicholas J. Horton is the Beitzel Professor of Technology and Society (Statistics and Data Science) at Amherst College. He is a Fellow of the ASA and the AAAS, co-chair of the National Academies Committee on Applied and Theoretical Statistics, recipient of a number of national teaching awards, author of a series of books on statistical computing, and actively involved in data science curriculum efforts to help students “think with data”.



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

Preface

Background and motivation

The increasing volume and sophistication of data poses new challenges for analysts, who need to be able to transform complex data sets to answer important statistical questions. A consensus report on data science for undergraduates (National Academies of Science, Engineering, and Medicine, 2018) noted that data science is revolutionizing science and the workplace. They defined a data scientist as “a knowledge worker who is principally occupied with analyzing complex and massive data resources.”

Michael I. Jordan has described data science as the marriage of computational thinking and inferential (statistical) thinking. Without the skills to be able to “wrangle” or “marshal” the increasingly rich and complex data that surround us, analysts will not be able to use these data to make better decisions.

Demand is strong for graduates with these skills. According to the company ratings site **Glassdoor**, “data scientist” was the best job in America every year from 2016–2019 (Columbus, 2019).

New data technologies make it possible to extract data from more sources than ever before. Streamlined data processing libraries enable data scientists to express how to restructure those data into a form suitable for analysis. Database systems facilitate the storage and retrieval of ever-larger collections of data. State-of-the-art workflow tools foster well-documented and reproducible analysis. Modern statistical and machine learning methods allow the analyst to fit and assess models as well as to undertake supervised or unsupervised learning to glean information about the underlying real-world phenomena. Contemporary data science requires tight integration of these statistical, computing, data-related, and communication skills.

Intended audience

This book is intended for readers who want to develop the appropriate skills to tackle complex data science projects and “think with data” (as coined by Diane Lambert of Google). The desire to solve problems using data is at the heart of our approach.

We acknowledge that it is impossible to cover all these topics in any level of detail within a single book: Many of the chapters could productively form the basis for a course or series of courses. Instead, our goal is to lay a foundation for analysis of real-world data and to ensure that analysts see the power of statistics and data analysis. After reading this book, readers will have greatly expanded their skill set for working with these data, and should have a newfound confidence about their ability to learn new technologies on-the-fly.

This book was originally conceived to support a one-semester, 13-week undergraduate course in data science. We have found that the book will be useful for more advanced students in related disciplines, or analysts who want to bolster their data science skills. At the same time, **Part I** of the book is accessible to a general audience with no programming or statistics experience.

Key features of this book

Focus on case studies and extended examples

We feature a series of complex, real-world extended case studies and examples from a broad range of application areas, including politics, transportation, sports, environmental science, public health, social media, and entertainment. These rich data sets require the use of sophisticated data extraction techniques, modern data visualization approaches, and refined computational approaches.

Context is king for such questions, and we have structured the book to foster the parallel developments of statistical thinking, data-related skills, and communication. Each chapter focuses on a different extended example with diverse applications, while exercises allow for the development and refinement of the skills learned in that chapter.

Structure

The book has three main sections plus supplementary appendices. **Part I** provides an introduction to data science, which includes an introduction to data visualization, a foundation for data management (or “wrangling”), and ethics. **Part II** extends key modeling notions from introductory statistics, including regression modeling, classification and prediction, statistical foundations, and simulation. **Part III** introduces more advanced topics, including interactive data visualization, SQL and relational databases, geospatial data, text mining, and network science.

We conclude with appendices that introduce the book’s **R** package, **R** and **RStudio**, key aspects of algorithmic thinking, reproducible analysis, a review of regression, and how to set up a local SQL database.

We provide two indices: one organized by subject and the other organized by **R** function and package. In addition, the book features extensive cross-referencing (given the inherent connections between topics and approaches).

Supporting materials

In addition to many examples and extended case studies, the book incorporates exercises at the end of each chapter along with supplementary exercises available online. Many of the exercises are quite open-ended, and are designed to allow students to explore their creativity in tackling data science questions. (A solutions manual for instructors is available from the publisher.)

The book website at <https://mdsr-book.github.io/mdsr2e> includes the table of contents, the full text of each chapter, bibliography, and subject and **R** indices. The instructor’s

website at <https://mdsr-book.github.io/> contains code samples, supplementary exercises, additional activities, and a list of errata.

Changes in the second edition

Data science moves quickly. A lot has changed since we wrote the first edition. We have updated all chapters to account for many of these changes and to take advantage of state-of-the-art **R** packages.

First, the chapter on working with geospatial data has been expanded and split into two chapters. The first focuses on working with geospatial data, and the second focuses on geospatial computations. Both chapters now use the **sf** package and the new **geom_sf()** function in **ggplot2**. These changes allow students to penetrate deeper into the world of geospatial data analysis.

Second, the chapter on *tidy data* has undergone significant revisions. A new section on list-columns has been added, and the section on iteration has been expanded into a full chapter. This new chapter makes consistent use of the functional programming style provided by the **purrr** package. These changes help students develop a habit of mind around scalability: if you are copying-and-pasting code more than twice, there is probably a more efficient way to do it.

Third, the chapter on supervised learning has been split into two chapters and updated to use the **tidymodels** suite of packages. The first chapter now covers model evaluation in generality, while the second introduces several models. The **tidymodels** ecosystem provides a consistent syntax for fitting, interpreting, and evaluating a wide variety of machine learning models, all in a manner that is consistent with the **tidyverse**. These changes significantly reduce the cognitive overhead of the code in this chapter.

The content of several other chapters has undergone more minor—but nonetheless substantive—revisions. All of the code in the book has been revised to adhere more closely to the **tidyverse** syntax and style. Exercises and solutions from the first edition have been revised, and new exercises have been added. The code from each chapter is now available on the book website. The book has been ported to **bookdown**, so that a full version can be found online at <https://mdsr-book.github.io/mdsr2e>.

Key role of technology

While many tools can be used effectively to undertake data science, and the technologies to undertake analyses are quickly changing, **R** and Python have emerged as two powerful and extensible environments. While it is important for data scientists to be able to use multiple technologies for their analyses, we have chosen to focus on the use of **R** and **RStudio** to avoid cognitive overload. We describe a powerful and coherent set of tools that can be introduced within the confines of a single semester and that provide a foundation for data wrangling and exploration.

We take full advantage of the **RStudio** environment. This powerful and easy-to-use front

end adds innumerable features to **R** including package support, code-completion, integrated help, a debugger, and other coding tools. In our experience, the use of **RStudio** dramatically increases the productivity of **R** users, and by tightly integrating reproducible analysis tools, helps avoid error-prone “cut-and-paste” workflows. Our students and colleagues find **RStudio** to be an accessible interface. No prior knowledge or experience with **R** or **RStudio** is required: we include an introduction within the Appendix.

As noted earlier, we have comprehensively integrated many substantial improvements in the **tidyverse**, an opinionated set of packages that provide a more consistent interface to **R** (Wickham, 2019h). Many of the design decisions embedded in the **tidyverse** packages address issues that have traditionally complicated the use of **R** for data analysis. These decisions allow novice users to make headway more quickly and develop good habits.

We used a reproducible analysis system (**knitr**) to generate the example code and output in this book. Code extracted from these files is provided on the book’s website. We provide a detailed discussion of the philosophy and use of these systems. In particular, we feel that the **knitr** and **rmarkdown** packages for **R**, which are tightly integrated with **RStudio**, should become a part of every **R** user’s toolbox. We can’t imagine working on a project without them (and we’ve incorporated reproducibility into all of our courses).

Modern data science is a team sport. To be able to fully engage, analysts must be able to pose a question, seek out data to address it, ingest this into a computing environment, model and explore, then communicate results. This is an iterative process that requires a blend of statistics and computing skills.

How to use this book

The material from this book has supported several courses to date at Amherst, Smith, and Macalester Colleges, as well as many others around the world. From our personal experience, this includes an intermediate course in data science (in 2013 and 2014 at Smith College and since 2017 at Amherst College), an introductory course in data science (since 2016 at Smith), and a capstone course in advanced data analysis (multiple years at Amherst).

The introductory data science course at Smith has no prerequisites and includes the following subset of material:

- Data Visualization: three weeks, covering [Chapters 1–3](#)
- Data Wrangling: five weeks, covering [Chapters 4–7](#)
- Ethics: one week, covering [Chapter 8](#)
- Database Querying: two weeks, covering [Chapter 15](#)
- Geospatial Data: two weeks, covering [Chapter 17](#) and some of [Chapter 18](#)

A intermediate course at Amherst followed the approach of Baumer (2015b) with a prerequisite of some statistics and some computer science and an integrated final project. The course generally covers the following chapters:

- Data Visualization: two weeks, covering [Chapters 1–3](#) and [14](#)
- Data Wrangling: four weeks, covering [Chapters 4–7](#)
- Ethics: one week, covering [Chapter 8](#)
- Unsupervised Learning: one week, covering [Chapter 12](#)
- Database Querying: one week, covering [Chapter 15](#)

- Geospatial Data: one week, covering [Chapter 17](#) and some of [Chapter 18](#)
- Text Mining: one week, covering [Chapter 19](#)
- Network Science: one week, covering [Chapter 20](#)

The capstone course at Amherst reviewed much of that material in more depth:

- Data Visualization: three weeks, covering [Chapters 1–3](#) and [14](#)
- Data Wrangling: two weeks, covering [Chapters 4–7](#)
- Ethics: one week, covering [Chapter 8](#)
- Simulation: one week, covering [Chapter 13](#)
- Statistical Learning: two weeks, covering [Chapters 10–12](#)
- Databases: one week, covering [Chapter 15](#) and [Appendix F](#)
- Text Mining: one week, covering [Chapter 19](#)
- Spatial Data: one week, covering [Chapter 17](#)
- Big Data: one week, covering [Chapter 21](#)

We anticipate that this book could serve as the primary text for a variety of other courses, with or without additional supplementary material.

The content in [Part I](#)—particularly the **ggplot2** visualization concepts presented in [Chapter 3](#) and the **dplyr** data wrangling operations presented in [Chapter 4](#)—is fundamental and is assumed in [Parts II](#) and [III](#). Each of the topics in [Part III](#) are independent of each other and the material in [Part II](#). Thus, while most instructors will want to cover most (if not all) of [Part I](#) in any course, the material in [Parts II](#) and [III](#) can be added with almost total freedom.

The material in [Part II](#) is designed to expose students with a beginner’s understanding of statistics (i.e., basic inference and linear regression) to a richer world of statistical modeling and statistical inference.

Acknowledgments

We would like to thank John Kimmel at Informa CRC/Chapman and Hall for his support and guidance. We also thank Jim Albert, Nancy Boynton, Jon Caris, Mine Çetinkaya-Rundel, Jonathan Che, Patrick Frenett, Scott Gilman, Maria-Cristiana Gírjäu, Johanna Hardin, Alana Horton, John Horton, Kinari Horton, Azka Javaid, Andrew Kim, Eunice Kim, Caroline Kusiak, Ken Kleinman, Priscilla (Wencong) Li, Amelia McNamara, Melody Owen, Randall Pruim, Tanya Riseman, Gabriel Sosa, Katie St. Clair, Amy Wagaman, Susan (Xiaofei) Wang, Hadley Wickham, J. J. Allaire and the RStudio developers, the anonymous reviewers, multiple classes at Smith and Amherst Colleges, and many others for contributions to the **R** and **RStudio** environment, comments, guidance, and/or helpful suggestions on drafts of the manuscript. Rose Porta was instrumental in proofreading and easing the transition from Sweave to R Markdown. Jessica Yu converted and tagged most of the exercises from the first edition to the new format based on **etude**.

Above all we greatly appreciate Cory, Maya, and Julia for their patience and support.

*Northampton, MA and St. Paul, MN
December 2020*



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

Part I

Part I: Introduction to Data Science



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

Prologue: Why data science?

Information is what we want, but data are what we've got. The techniques for transforming data into information go back hundreds of years. A good starting point is 1592 with the publication of John Graunt's weekly "bills of mortality" in London (see Figure 1.1). These "bills" are tabulations—a condensation of data on individual events into a form more readily assimilated by the human reader. Constructing such tabulations was a manual operation.

<p><i>Natural and Political</i> OBSERVATIONS Mentioned in a following INDEX, and made upon the Bills of Mortality.</p> <p>By <i>JOHN GRAUNT</i>, Citizen of LONDON.</p> <p>With reference to the <i>Government, Religion, Trade, Growth, Ayre, Diseases</i>, and the several Changes of the said CITY.</p> <p>— <i>Non, me ut miretur Turba, labore.</i> <i>Contentus paucis Lettoribus</i> —</p> <p>LONDON, Printed by <i>Tho: Roycroft, for John Martin, James Allestry, and Tbo: Dicas</i>, at the Sign of the Bell in St. Paul's Church-yard, MDCLXII.</p>	<p><i>Of the Plague.</i></p> <p>1. Before we leave to discourse of the <i>Casualties</i>, we shall add something concerning that greatest <i>Disease</i>, or <i>Casualty</i> of all, <i>The Plague</i>. There have been in <i>London</i>, within this Age, four Times of great <i>Mortality</i>, that is to say, the years 1592, and 1593, 1603, 1625, and 1636. There died <i>Anno 1592</i> from <i>March to December</i>, 25886 Whereof of the <i>Plague</i>..... 11503 <i>Anno 1593</i>..... 17844 Whereof of the <i>Plague</i>..... 10662 <i>Christned</i> in the said year..... 4021 <i>Anno 1603</i> within the same space of time, were Buried 37294 Whereof of the <i>Plague</i>..... 30561 <i>Anno 1625</i>, within the same space,..... 51758 Whereof of the <i>Plague</i>..... 35417 <i>Anno 1636</i>, from <i>April to December</i>..... 23359 Whereof of the <i>Plague</i>..... 10400</p> <p>2. Now it is manifest of it self, in which of these years most died; but in which of them was the greatest <i>Mortality</i> of all Diseases in general, or of the <i>Plague</i> in particular, we discover thus. In the year 1592, and 1636, we finde the proportion of those dying of the <i>Plague</i> in the whole to be [34] near alike, that is about 10 to 23, or 11 to 25, or as about two to five.</p> <p>3. In the year 1625, we finde the <i>Plague</i> to bear unto the whole in proportion as 35 to 51, or 7 to 10, that is almost the triplicate of the former proportion, for the <i>Cube</i> of 7.</p>
---	--

Figure 1.1: Excerpt from Graunt's bills of mortality. At left, the title page. At right, an excerpt on the plague.

Over the centuries, as data became larger, machines were introduced to speed up the tabulations. A major step was Herman Hollerith's development of punched cards and an electrical tabulating system for the *United States Census of 1890*. This was so successful that Hollerith started a company, *International Business Machines Corporation* (IBM), that came to play an important role in the development of today's electronic computers.

Also in the late 19th century, statistical methods began to develop rapidly. These methods have been tremendously important in interpreting data, but they were not intrinsically tied

to mechanical data processing. Generations of students have learned to carry out statistical operations by hand on small sets of data.

Nowadays, it is common to have data sets that are so large they can be processed only by machine. In this era of *big data*, data are amassed by networks of instruments and computers. The settings where such data arise are diverse: the genome, satellite observations of Earth, entries by Web users, sales transactions, etc. There are new opportunities for finding and characterizing patterns using techniques described as data mining, machine learning, data visualization, and so on. Such techniques require computer processing. Among the tasks that need performing are data cleaning, combining data from multiple sources, and reshaping data into a form suitable as input to data-summarization operations for visualization and modeling.

In writing this book we hope to help people gain the understanding and skills for *data wrangling* (a process of preparing data for visualization and other modern techniques of statistical interpretation) and using those data to answer statistical questions via modeling and visualization. Doing so inevitably involves, at the center, the ability to reason statistically and utilize computational and algorithmic capacities.

The National Academies “Data Science for Undergraduates” consensus report (National Academies of Science, Engineering, and Medicine, 2018) noted that the vital new field of data science spans a wide range of capacities that they described as “data acumen.” Key components that are part of data acumen include mathematical, computational, and statistical foundations, data management and curation, data description and visualization, data modeling and assessment, workflow and reproducibility, communication and teamwork, domain-specific considerations, and ethical problem solving. They suggested that *all* students would benefit from “awareness and competence” in data science.

Is an extended study of computer programming necessary to engage in sophisticated computing? Our view is that it is not.

First, over the last half century, a coherent set of simple data operations have been developed that can be used as the building blocks of sophisticated data wrangling processes. The trick is not mastering programming but rather learning to think in terms of these operations. Much of this book is intended to help you master such thinking. Moreover, while data science involves programming, it is more than just programming.

Second, it is possible to use recent developments in software to vastly reduce the amount of programming needed to use these data operations. We have drawn on such software—particularly **R** and the packages **dplyr** and **ggplot2**—to focus on a small subset of functions that accomplish data wrangling tasks in a concise and expressive way. The programming syntax is consistent enough that with a little practice you should be able to adapt the code contained in this book to solve your own problems. (Experienced **R** programmers will note the distinctive style of **R** statements in this book, including a consistent focus on a small set of functions and extensive use of the “*pipe*” operator.) **Part I** of this book focuses on data wrangling and data visualization as key building blocks for data science.

1.1 What is data science?

We hold a broad view of *data science*—we see it as the science of extracting meaningful information from data. There are several key ideas embedded in that simple definition. First,

data science is a *science*, a rigorous discipline combining elements of statistics and computer science, with roots in mathematics. Michael I. Jordan from the University of California, Berkeley has described data science as a fine-grained blend of intellectual traditions from statistics and computer science:

Computer science is more than just programming; it is the creation of appropriate abstractions to express computational structures and the development of algorithms that operate on those abstractions. Similarly, statistics is more than just collections of estimators and tests; it is the interplay of general notions of sampling, models, distributions and decision-making. [Data science] is based on the idea that these styles of thinking support each other (Pierson, 2016).

Second, data science is best applied in the context of expert knowledge about the domain from which the data originate. This domain might be anything from astronomy to zoology; business and health care are two particularly important domain areas. Third, the distinction between *data* and *information* is the raison d'être of data science. Data scientists are people who are interested in converting the data that is now abundant into actionable information that always seems to be scarce.

Some statisticians might say: “But we already have a field for that: it’s called *statistics*!” The goals of data scientists and statisticians are the same: They both want to extract meaningful information from data. Much of statistical technique was originally developed in an environment where data were scarce and difficult or expensive to collect, so statisticians focused on creating methods that would maximize the strength of inference one is able to make, given the least amount of data. These techniques were often ingenious, involved sophisticated mathematics, and have proven invaluable to the empirical sciences for going on a century. While several of the most influential early statisticians saw computing as an integral part of statistics, it is also true that much of the development of statistical theory was to find mathematical approximations for things that we couldn’t yet compute (Cobb, 2007).

Today, the manner in which we extract meaning from data is different in two ways—both due primarily to advances in computing:

- we are able to compute many more things than we could before, and,
- we have a *lot* more data than we had before.

The first change means that some of the techniques that were ubiquitous in statistics education in the 20th century (e.g., the *t-test*, *ANOVA*) are being replaced by computational techniques that are conceptually simpler but were simply infeasible until the *microcomputer* revolution (e.g., the *bootstrap*, *permutation tests*). The second change means that many of the data we now collect are observational—they don’t come from a designed experiment, and they aren’t really sampled at random. This makes developing realistic probability models for these data much more challenging, which in turn makes formal statistical inference a more challenging (and perhaps less relevant) problem. In some settings (e.g., *clinical trials* and *A/B tests*) the careful estimation of a model parameter using inferential statistical methods is still the primary goal. But in an array of academic, government, and industrial

settings, the end result may instead be a *predictive model*, an interactive visualization of the data, or a *web application* that allows the user to engage with the data to explore questions and extract meaning. We explore issues related to statistical inference and modeling in greater depth in [Part II](#) of this book.

The increasing complexity and heterogeneity of modern data means that each data analysis project needs to be custom-built. Simply put, the modern data analyst needs to be able to read and write computer instructions, the “code” from which data analysis projects are built. [Part I](#) of this book develops foundational abilities in data visualization and data wrangling—two essential skills for the modern data scientist. These chapters focus on the traditional two-dimensional representation of data: rows and columns in a data table, and horizontal and vertical in a data graphic. In [Part III](#), we explore a variety of non-traditional data types (e.g., geospatial, text, network, “big”) and interactive data graphics.

As you work through this book, you will develop computational skills that we describe as “precursors” to big data (Horton et al., 2015). In [Chapter 21](#), we point to some tools for working with truly big data. One has to learn to crawl before one can walk, and we argue that for most people the skills developed herein are more germane to the kinds of problems that you are likely to encounter.

1.2 Case study: The evolution of sabermetrics

The evolution of baseball analytics (often called *sabermetrics*) in many ways recapitulates the evolution of analytics in other domains. Although *domain knowledge* is always useful in data science, no background in baseball is required for this section.¹

The *use* of statistics in baseball has a long and storied history—in part because the game itself is naturally *discrete*, and in part because Henry Chadwick began publishing boxscores in the early 1900s (Schwarz, 2005). For these reasons, a rich catalog of baseball data began to accumulate.

However, while more and more baseball data were piling up, *analysis* of that data was not so prevalent. That is, the extant data provided a means to keep records, and as a result some numerical elements of the game’s history took on a life of their own (e.g., Babe Ruth’s 714 home runs). But it is not as clear how much people were learning about the game of baseball from the data. Knowing that Babe Ruth hit more home runs than Mel Ott tells us something about two players, but doesn’t provide any insight into the nature of the game itself.

In 1947—Jackie Robinson’s rookie season—*Brooklyn Dodgers*’ general manager Branch Rickey made another significant innovation: He hired Allan Roth to be baseball’s first statistical analyst. Roth’s *analysis* of baseball data led to insights that the Dodgers used to win more games. In particular, Roth convinced Rickey that a measurement of how often a batter reaches first base via any means (e.g., hit, walk, or being hit by the pitch) was a better indicator of that batter’s value than how often he reaches first base via a hit (which was—and probably still is—the most commonly cited batting statistic). The logic support-

¹The main rules of baseball are these: Two teams of nine players alternate trying to score runs on a field with four bases (first base, second base, third base, and home). The defensive team pitches while one member of the offensive team bats while standing by home base. A run is scored when an offensive player crosses home plate after advancing in order through the other bases.

ing this insight was based on both Roth’s understanding of the game of baseball (what we call *domain knowledge*) and his statistical analysis of baseball data.

During the next 50 years, many important contributions to baseball analytics were made by a variety of people, most notably “The Godfather of Sabermetrics” Bill James (James, 1986). Most of these sabermetricians had little formal training in statistics. Their tool of choice was often a *spreadsheet*. They were able to use their creativity, domain knowledge, and a keen sense of what the interesting questions were to make ground-breaking discoveries.

The 2003 publication of *Moneyball* (Lewis, 2003)—which showcased how Billy Beane and Paul DePodesta used statistical analysis to run the *Oakland A’s*—triggered a revolution in how front offices in baseball were managed (Baumer and Zimbalist, 2014). Over the next decade, the size of the data expanded so rapidly that a spreadsheet was no longer a viable mechanism for storing—let alone analyzing—all of the available data. Today, many professional sports teams have research and development groups headed by people with Ph.D.’s in statistics or computer science along with graduate training in machine learning (Baumer, 2015a). This is not surprising given that revenue estimates for major league baseball top \$8 billion per year.

The contributions made by the next generation of baseball analysts will *require* coding ability. The creativity and domain knowledge that fueled the work of Allan Roth and Bill James remain necessary traits for success, but they are no longer sufficient. There is nothing special about baseball in this respect—a similar profusion of data are now available in many other areas, including astronomy, health services research, genomics, and climate change, among others. For data scientists of all application domains, creativity, domain knowledge, and technical ability are absolutely essential.

1.3 Datasets

There are many data sets used in this book. The smaller ones are available through either the **mdsr** (see Appendix A) or **mosaicData** packages. Some other data used in this book are pulled directly from the internet—URLs for these data are embedded in the text. There are a few larger, more complicated data sets that we use repeatedly and that warrant some explication here.

- Airline Delays: The *United States Bureau of Transportation Statistics* has collected data on more than 169 million domestic flights dating back to October 1987. These data were used for the 2009 ASA Data Expo (Wickham, 2011) (a subset are available in the MySQL database we have made available through the **mdsr** package). The **ny-cflights13** package contains a proper subset of these data (flights leaving the three most prominent New York City airports in 2013).
- Baseball: The **Lahman** database is maintained by Sean Lahman, a database journalist. Compiled by a team of volunteers, it contains complete seasonal records going back to 1871 and is usually updated yearly. It is available for download both as a pre-packaged SQL file and as an **R** package (Friendly et al., 2020).
- Baby Names: The **babynames** package for **R** provides data about the popularity of individual baby names from the *United States Social Security Administration* (Wickham, 2019c). These data can be used, for example, to track the popularity of certain names over time.
- Federal Election Commission: The **fec** package (Baumer and Gjekmarkaj, 2017) pro-

vides access to campaign spending data for recent federal elections maintained by the Federal Election Commission. These data include contributions by individuals to committees, spending by those committees on behalf, or against individual candidates for president, the Senate, and the House of Representatives, as well information about those committees and candidates. The **fec12** and **fec16** packages provide that information for single election cycles in a simplified form (Tapal et al., 2020b).

- MacLeish: The Ada and Archibald MacLeish field station is a 260-acre plot of land owned and operated by *Smith College*. It is used by faculty, students, and members of the local community for environmental research, outdoor activities, and recreation. The **macleish** package allows you to download and process weather data as a time series from the MacLeish Field Station using the **etl** framework (Baumer et al., 2020). It also contains shapefiles for contextualizing spatial information.
 - Movies: The Internet Movie Database is a massive repository of information about movies (IMDb.com, 2013). The easiest way to get the IMDb data into SQL is by using the open-source **IMDbPY** Python package (Alberani, 2014).
 - Restaurant Violations: The **mdsr** package contains data on restaurant health inspections made by the New York City Health Department.
-

1.4 Further resources

Each chapter features a list of additional resources that serve as a definitive reference for a given topic, provide further detail, or suggest additional material to explore. Definitions of data science and data analytics abound. See Donoho (2017), De Veaux et al. (2017), Cobb (2015), Horton and Hardin (2015), Hardin et al. (2015), Finzer (2013), Provost and Fawcett (2013), and Cleveland (2001) for some examples. More information regarding the components of data acumen can be found in National Academies of Science, Engineering, and Medicine (2018).

2

Data visualization

Data graphics provide one of the most accessible, compelling, and expressive modes to investigate and depict patterns in data. This chapter will motivate why well-designed data graphics are important and describe a taxonomy for understanding their composition. If you are seeing this material for the first time, you will never look at data graphics the same way again—yours will soon be a more critical lens.

2.1 The 2012 federal election cycle

Every four years, the presidential election draws an enormous amount of interest in the United States. The most prominent candidates announce their candidacy nearly two years before the November elections, beginning the process of raising the hundreds of millions of dollars necessary to orchestrate a national campaign. In many ways, the experience of running a successful presidential campaign is in itself evidence of the leadership and organizational skills necessary to be *commander-in-chief*.

Voices from all parts of the political spectrum are critical of the influence of money upon political campaigns. While the contributions from individual citizens to individual candidates are limited in various ways, the Supreme Court's decision in *Citizens United v. Federal Election Commission* allows unlimited political spending by corporations (non-profit or otherwise). This has resulted in a system of committees (most notably, *political action committees*, PACs) that can accept unlimited contributions and spend them on behalf of (or against) a particular candidate or set of candidates. Unraveling the complicated network of campaign spending is a subject of great interest.

To perform that unraveling is an exercise in data science. The *Federal Election Commission* (FEC) maintains a website with logs of not only all of the (\$200 or more) contributions made by individuals to candidates and committees, but also of spending by committees on behalf of (and against) candidates. Of course, the FEC also maintains data on which candidates win elections, and by how much. These data sources are separate, and it requires some ingenuity to piece them together. We will develop these skills in [Chapters 4–6](#), but for now, we will focus on graphical displays of the information that can be gleaned from these data. Our emphasis at this stage is on making intelligent decisions about how to display certain data, so that a clear (and correct) message is delivered.

Among the most basic questions is: How much money did each candidate raise? However, the convoluted campaign finance network makes even this simple question difficult to answer, and—perhaps more importantly—less meaningful than we might think. A better question is: On whose candidacy was the most money spent? In [Figure 2.1](#), we show a bar graph of the amount of money (in millions of dollars) that were spent by committees on particular candidates during the general election phase of the *2012 federal election cycle*. This includes

candidates for president, the *United States Senate*, and the *United States House of Representatives*. Only candidates on whose campaign at least \$4 million was spent are included in Figure 2.1.

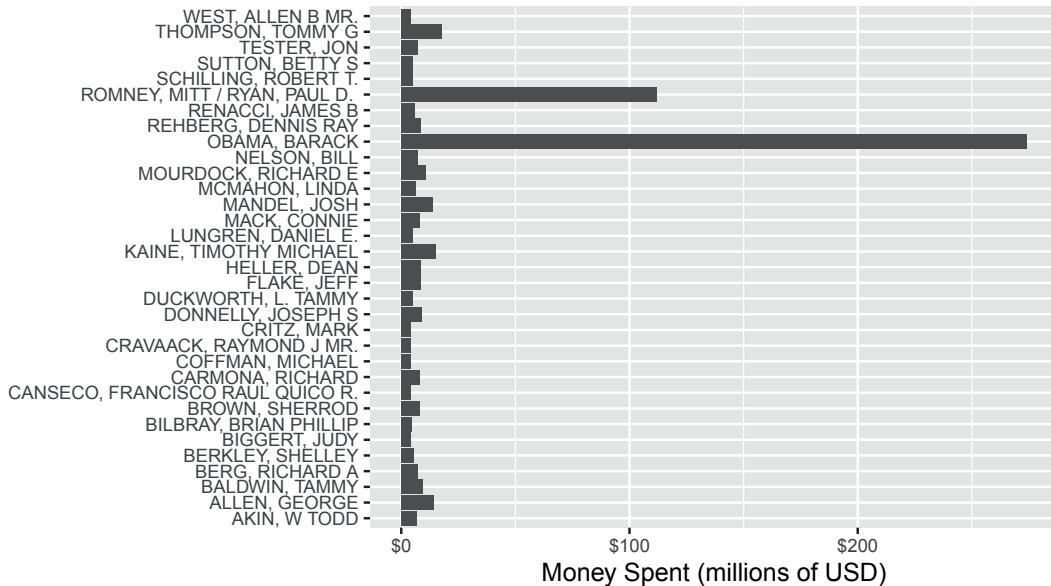


Figure 2.1: Amount of money spent on individual candidates in the general election phase of the 2012 federal election cycle, in millions of dollars. Candidacies with at least \$4 million in spending are depicted.

It seems clear from Figure 2.1 that President Barack Obama’s re-election campaign spent far more money than any other candidate, in particular more than doubling the amount of money spent by his *Republican* challenger, Mitt Romney. However, committees are not limited to spending money in support of a candidate—they can also spend money **against** a particular candidate (*attack ads*). In Figure 2.2, we separate the same spending shown in Figure 2.1 by whether the money was spent for or against the candidate.

In these elections, most of the money was spent against each candidate, and in particular, \$251 million of the \$274 million spent on President Obama’s campaign was spent against his candidacy. Similarly, most of the money spent on Mitt Romney’s campaign was against him, but the percentage of negative spending on Romney’s campaign (70%) was lower than that of Obama (92%).

The difference between Figure 2.1 and Figure 2.2 is that in the latter we have used color to bring a third variable (type of spending) into the plot. This allows us to make a clear comparison that importantly changes the conclusions we might draw from the former plot. In particular, Figure 2.1 makes it appear as though President Obama’s *war chest* dwarfed that of Romney, when in fact the opposite was true.

2.1.1 Are these two groups different?

Since so much more money was spent attacking Obama’s campaign than Romney’s, you might conclude from Figure 2.2 that Republicans were more successful in fundraising during this election cycle. In Figure 2.3, we can confirm that this was indeed the case, since more money was spent supporting Republican candidates than Democrats, and more money was

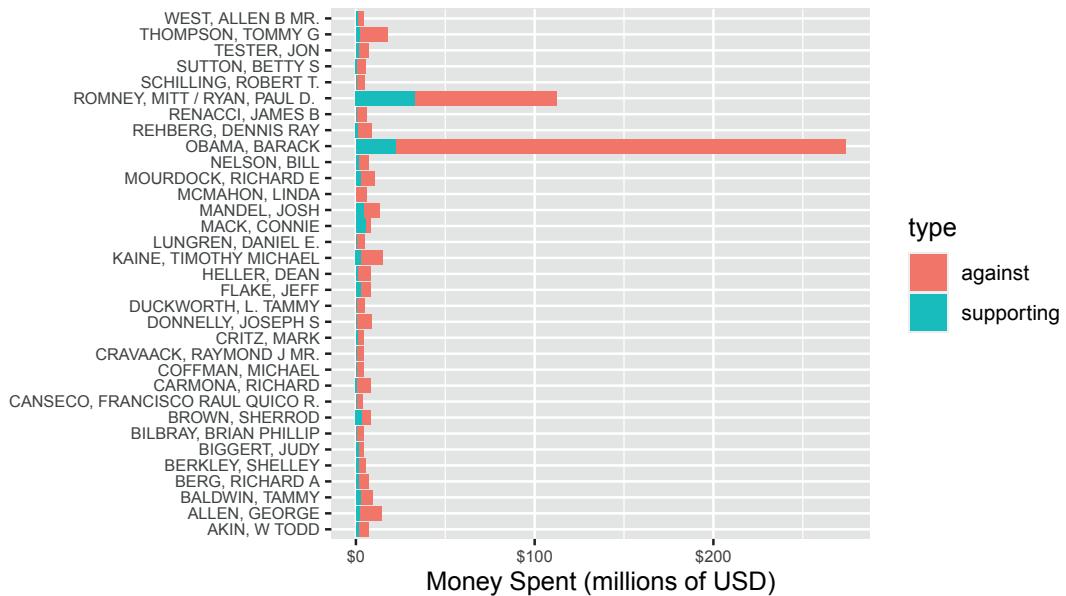


Figure 2.2: Amount of money spent on individual candidates in the general election phase of the 2012 federal election cycle, in millions of dollars, broken down by type of spending. Candidacies with at least \$4 million in spending are depicted.

spent attacking Democratic candidates than Republican. It also seems clear from Figure 2.3 that nearly all of the money was spent on either Democrats or Republicans.¹

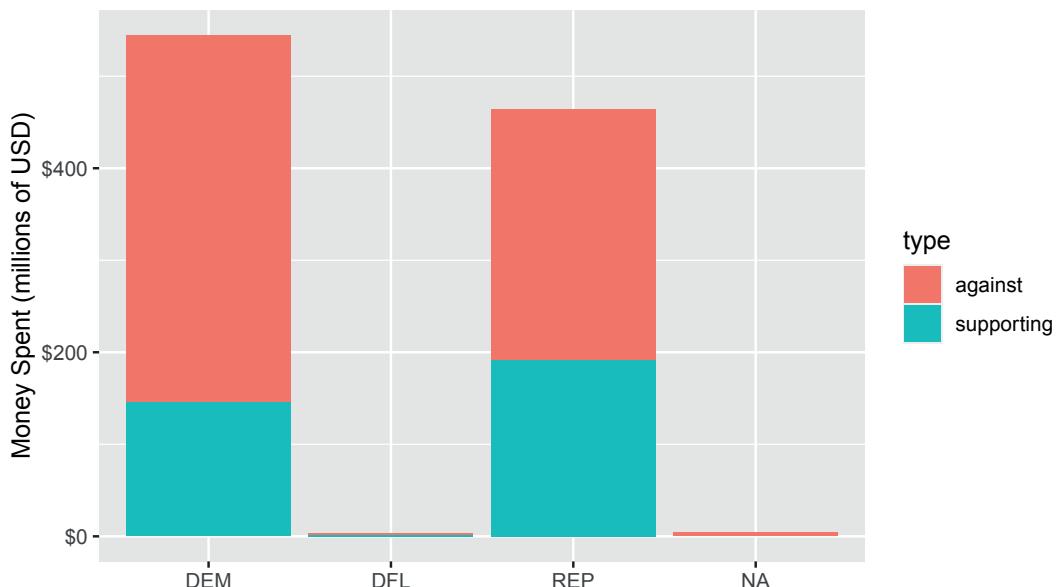


Figure 2.3: Amount of money spent on individual candidacies by political party affiliation during the general election phase of the 2012 federal election cycle.

¹DFL stands for the *Minnesota Democratic–Farmer–Labor Party*, which is affiliated with the Democratic Party.

However, the question of whether the money spent on candidates really differed by party affiliation is a bit thornier. As we saw above, the presidential election dominated the political donations in this election cycle. Romney faced a serious disadvantage in trying to unseat an incumbent president. In this case, the office being sought is a confounding variable. By further subdividing the contributions in [Figure 2.3](#) by the office being sought, we can see in [Figure 2.4](#) that while more money was spent supporting Republican candidates for all elective branches of government, it was only in the presidential election that more money was spent attacking Democratic candidates. In fact, slightly more money was spent attacking Republican House and Senate candidates.

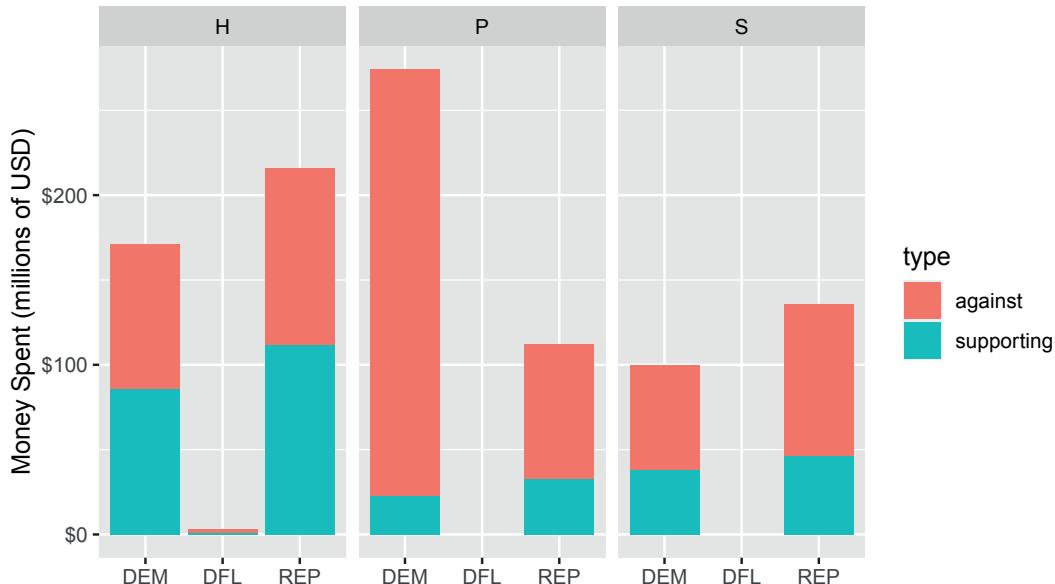


Figure 2.4: Amount of money spent on individual candidacies by political party affiliation during the general election phase of the 2012 federal election cycle, broken down by office being sought (House, President, or Senate).

Note that [Figures 2.3](#) and [2.4](#) display the same data. In [Figure 2.4](#), we have an additional variable that provides an important clue into the mystery of campaign finance. Our choice to include that variable results in [Figure 2.4](#) conveying substantially more meaning than [Figure 2.3](#), even though both figures are “correct.” In this chapter, we will begin to develop a framework for creating principled data graphics.

2.1.2 Graphing variation

One theme that arose during the presidential election was the allegation that Romney’s campaign was supported by a few rich donors, whereas Obama’s support came from people across the economic spectrum. If this were true, then we would expect to see a difference in the distribution of donation amounts between the two candidates. In particular, we would expect to see this in the histograms shown in [Figure 2.5](#), which summarize the more than one million donations made by individuals to the two major committees that supported each candidate (for Obama, Obama for America, and the Obama Victory Fund 2012; for Romney, Romney for President, and Romney Victory 2012). We do see some evidence for this claim in [Figure 2.5](#), Obama did appear to receive more smaller donations, but the evidence is far from conclusive. One problem is that both candidates received many small donations but

just a few larger donations; the scale on the horizontal axis makes it difficult to actually see what is going on. Secondly, the histograms are hard to compare in a side-by-side placement. Finally, we have lumped all of the donations from both phases of the presidential election (i.e., primary vs. general) in together.

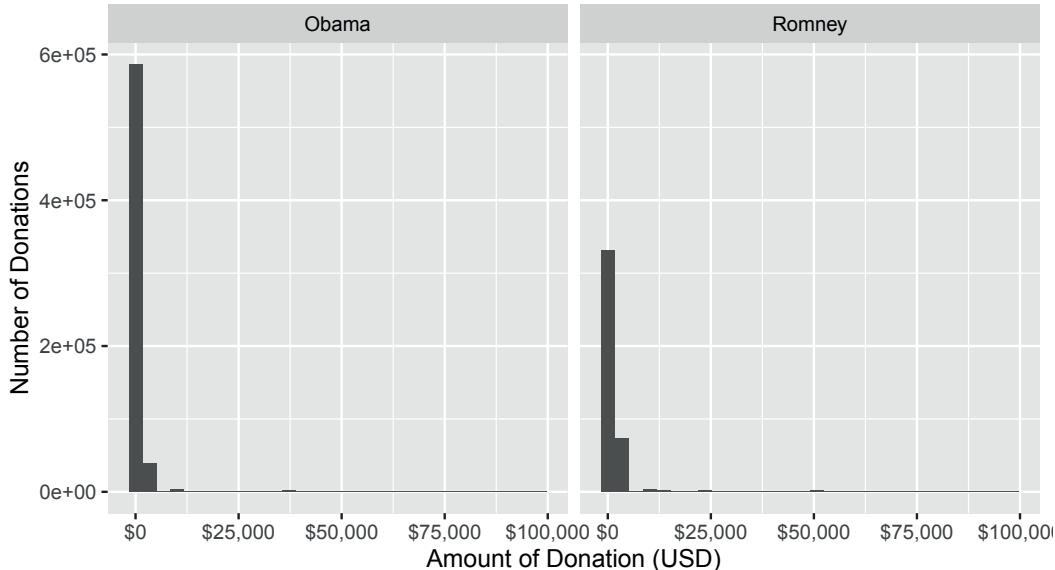


Figure 2.5: Donations made by individuals to the PACs supporting the two major presidential candidates in the 2012 election.

In [Figure 2.6](#), we remedy these issues by (1) using density curves instead of histograms, so that we can compare the distributions directly, (2) plotting the logarithm of the donation amount on the horizontal scale to focus on the data that are important, and (3) separating the donations by the phase of the election. [Figure 2.6](#) allows us to make more nuanced conclusions. The right panel supports the allegation that Obama's donations came from a broader base during the primary election phase. It does appear that more of Obama's donations came in smaller amounts during this phase of the election. However, in the general phase, there is virtually no difference in the distribution of donations made to either campaign.

2.1.3 Examining relationships among variables

Naturally, the biggest questions raised by the *Citizens United* decision are about the influence of money in elections. If campaign spending is unlimited, does this mean that the candidate who generates the most spending on their behalf will earn the most votes? One way that we might address this question is to compare the amount of money spent on each candidate in each election with the number of votes that candidate earned. Statisticians will want to know the *correlation* between these two quantities—when one is high, is the other one likely to be high as well?

Since all 435 members of the United States House of Representatives are elected every two years, and the districts contain roughly the same number of people, House elections provide a nice data set to make this type of comparison. In [Figure 2.7](#), we show a simple scatterplot relating the number of dollars spent on behalf of the Democratic candidate against the number of votes that candidate earned for each of the House elections.

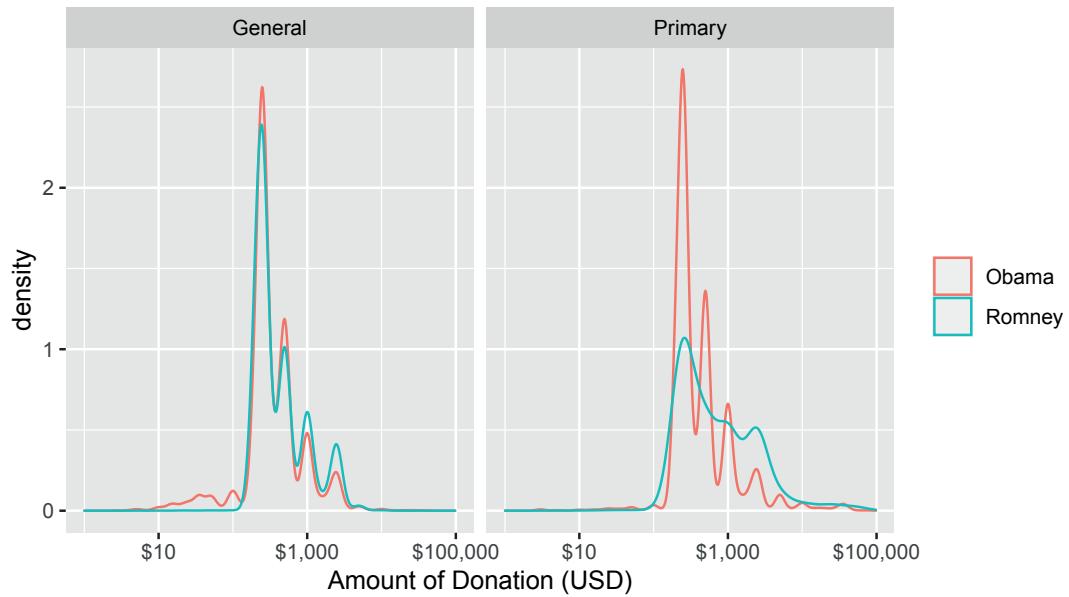


Figure 2.6: Donations made by individuals to the PACs supporting the two major presidential candidates in the 2012 election, separated by election phase.

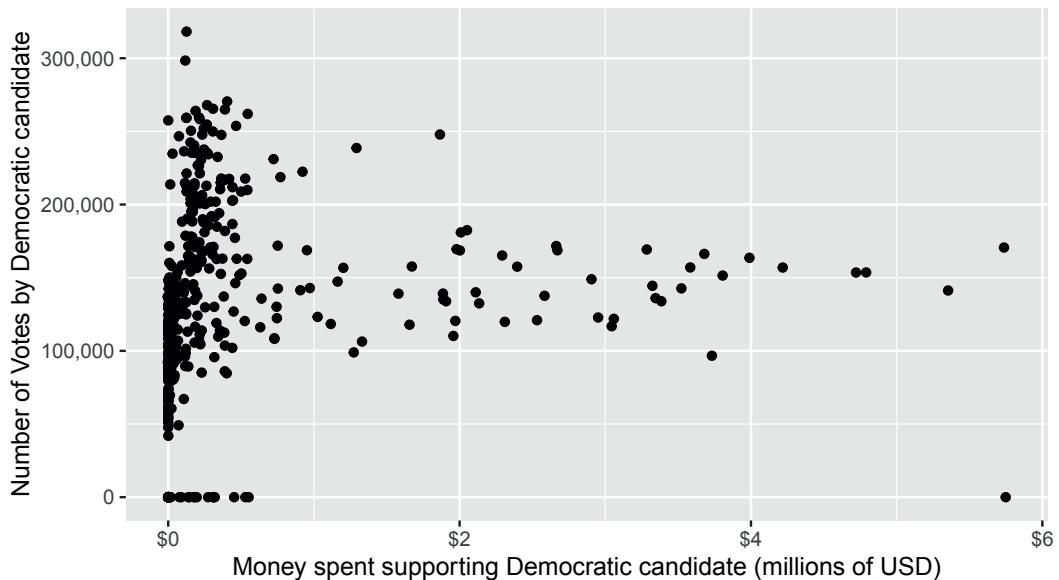


Figure 2.7: Scatterplot illustrating the relationship between number of dollars spent supporting and number of votes earned by Democrats in 2012 elections for the House of Representatives.

The relationship between the two quantities depicted in [Figure 2.7](#) is very weak. It does not appear that candidates who benefited more from campaign spending earned more votes. However, the comparison in [Figure 2.7](#) is misleading. On both axes, it is not the *amount* that is important, but the *proportion*. Although the population of each congressional district is similar, they are not the same, and voter turnout will vary based on a variety of factors. By comparing the proportion of the vote, we can control for the size of the voting population in each district. Similarly, it makes less sense to focus on the total amount of money spent, as opposed to the proportion of money spent. In [Figure 2.8](#), we present the same comparison, but with both axes scaled to proportions.

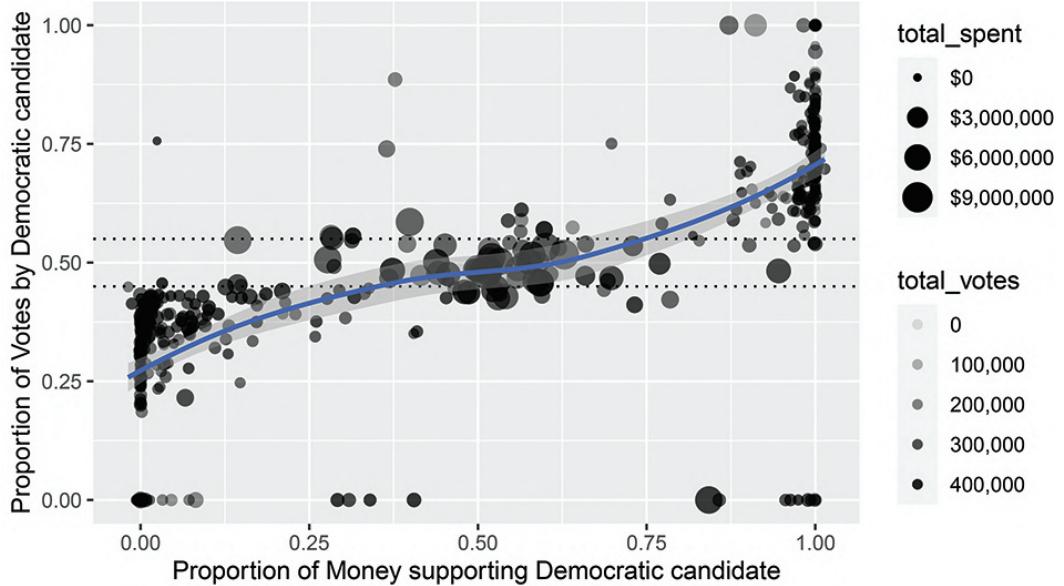


Figure 2.8: Scatterplot illustrating the relationship between proportion of dollars spent supporting and proportion of votes earned by Democrats in the 2012 House of Representatives elections. Each dot represents one district. The size of each dot is proportional to the total spending in that election, and the alpha transparency of each dot is proportional to the total number of votes in that district.

[Figure 2.8](#) captures many nuances that were impossible to see in [Figure 2.7](#). First, there *does* appear to be a positive association between the percentage of money supporting a candidate and the percentage of votes that they earn. However, that relationship is of greatest interest towards the center of the plot, where elections are actually contested. Outside of this region, one candidate wins more than 55% of the vote. In this case, there is usually very little money spent. These are considered “safe” House elections—you can see these points on the plot because most of them are close to $x = 0$ or $x = 1$, and the dots are very small. For example, one of the points in the lower-left corner is the *8th district in Ohio*, which was won by the then *Speaker of the House* John Boehner, who ran unopposed. The election in which the most money was spent (over \$11 million) was also in Ohio. In the 16th district, Republican incumbent Jim Renacci narrowly defeated Democratic challenger Betty Sutton, who was herself an incumbent from the 13th district. This battle was made possible through decennial redistricting (see [Chapter 17](#)). Of the money spent in this election, 51.2% was in support of Sutton but she earned only 48.0% of the votes.

In the center of the plot, the dots are bigger, indicating that more money is being spent on

these contested elections. Of course this makes sense, since candidates who are fighting for their political lives are more likely to fundraise aggressively. Nevertheless, the evidence that more financial support correlates with more votes in contested elections is relatively weak.

2.1.4 Networks

Not all relationships among variables are sensibly expressed by a scatterplot. Another way in which variables can be related is in the form of a network (we will discuss these in more detail in [Chapter 20](#)). In this case, campaign funding has a network structure in which individuals donate money to committees, and committees then spend money on behalf of candidates. While the national campaign funding network is far too complex to show here, in [Figure 2.9](#) we display the funding network for candidates from *Massachusetts*.

In [Figure 2.9](#), we see that the two campaigns that benefited the most from committee spending were Republicans Mitt Romney and Scott Brown. This is not surprising, since Romney was running for president and received massive donations from the Republican National Committee, while Brown was running to keep his Senate seat in a heavily Democratic state against a strong challenger, Elizabeth Warren. Both men lost their elections. The constellation of blue dots are the congressional delegation from Massachusetts, all of whom are Democrats.

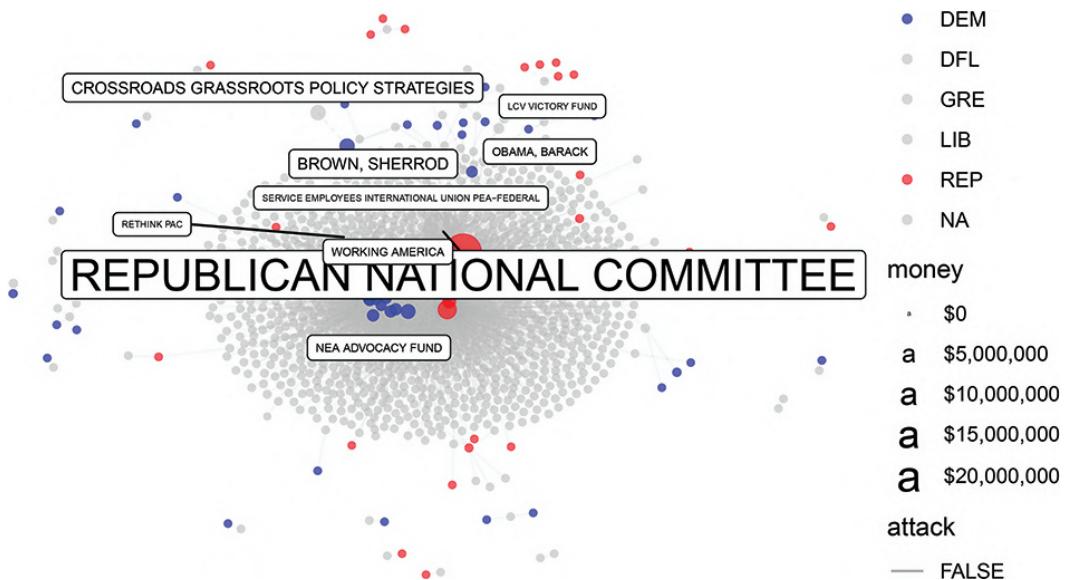


Figure 2.9: Campaign funding network for candidates from Massachusetts, 2012 federal elections. Each edge represents a contribution from a PAC to a candidate.

2.2 Composing data graphics

Former *New York Times* intern and FlowingData.com creator Nathan Yau makes the analogy that creating data graphics is like cooking: Anyone can learn to type graphical commands and generate plots on the computer. Similarly, anyone can heat up food in a mi-

Table 2.1: Visual cues and what they signify.

Visual Cue	Variable Type	Question
Position	numerical	where in relation to other things?
Length	numerical	how big (in one dimension)?
Angle	numerical	how wide? parallel to something else?
Direction	numerical	at what slope? in a time series, going up or down?
Shape	categorical	belonging to which group?
Area	numerical	how big (in two dimensions)?
Volume	numerical	how big (in three dimensions)?
Shade	either	to what extent? how severely?
Color	either	to what extent? how severely?

crowd. What separates a high-quality visualization from a plain one are the same elements that separate great chefs from novices: mastery of their tools, knowledge of their ingredients, insight, and creativity (Yau, 2013). In this section, we present a framework—rooted in scientific research—for understanding data graphics. Our hope is that by internalizing these ideas you will refine your data graphics palette.

2.2.1 A taxonomy for data graphics

The taxonomy presented in Yau (2013) provides a systematic way of thinking about how data graphics convey specific pieces of information and how they could be improved. A complementary *grammar* of graphics (Wilkinson et al., 2005) is implemented by Hadley Wickham in the **ggplot2** graphics package (Wickham, 2016), albeit using slightly different terminology. For clarity, we will postpone discussion of **ggplot2** until [Chapter 3](#). (To extend our cooking analogy, you must learn to taste before you can learn to cook well.)

In this framework, data graphics can be understood in terms of four basic elements: visual cues, coordinate systems, scale, and context. In what follows, we explicate this vision and append a few additional items (facets and layers). This section should equip the careful reader with the ability to systematically break down data graphics, enabling a more critical analysis of their content.

2.2.1.1 Visual Cues

Visual cues are graphical elements that draw the eye to what you want your audience to focus upon. They are the fundamental building blocks of data graphics, and the choice of which visual cues to use to represent which quantities is the central question for the data graphic composer. [Table 2.1](#) identifies nine distinct visual cues, for which we also list whether that cue is used to encode a numerical or categorical quantity:

Research into graphical perception (dating back to the mid-1980s) has shown that human beings' ability to perceive differences in magnitude accurately descends in this order (Cleveland and McGill, 1984). That is, humans are quite good at accurately perceiving differences in position (e.g., how much taller one bar is than another), but not as good at perceiving differences in angles. This is one reason why many people prefer bar charts to *pie charts*. Our relatively poor ability to perceive differences in color is a major factor in the relatively low opinion of *heat maps* that many data scientists have.

2.2.1.2 Coordinate systems

How are the data points organized? While any number of coordinate systems are possible, three are most common:

- **Cartesian:** The familiar (x, y) -rectangular coordinate system with two perpendicular axes.
- **Polar:** The radial analog of the Cartesian system with points identified by their radius ρ and angle θ .
- **Geographic:** The increasingly important system in which we have locations on the curved surface of the Earth, but we are trying to represent these locations in a flat two-dimensional plane. We will discuss such geospatial analyses in [Chapter 17](#).

An appropriate choice for a coordinate system is critical in representing one's data accurately, since, for example, displaying geospatial data like airline routes on a flat *Cartesian plane* can lead to gross distortions of reality (see [Section 17.3.2](#)).

2.2.1.3 Scale

Scales translate values into visual cues. The choice of scale is often crucial. The central question is *how* does distance in the data graphic translate into meaningful differences in quantity? Each coordinate axis can have its own scale, for which we have three different choices:

- **Numeric:** A numeric quantity is most commonly set on a *linear*, *logarithmic*, or *percentage* scale. Note that a logarithmic scale does not have the property that, say, a one-centimeter difference in position corresponds to an equal difference in quantity anywhere on the scale.
- **Categorical:** A categorical variable may have no ordering (e.g., Democrat, Republican, or Independent), or it may be *ordinal* (e.g., never, former, or current smoker).
- **Time:** A numeric quantity that has some special properties. First, because of the calendar, it can be demarcated by a series of different units (e.g., year, month, day, etc.). Second, it can be considered periodically (or cyclically) as a “wrap-around” scale. Time is also so commonly used and misused that it warrants careful consideration.

Misleading with scale is easy, since it has the potential to completely distort the relative positions of data points in any graphic.

2.2.1.4 Context

The purpose of data graphics is to help the viewer make *meaningful* comparisons, but a bad data graphic can do just the opposite: It can instead focus the viewer's attention on meaningless artifacts, or ignore crucial pieces of relevant but external knowledge. Context can be added to data graphics in the form of titles or subtitles that explain what is being shown, axis labels that make it clear how units and scale are depicted, or reference points or lines that contribute relevant external information. While one should avoid cluttering up a data graphic with excessive annotations, it is necessary to provide proper context.

2.2.1.5 Small multiples and layers

One of the fundamental challenges of creating data graphics is condensing multivariate information into a two-dimensional image. While three-dimensional images are occasionally useful, they are often more confusing than anything else. Instead, here are three common ways of incorporating more variables into a two-dimensional data graphic:

- **Small multiples:** Also known as *facets*, a single data graphic can be composed of several small multiples of the same basic plot, with one (discrete) variable changing in each of the small sub-images.
- **Layers:** It is sometimes appropriate to draw a new layer on top of an existing data graphic. This new layer can provide context or comparison, but there is a limit to how many layers humans can reliably parse.
- **Animation:** If time is the additional variable, then an animation can sometimes effectively convey changes in that variable. Of course, this doesn't work on the printed page and makes it impossible for the user to see all the data at once.

2.2.2 Color

Color is one of the flashiest, but most misperceived and misused visual cues. In making color choices, there are a few key ideas that are important for any data scientist to understand.

First, as we saw above, color and its monochromatic cousin *shade* are two of the most poorly perceived visual cues. Thus, while potentially useful for a small number of levels of a categorical variable, color and shade are not particularly faithful ways to represent numerical variables—especially if small differences in those quantities are important to distinguish. This means that while color can be visually appealing to humans, it often isn't as informative as we might hope. For two numeric variables, it is hard to think of examples where color and shade would be more useful than position. Where color can be most effective is to represent a *third* or *fourth* numeric quantity on a scatterplot—once the two position cues have been exhausted.

Second, approximately 8% of the population—most of whom are men—have some form of color blindness. Most commonly, this renders them incapable of seeing colors accurately, most notably of distinguishing between red and green. Compounding the problem, many of these people do not know that they are color-blind. Thus, for professional graphics it is worth thinking carefully about which colors to use. The *National Football League* famously failed to account for this in a 2015 game in which the *Buffalo Bills* wore all-red jerseys and the *New York Jets* wore all-green, leaving colorblind fans unable to distinguish one team from the other!

Pro Tip 1. *To prevent issues with color blindness, avoid contrasting red with green in data graphics. As a bonus, your plots won't seem Christmas-y!*

Thankfully, we have been freed from the burden of having to create such intelligent palettes by the research of Cynthia Brewer, creator of the ColorBrewer website (and inspiration for the **RColorBrewer R package**). Brewer has created colorblind-safe palettes in a variety of hues for three different types of numeric data in a single variable:

- **Sequential:** The ordering of the data has only one direction. Positive integers are sequential because they can only go up: they can't go past 0. (Thus, if 0 is encoded as white, then any darker shade of gray indicates a larger number.)
- **Diverging:** The ordering of the data has two directions. In an election forecast, we commonly see states colored based on how they are expected to vote for the president. Since red is associated with Republicans and blue with Democrats, states that are solidly red or blue are on opposite ends of the scale. But “swing states” that could go either way may appear purple, white, or some other neutral color that is “between” red and blue (see [Figure 2.10](#)).

- **Qualitative:** There is no ordering of the data, and we simply need color to differentiate different categories.

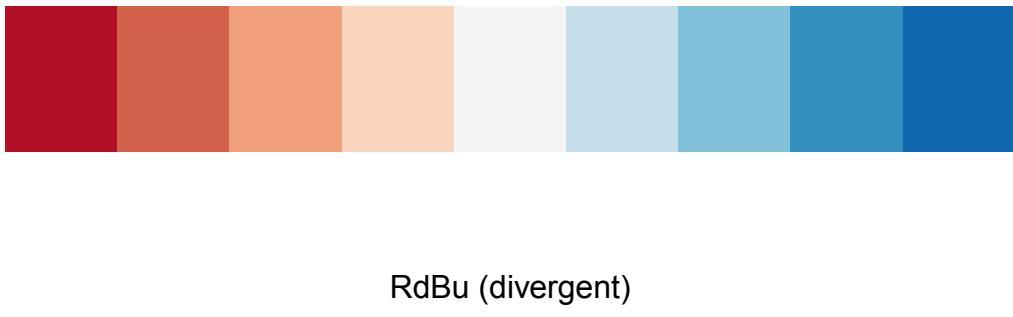


Figure 2.10: Diverging red-blue color palette.

The **RColorBrewer** package provides functionality to use these palettes directly in **R**. [Figure 2.11](#) illustrates the sequential, qualitative, and diverging palettes built into **RColorBrewer**.

Pro Tip 2. *Take the extra time to use a well-designed color palette. Accept that those who work with color for a living will probably choose better colors than you.*

Other excellent perceptually distinct color palettes are provided by the **viridis** package. These palettes mimic those that are used in the *matplotlib* plotting library for **Python**. The **viridis** palettes are also accessible in **ggplot2** through, for example, the `scale_color_viridis()` function.

2.2.3 Dissecting data graphics

With a little practice, one can learn to dissect data graphics in terms of the taxonomy outlined above. For example, your basic scatterplot uses *position* in the *Cartesian* plane with *linear* scales to show the relationship between two variables. In what follows, we identify the visual cues, coordinate system, and scale in a series of simple data graphics.

1. The bar graph in [Figure 2.12](#) displays the average score on the math portion of the 1994–1995 *SAT* (with possible scores ranging from 200 to 800) among states for whom at least two-thirds of the students took the SAT.

This plot uses the visual cue of *length* to represent the math SAT score on the vertical axis with a *linear* scale. The *categorical* variable of *state* is arrayed on the horizontal axis. Although the states are ordered alphabetically, it would not be appropriate to consider the *state* variable to be ordinal, since the ordering is not meaningful in the context of math SAT scores. The coordinate system is *Cartesian*, although as noted previously, the horizontal coordinate is meaningless. Context is provided by the axis labels and title. Note also that since 200 is the minimum score possible on each section of the SAT, the vertical axis has been constrained to start at 200.

2. Next, we consider a time series that shows the progression of the world record times in the *100-meter freestyle* swimming event for men and women. [Figure 2.13](#) displays the times as a function of the year in which the new record was set.

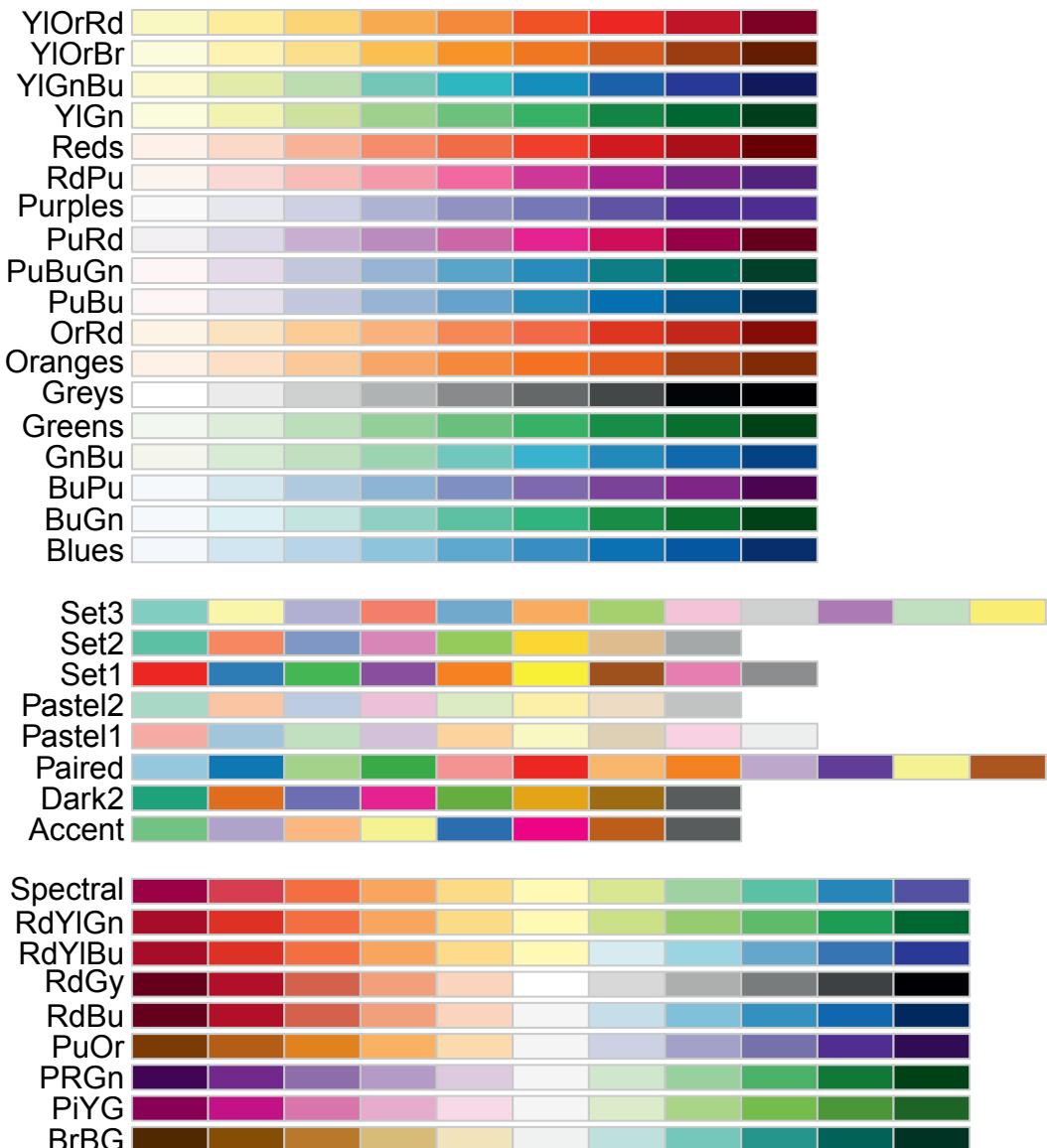


Figure 2.11: Palettes available through the **RColorBrewer** package.

At some level this is simply a scatterplot that uses *position* on both the vertical and horizontal axes to indicate swimming time and chronological time, respectively, in a *Cartesian plane*. The numeric scale on the vertical axis is linear, in units of seconds, while the scale on the horizontal axis is also linear, measured in years. But there is more going on here. Color is being used as a visual cue to distinguish the categorical variable *sex*. Furthermore, since the points are connected by lines, *direction* is being used to indicate the progression of the record times. (In this case, the records can only get faster, so the direction is always down.) One might even argue that *angle* is being used to compare the descent of the world records across time and/or gender. In fact, in this case *shape* is also being used to distinguish *sex*.

3. Next, we present two pie charts in [Figure 2.14](#) indicating the different substance of

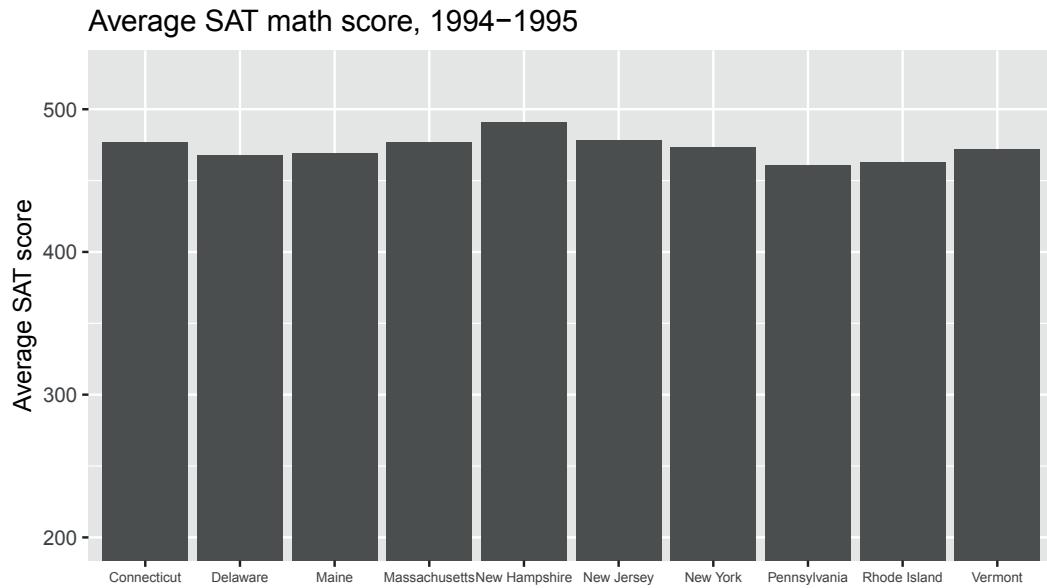


Figure 2.12: Bar graph of average SAT scores among states with at least two-thirds of students taking the test.

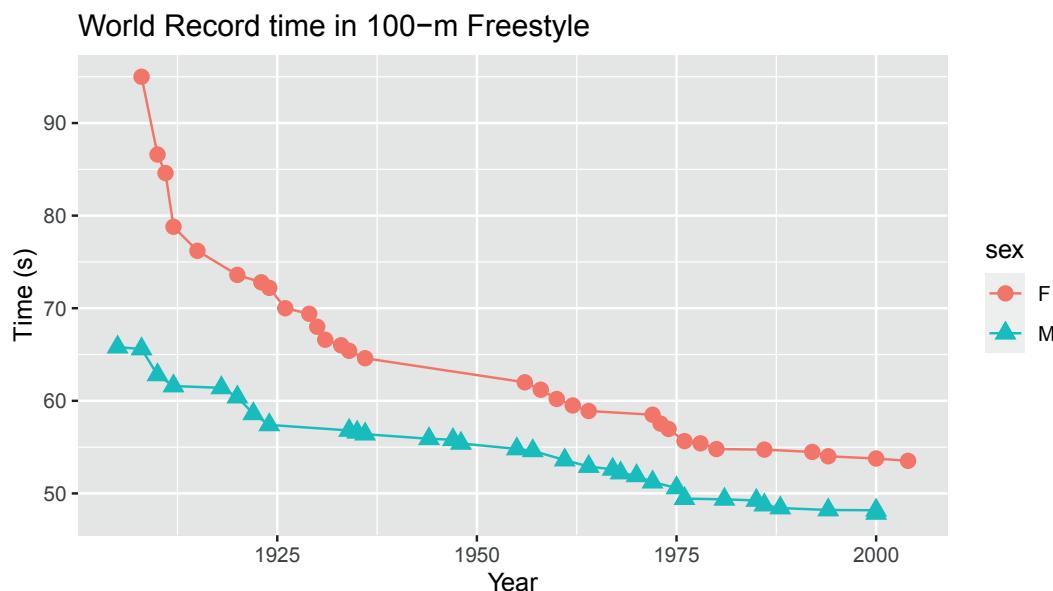


Figure 2.13: Scatterplot of world record time in 100-m freestyle swimming.

abuse for subjects in the *Health Evaluation and Linkage to Primary Care* (HELP) clinical trial (Samet et al., 2003). Each subject was identified with involvement with one primary substance (alcohol, cocaine, or heroin). On the right, we see the distribution of substance for housed (no nights in shelter or on the street) participants is fairly evenly distributed, while on the left, we see the same distribution for those who were homeless one or more nights (more likely to have alcohol as their primary substance of abuse).

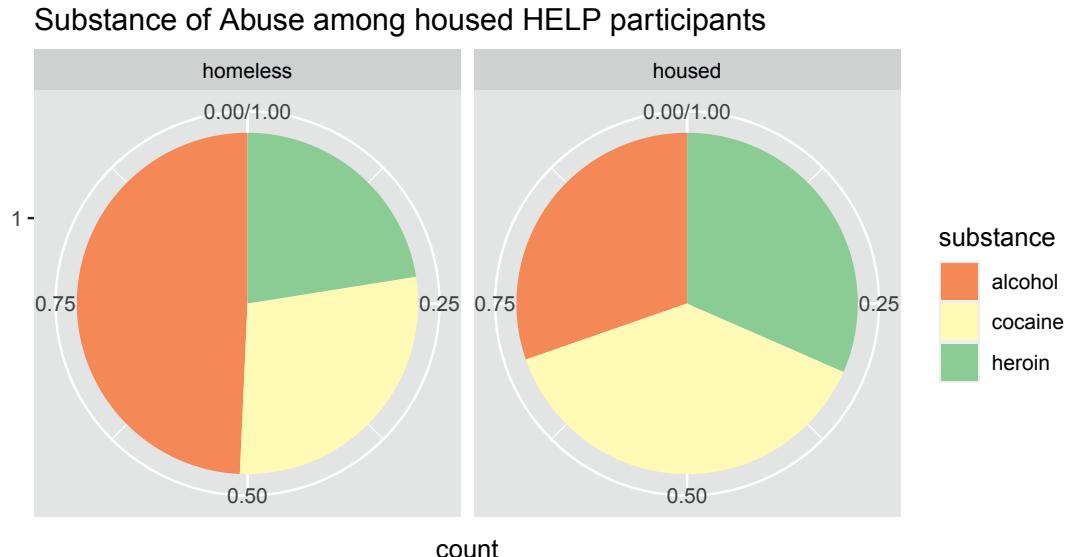


Figure 2.14: Pie charts showing the breakdown of substance of abuse among HELP study participants, faceted by homeless status. Compare this to [Figure 3.13](#).

This graphic uses a *radial* coordinate system and the visual cue of *color* to distinguish the three levels of the *categorical* variable *substance*. The visual cue of *angle* is being used to quantify the differences in the proportion of patients using each substance. Are you able to accurately identify these percentages from the figure? The actual percentages are shown as follows.

```
# A tibble: 3 x 3
  substance Homeless      Housed
  <fct>     <chr>        <chr>
1 alcohol    n = 103 (49.3%) n = 74 (30.3%)
2 cocaine    n = 59 (28.2%) n = 93 (38.1%)
3 heroin     n = 47 (22.5%) n = 77 (31.6%)
```

This is a case where a simple table of these proportions is more effective at communicating the true differences than this—and probably any—data graphic. Note that there are only six data points presented, so any graphic is probably gratuitous.

Pro Tip 3. *Don't use pie charts, except perhaps in small multiples.*

4. Finally, in [Figure 2.15](#) we present a *choropleth map* showing the population of Massachusetts by the 2010 Census tracts.

Massachusetts Census Tracts by Population

Based on 2010 US Census

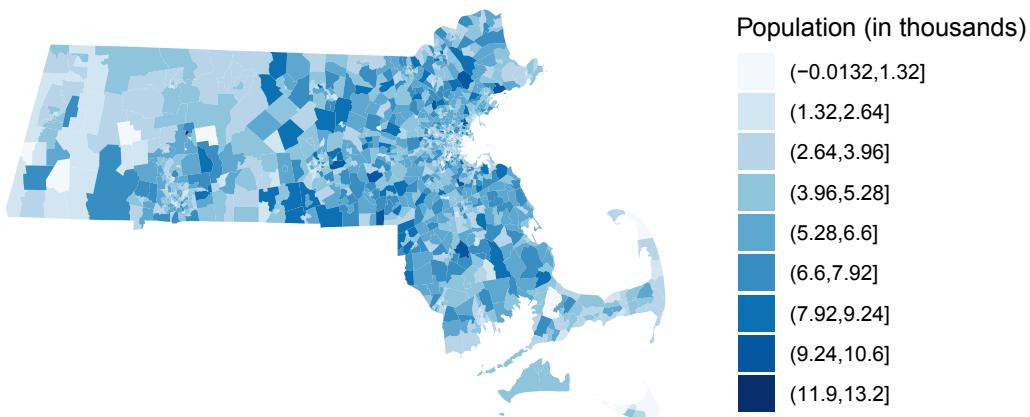


Figure 2.15: Choropleth map of population among Massachusetts Census tracts, based on 2010 US Census.

Clearly, we are using a *geographic* coordinate system here, with *latitude* and *longitude* on the vertical and horizontal axes, respectively. (This plot is not projected: More information about projection systems is provided in [Chapter 17](#).) *Shade* is once again being used to represent the quantity *population*, but here the scale is more complicated. The ten shades of blue have been mapped to the *deciles* of the census tract populations, and since the distribution of population across these tracts is *right-skewed*, each shade does not correspond to a range of people of the same width, but rather to the same number of tracts that have a population in that range. Helpful context is provided by the title, subtitle, and legend.

2.3 Importance of data graphics: *Challenger*

On January 27th, 1986, engineers at *Morton Thiokol*, who supplied solid rocket motors (SRMs) to *NASA* for the *space shuttle*, recommended that NASA delay the launch of the space shuttle *Challenger* due to concerns that the cold weather forecast for the next day's launch would jeopardize the stability of the rubber *O-rings* that held the rockets together. These engineers provided 13 charts that were reviewed over a two-hour conference call involving the engineers, their managers, and NASA. The engineers' recommendation was overruled due to a lack of persuasive evidence, and the launch proceeded on schedule. The O-rings failed in exactly the manner the engineers had feared 73 seconds after launch, *Challenger* exploded, and all seven astronauts on board died (Tufte, 1997).

In addition to the tragic loss of life, the incident was a devastating blow to NASA and the United States space program. The hand-wringing that followed included a two-and-a-half year hiatus for NASA and the formation of the *Rogers Commission* to study the disaster. What became clear is that the Morton Thiokol engineers had correctly identified the key causal link between *temperature* and *O-ring damage*. They did this using statistical data analysis combined with a plausible physical explanation: in short, that the rubber O-rings became brittle in low temperatures. (This link was famously demonstrated by legendary

physicist and Rogers Commission member Richard Feynman during the hearings, using a glass of water and some ice cubes (Tufte, 1997). Thus, the engineers were able to identify the critical weakness using their *domain knowledge*—in this case, rocket science—and their data analysis.

Their failure—and its horrific consequences—was one of persuasion: They simply did not present their evidence in a convincing manner to the NASA officials who ultimately made the decision to proceed with the launch. More than 30 years later this tragedy remains critically important. The evidence brought to the discussions about whether to launch was in the form of hand written data tables (or “charts”), but none were graphical. In his sweeping critique of the incident, Edward Tufte created a powerful scatterplot similar to Figures 2.16 and 2.17, which were derived from data that the engineers had at the time, but in a far more effective presentation (Tufte, 1997).

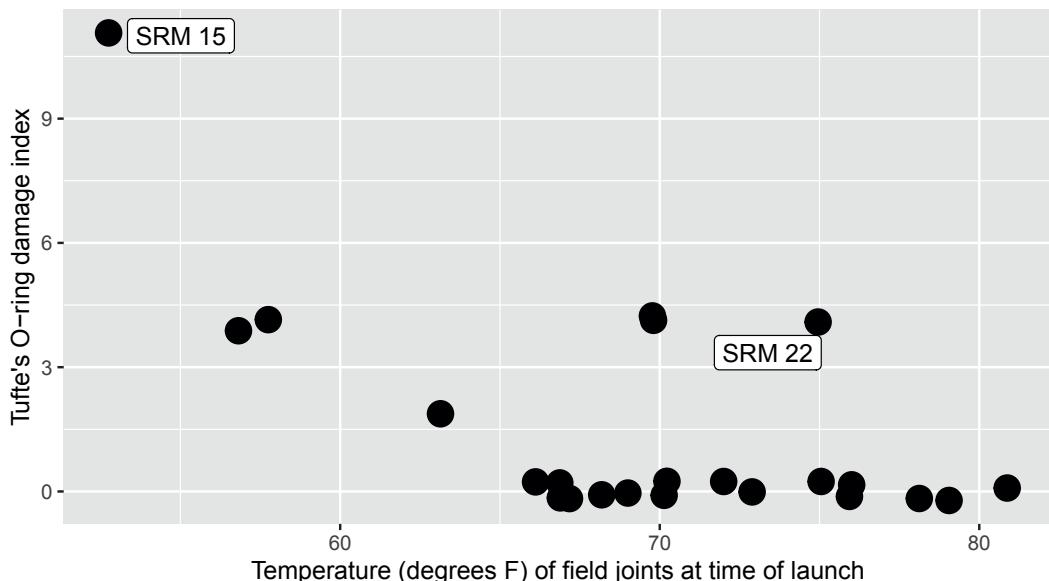


Figure 2.16: A scatterplot with smoother demonstrating the relationship between temperature and O-ring damage on solid rocket motors. The dots are semi-transparent, so that darker dots indicate multiple observations with the same values.

Figure 2.16 indicates a clear relationship between the ambient temperature and O-ring damage on the solid rocket motors. To demonstrate the dramatic extrapolation made to the predicted temperature on January 27th, 1986, Tufte extended the horizontal axis in his scatterplot (Figure 2.17) to include the forecast temperature. The huge gap makes plain the problem with extrapolation. Reprints of two Morton Thiokol data graphics are shown in Figures 2.18 and 2.19 (Tufte, 1997).

Tufte provides a full critique of the engineers’ failures (Tufte, 1997), many of which are instructive for data scientists.

- **Lack of authorship:** There were no names on any of the charts. This creates a lack of accountability. No single person was willing to take responsibility for the data contained in any of the charts. It is much easier to refute an argument made by a group of nameless people, than to a single or group of named people.
- **Univariate analysis:** The engineers provided several data tables, but all were essentially

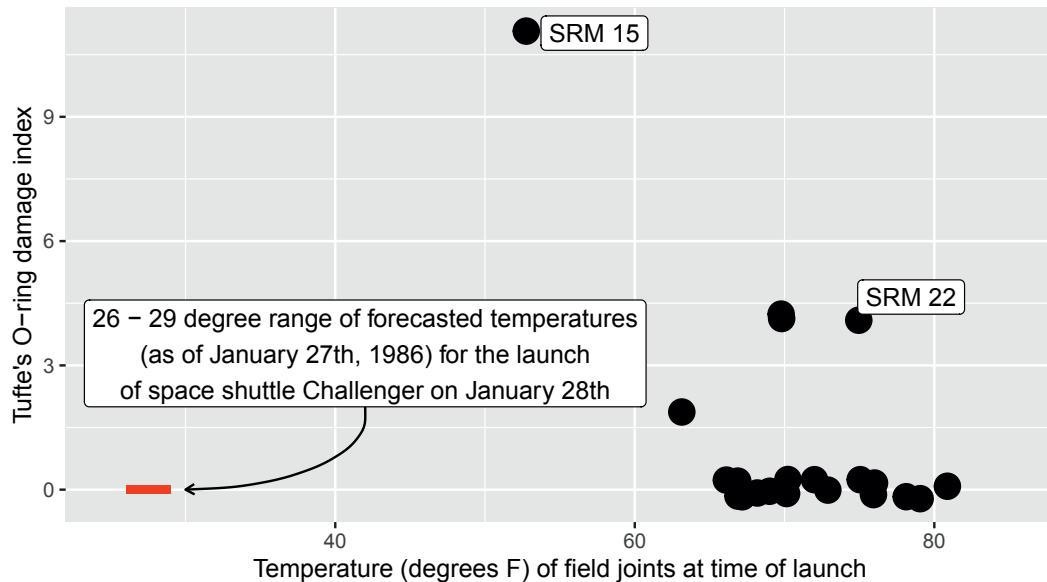


Figure 2.17: A recreation of Tufte's scatterplot demonstrating the relationship between temperature and O-ring damage on solid rocket motors.

HISTORY OF O-RING TEMPERATURES (DEGREES - F)				
MOTOR	MBT	AMB	O-RING	WIND
OM-1	68	36	47	10 MPH
OM-2	76	45	52	10 MPH
QM-3	72.5	40	48	10 MPH
QM-4	76	48	51	10 MPH
SRM-15	52	64	53	10 MPH
SRM-22	77	78	75	10 MPH
SRM-25	55	26	29 27	10 MPH 25 MPH

Figure 2.18: One of the original 13 charts presented by Morton Thiokol engineers to NASA on the conference call the night before the *Challenger* launch. This is one of the more data-intensive charts.

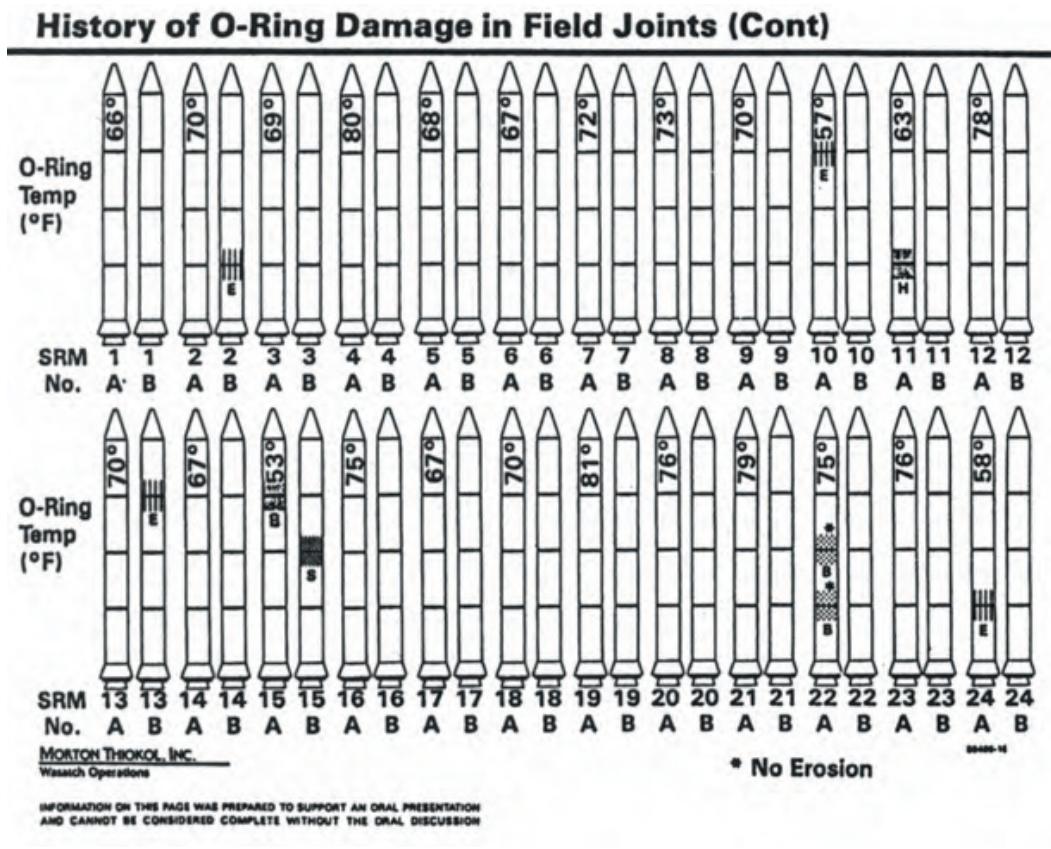


Figure 2.19: Evidence presented during the congressional hearings after the *Challenger* explosion.

univariate. That is, they presented data on a single variable, but did not illustrate the relationship between two variables. Note that while Figure 2.18 does show data for two different variables, it is very hard to see the connection between the two in tabular form. Since the crucial connection here was between temperature and O-ring damage, this lack of bivariate analysis was probably the single most damaging omission in the engineers' presentation.

- **Anecdotal evidence:** With such a small sample size, anecdotal evidence can be particularly challenging to refute. In this case, a bogus comparison was made based on two observations. While the engineers argued that SRM-15 had the most damage on the coldest previous launch date (see Figure 2.17), NASA officials were able to counter that SRM-22 had the second-most damage on one of the warmer launch dates. These anecdotal pieces of evidence fall apart when all of the data are considered in context—in Figure 2.17, it is clear that SRM-22 is an outlier that deviates from the general pattern—but the engineers never presented all of the data in context.
- **Omitted data:** For some reason, the engineers chose not to present data from 22 other flights, which collectively represented 92% of launches. This may have been due to time constraints. This dramatic reduction in the accumulated evidence played a role in enabling the anecdotal evidence outlined above.
- **Confusion:** No doubt working against the clock, and most likely working in tandem, the engineers were not always clear about two different types of damage: *erosion* and *blow-by*.

A failure to clearly define these terms may have hindered understanding on the part of NASA officials.

- **Extrapolation:** Most forcefully, the failure to include a simple scatterplot of the full data obscured the “stupendous extrapolation” (Tufte, 1997) necessary to justify the launch. The bottom line was that the forecast launch temperature (between 26 and 29 degrees *Fahrenheit*) was so much colder than anything that had occurred previously, any model for O-ring damage as a function of temperature would be untested.

Pro Tip 4. *When more than a handful of observations are present, data graphics are often more revealing than tables. Always consider alternative representations to improve communication.*

Tufte notes that the cardinal sin of the engineers was a failure to frame the data *in relation to what?* The notion that certain data may be understood in relation to something is perhaps the fundamental and defining characteristic of statistical reasoning. We will follow this thread throughout the book.

Pro Tip 5. *Always ensure that graphical displays are clearly described with appropriate axis labels, additional text descriptions, and a caption.*

We present this tragic episode in this chapter as motivation for a careful study of data visualization. It illustrates a critical truism for practicing data scientists: Being right isn’t enough—you have to be *convincing*. Note that Figure 2.19 contains the same data that are present in Figure 2.17 but in a far less suggestive format. It just so happens that for most human beings, graphical explanations are particularly persuasive. Thus, to be a successful data analyst, one must master at least the basics of data visualization.

2.4 Creating effective presentations

Giving effective presentations is an important skill for a data scientist. Whether these presentations are in academic conferences, in a classroom, in a boardroom, or even on stage, the ability to communicate to an audience is of immeasurable value. While some people may be naturally more comfortable in the limelight, everyone can improve the quality of their presentations.

A few pieces of general advice are warranted (Ludwig, 2012):

- **Budget your time:** Often you will only have a few minutes to speak and usually a few additional minutes to answer questions. If your talk runs too short or too long, it makes you seem unprepared. Rehearse your talk several times in order to get a better feel for your timing. Note also that you may have a tendency to talk faster during your actual talk than you will during your rehearsal. Talking faster in order to speed up is a bad strategy—you are much better off simply cutting material ahead of time. You will probably have a hard time getting through x slides in x minutes.

Pro Tip 6. *Talking faster in order to speed up is not a good strategy—you are much better off simply cutting material ahead of time or moving to a key slide or conclusion.*

- **Don't write too much on each slide:** You don't want people to have to read your slides, because if the audience is reading your slides, then they aren't listening to you. You want your slides to provide visual cues to the points that you are making—not substitute for your spoken words. Concentrate on graphical displays and bullet-pointed lists of ideas.
 - **Put your problem in context:** Remember that (in most cases) most of your audience will have little or no knowledge of your subject matter. The easiest way to lose people is to dive right into technical details that require prior domain knowledge. Spend a few minutes at the beginning of your talk introducing your audience to the most basic aspects of your topic and presenting some motivation for what you are studying.
 - **Speak loudly and clearly:** Remember that (in most cases) you know more about your topic than anyone else in the room, so speak and act with confidence!
 - **Tell a story, but not necessarily the whole story:** It is unrealistic to expect that you can tell your audience everything that you know about your topic in x minutes. You should strive to convey the big ideas in a clear fashion but not dwell on the details. Your talk will be successful if your audience is able to walk away with an understanding of what your research question was, how you addressed it, and what the implications of your findings are.
-

2.5 The wider world of data visualization

Thus far our discussion of data visualization has been limited to static, two-dimensional data graphics. However, there are many additional ways to visualize data. While [Chapter 3](#) focuses on static data graphics, [Chapter 14](#) presents several cutting-edge tools for making interactive data visualizations. Even more broadly, the field of *visual analytics* is concerned with the science behind building interactive visual interfaces that enhance one's ability to reason about data.

Finally, we have *data art*. You can do many things with data. On one end of the spectrum, you might be focused on predicting the outcome of a specific response variable. In such cases, your goal is very well-defined and your success can be quantified. On the other end of the spectrum are projects called *data art*, wherein the meaning of what you are doing with the data is elusive, but the experience of viewing the data in a new way is in itself meaningful.

Consider Memo Akten and Quayola's *Forms*, which was inspired by the physical movement of athletes in the *Commonwealth Games*. Through video analysis, these movements were translated into three-dimensional digital objects shown in [Figure 2.20](#). Note how the image in the upper-left is evocative of a swimmer surfacing after a dive. When viewed as a movie, *Forms* is an arresting example of data art.

Pro Tip 7. Watch *Forms* (process) from Memo Akten on [Vimeo](#).

Successful data art projects require both artistic talent and technical ability. *Before Us is the Salesman's House* is a live, continuously-updating exploration of the online marketplace *eBay*. This installation was created by statistician Mark Hansen and digital artist Jer Thorpe and is projected on a big screen as you enter eBay's campus.

Pro Tip 8. Watch *Before us is the Salesman's House—Three Cycles* from [blprnt](#) on [Vimeo](#).



Figure 2.20: Still images from *Forms*, by Memo Akten and Quayola. Each image represents an athletic movement made by a competitor at the Commonwealth Games, but reimagined as a collection of moving three-dimensional digital objects. Reprinted with permission.

The display begins by pulling up Arthur Miller’s classic play *Death of a Salesman*, and “reading” the text of the first chapter. Along the way, several nouns are plucked from the text (e.g., flute, refrigerator, chair, bed, trophy, etc.). For each in succession, the display then shifts to a geographic display of where things with that noun in the description are *currently* being sold on eBay, replete with price and auction information. (Note that these descriptions are not always perfect. In the video, a search for “refrigerator” turns up a T-shirt of former *Chicago Bears* defensive end William [Refrigerator] Perry.)

Next, one city where such an item is being sold is chosen, and any classic books of American literature being sold nearby are collected. One is chosen, and the cycle returns to the beginning by “reading” the first page of that book. This process continues indefinitely. When describing the exhibit, Hansen spoke of “one data set reading another.” It is this interplay of data and literature that makes such data art projects so powerful.

Finally, we consider another Mark Hansen collaboration, this time with Ben Rubin and Michele Gorman. In *Shakespeare Machine*, 37 digital LCD blades—each corresponding to one of Shakespeare’s plays—are arrayed in a circle. The display on each blade is a pattern of words culled from the text of these plays. First, pairs of hyphenated words are shown. Next, Boolean pairs (e.g., “good or bad”) are found. Third, articles and adjectives modifying nouns (e.g., “the holy father”). In this manner, the artistic masterpieces of Shakespeare are shattered into formulaic chunks. In [Chapter 19](#), we will learn how to use *regular expressions* to find the data for *Shakespeare Machine*.

Pro Tip 9. Watch *Shakespeare Machine* by Ben Rubin, Mark Hansen, Michele Gorman on Vimeo.

2.6 Further resources

While issues related to data visualization pervade this entire text, they will be the particular focus of [Chapters 3](#) (Data visualization II), 14 (Data visualization III), and 17 (Geospatial data).

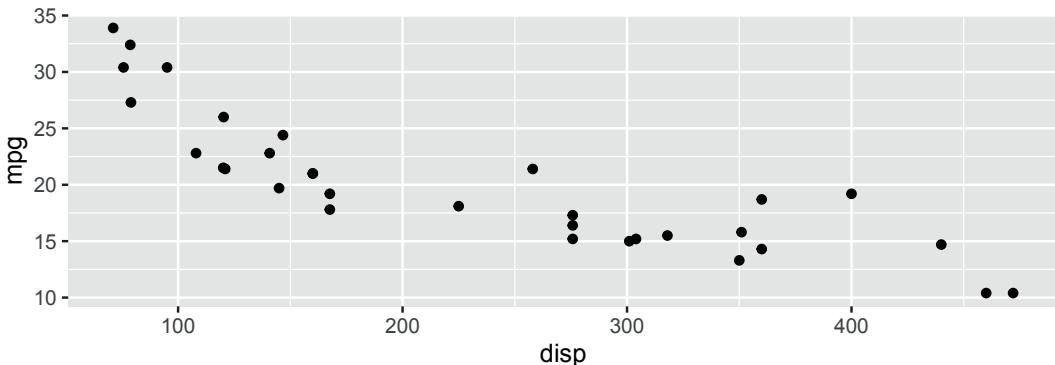
No education in data graphics is complete without reading Tufte’s *Visual Display of Quantitative Information* (Tufte, 2001), which also contains a description of John Snow’s cholera map (see [Chapter 17](#)). For a full description of the *Challenger* incident, see (Tufte, 1997). Tufte has also published two other landmark books (Tufte, 1990, 2006), as well as reasoned polemics about the shortcomings of *PowerPoint* (Tufte, 2003). Cleveland and McGill (1984) provide the foundation for Yau’s taxonomy (Yau, 2013). Yau (2011) provides many examples of thought-provoking data visualizations, particularly data art. The grammar of graphics was first described by Wilkinson et al. (2005). Wickham (2016) implemented **ggplot2** based on this formulation.

Many important data graphics were developed by Tukey (1990). Gelman et al. (2002) have also written persuasively about data graphics in statistical journals. Gelman discusses a set of canonical data graphics as well as Tufte’s suggested modifications to them. Nolan and Perrett (2016) discuss data visualization assignments and rubrics that can be used to grade them. Steven J. Murdoch has created some **R** functions for drawing the kind of modified diagrams described in Tufte (2001). These also appear in the **ggtreemap** package (Arnold, 2019a).

Cynthia Brewer's color palettes are available at <http://colorbrewer2.org> and through the **RColorBrewer** package. Her work is described in more detail in Brewer (1994) and Brewer (1999). The **viridis** (Garnier, 2018a) and **viridisLite** (Garnier, 2018b) packages provide *matplotlib*-like palettes for **R**. Ram and Wickham (2018) created the whimsical color palette that evokes Wes Anderson's distinctive movies. Technically Speaking is an NSF-funded project for presentation advice that contains instructional videos for students (Ludwig, 2012).

2.7 Exercises

Problem 1 (Easy): Consider the following data graphic.



The **am** variable takes the value **0** if the car has automatic transmission and **1** if the car has manual transmission. How could you differentiate the cars in the graphic based on their transmission type?

Problem 2 (Medium): Pick one of the Science Notebook entries at <https://www.edwardtufte.com/tufte> (e.g., "Making better inferences from statistical graphics"). Write a brief reflection on the graphical principles that are illustrated by this entry.

Problem 3 (Medium): Find two graphs published in a newspaper or on the internet in the last two years.

- a. Identify a graphical display that you find compelling. What aspects of the display work well, and how do these relate to the principles established in this chapter? Include a screen shot of the display along with your solution.
- b. Identify a graphical display that you find less than compelling. What aspects of the display don't work well? Are there ways that the display might be improved? Include a screen shot of the display along with your solution.

Problem 4 (Medium): Find two scientific papers from the last two years in a peer-reviewed journal (*Nature* and *Science* are good choices).

- a. Identify a graphical display that you find compelling. What aspects of the display work well, and how do these relate to the principles established in this chapter? Include a screen shot of the display along with your solution.

- b. Identify a graphical display that you find less than compelling. What aspects of the display don't work well? Are there ways that the display might be improved? Include a screen shot of the display along with your solution.

Problem 5 (Medium): Consider the two graphics related to *The New York Times* "Taxmageddon" article at <http://www.nytimes.com/2012/04/15/sunday-review/coming-soon-taxmageddon.html>. The first is "Whose Tax Rates Rose or Fell" and the second is "Who Gains Most From Tax Breaks."

- a. Examine the two graphics carefully. Discuss what you think they convey. What story do the graphics tell?
- b. Evaluate both graphics in terms of the taxonomy described in this chapter. Are the scales appropriate? Consistent? Clearly labeled? Do variable dimensions exceed data dimensions?
- c. What, if anything, is misleading about these graphics?

Problem 6 (Medium): Consider the data graphic <http://tinyurl.com/nytimes-unplanned> about birth control methods.

- a. What quantity is being shown on the y -axis of each plot?
- b. List the variables displayed in the data graphic, along with the units and a few typical values for each.
- c. List the visual cues used in the data graphic and explain how each visual cue is linked to each variable.
- d. Examine the graphic carefully. Describe, in words, what *information* you think the data graphic conveys. Do not just summarize the *data*—interpret the data in the context of the problem and tell us what it means. (Note: *information* is meaningful to human beings—it is not the same thing as *data*.)

2.8 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-vizI.html#datavizI-online-exercises>



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

3

A grammar for graphics

In [Chapter 2](#), we presented a taxonomy for understanding data graphics. In this chapter, we illustrate how the **ggplot2** package can be used to create data graphics. Other packages for creating static, two-dimensional data graphics in **R** include **base** graphics and the **lattice** system. We employ the **ggplot2** system because it provides a unifying framework—a grammar—for describing and specifying graphics. The grammar for specifying graphics will allow the creation of custom data graphics that support visual display in a purposeful way. We note that while the terminology used in **ggplot2** is not the same as the taxonomy we outlined in [Chapter 2](#), there are many close parallels, which we will make explicit.

3.1 A grammar for data graphics

The **ggplot2** package is one of the many creations of prolific **R** programmer Hadley Wickham. It has become one of the most widely-used **R** packages, in no small part because of the way it builds data graphics incrementally from small pieces of code.

In the grammar of **ggplot2**, an *aesthetic* is an explicit mapping between a variable and the visual cues that represent its values. A *glyph* is the basic graphical element that represents one case (other terms used include “mark” and “symbol”). In a scatterplot, the *positions* of a glyph on the plot—in both the horizontal and vertical senses—are the *visual cues* that help the viewer understand how big the corresponding quantities are. The *aesthetic* is the mapping that defines these correspondences. When more than two variables are present, additional aesthetics can marshal additional visual cues. Note also that some visual cues (like *direction* in a time series) are implicit and do not have a corresponding aesthetic.

For many of the chapters in this book, the first step in following these examples will be to load the **mdsr** package, which contains many of the data sets referenced in this book. In addition, we load the **tidyverse** package, which in turn loads **dplyr** and **ggplot2**. (For more information about the **mdsr** package see [Appendix A](#). If you are using **R** for the first time, please see [Appendix B](#) for an introduction.)

```
library(mdsr)
library(tidyverse)
```

Pro Tip 10. *If you want to learn how to use a particular command, we highly recommend running the example code on your own.*

We begin with a data set that includes measures that are relevant to answer questions about economic productivity. The **CIAcountries** data table contains seven variables collected for each of 236 countries: population (**pop**), area (**area**), gross domestic product (**gdp**), percentage of GDP spent on education (**educ**), length of roadways per unit area (**roadways**), Internet

Table 3.1: A selection of variables from the first six rows of the CIA countries data table.

country	oil_prod	gdp	educ	roadways	net_users
Afghanistan	0	1900	NA	0.065	>5%
Albania	20510	11900	3.3	0.626	>35%
Algeria	1420000	14500	4.3	0.048	>15%
American Samoa	0	13000	NA	1.211	NA
Andorra	NA	37200	NA	0.684	>60%
Angola	1742000	7300	3.5	0.041	>15%

use as a fraction of the population (`net_users`), and the number of barrels of oil produced per day (`oil_prod`). **Table 3.1** displays a selection of variables for the first six countries.

3.1.1 Aesthetics

In the simple scatterplot shown in [Figure 3.1](#), we employ the grammar of graphics to build a multivariate data graphic. In **ggplot2**, the `ggplot()` command creates a plot object `g`, and any arguments to that function are applied across any subsequent plotting directives. In this case, this means that any variables mentioned anywhere in the plot are understood to be within the `CIACountries` data frame, since we have specified that in the `data` argument. Graphics in **ggplot2** are built incrementally by elements. In this case, the only glyphs added are points, which are plotted using the `geom_point()` function. The arguments to `geom_point()` specify *where* and *how* the points are drawn. Here, the two *aesthetics* (`aes()`) map the vertical (`y`) coordinate to the `gdp` variable, and the horizontal (`x`) coordinate to the `educ` variable. The `size` argument to `geom_point()` changes the size of all of the glyphs. Note that here, every dot is the same size. Thus, `size` is *not* an aesthetic, since it does not map a variable to a visual cue. Since each case (i.e., row in the data frame) is a country, each dot represents one country.

```
g <- ggplot(data = CIACountries, aes(y = gdp, x = educ))
g + geom_point(size = 3)
```

In [Figure 3.1](#) the glyphs are simple. Only position in the frame distinguishes one glyph from another. The shape, size, etc. of all of the glyphs are identical—there is nothing about the glyph itself that identifies the country.

However, it is possible to use a glyph with several attributes. We can define additional aesthetics to create new visual cues. In [Figure 3.2](#), we have extended the previous example by mapping the color of each dot to the categorical `net_users` variable.

```
g + geom_point(aes(color = net_users), size = 3)
```

Changing the glyph is as simple as changing the function that draws that glyph—the aesthetic can often be kept exactly the same. In [Figure 3.3](#), we plot text instead of a dot.

```
g + geom_text(aes(label = country, color = net_users), size = 3)
```

Of course, we can employ multiple aesthetics. There are four aesthetics in [Figure 3.4](#). Each of the four aesthetics is set in correspondence with a variable—we say the variable is *mapped* to the aesthetic. Educational attainment is being mapped to horizontal position, GDP to vertical position, Internet connectivity to color, and length of roadways to size. Thus, we

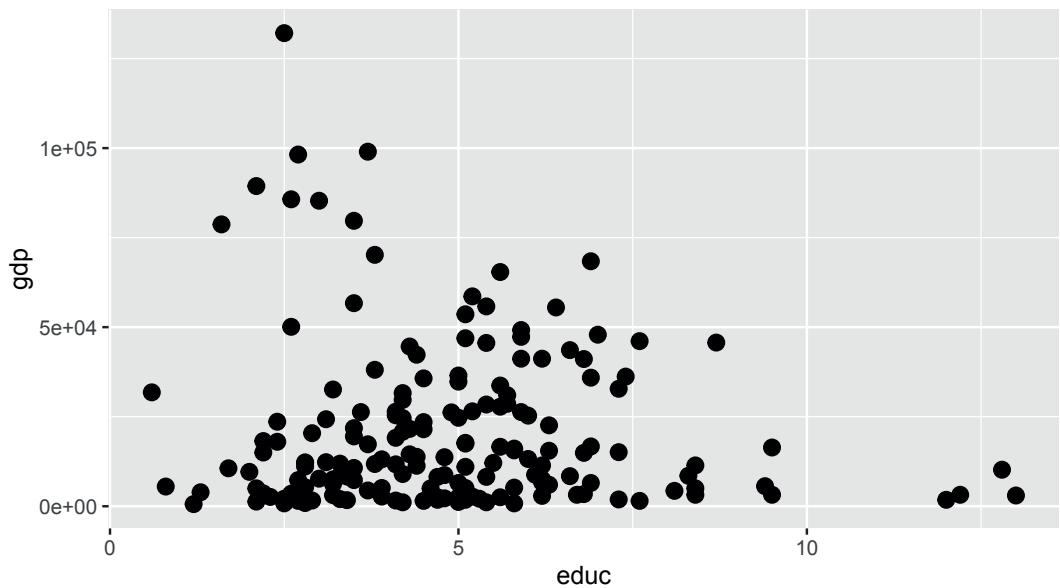


Figure 3.1: Scatterplot using only the position aesthetic for glyphs.

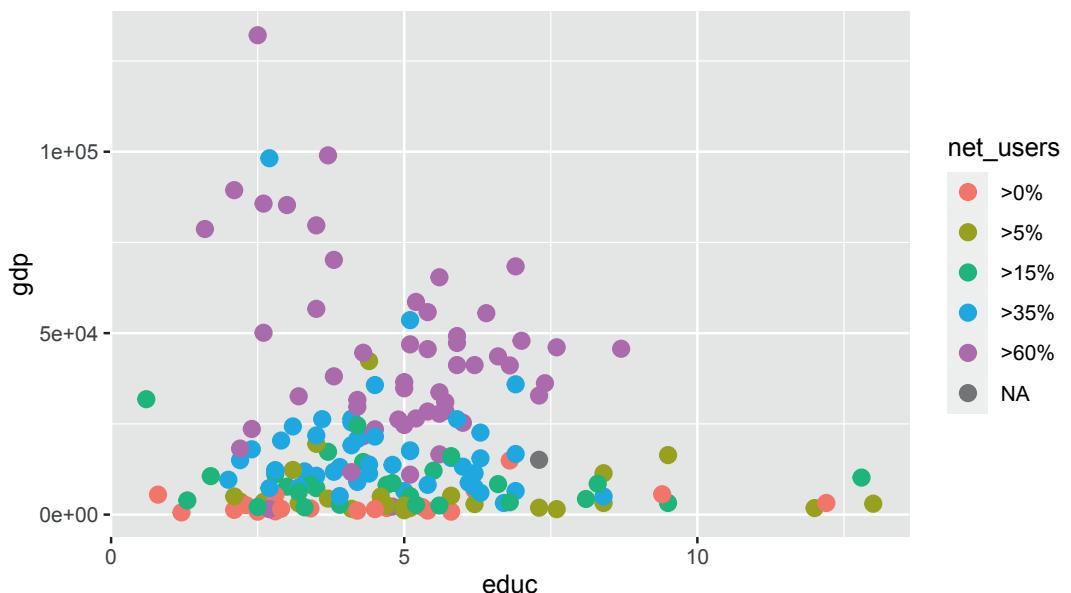


Figure 3.2: Scatterplot in which net_users is mapped to color.

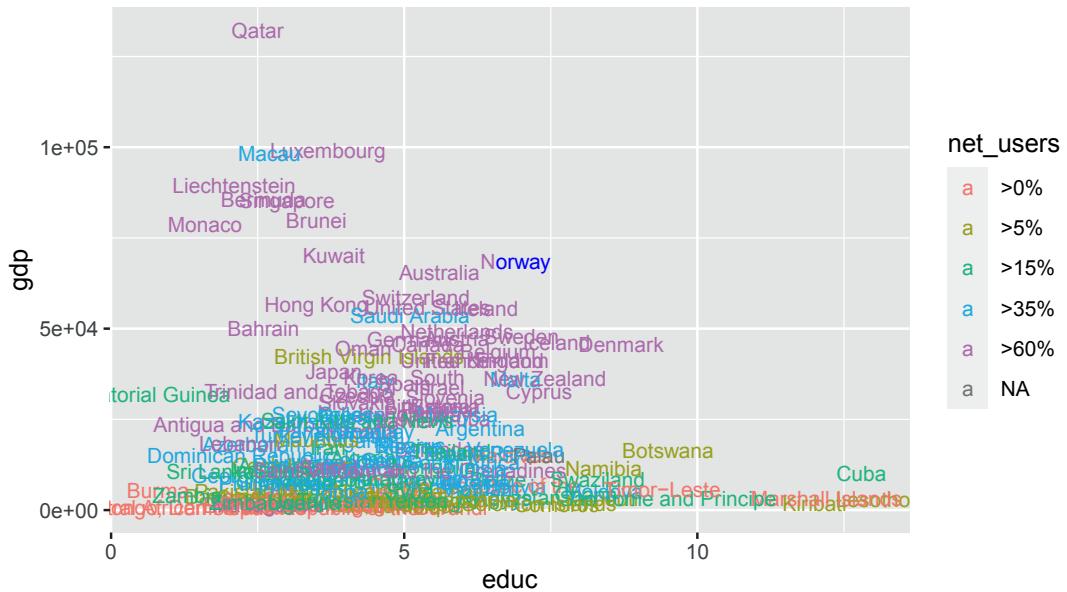


Figure 3.3: Scatterplot using both location and label as aesthetics.

encode four variables (`gdp`, `educ`, `net_users`, and `roadways`) using the visual cues of position, color, and area, respectively.

```
g + geom_point(aes(color = net_users, size = roadways))
```

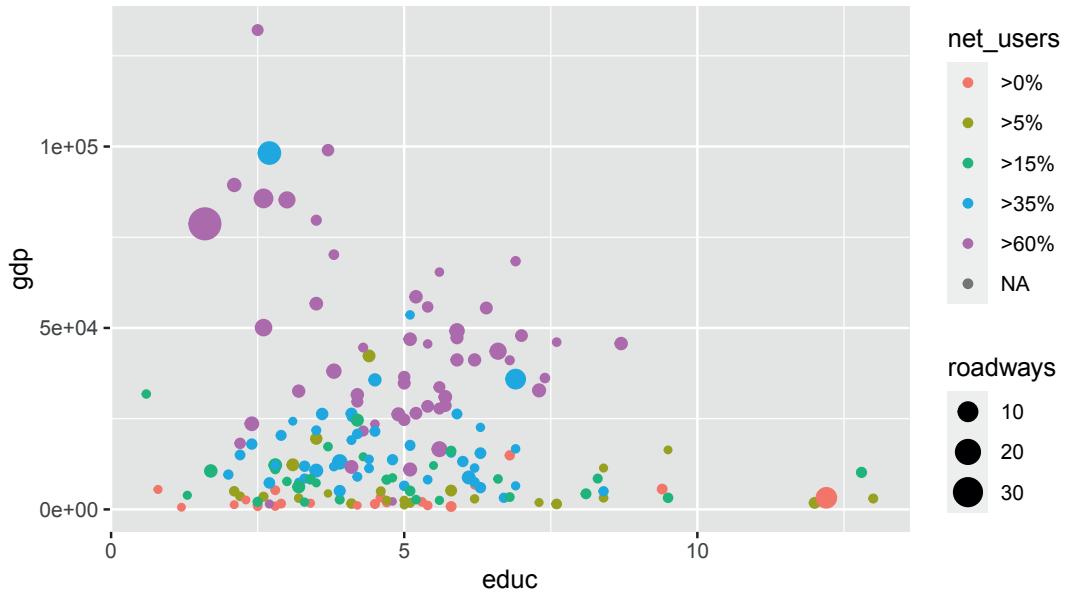


Figure 3.4: Scatterplot in which `net_users` is mapped to color and `educ` mapped to size. Compare this graphic to [Figure 3.7](#), which displays the same data using facets.

A data table provides the basis for drawing a data graphic. The relationship between a data table and a graphic is simple: Each case (row) in the data table becomes a mark in the

graph (we will return to the notion of *glyph-ready data* in [Chapter 6](#)). As the designer of the graphic, you choose which variables the graphic will display and how each variable is to be represented graphically: position, size, color, and so on.

3.1.2 Scales

Compare [Figure 3.4](#) to [Figure 3.5](#). In the former, it is hard to discern differences in GDP due to its right-skewed distribution and the choice of a *linear* scale. In the latter, the *logarithmic* scale on the vertical axis makes the scatterplot more readable. Of course, this makes interpreting the plot more complex, so we must be very careful when doing so. Note that the only difference in the code is the addition of the `coord_trans()` directive.

```
g +
  geom_point(aes(color = net_users, size = roadways)) +
  coord_trans(y = "log10")
```

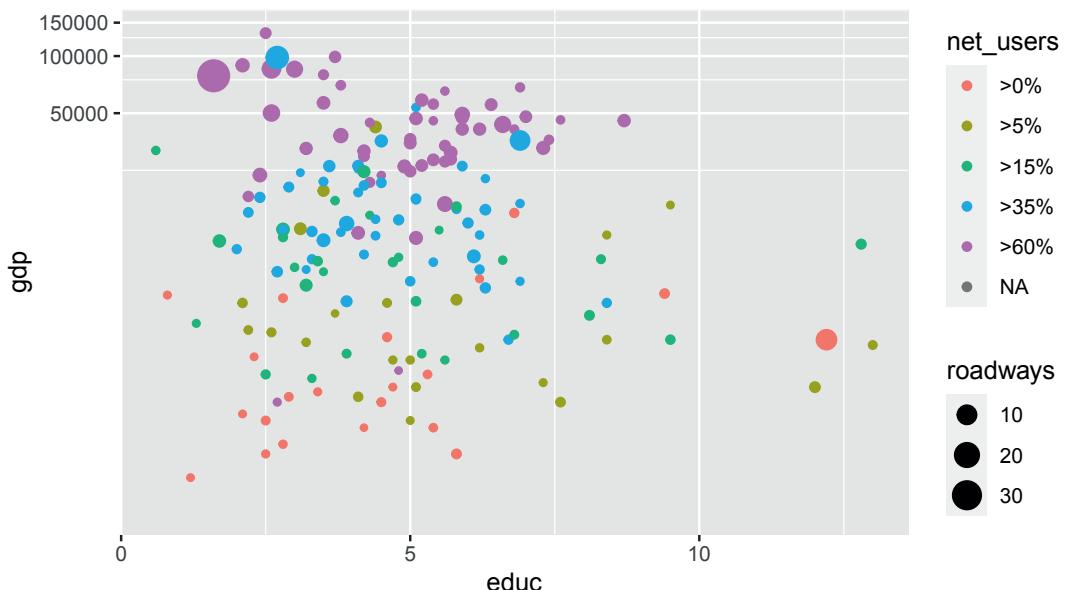


Figure 3.5: Scatterplot using a logarithmic transformation of GDP that helps to mitigate visual clustering caused by the right-skewed distribution of GDP among countries.

Scales can also be manipulated in `ggplot2` using any of the scale functions. For example, instead of using the `coord_trans()` function as we did above, we could have achieved a similar plot through the use of the `scale_y_continuous()` function, as illustrated in [Figure 3.6](#). In either case, the points will be drawn in the same location—the difference in the two plots is how and where the major tick marks and axis labels are drawn. We prefer to use `coord_trans()` in [Figure 3.5](#) because it draws attention to the use of the log scale (compare with [Figure 3.6](#)). Similarly-named functions (e.g., `scale_x_continuous()`, `scale_x_discrete()`, `scale_color()`, etc.) perform analogous operations on different aesthetics.

```
g +
  geom_point(aes(color = net_users, size = roadways)) +
  scale_y_continuous(
    name = "Gross Domestic Product",
```

```
trans = "log10",
labels = scales::comma
)
```

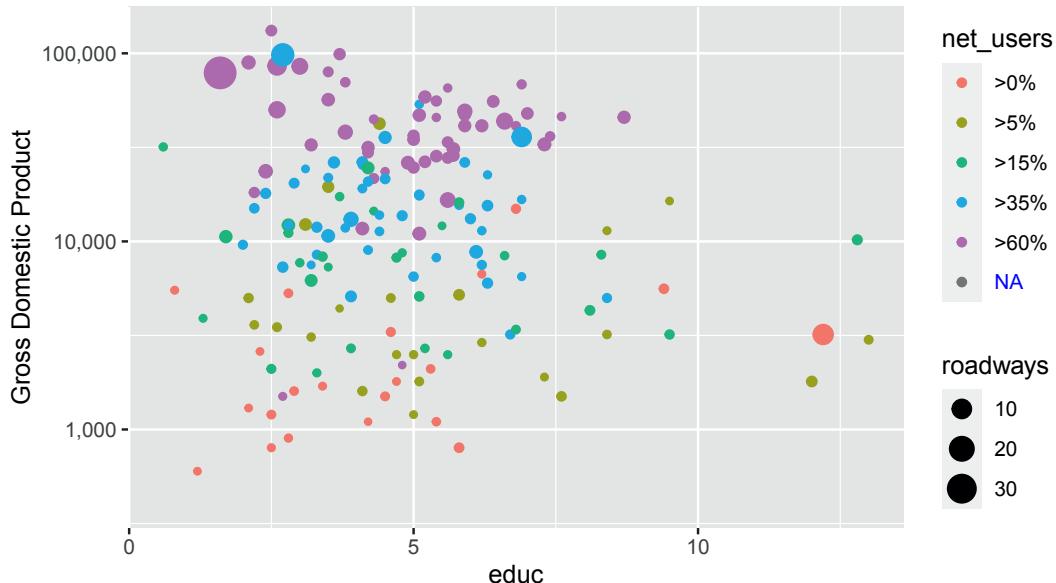


Figure 3.6: Scatterplot using a logarithmic transformation of GDP. The use of a log scale on the *y*-axis is less obvious than it is in [Figure 3.5](#) due to the uniformly-spaced horizontal grid lines.

Not all scales are about position. For instance, in [Figure 3.4](#), *net_users* is translated to color. Similarly, *roadways* is translated to size: the largest dot corresponds to a value of 30 roadways per unit area.

3.1.3 Guides

Context is provided by *guides* (more commonly called legends). A guide helps a human reader understand the meaning of the visual cues by providing context.

For position visual cues, the most common sort of guide is the familiar axis with its tick marks and labels. But other guides exist. In [Figures 3.4](#) and [3.5](#), legends relate how dot color corresponds to internet connectivity, and how dot size corresponds to length of roadways (note the use of a log scale). The `geom_text()` and `geom_label()` functions can also be used to provide specific textual annotations on the plot. Examples of how to use these functions for annotations are provided in [Section 3.3](#).

3.1.4 Facets

Using multiple aesthetics such as shape, color, and size to display multiple variables can produce a confusing, hard-to-read graph. *Facets*—multiple side-by-side graphs used to display levels of a categorical variable—provide a simple and effective alternative. [Figure 3.7](#) uses facets to show different levels of Internet connectivity, providing a better view than [Figure 3.4](#). There are two functions that create facets: `facet_wrap()` and `facet_grid()`. The former

creates a facet for each level of a single categorical variable, whereas the latter creates a facet for each combination of two categorical variables, arranging them in a grid.

```
g +
  geom_point(alpha = 0.9, aes(size = roadways)) +
  coord_trans(y = "log10") +
  facet_wrap(~net_users, nrow = 1) +
  theme(legend.position = "top")
```

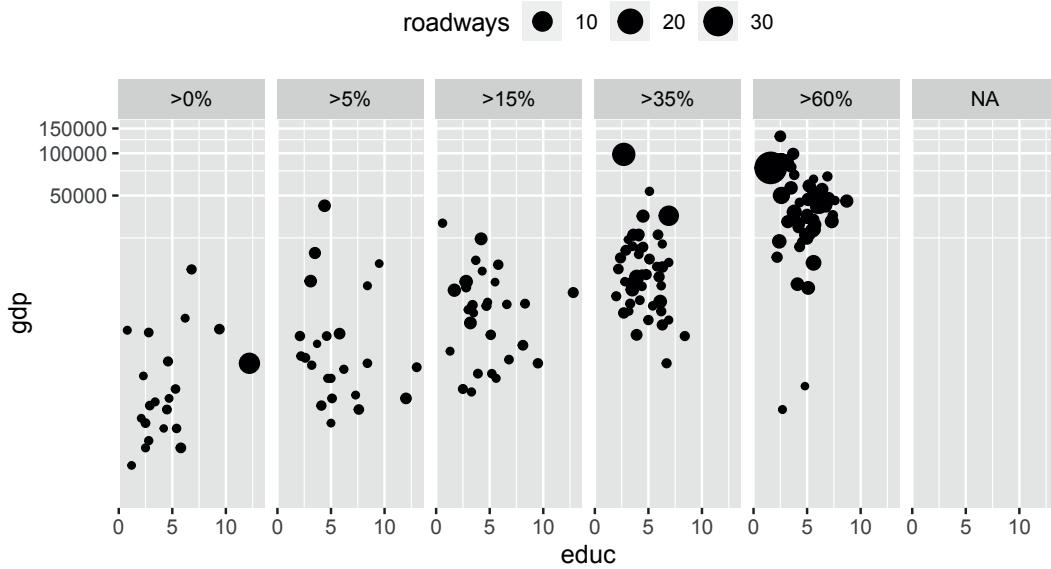


Figure 3.7: Scatterplot using facets for different ranges of Internet connectivity.

3.1.5 Layers

On occasion, data from more than one data table are graphed together. For example, the `MedicareCharges` and `MedicareProviders` data tables provide information about the average cost of each medical procedure in each state. If you live in *New Jersey*, you might wonder how providers in your state charge for different medical procedures. However, you will certainly want to understand those averages in the context of the averages across all states. In the `MedicareCharges` table, each row represents a different medical procedure (`drg`) with its associated average cost in each state. We also create a second data table called `ChargesNJ`, which contains only those rows corresponding to providers in the state of New Jersey. Do not worry if these commands aren't familiar—we will learn these in [Chapter 4](#).

```
ChargesNJ <- MedicareCharges %>%
  filter(stateProvider == "NJ")
```

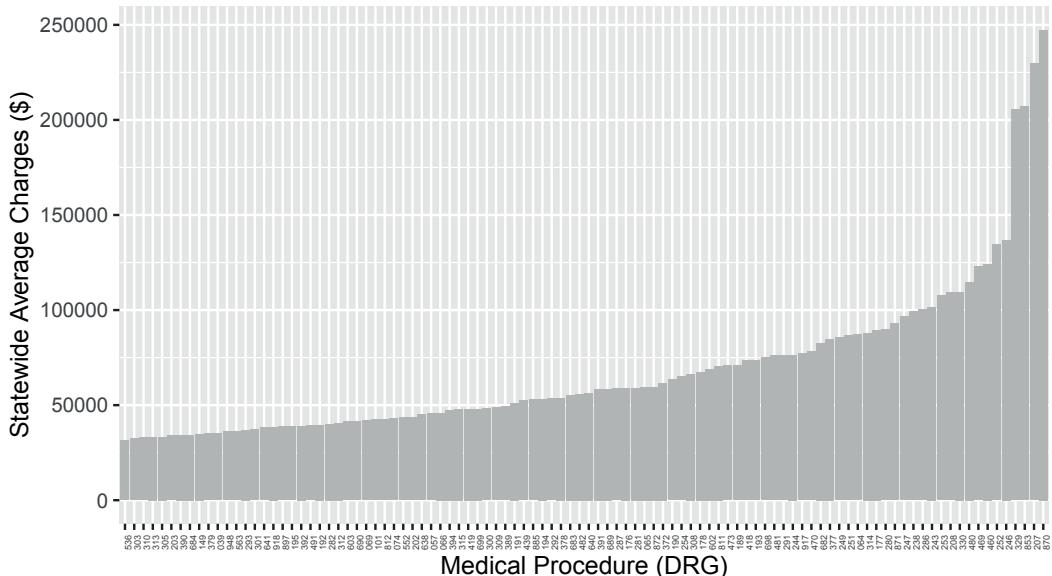
The first few rows from the data table for New Jersey are shown in [Table 3.2](#). This glyph-ready table (see [Chapter 6](#)) can be translated to a chart ([Figure 3.8](#)) using bars to represent the average charges for different medical procedures in New Jersey. The `geom_col()` function creates a separate bar for each of the 100 different medical procedures.

```
p <- ggplot(
  data = ChargesNJ,
  aes(x = reorder(drg, mean_charge), y = mean_charge)
```

Table 3.2: Glyph-ready data for the barplot layer.

drg	stateProvider	num_charges	mean_charge
039	NJ	31	35104
057	NJ	55	45692
064	NJ	55	87042
065	NJ	59	59576
066	NJ	56	45819
069	NJ	61	41917
074	NJ	41	42993
101	NJ	58	42314
149	NJ	50	34916
176	NJ	36	58941

```
) +
  geom_col(fill = "gray") +
  ylab("Statewide Average Charges ($)") +
  xlab("Medical Procedure (DRG)") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1, size = rel(0.5)))
p
```

**Figure 3.8:** Bar graph of average charges for medical procedures in New Jersey.

How do the charges in New Jersey compare to those in other states? The two data tables, one for New Jersey and one for the whole country, can be plotted with different glyph types: bars for New Jersey and dots for the states across the whole country as in [Figure 3.9](#).

```
p + geom_point(data = MedicareCharges, size = 1, alpha = 0.3)
```

With the context provided by the individual states, it is easy to see that the charges in New Jersey are among the highest in the country for each medical procedure.

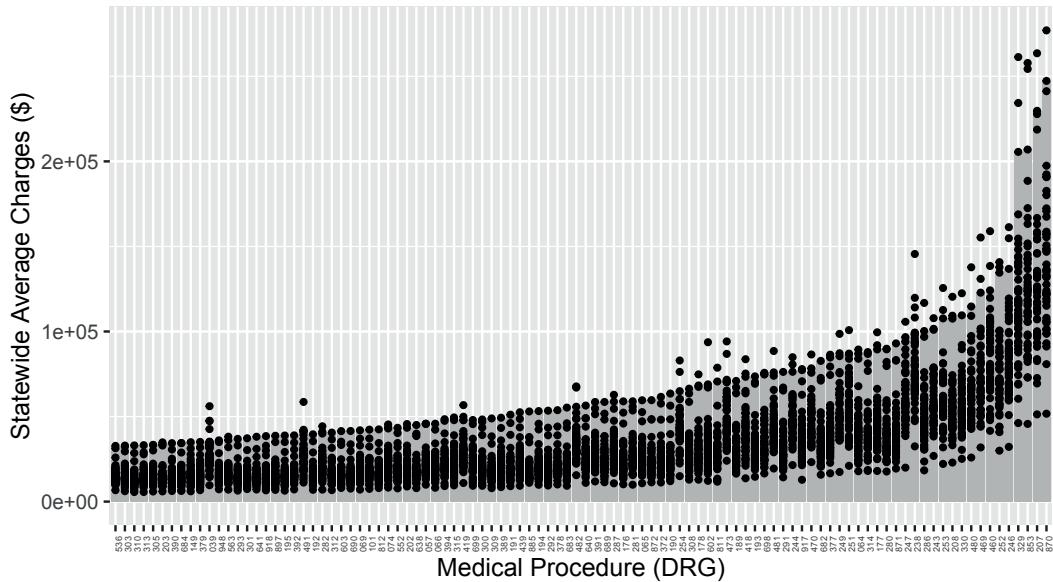


Figure 3.9: Bar graph adding a second layer to provide a comparison of New Jersey to other states. Each dot represents one state, while the bars represent New Jersey.

3.2 Canonical data graphics in R

Over time, statisticians have developed standard data graphics for specific use cases (Tukey, 1990). While these data graphics are not always mesmerizing, they are hard to beat for simple effectiveness. Every data scientist should know how to make and interpret these canonical data graphics—they are ignored at your peril.

3.2.1 Univariate displays

It is generally useful to understand how a single variable is distributed. If that variable is numeric, then its distribution is commonly summarized graphically using a *histogram* or *density plot*. Using the **ggplot2** package, we can display either plot for the `math` variable in the `SAT_2010` data frame by binding the `math` variable to the `x` aesthetic.

```
g <- ggplot(data = SAT_2010, aes(x = math))
```

Then we only need to choose either `geom_histogram()` or `geom_density()`. Both [Figures 3.10](#) and [3.11](#) convey the same information, but whereas the histogram uses pre-defined bins to create a discrete distribution, a density plot uses a *kernel smoother* to make a continuous curve.

```
g + geom_histogram(binwidth = 10) + labs(x = "Average math SAT score")
```

Note that the `binwidth` argument is being used to specify the width of bins in the histogram. Here, each bin contains a 10-point range of SAT scores. In general, the appearance of a histogram can vary considerably based on the choice of bins, and there is no one “best” choice (Lunzer and McNamara, 2017). You will have to decide what bin width is most appropriate for your data.

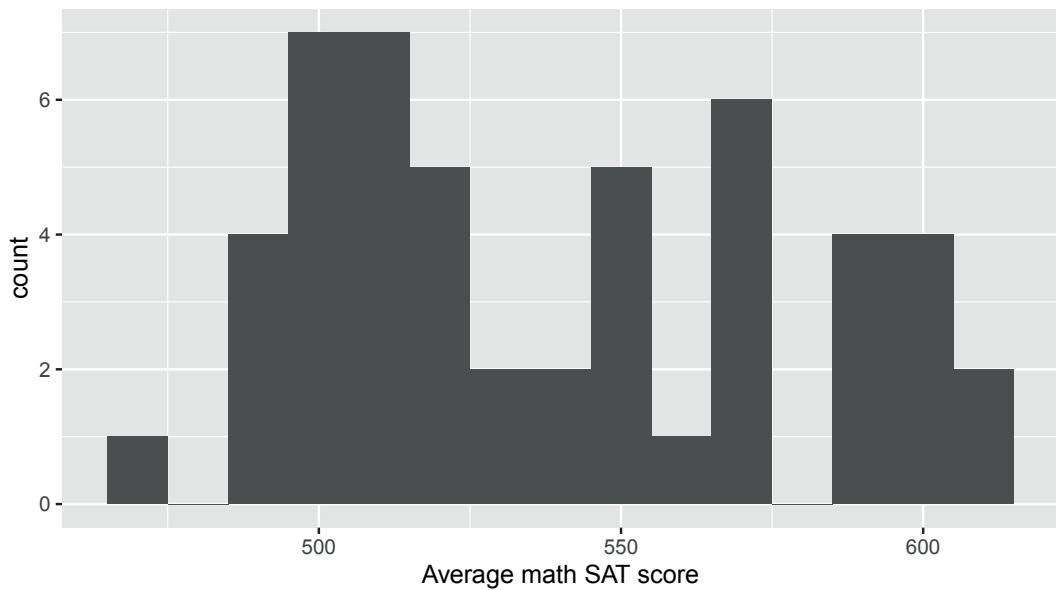


Figure 3.10: Histogram showing the distribution of math SAT scores by state.

```
g + geom_density(adjust = 0.3)
```

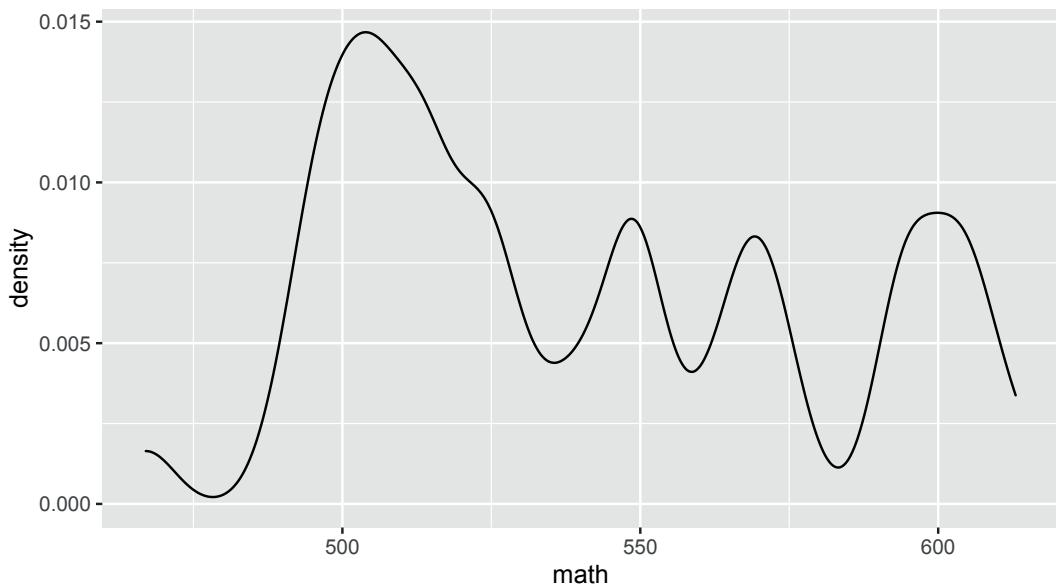


Figure 3.11: Density plot showing the distribution of average math SAT scores by state.

Similarly, in the density plot shown in [Figure 3.11](#) we use the `adjust` argument to modify the *bandwidth* being used by the kernel smoother. In the taxonomy defined above, a density plot uses position and direction in a Cartesian plane with a horizontal scale defined by the units in the data.

If your variable is categorical, it doesn't make sense to think about the values as having a

continuous density. Instead, we can use a *bar graph* to display the distribution of a categorical variable.

To make a simple bar graph for `math`, identifying each bar by the label `state`, we use the `geom_col()` command, as displayed in [Figure 3.12](#). Note that we add a few wrinkles to this plot. First, we use the `head()` function to display only the first 10 states (in alphabetical order). Second, we use the `reorder()` function to sort the state names in order of their average `math` SAT score.

```
ggplot(
  data = head(SAT_2010, 10),
  aes(x = reorder(state, math), y = math)
) +
  geom_col() +
  labs(x = "State", y = "Average math SAT score")
```

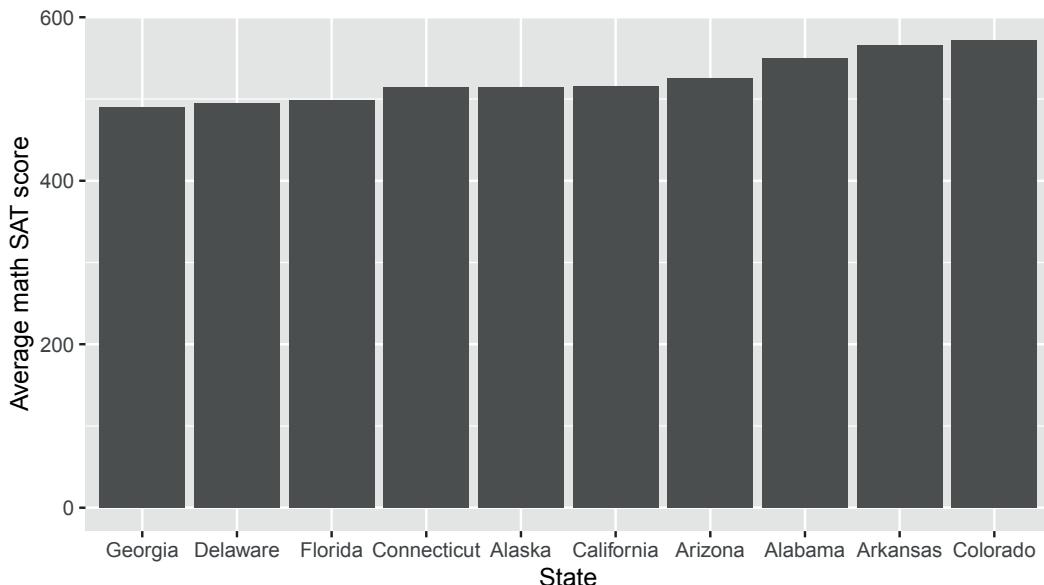


Figure 3.12: A bar plot showing the distribution of average math SAT scores for a selection of states.

As noted earlier, we recommend against the use of pie charts to display the distribution of a categorical variable since, in most cases, a table of frequencies is more informative. An informative graphical display can be achieved using a *stacked bar plot*, such as the one shown in [Figure 3.13](#) using the `geom_bar()` function. Note that we have used the `coord_flip()` function to display the bars horizontally instead of vertically.

```
ggplot(data = mosaicData::HELPrc, aes(x = homeless)) +
  geom_bar(aes(fill = substance), position = "fill") +
  scale_fill_brewer(palette = "Spectral") +
  coord_flip()
```

This method of graphical display enables a more direct comparison of proportions than would be possible using two pie charts. In this case, it is clear that homeless participants were more likely to identify as being involved with alcohol as their primary substance of abuse. However, like pie charts, bar charts are sometimes criticized for having a low *data-*

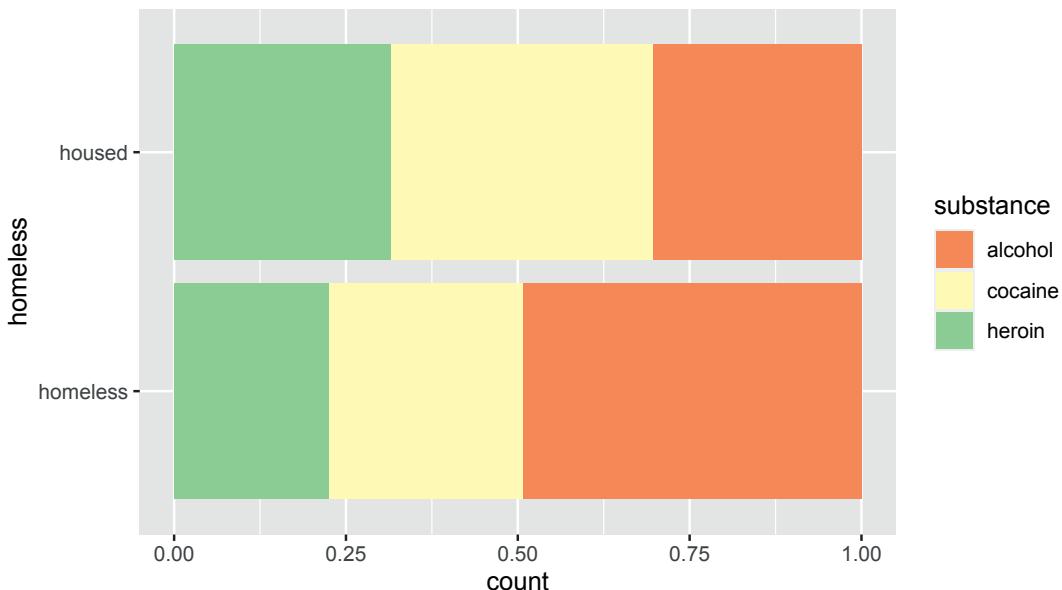


Figure 3.13: A stacked bar plot showing the distribution of substance of abuse for participants in the HELP study. Compare this to [Figure 2.14](#).

to-ink ratio. That is, they use a comparatively large amount of ink to depict relatively few data points.

3.2.2 Multivariate displays

Multivariate displays are the most effective way to convey the relationship between more than one variable. The venerable *scatterplot* remains an excellent way to display observations of two quantitative (or numerical) variables. The scatterplot is provided in **ggplot2** by the `geom_point()` command. The main purpose of a scatterplot is to show the relationship between two variables across many cases. Most often, there is a Cartesian coordinate system in which the *x*-axis represents one variable and the *y*-axis the value of a second variable.

```
g <- ggplot(
  data = SAT_2010,
  aes(x = expenditure, y = math)
) +
  geom_point()
```

We will also add a smooth trend line and some more specific axis labels. We use the `geom_smooth()` function in order to plot the simple linear regression line (`method = "lm"`) through the points (see [Section 9.6](#) and [Appendix E](#)).

```
g <- g +
  geom_smooth(method = "lm", se = FALSE) +
  xlab("Average expenditure per student ($1000)") +
  ylab("Average score on math SAT")
```

In [Figures 3.14](#) and [3.15](#), we plot the relationship between the average SAT math score and the expenditure per pupil (in thousands of United States dollars) among states in 2010. A third (categorical) variable can be added through *faceting* and/or *layering*. In this case, we

use the `mutate()` function (see [Chapter 4](#)) to create a new variable called `SAT_rate` that places states into bins (e.g., high, medium, low) based on the percentage of students taking the SAT. Additionally, in order to include that new variable in our plots, we use the `%+%` operator to update the data frame that is bound to our plot.

```
SAT_2010 <- SAT_2010 %>%
  mutate(
    SAT_rate = cut(
      sat_pct,
      breaks = c(0, 30, 60, 100),
      labels = c("low", "medium", "high")
    )
  )
g <- g %+% SAT_2010
```

In [Figure 3.14](#), we use the `color` aesthetic to separate the data by `SAT_rate` on a single plot (i.e., layering). Compare this with [Figure 3.15](#), where we add a `facet_wrap()` mapped to `SAT_rate` to separate by facet.

```
g + aes(color = SAT_rate)
```

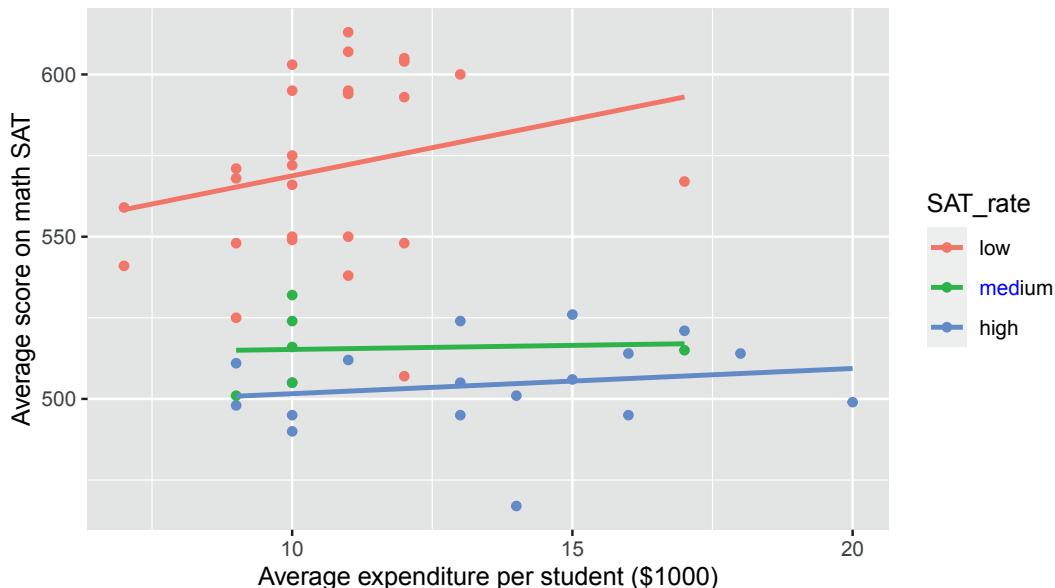


Figure 3.14: Scatterplot using the `color` aesthetic to separate the relationship between two numeric variables by a third categorical variable.

```
g + facet_wrap(~ SAT_rate)
```

The `NHANES` data table provides medical, behavioral, and morphometric measurements of individuals. The scatterplot in [Figure 3.16](#) shows the relationship between two of the variables, height and age. Each dot represents one person and the position of that dot signifies the value of the two variables for that person. Scatterplots are useful for visualizing a simple relationship between two variables. For instance, you can see in [Figure 3.16](#) the familiar pattern of growth in height from birth to the late teens.

It's helpful to do a bit more wrangling (more on this later) to ensure that the spatial

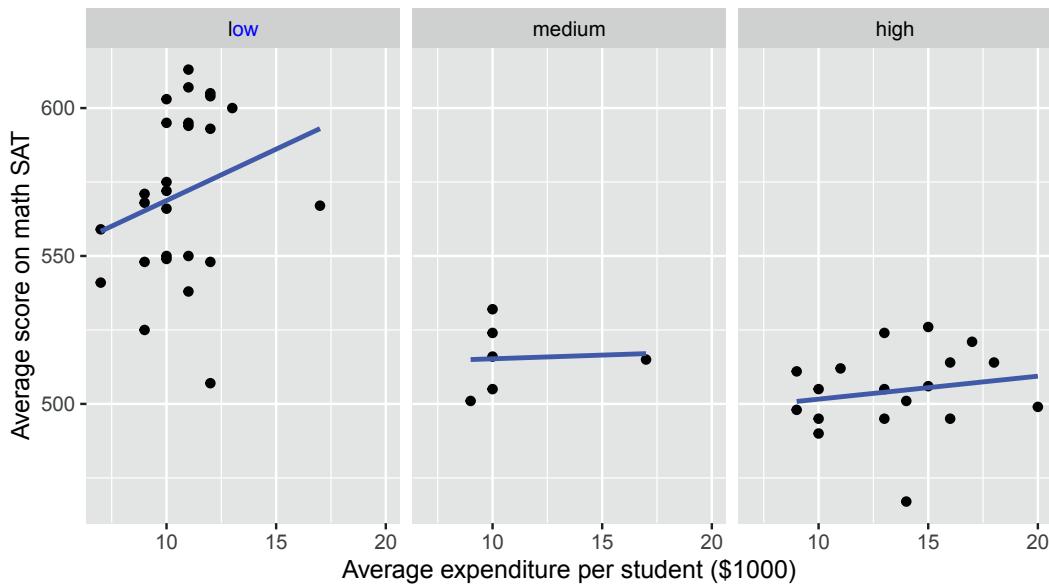


Figure 3.15: Scatterplot using a `facet_wrap()` to separate the relationship between two numeric variables by a third categorical variable.

relationship of the lines (adult men tend to be taller than adult women) matches the ordering of the legend labels. Here we use the `fct_relevel()` function (from the `forcats` package) to reset the factor levels.

```
library(NHANES)
ggplot(
  data = slice_sample(NHANES, n = 1000),
  aes(x = Age, y = Height, color = fct_relevel(Gender, "male"))
) +
  geom_point() +
  geom_smooth() +
  xlab("Age (years)") +
  ylab("Height (cm)") +
  labs(color = "Gender")
```

Some scatterplots have special meanings. A *time series*—such as the one shown in Figure 3.17—is just a scatterplot with time on the horizontal axis and points connected by lines to indicate temporal continuity. In Figure 3.17, the temperature at a weather station in western Massachusetts is plotted over the course of the year. The familiar fluctuations based on the seasons are evident. Be especially aware of dubious causality in these plots: Is time really a good explanatory variable?

```
library(macleish)
ggplot(data = whately_2015, aes(x = when, y = temperature)) +
  geom_line(color = "darkgray") +
  geom_smooth() +
  xlab(NULL) +
  ylab("Temperature (degrees Celsius)")
```

For displaying a numerical response variable against a categorical explanatory variable, a

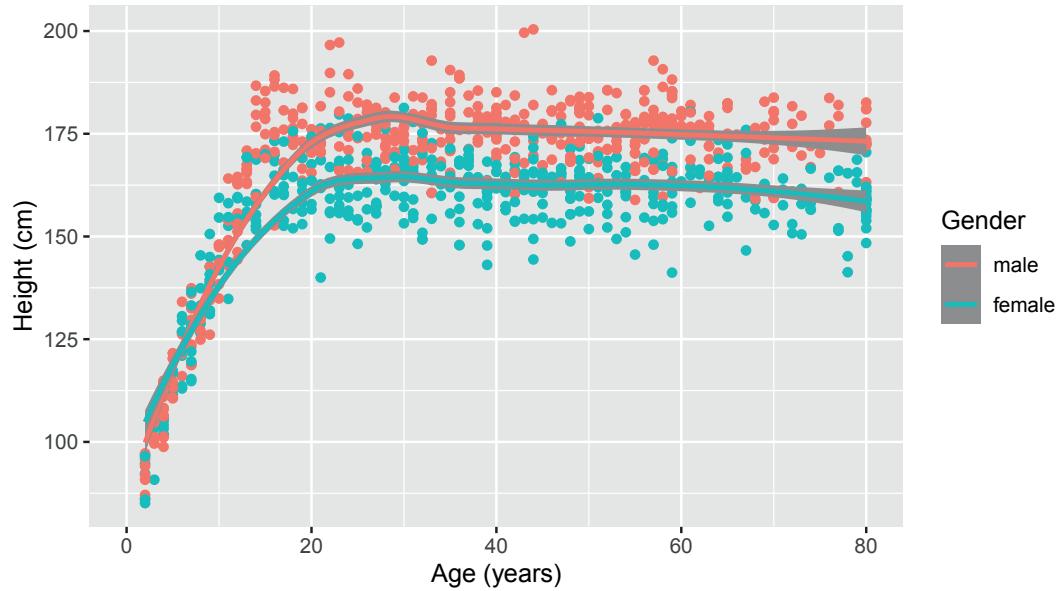


Figure 3.16: A scatterplot for 1,000 random individuals from the **NHANES** study. Note how mapping gender to color illuminates the differences in height between men and women.

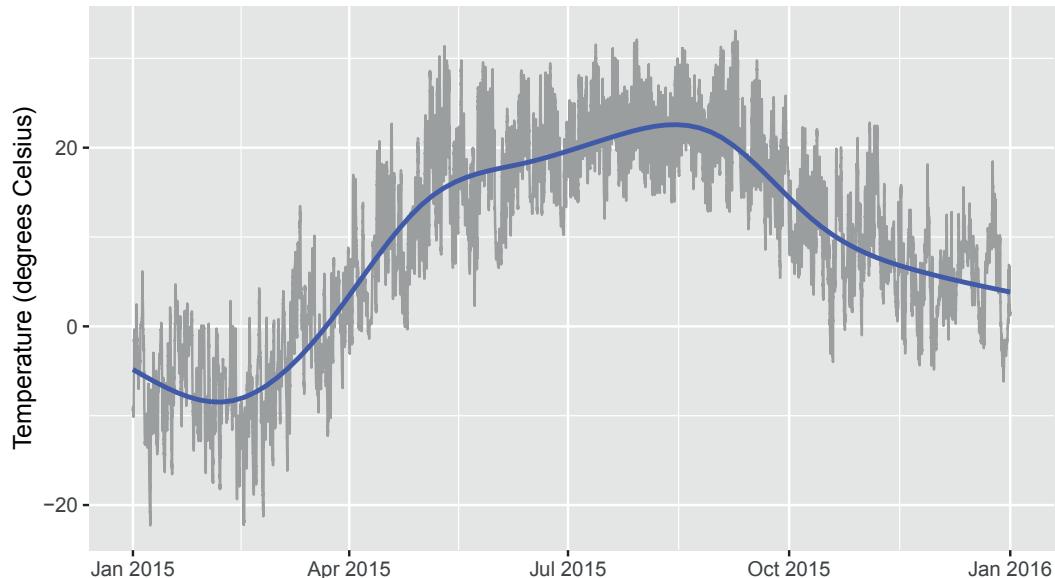


Figure 3.17: A time series showing the change in temperature at the MacLeish field station in 2015.

common choice is a *box-and-whisker* (or box) plot, as shown in [Figure 3.18](#). (More details about the data wrangling needed to create the categorical `month` variable will be provided in later chapters.) It may be easiest to think about this as simply a graphical depiction of the *five-number summary* (minimum [0th percentile], Q1 [25th percentile], median [50th percentile], Q3 [75th percentile], and maximum [100th percentile]).

```
whately_2015 %>%
  mutate(month = as.factor(lubridate::month(when, label = TRUE))) %>%
  group_by(month) %>%
  skim(temperature) %>%
  select(-na)

-- Variable type: numeric -----
#> #>   var      month     n   mean    sd    p0    p25    p50    p75   p100
#> #>   <chr>    <ord> <int>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
#> 1 temperature Jan    4464 -6.37   5.14 -22.3  -10.3  -6.25  -2.35   6.16
#> 2 temperature Feb    4032 -9.26   5.11 -22.2  -12.3  -9.43  -5.50   4.27
#> 3 temperature Mar    4464 -0.873  5.06 -16.2  -4.61 -0.550  2.99  13.5
#> 4 temperature Apr    4320  8.04   5.51 -3.04   3.77  7.61  11.8   22.7
#> 5 temperature May    4464 17.4    5.94  2.29  12.8   17.5  21.4   31.4
#> 6 temperature Jun    4320 17.7    5.11  6.53  14.2   18.0  21.2   29.4
#> 7 temperature Jul    4464 21.6    3.90  12.0   18.6  21.2   24.3   32.1
#> 8 temperature Aug    4464 21.4    3.79  12.9   18.4  21.1   24.3   31.2
#> 9 temperature Sep    4320 19.3    5.07  5.43  15.8   19     22.5   33.1
#> 10 temperature Oct   4464  9.79   5.00 -3.97   6.58  9.49  13.3   22.3
#> 11 temperature Nov   4320  7.28   5.65 -4.84   3.14  7.11  10.8   22.8
#> 12 temperature Dec   4464  4.95   4.59 -6.16   1.61  5.15  8.38  18.4

ggplot(
  data = whately_2015,
  aes(
    x = lubridate::month(when, label = TRUE),
    y = temperature
  )
) +
  geom_boxplot() +
  xlab("Month") +
  ylab("Temperature (degrees Celsius)")
```

When both the explanatory and response variables are categorical (or binned), points and lines don't work as well. How likely is a person to have *diabetes*, based on their age and *BMI* (body mass index)? In the *mosaic plot* (or eikosogram) shown in [Figure 3.19](#) the number of observations in each cell is proportional to the area of the box. Thus, you can see that *diabetes* tends to be more common for older people as well as for those who are obese, since the blue-shaded regions are larger than expected under an independence model while the pink are less than expected. These provide a more accurate depiction of the intuitive notions of probability familiar from *Venn diagrams* (Olford and Cherry, 2003).

In [Table 3.3](#) we summarize the use of `ggplot2` plotting commands and their relationship to canonical data graphics. Note that the `geom_mosaic()` function is not part of `ggplot2` but rather is available through the `ggmosaic` package.

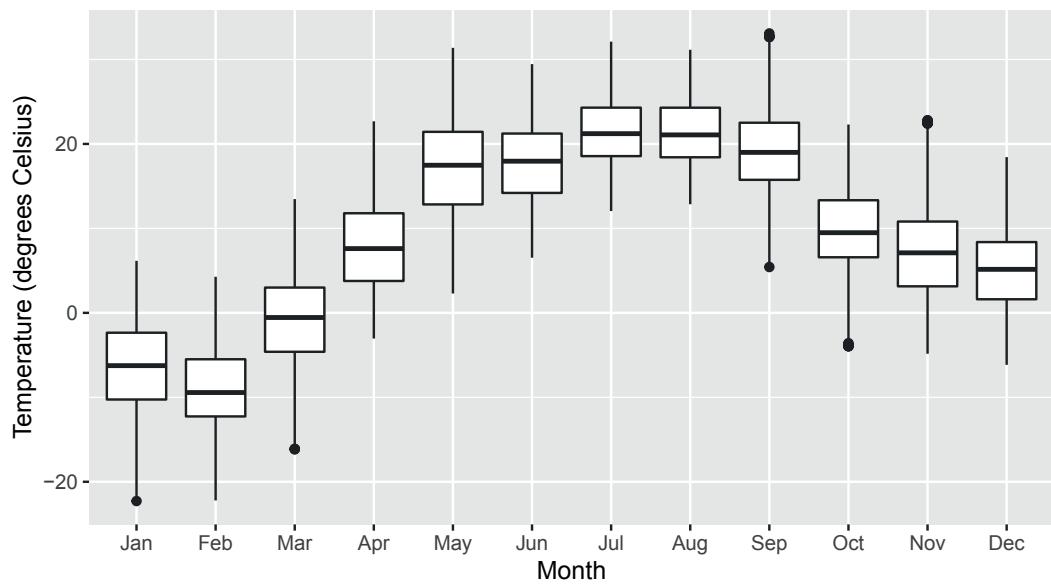


Figure 3.18: A box-and-whisker of temperatures by month at the MacLeish field station.

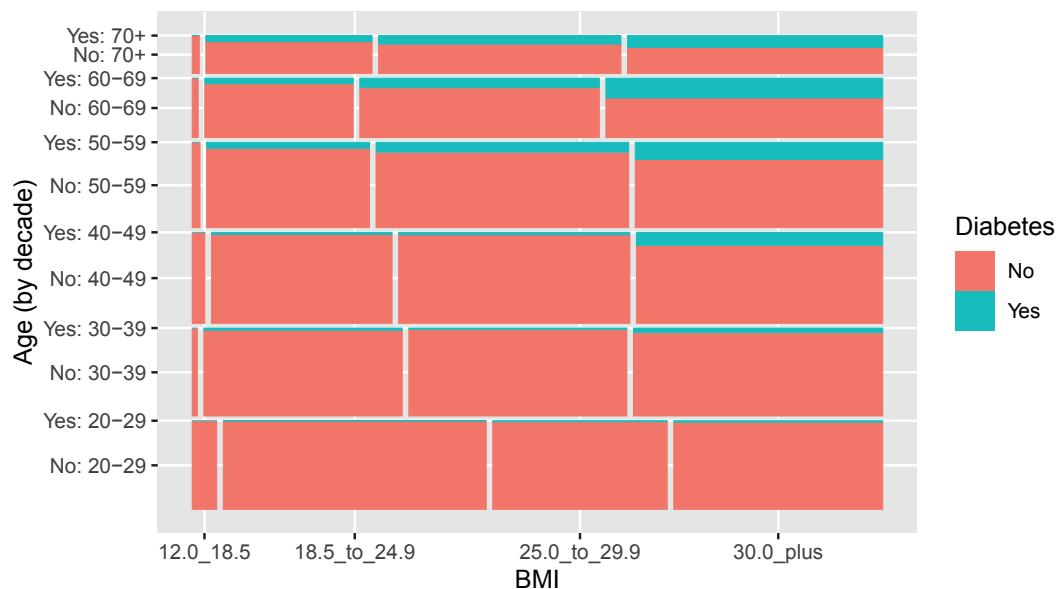


Figure 3.19: Mosaic plot (eikosogram) of diabetes by age and weight status (BMI).

Table 3.3: Table of canonical data graphics and their corresponding **ggplot2** commands. Note that the mosaic plot function is not part of the **ggplot2** package.

response (y)	explanatory (x)	plot type	geom_*(())
numeric	numeric	histogram, density	geom_histogram(), geom_density()
	categorical	stacked bar	geom_bar()
	numeric	scatter	geom_point()
	categorical	box	geom_boxplot()
categorical	categorical	mosaic	geom_mosaic()

3.2.3 Maps

Using a map to display data geographically helps both to identify particular cases and to show spatial patterns and discrepancies. In [Figure 3.20](#), the shading of each country represents its oil production. This sort of map, where the fill color of each region reflects the value of a variable, is sometimes called a *choropleth map*. We will learn more about mapping and how to work with spatial data in [Chapter 17](#).

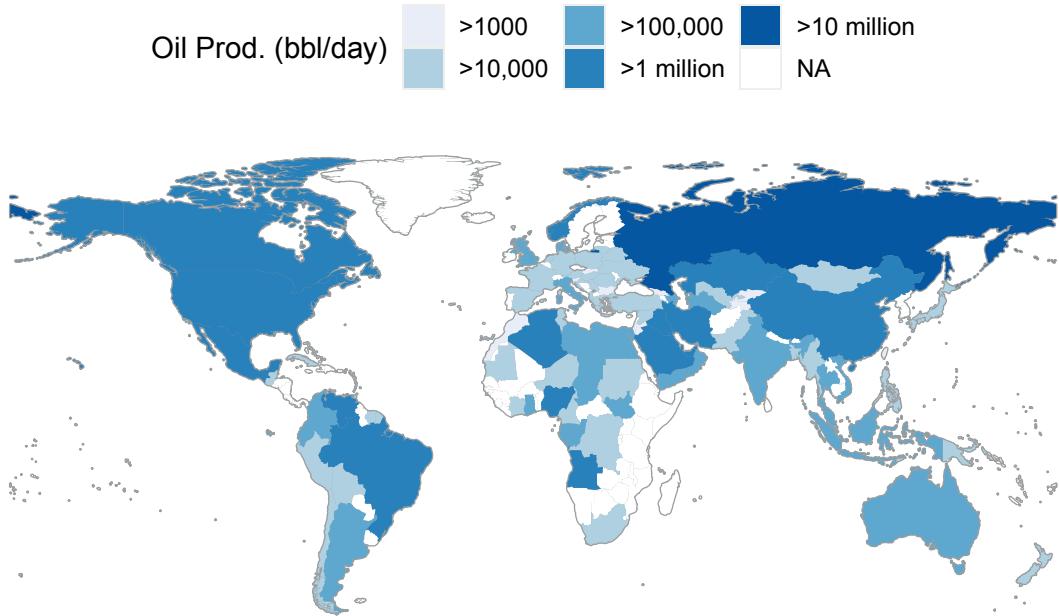


Figure 3.20: A choropleth map displaying oil production by countries around the world in barrels per day.

3.2.4 Networks

A *network* is a set of connections, called *edges*, between nodes, called *vertices*. A vertex represents an entity. The edges indicate pairwise relationships between those entities.

The `NCI60` data set is about the genetics of cancer. The data set contains more than 40,000 probes for the expression of genes, in each of 60 cancers. In the network displayed in [Figure 3.21](#), a vertex is a given cell line, and each is depicted as a dot. The dot's color and label gives

the type of cancer involved. These are ovarian, colon, central nervous system, melanoma, renal, breast, and lung cancers. The edges between vertices show pairs of cell lines that had a strong correlation in gene expression.

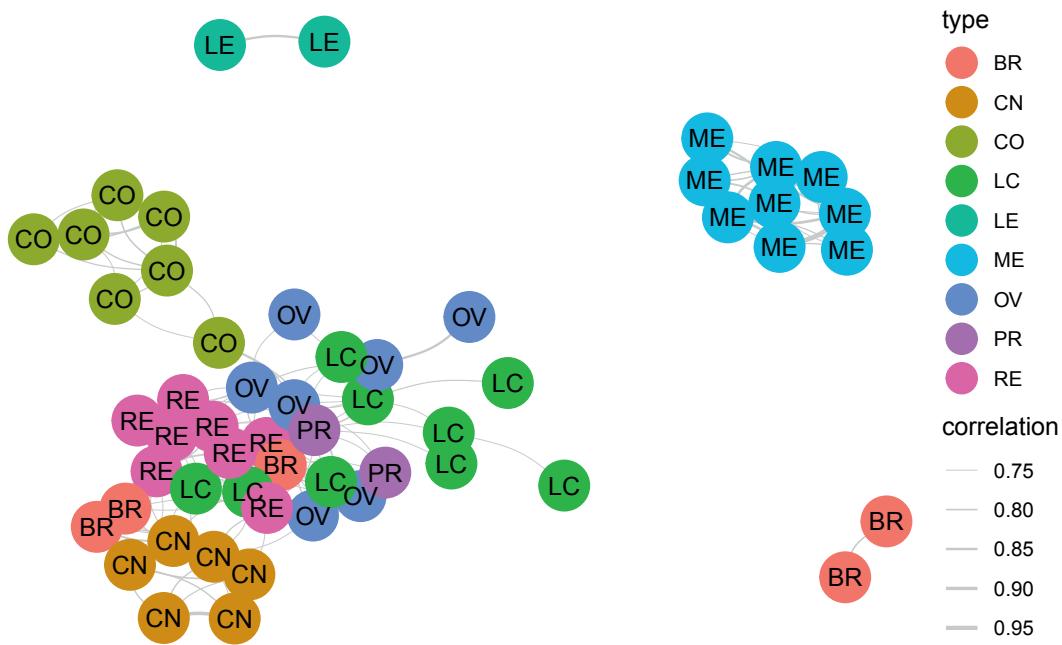


Figure 3.21: A network diagram displaying the relationship between types of cancer cell lines.

The network shows that the melanoma cell lines (ME) are closely related to each other but not so much to other cell lines. The same is true for colon cancer cell lines (CO) and for central nervous system (CN) cell lines. Lung cancers, on the other hand, tend to have associations with multiple other types of cancers. We will explore the topic of *network science* in greater depth in [Chapter 20](#).

3.3 Extended example: Historical baby names

For many of us, there are few things that are more personal than our name. It is impossible to remember a time when you didn't have your name, and you carry it with you wherever you go. You instinctively react when you hear it. And yet, you didn't choose your name—your parents did (unless you've changed your name).

How do parents go about choosing names? Clearly, there seem to be both short- and long-term trends in baby names. The popularity of the name “Bella” spiked after the lead character in *Twilight* became a cultural phenomenon. Other once-popular names seem to have fallen out of favor—writers at *FiveThirtyEight* asked, “where have all the Elmer's gone?”

Using data from the **babynames** package, which uses public data from the *Social Security Administration* (SSA), we can recreate many of the plots presented in the *FiveThirtyEight*

article, and in the process learn how to use **ggplot2** to make production-quality data graphics.

In the link in the footnote¹, FiveThirtyEight presents an informative, annotated data graphic that shows the relative ages of American males named “Joseph.” Drawing on what you have learned in Chapter 2, take a minute to jot down the visual cues, coordinate system, scales, and context present in this plot. This analysis will facilitate our use of **ggplot2** to re-construct it. (Look ahead to Figure 3.22 to see our recreation.)

The key insight of the FiveThirtyEight work is the estimation of the number of people with each name who are currently alive. The `lifetables` table from the **babynames** package contains actuarial estimates of the number of people per 100,000 who are alive at age x , for every $0 \leq x \leq 114$. The `make_babynames_dist()` function in the **mdsr** package adds some more convenient variables and filters for only the data that is relevant to people alive in 2014.²

```
library(babynames)
BabynamesDist <- make_babynames_dist()
BabynamesDist

# A tibble: 1,639,722 x 9
  year sex   name     n    prop alive_prob count_thousands age_today
  <dbl> <chr> <chr> <int>   <dbl>        <dbl>          <dbl>      <dbl>
1 1900 F   Mary   16706 0.0526       0         16.7        114
2 1900 F   Helen  6343  0.0200       0         6.34        114
3 1900 F   Anna   6114  0.0192       0         6.11        114
4 1900 F   Marg~  5304  0.0167       0         5.30        114
5 1900 F   Ruth   4765  0.0150       0         4.76        114
6 1900 F   Eliz~  4096  0.0129       0         4.10        114
7 1900 F   Flor~  3920  0.0123       0         3.92        114
8 1900 F   Ethel  3896  0.0123       0         3.90        114
9 1900 F   Marie  3856  0.0121       0         3.86        114
10 1900 F   Lill~  3414  0.0107      0         3.41        114
# ... with 1,639,712 more rows, and 1 more variable: est_alive_today <dbl>
```

To find information about a specific name, we use the `filter()` function.

```
BabynamesDist %>%
  filter(name == "Benjamin")
```

3.3.1 Percentage of people alive today

How did you break down Figure 3.22? There are two main data elements in that plot: a thick black line indicating the number of Josephs born each year, and the thin light blue bars indicating the number of Josephs born in each year that are expected to still be alive today. In both cases, the vertical axis corresponds to the number of people (in thousands), and the horizontal axis corresponds to the year of birth.

We can compose a similar plot in **ggplot2**. First we take the relevant subset of the data and set up the initial **ggplot2** object. The data frame `joseph` is bound to the plot, since this contains all of the data that we need for this plot, but we will be using it with multiple

¹<https://fivethirtyeight.com/wp-content/uploads/2014/05/silver-feature-joseph2.png>

²See the SSA documentation for more information.

geoms. Moreover, the `year` variable is mapped to the `x`-axis as an aesthetic. This will ensure that everything will line up properly.

```
joseph <- BabynamesDist %>%
  filter(name == "Joseph" & sex == "M")
name_plot <- ggplot(data = joseph, aes(x = year))
```

Next, we will add the bars.

```
name_plot <- name_plot +
  geom_col(
    aes(y = count_thousands * alive_prob),
    fill = "#b2d7e9",
    color = "white",
    size = 0.1
  )
```

The `geom_col()` function adds bars, which are filled with a light blue color and a white border. The height of the bars is an aesthetic that is mapped to the estimated number of people alive today who were born in each year.

The black line is easily added using the `geom_line()` function.

```
name_plot <- name_plot +
  geom_line(aes(y = count_thousands), size = 2)
```

Adding an informative label for the vertical axis and removing an uninformative label for the horizontal axis will improve the readability of our plot.

```
name_plot <- name_plot +
  ylab("Number of People (thousands)") +
  xlab(NULL)
```

Inspecting the `summary()` of our plot at this point can help us keep things straight—take note of the mappings. Do they accord with what you jotted down previously?

```
summary(name_plot)

data: year, sex, name, n, prop, alive_prob, count_thousands,
age_today, est_alive_today [111x9]
mapping: x = ~year
faceting: <ggproto object: Class FacetNull, Facet, gg>
  compute_layout: function
  draw_back: function
  draw_front: function
  draw_labels: function
  draw_panels: function
  finish_data: function
  init_scales: function
  map_data: function
  params: list
  setup_data: function
  setup_params: function
  shrink: TRUE
  train_scales: function
```

```

vars: function
super: <ggproto object: Class FacetNull, Facet, gg>
-----
mapping: y = ~count_thousands * alive_prob
geom_col: width = NULL, na.rm = FALSE
stat_identity: na.rm = FALSE
position_stack

mapping: y = ~count_thousands
geom_line: na.rm = FALSE, orientation = NA
stat_identity: na.rm = FALSE
position_identity

```

The final data-driven element of the FiveThirtyEight graphic is a darker blue bar indicating the median year of birth. We can compute this with the `wtd.quantile()` function in the **Hmisc** package. Setting the `probs` argument to 0.5 will give us the median year of birth, weighted by the number of people estimated to be alive today (`est_alive_today`). The `pull()` function simply extracts the `year` variable from the data frame returned by `summarize()`.

```

wtd_quantile <- Hmisc::wtd.quantile
median_yob <- joseph %>%
  summarize(
    year = wtd_quantile(year, est_alive_today, probs = 0.5)
  ) %>%
  pull(year)
median_yob

```

50%
1975

We can then overplot a single bar in a darker shade of blue. Here, we are using the `ifelse()` function cleverly. If the `year` is equal to the median year of birth, then the height of the bar is the estimated number of Josephs alive today. Otherwise, the height of the bar is zero (so you can't see it at all). In this manner, we plot only the one darker blue bar that we want to highlight.

```

name_plot <- name_plot +
  geom_col(
    color = "white", fill = "#008fd5",
    aes(y = ifelse(year == median_yob, est_alive_today / 1000, 0))
  )

```

Lastly, the FiveThirtyEight graphic contains many contextual elements specific to the name Joseph. We can add a title, annotated text, and an arrow providing focus to a specific element of the plot. [Figure 3.22](#) displays our reproduction. There are a few differences in the presentation of fonts, title, etc. These can be altered using **ggplot2**'s theming framework, but we won't explore these subtleties here (see [Section 14.5](#)).³ Here we create a `tribble()` (a row-wise simple data frame) to add annotations.

³You may note that our number of births per year are lower than FiveThirtyEight's beginning in about 1940. It is explained in a footnote in their piece that some of the SSA records are incomplete for privacy reasons, and thus they pro-rated their data based on United States Census estimates for the early years of the century. We have omitted this step, but the `births` table in the **babynames** package will allow you to perform it.

```

context <- tribble(
  ~year, ~num_people, ~label,
  1935, 40, "Number of Josephs\nborn each year",
  1915, 13, "Number of Josephs\nborn each year
\nestimated to be alive\non 1/1/2014",
  2003, 40, "The median\nliving Joseph\nis 37 years old",
)

name_plot +
  ggtitle("Age Distribution of American Boys Named Joseph") +
  geom_text(
    data = context,
    aes(y = num_people, label = label, color = label)
  ) +
  geom_curve(
    x = 1990, xend = 1974, y = 40, yend = 24,
    arrow = arrow(length = unit(0.3, "cm")), curvature = 0.5
  ) +
  scale_color_manual(
    guide = FALSE,
    values = c("black", "#b2d7e9", "darkgray")
  ) +
  ylim(0, 42)

```

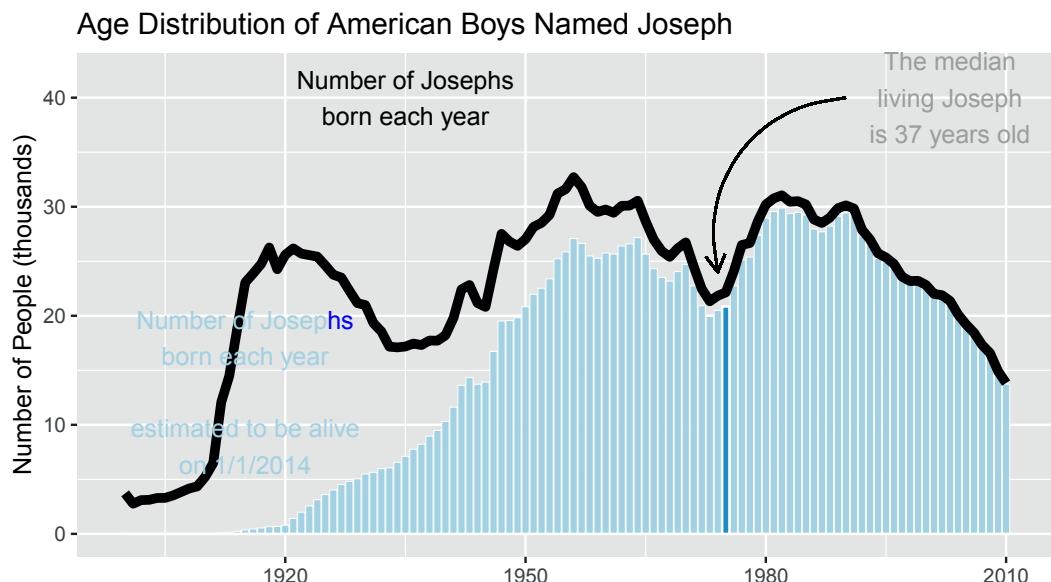


Figure 3.22: Recreation of the age distribution of “Joseph” plot.

Notice that we did not update the `name_plot` object with this contextual information. This was intentional, since we can update the `data` argument of `name_plot` and obtain an analogous plot for another name. This functionality makes use of the special `%+%` operator. As shown in [Figure 3.23](#), the name “Josephine” enjoyed a spike in popularity around 1920 that later subsided.

```
name_plot %>% filter(
  BabynamesDist,
  name == "Josephine" & sex == "F"
)
```

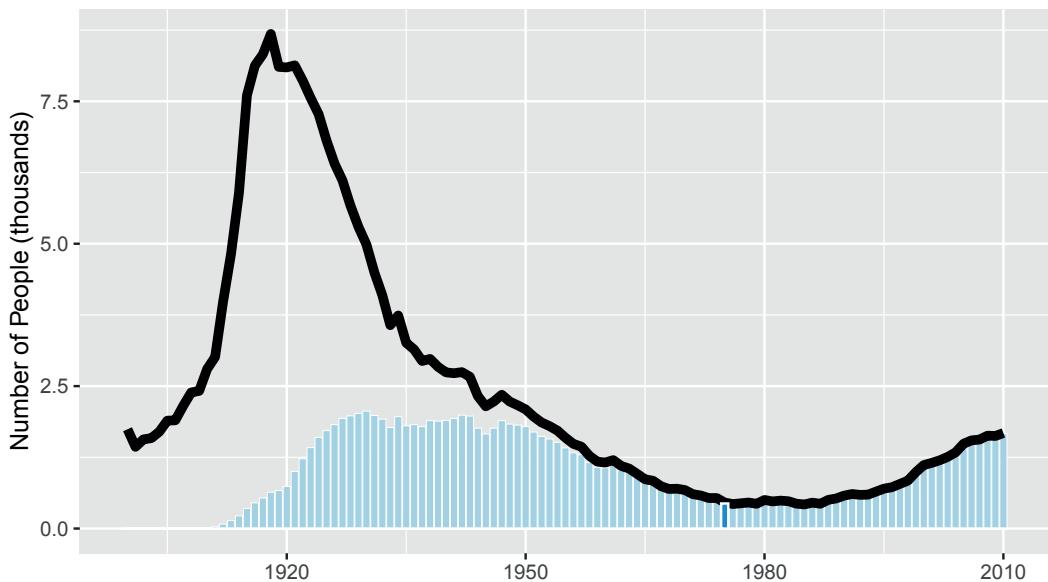


Figure 3.23: Age distribution of American girls named “Josephine.”

While some names are almost always associated with a particular gender, many are not. More interestingly, the proportion of people assigned male or female with a given name often varies over time. These data were presented nicely by Nathan Yau at FlowingData.

We can compare how our `name_plot` differs by gender for a given name using a `facet`. To do this, we will simply add a call to the `facet_wrap()` function, which will create small multiples based on a single categorical variable, and then feed a new data frame to the plot that contains data for both sexes assigned at birth. In [Figure 3.24](#), we show the prevalence of “Jessie” changed for the two sexes.

```
names_plot <- name_plot +
  facet_wrap(~sex)
names_plot %>% filter(BabynamesDist, name == "Jessie")
```

The plot at FlowingData shows the 35 most common “unisex” names—that is, the names that have historically had the greatest balance between those assigned male and female at birth. We can use a `facet_grid()` to compare the gender breakdown for a few of the most common of these, as shown in [Figures 3.25](#) and [3.26](#).

```
many_names_plot <- name_plot +
  facet_grid(name ~ sex)
mnp <- many_names_plot %>% filter(
  BabynamesDist,
  name %in% c("Jessie", "Marion", "Jackie")
)
mnp
```

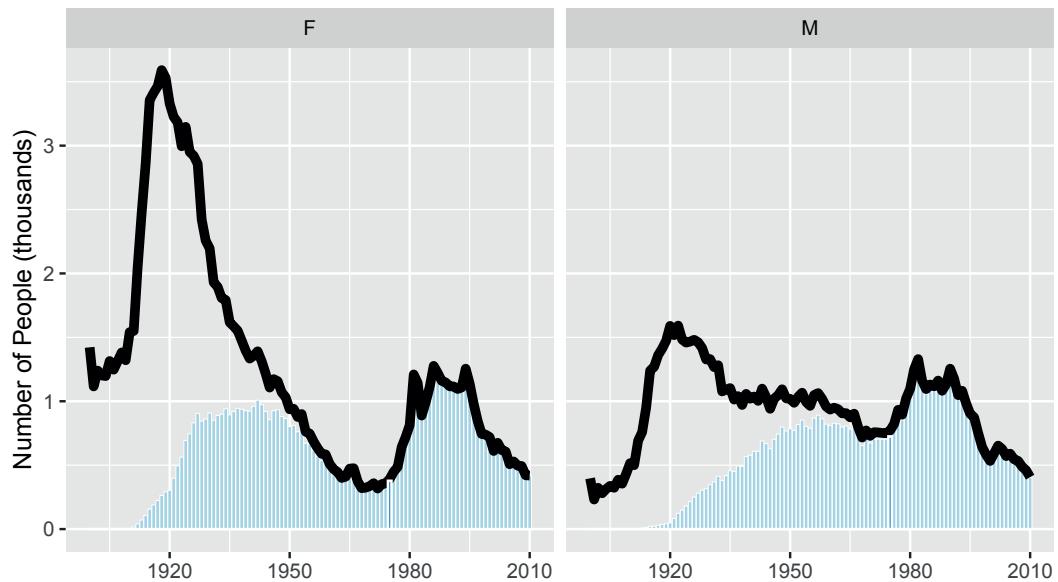


Figure 3.24: Comparison of the name “Jessie” across two genders.

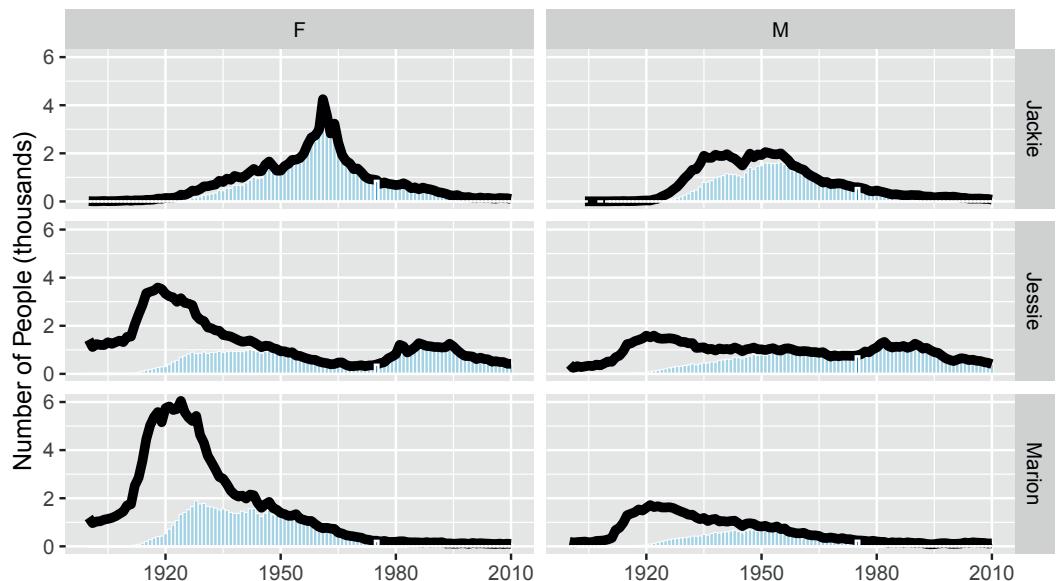


Figure 3.25: Gender breakdown for the three most unisex names.

Reversing the order of the variables in the call to `facet_grid()` flips the orientation of the facets.

```
mnp + facet_grid(sex ~ name)
```

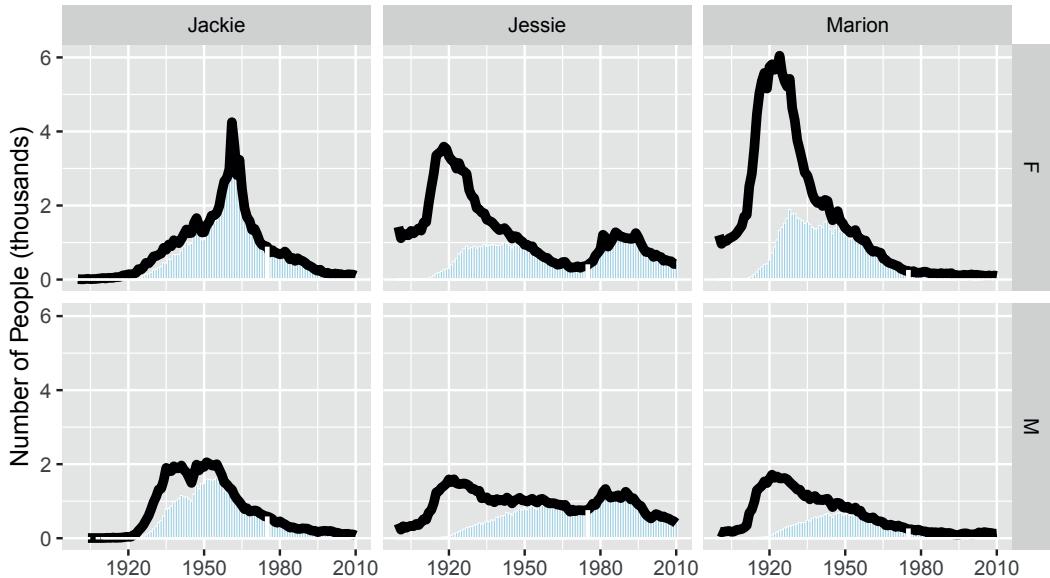


Figure 3.26: Gender breakdown for the three most unisex names, oriented vertically.

3.3.2 Most common names for women

A second interesting data graphic from the same FiveThirtyEight article is recreated in [Figure 3.27](#). Take a moment to jump ahead and analyze this data graphic. What are visual cues? What are the variables? How are the variables being mapped to the visual cues? What geoms are present?

To recreate this data graphic, we need to collect the right data. We begin by figuring out what the 25 most common female names are among those estimated to be alive today. We can do this by counting the estimated number of people alive today for each name, filtering for women, sorting by the number estimated to be alive, and then taking the top 25 results. We also need to know the median age, as well as the first and third quartiles for age among people having each name.

```
com_fem <- BabynamesDist %>%
  filter(n > 100, sex == "F") %>%
  group_by(name) %>%
  mutate(wgt = est_alive_today / sum(est_alive_today)) %>%
  summarize(
    N = n(),
    est_num_alive = sum(est_alive_today),
    quantiles = list(
      wtd_quantile(
        age_today, est_alive_today, probs = 1:3/4, na.rm = TRUE
      )
    )
  )
```

```
) %>%
  mutate(measures = list(c("q1_age", "median_age", "q3_age"))) %>%
  unnest(cols = c(quantiles, measures)) %>%
  pivot_wider(names_from = measures, values_from = quantiles) %>%
  arrange(desc(est_num_alive)) %>%
  head(25)
```

This data graphic is a bit trickier than the previous one. We'll start by binding the data, and defining the x and y aesthetics. We put the names on the x -axis and the `median_age` on the y —the reasons for doing so will be made clearer later. We will also define the title of the plot, and remove the x -axis label, since it is self-evident.

```
w_plot <- ggplot(
  data = com_fem,
  aes(x = reorder(name, -median_age), y = median_age)
) +
  xlab(NULL) +
  ylab("Age (in years)") +
  ggtitle("Median ages for females with the 25 most common names")
```

The next elements to add are the gold rectangles. To do this, we use the `geom_linerange()` function. It may help to think of these not as rectangles, but as really thick lines. Because we have already mapped the names to the x -axis, we only need to specify the mappings for `ymin` and `ymax`. These are mapped to the first and third quartiles, respectively. We will also make these lines very thick and color them appropriately. The `geom_linerange()` function only understands `ymin` and `ymax`—there is not a corresponding function with `xmin` and `xmax`. However, we will fix this later by transposing the figure. We have also added a slight `alpha` transparency to allow the gridlines to be visible underneath the gold rectangles.

```
w_plot <- w_plot +
  geom_linerange(
    aes(ymin = q1_age, ymax = q3_age),
    color = "#f3d478",
    size = 4.5,
    alpha = 0.8
  )
```

There is a red dot indicating the median age for each of these names. If you look carefully, you can see a white border around each red dot. The default glyph for `geom_point()` is a solid dot, which is shape 19. By changing it to shape 21, we can use both the `fill` and `color` arguments.

```
w_plot <- w_plot +
  geom_point(
    fill = "#ed3324",
    color = "white",
    size = 2,
    shape = 21
  )
```

It remains only to add the context and flip our plot around so the orientation matches the original figure. The `coord_flip()` function does exactly that.

```

context <- tribble(
  ~median_age, ~x, ~label,
  65, 24, "median",
  29, 16, "25th",
  48, 16, "75th percentile",
)
age_breaks <- 1:7 * 10 + 5

w_plot +
  geom_point(
    aes(y = 60, x = 24),
    fill = "#ed3324",
    color = "white",
    size = 2,
    shape = 21
) +
  geom_text(data = context, aes(x = x, label = label)) +
  geom_point(aes(y = 24, x = 16), shape = 17) +
  geom_point(aes(y = 56, x = 16), shape = 17) +
  geom_hline(
    data = tibble(x = age_breaks),
    aes(yintercept = x),
    linetype = 3
) +
  scale_y_continuous(breaks = age_breaks) +
  coord_flip()

```

You will note that the name “Anna” was fifth most common in the original FiveThirtyEight article but did not appear in [Figure 3.27](#). This appears to be a result of that name’s extraordinarily large range and the pro-rating that FiveThirtyEight did to their data. The “older” names—including Anna—were more affected by this alteration. Anna was the 47th most popular name by our calculations.

3.4 Further resources

The grammar of graphics was created by Wilkinson et al. (2005), and implemented in **ggplot2** by Wickham (2016), now in a second edition. The **ggplot2** cheat sheet produced by **RStudio** is an excellent reference for understanding the various features of **ggplot2**.

3.5 Exercises

Problem 1 (Easy): Angelica Schuyler Church (1756–1814) was the daughter of New York Governor Philip Schuyler and sister of Elizabeth Schuyler Hamilton. Angelica, New York

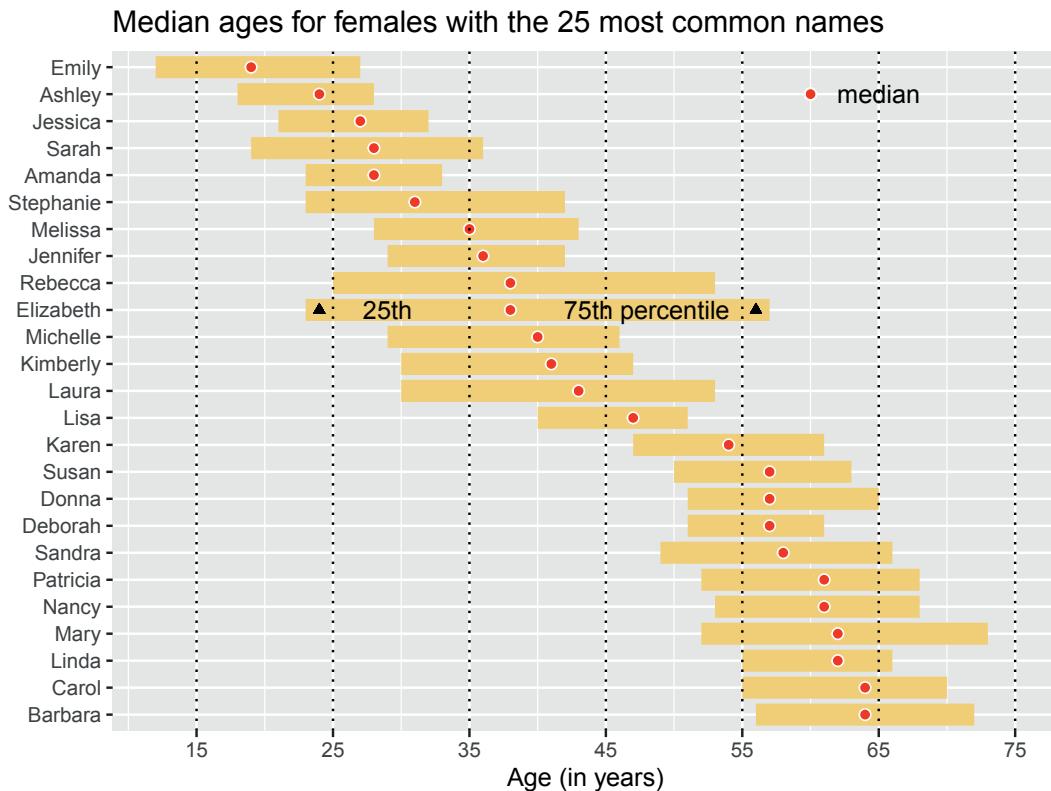


Figure 3.27: Recreation of FiveThirtyEight’s plot of the age distributions for the 25 most common women’s names.

was named after her. Using the `babynames` package generate a plot of the reported proportion of babies born with the name Angelica over time and interpret the figure.

Problem 2 (Easy): Using data from the `nasaweather` package, create a scatterplot between wind and pressure, with color being used to distinguish the type of storm.

Problem 3 (Medium): The following questions use the `Marriage` data set from the `mosaicData` package.

```
library(mosaicData)
```

- Create an informative and meaningful data graphic.
- Identify each of the visual cues that you are using, and describe how they are related to each variable.
- Create a data graphic with at least *five* variables (either quantitative or categorical). For the purposes of this exercise, do not worry about making your visualization meaningful—just try to encode five variables into one plot.

Problem 4 (Medium): The `macleish` package contains weather data collected every 10 minutes in 2015 from two weather stations in Whately, MA.

```
library(tidyverse)
```

```
library(macleish)
glimpse(whately_2015)
```

```
Rows: 52,560
Columns: 8
$ when      <dttm> 2015-01-01 00:00:00, 2015-01-01 00:10:00, 2015...
$ temperature <dbl> -9.32, -9.46, -9.44, -9.30, -9.32, -9.34, -9.30...
$ wind_speed   <dbl> 1.40, 1.51, 1.62, 1.14, 1.22, 1.09, 1.17, 1.31, ...
$ wind_dir     <dbl> 225, 248, 258, 244, 238, 242, 242, 244, 226, 22...
$ rel_humidity <dbl> 54.5, 55.4, 56.2, 56.4, 56.9, 57.2, 57.7, 58.2, ...
$ pressure     <int> 985, 985, 985, 985, 984, 984, 984, 984, 984, 98...
$ solar_radiation <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
$ rainfall      <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
```

Using `ggplot2`, create a data graphic that displays the average temperature over each 10-minute interval (`temperature`) as a function of time (`when`).

Problem 5 (Medium): Use the `MLB_teams` data in the `mdsr` package to create an informative data graphic that illustrates the relationship between winning percentage and payroll in context.

Problem 6 (Medium): The `MLB_teams` data set in the `mdsr` package contains information about Major League Baseball teams from 2008–2014. There are several quantitative and a few categorical variables present. See how many variables you can illustrate on a single plot in R. The current record is 7. (Note: This is *not* good graphical practice—it is merely an exercise to help you understand how to use visual cues and aesthetics!)

Problem 7 (Medium): The `RailTrail` data set from the `mosaicData` package describes the usage of a rail trail in Western Massachusetts. Use these data to answer the following questions.

- Create a scatterplot of the number of crossings per day `volume` against the high temperature that day
- Separate your plot into facets by `weekday` (an indicator of weekend/holiday vs. weekday)
- Add regression lines to the two facets

Problem 8 (Medium): Using data from the `nasaweather` package, use the `geom_path` function to plot the path of each tropical storm in the `storms` data table. Use color to distinguish the storms from one another, and use faceting to plot each year in its own panel.

Problem 9 (Medium): Using the `penguins` data set from the `palmerpenguins` package:

- Create a scatterplot of `bill_length_mm` against `bill_depth_mm` where individual species are colored and a regression line is added to each species. Add regression lines to all of your facets. What do you observe about the association of bill depth and bill length?
- Repeat the same scatterplot but now separate your plot into facets by `species`. How would you summarize the association between bill depth and bill length?

Problem 10 (Hard): Use the `make_babynames_dist()` function in the `mdsr` package to

recreate the “Deadest Names” graphic from FiveThirtyEight (<https://fivethirtyeight.com/features/how-to-tell-someones-age-when-all-you-know-is-her-name>).

```
library(tidyverse)
library(mdsr)
babynames_dist <- make_babynames_dist()
```

3.6 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-vizII.html#datavizII-online-exercises>



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

Data wrangling on one table

This chapter introduces basics of how to wrangle data in **R**. Wrangling skills will provide an intellectual and practical foundation for working with modern data.

4.1 A grammar for data wrangling

In much the same way that **ggplot2** presents a grammar for data graphics, the **dplyr** package presents a grammar for data wrangling (Wickham and Francois, 2020). This package is loaded when `library(tidyverse)` is run. Hadley Wickham, one of the authors of **dplyr** and the **tidyverse**, has identified five *verbs* for working with data in a data frame:

- `select()`: take a subset of the columns (i.e., features, variables)
- `filter()`: take a subset of the rows (i.e., observations)
- `mutate()`: add or modify existing columns
- `arrange()`: sort the rows
- `summarize()`: aggregate the data across rows (e.g., group it according to some criteria)

Each of these functions takes a data frame as its first argument, and returns a data frame. These five verbs can be used in conjunction with each other to provide a powerful means to slice-and-dice a single table of data. As with any grammar, what these verbs mean on their own is one thing, but being able to combine these verbs with nouns (i.e., data frames) and adverbs (i.e., arguments) creates a flexible and powerful way to wrangle data. Mastery of these five verbs can make the computation of most any descriptive statistic a breeze and facilitate further analysis. Wickham’s approach is inspired by his desire to blur the boundaries between **R** and the ubiquitous *relational database* querying syntax *SQL*. When we revisit SQL in [Chapter 15](#), we will see the close relationship between these two computing paradigms. A related concept more popular in business settings is the *OLAP* (online analytical processing) hypercube, which refers to the process by which multidimensional data is “sliced-and-diced.”

4.1.1 `select()` and `filter()`

The two simplest of the five verbs are `filter()` and `select()`, which return a subset of the rows or columns of a data frame, respectively. Generally, if we have a data frame that consists of n rows and p columns, [Figures 4.1](#) and [4.2](#) illustrate the effect of filtering this data frame based on a condition on one of the columns, and selecting a subset of the columns, respectively.

We will demonstrate the use of these functions on the `presidential` data frame (from the **ggplot2** package), which contains $p = 4$ variables about the terms of $n = 11$ recent U.S. presidents.

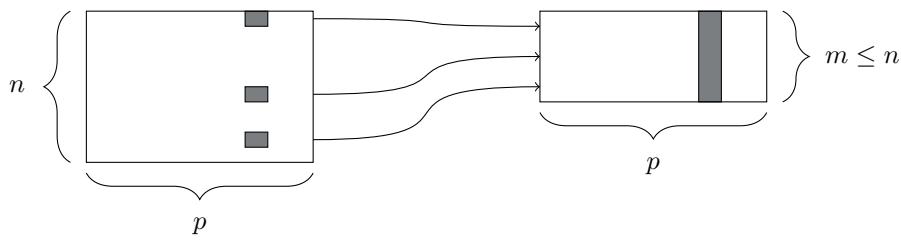


Figure 4.1: The `filter()` function. At left, a data frame that contains matching entries in a certain column for only a subset of the rows. At right, the resulting data frame after filtering.

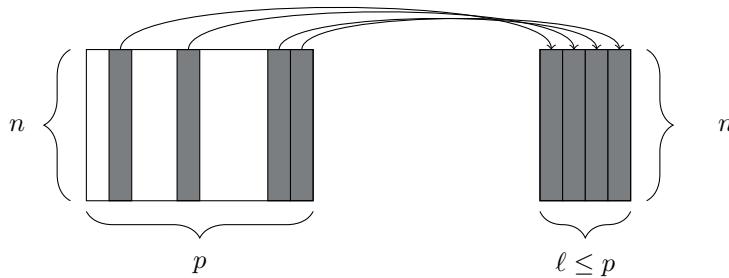


Figure 4.2: The `select()` function. At left, a data frame, from which we retrieve only a few of the columns. At right, the resulting data frame after selecting those columns.

```
library(tidyverse)
library(mdsr)
presidential
```

```
# A tibble: 11 x 4
  name      start      end    party
  <chr>     <date>     <date>   <chr>
1 Eisenhower 1953-01-20 1961-01-20 Republican
2 Kennedy    1961-01-20 1963-11-22 Democratic
3 Johnson    1963-11-22 1969-01-20 Democratic
4 Nixon      1969-01-20 1974-08-09 Republican
5 Ford        1974-08-09 1977-01-20 Republican
6 Carter      1977-01-20 1981-01-20 Democratic
7 Reagan      1981-01-20 1989-01-20 Republican
8 Bush        1989-01-20 1993-01-20 Republican
9 Clinton     1993-01-20 2001-01-20 Democratic
10 Bush       2001-01-20 2009-01-20 Republican
11 Obama      2009-01-20 2017-01-20 Democratic
```

To retrieve only the names and party affiliations of these presidents, we would use `select()`. The first *argument* to the `select()` function is the data frame, followed by an arbitrarily long list of column names, separated by commas.

```
select(presidential, name, party)
```

```
# A tibble: 11 x 2
```

```

name      party
<chr>    <chr>
1 Eisenhower Republican
2 Kennedy   Democratic
3 Johnson   Democratic
4 Nixon    Republican
5 Ford     Republican
6 Carter   Democratic
7 Reagan   Republican
8 Bush     Republican
9 Clinton  Democratic
10 Bush    Republican
11 Obama   Democratic

```

Similarly, the first argument to `filter()` is a data frame, and subsequent arguments are logical conditions that are evaluated on any involved columns. If we want to retrieve only those rows that pertain to Republican presidents, we need to specify that the value of the `party` variable is equal to `Republican`.

```
filter(presidential, party == "Republican")
```

```

# A tibble: 6 x 4
  name      start      end      party
  <chr>    <date>    <date>    <chr>
1 Eisenhower 1953-01-20 1961-01-20 Republican
2 Nixon      1969-01-20 1974-08-09 Republican
3 Ford       1974-08-09 1977-01-20 Republican
4 Reagan     1981-01-20 1989-01-20 Republican
5 Bush       1989-01-20 1993-01-20 Republican
6 Bush       2001-01-20 2009-01-20 Republican

```

Note that the `==` is a *test for equality*. If we were to use only a single equal sign here, we would be asserting that the value of `party` was `Republican`. This would result in an error. The quotation marks around `Republican` are necessary here, since `Republican` is a literal value, and not a variable name.

Combining the `filter()` and `select()` commands enables one to drill down to very specific pieces of information. For example, we can find which Democratic presidents served since *Watergate*.

```
select(
  filter(presidential, lubridate::year(start) > 1973 & party == "Democratic"),
  name
)
```

```

# A tibble: 3 x 1
  name
  <chr>
1 Carter
2 Clinton
3 Obama

```

In the syntax demonstrated above, the `filter()` operation is *nested* inside the `select()` operation. As noted above, each of the five verbs takes and returns a data frame, which

makes this type of nesting possible. Shortly, we will see how these verbs can be chained together to make rather long expressions that can become very difficult to read. Instead, we recommend the use of the `%>%` (pipe) operator. Pipe-forwarding is an alternative to nesting that yields code that can be easily read from top to bottom. With the pipe, we can write the same expression as above in this more readable syntax.

```
presidential %>%
  filter(lubridate::year(start) > 1973 & party == "Democratic") %>%
  select(name)
```

```
# A tibble: 3 x 1
  name
  <chr>
1 Carter
2 Clinton
3 Obama
```

This expression is called a *pipeline*. Notice how the expression

```
dataframe %>% filter(condition)
```

is equivalent to `filter(dataframe, condition)`. In later examples, we will see how this operator can make code more readable and efficient, particularly for complex operations on large data sets.

4.1.2 `mutate()` and `rename()`

Frequently, in the process of conducting our analysis, we will create, re-define, and rename some of our variables. The functions `mutate()` and `rename()` provide these capabilities. A graphical illustration of the `mutate()` operation is shown in Figure 4.3.

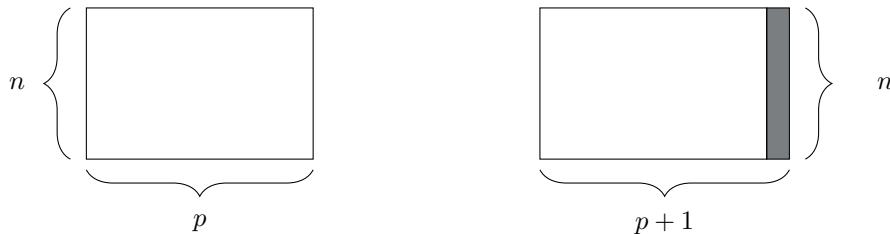


Figure 4.3: The `mutate()` function. At right, the resulting data frame after adding a new column.

While we have the raw data on when each of these presidents took and relinquished office, we don't actually have a numeric variable giving the length of each president's term. Of course, we can derive this information from the dates given, and add the result as a new column to our data frame. This date arithmetic is made easier through the use of the `lubridate` package, which we use to compute the number of years (`dyears()`) that elapsed since during the `interval()` from the `start` until the `end` of each president's term.

In this situation, it is generally considered good style to create a new object rather than *clobbering* the one that comes from an external source. To preserve the existing `presidential` data frame, we save the result of `mutate()` as a new object called `my_presidents`.

```
library(lubridate)
my_presidents <- presidential %>%
  mutate(term.length = interval(start, end) / dyears(1))
my_presidents
```

	# A tibble: 11 × 5	name	start	end	party	term.length
		<chr>	<date>	<date>	<chr>	<dbl>
1	Eisenhower	1953-01-20	1961-01-20	Republican	8	
2	Kennedy	1961-01-20	1963-11-22	Democratic	2.84	
3	Johnson	1963-11-22	1969-01-20	Democratic	5.16	
4	Nixon	1969-01-20	1974-08-09	Republican	5.55	
5	Ford	1974-08-09	1977-01-20	Republican	2.45	
6	Carter	1977-01-20	1981-01-20	Democratic	4	
7	Reagan	1981-01-20	1989-01-20	Republican	8	
8	Bush	1989-01-20	1993-01-20	Republican	4	
9	Clinton	1993-01-20	2001-01-20	Democratic	8	
10	Bush	2001-01-20	2009-01-20	Republican	8	
11	Obama	2009-01-20	2017-01-20	Democratic	8	

The `mutate()` function can also be used to modify the data in an existing column. Suppose that we wanted to add to our data frame a variable containing the year in which each president was elected. Our first (naïve) attempt might assume that every president was elected in the year before he took office. Note that `mutate()` returns a data frame, so if we want to modify our existing data frame, we need to overwrite it with the results.

```
my_presidents <- my_presidents %>%
  mutate(elected = year(start) - 1)
my_presidents
```

	# A tibble: 11 × 6	name	start	end	party	term.length	elected
		<chr>	<date>	<date>	<chr>	<dbl>	<dbl>
1	Eisenhower	1953-01-20	1961-01-20	Republican	8	1952	
2	Kennedy	1961-01-20	1963-11-22	Democratic	2.84	1960	
3	Johnson	1963-11-22	1969-01-20	Democratic	5.16	1962	
4	Nixon	1969-01-20	1974-08-09	Republican	5.55	1968	
5	Ford	1974-08-09	1977-01-20	Republican	2.45	1973	
6	Carter	1977-01-20	1981-01-20	Democratic	4	1976	
7	Reagan	1981-01-20	1989-01-20	Republican	8	1980	
8	Bush	1989-01-20	1993-01-20	Republican	4	1988	
9	Clinton	1993-01-20	2001-01-20	Democratic	8	1992	
10	Bush	2001-01-20	2009-01-20	Republican	8	2000	
11	Obama	2009-01-20	2017-01-20	Democratic	8	2008	

Some entries in this data set are wrong, because presidential elections are only held every four years. Lyndon Johnson assumed the office after President John Kennedy was assassinated in 1963, and Gerald Ford took over after President Richard Nixon resigned in 1974. Thus, there were no presidential elections in 1962 or 1973, as suggested in our data frame. We should overwrite these values with `NA`'s—which is how **R** denotes missing values. We can use the `ifelse()` function to do this. Here, if the value of `elected` is either 1962 or 1973,

we overwrite that value with `NA`.¹ Otherwise, we overwrite it with the same value that it currently has. In this case, instead of checking to see whether the value of `elected` equals 1962 or 1973, for brevity we can use the `%in%` operator to check to see whether the value of `elected` belongs to the vector consisting of 1962 and 1973.

```
my_presidents <- my_presidents %>%
  mutate(elected = ifelse(elected %in% c(1962, 1973), NA, elected))
my_presidents
```

	# A tibble: 11 × 6				
	name	start	end	party	term.length
	<chr>	<date>	<date>	<chr>	<dbl>
1	Eisenhower	1953-01-20	1961-01-20	Republican	8
2	Kennedy	1961-01-20	1963-11-22	Democratic	2.84
3	Johnson	1963-11-22	1969-01-20	Democratic	5.16
4	Nixon	1969-01-20	1974-08-09	Republican	5.55
5	Ford	1974-08-09	1977-01-20	Republican	2.45
6	Carter	1977-01-20	1981-01-20	Democratic	4
7	Reagan	1981-01-20	1989-01-20	Republican	8
8	Bush	1989-01-20	1993-01-20	Republican	4
9	Clinton	1993-01-20	2001-01-20	Democratic	8
10	Bush	2001-01-20	2009-01-20	Republican	8
11	Obama	2009-01-20	2017-01-20	Democratic	8

Finally, it is considered bad practice to use periods in the name of functions, data frames, and variables in **R**. Ill-advised periods could conflict with **R**'s use of *generic functions* (i.e., **R**'s mechanism for method overloading). Thus, we should change the name of the `term.length` column that we created earlier. We can achieve this using the `rename()` function. In this book, we will use *snake_case* for function and variable names.

Pro Tip 11. *Don't use periods in the names of functions, data frames, or variables, as this can be confused with the object-oriented programming model.*

```
my_presidents <- my_presidents %>%
  rename(term_length = term.length)
my_presidents
```

	# A tibble: 11 × 6				
	name	start	end	party	term_length
	<chr>	<date>	<date>	<chr>	<dbl>
1	Eisenhower	1953-01-20	1961-01-20	Republican	8
2	Kennedy	1961-01-20	1963-11-22	Democratic	2.84
3	Johnson	1963-11-22	1969-01-20	Democratic	5.16
4	Nixon	1969-01-20	1974-08-09	Republican	5.55
5	Ford	1974-08-09	1977-01-20	Republican	2.45
6	Carter	1977-01-20	1981-01-20	Democratic	4
7	Reagan	1981-01-20	1989-01-20	Republican	8
8	Bush	1989-01-20	1993-01-20	Republican	4
9	Clinton	1993-01-20	2001-01-20	Democratic	8
10	Bush	2001-01-20	2009-01-20	Republican	8
11	Obama	2009-01-20	2017-01-20	Democratic	8

¹Johnson was elected in 1964 as an incumbent.

4.1.3 `arrange()`

The function `sort()` will sort a vector but not a data frame. The function that will sort a data frame is called `arrange()`, and its behavior is illustrated in [Figure 4.4](#).

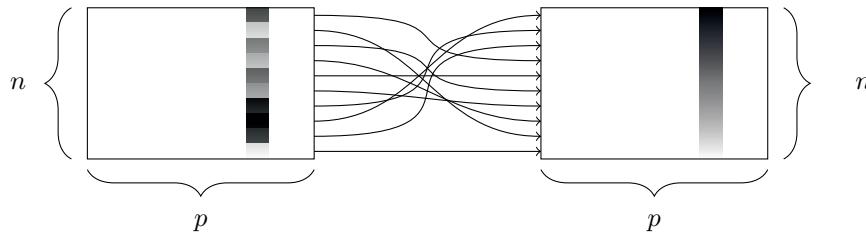


Figure 4.4: The `arrange()` function. At left, a data frame with an ordinal variable. At right, the resulting data frame after sorting the rows in descending order of that variable.

In order to use `arrange()` on a data frame, you have to specify the data frame, and the column by which you want it to be sorted. You also have to specify the direction in which you want it to be sorted. Specifying multiple sort conditions will help break ties. To sort our `presidential` data frame by the length of each president's term, we specify that we want the column `term_length` in descending order.

```
my_presidents %>%
  arrange(desc(term_length))
```

```
# A tibble: 11 x 6
  name    start      end    party  term_length elected
  <chr>   <date>    <date>  <chr>     <dbl>    <dbl>
1 Eisenhower 1953-01-20 1961-01-20 Republican     8      1952
2 Reagan     1981-01-20 1989-01-20 Republican     8      1980
3 Clinton    1993-01-20 2001-01-20 Democratic     8      1992
4 Bush       2001-01-20 2009-01-20 Republican     8      2000
5 Obama      2009-01-20 2017-01-20 Democratic     8      2008
6 Nixon      1969-01-20 1974-08-09 Republican    5.55    1968
7 Johnson    1963-11-22 1969-01-20 Democratic    5.16    NA
8 Carter      1977-01-20 1981-01-20 Democratic    4      1976
9 Bush       1989-01-20 1993-01-20 Republican    4      1988
10 Kennedy    1961-01-20 1963-11-22 Democratic    2.84    1960
11 Ford       1974-08-09 1977-01-20 Republican    2.45    NA
```

A number of presidents completed either one or two full terms, and thus have the exact same term length (4 or 8 years, respectively). To break these ties, we can further sort by `party` and `elected`.

```
my_presidents %>%
  arrange(desc(term_length), party, elected)
```

```
# A tibble: 11 x 6
  name    start      end    party  term_length elected
  <chr>   <date>    <date>  <chr>     <dbl>    <dbl>
1 Clinton  1993-01-20 2001-01-20 Democratic     8      1992
2 Obama    2009-01-20 2017-01-20 Democratic     8      2008
3 Eisenhower 1953-01-20 1961-01-20 Republican     8      1952
```

4	Reagan	1981-01-20	1989-01-20	Republican	8	1980
5	Bush	2001-01-20	2009-01-20	Republican	8	2000
6	Nixon	1969-01-20	1974-08-09	Republican	5.55	1968
7	Johnson	1963-11-22	1969-01-20	Democratic	5.16	NA
8	Carter	1977-01-20	1981-01-20	Democratic	4	1976
9	Bush	1989-01-20	1993-01-20	Republican	4	1988
10	Kennedy	1961-01-20	1963-11-22	Democratic	2.84	1960
11	Ford	1974-08-09	1977-01-20	Republican	2.45	NA

Note that the default sort order is *ascending order*, so we do not need to specify an order if that is what we want.

4.1.4 `summarize()` with `group_by()`

Our last of the five verbs for single-table analysis is `summarize()`, which is nearly always used in conjunction with `group_by()`. The previous four verbs provided us with means to manipulate a data frame in powerful and flexible ways. But the extent of the analysis we can perform with these four verbs alone is limited. On the other hand, `summarize()` with `group_by()` enables us to make comparisons.

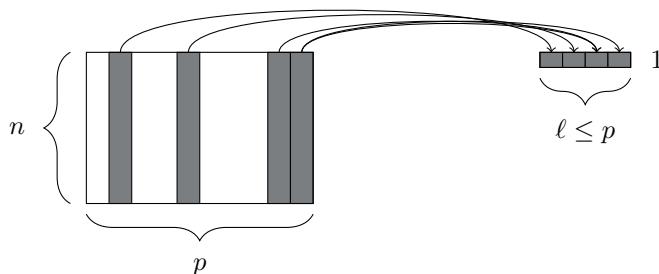


Figure 4.5: The `summarize()` function. At left, a data frame. At right, the resulting data frame after aggregating four of the columns.

When used alone, `summarize()` collapses a data frame into a single row². This is illustrated in Figure 4.5. Critically, we have to specify *how* we want to reduce an entire column of data into a single value. The method of aggregation that we specify controls what will appear in the output.

```
my_presidents %>%
  summarize(
    N = n(),
    first_year = min(year(start)),
    last_year = max(year(end)),
    num_dems = sum(party == "Democratic"),
    years = sum(term_length),
    avg_term_length = mean(term_length)
  )

# A tibble: 1 x 6
#>   N first_year last_year num_dems years avg_term_length
#>   <dbl>     <dttm>     <dttm>     <dbl>   <dbl>            <dbl>
#> 1 11 1961-01-20 2009-01-20  6       48      41.8
```

²This is no longer technically true, but it is still a helpful conceptual construction, especially for novice learners.

	<int>	<dbl>	<dbl>	<int>	<dbl>	<dbl>
1	11	1953	2017	5	64	5.82

The first argument to `summarize()` is a data frame, followed by a list of variables that will appear in the output. Note that every variable in the output is defined by operations performed on *vectors*—not on individual values. This is essential, since if the specification of an output variable is not an operation on a vector, there is no way for **R** to know how to collapse each column.

In this example, the function `n()` simply counts the number of rows. This is often useful information.

Pro Tip 12. *To help ensure that data aggregation is being done correctly, use `n()` every time you use `summarize()`.*

The next two variables determine the first year that one of these presidents assumed office. This is the smallest year in the `start` column. Similarly, the most recent year is the largest year in the `end` column. The variable `num_dems` simply counts the number of rows in which the value of the `party` variable was Democratic. Finally, the last two variables compute the sum and average of the `term_length` variable. We see that 5 of the 11 presidents who served from 1953 to 2017 were Democrats, and the average term length over these 64 years was about 5.8 years.

This begs the question of whether Democratic or Republican presidents served a longer average term during this time period. To figure this out, we can just execute `summarize()` again, but this time, instead of the first argument being the data frame `my_presidents`, we will specify that the rows of the `my_presidents` data frame should be grouped by the values of the `party` variable. In this manner, the same computations as above will be carried out for each party separately.

```
my_presidents %>%
  group_by(party) %>%
  summarize(
    N = n(),
    first_year = min(year(start)),
    last_year = max(year(end)),
    num_dems = sum(party == "Democratic"),
    years = sum(term_length),
    avg_term_length = mean(term_length)
  )

# A tibble: 2 x 7
#> #> #> #> #> #> #>
#>   party      N first_year last_year num_dems years avg_term_length
#>   <chr>     <int>     <dbl>     <dbl>     <int> <dbl>       <dbl>
#> 1 Democratic  5       1961     2017      5     28        5.6
#> 2 Republican 6       1953     2009      0     36         6
```

This provides us with the valuable information that the six Republican presidents served an average of 6 years in office, while the five Democratic presidents served an average of only 5.6. As with all of the `dplyr` verbs, the final output is a data frame.

Pro Tip 13. *In this chapter, we are using the `dplyr` package. The most common way to extract data from data tables is with SQL (structured query language). We'll introduce SQL*

in [Chapter 15](#). The `dplyr` package provides an interface that fits more smoothly into an overall data analysis workflow and is, in our opinion, easier to learn. Once you understand data wrangling with `dplyr`, it's straightforward to learn SQL if needed. `dplyr` can also work as an interface to many systems that use SQL internally.

4.2 Extended example: Ben's time with the Mets

In this extended example, we will continue to explore Sean Lahman's historical baseball database, which contains complete seasonal records for all players on all *Major League Baseball* (MLB) teams going back to 1871. These data are made available in **R** via the **Lahman** package (Friendly et al., 2020). Here again, while domain knowledge may be helpful, it is not necessary to follow the example. To flesh out your understanding, try reading the Wikipedia entry on Major League Baseball.

```
library(Lahman)
dim(Teams)
```

```
[1] 2925   48
```

The `Teams` table contains the seasonal results of every major league team in every season since 1871. There are 2925 rows and 48 columns in this table, which is far too much to show here, and would make for a quite unwieldy spreadsheet. Of course, we can take a peek at what this table looks like by printing the first few rows of the table to the screen with the `head()` command, but we won't print that on the page of this book.

Ben Baumer worked for the New York Mets from 2004 to 2012. How did the team do during those years? We can use `filter()` and `select()` to quickly identify only those pieces of information that we care about.

```
mets <- Teams %>%
  filter(teamID == "NYN")
my_mets <- mets %>%
  filter(yearID %in% 2004:2012)
my_mets %>%
  select(yearID, teamID, W, L)
```

	yearID	teamID	W	L
1	2004	NYN	71	91
2	2005	NYN	83	79
3	2006	NYN	97	65
4	2007	NYN	88	74
5	2008	NYN	89	73
6	2009	NYN	70	92
7	2010	NYN	79	83
8	2011	NYN	77	85
9	2012	NYN	74	88

Notice that we have broken this down into three steps. First, we filter the rows of the `Teams`

data frame into only those teams that correspond to the New York Mets.³ There are 58 of those, since the Mets joined the *National League* in 1962.

```
nrow(mets)
```

```
[1] 58
```

Next, we filtered these data so as to include only those seasons in which Ben worked for the team—those with `yearID` between 2004 and 2012. Finally, we printed to the screen only those columns that were relevant to our question: the year, the team's ID, and the number of wins and losses that the team had.

While this process is logical, the code can get unruly, since two ancillary data frames (`mets` and `my_mets`) were created during the process. It may be the case that we'd like to use data frames later in the analysis. But if not, they are just cluttering our workspace, and eating up memory. A more streamlined way to achieve the same result would be to *nest* these commands together.

```
select(filter(Teams, teamID == "NYN" & yearID %in% 2004:2012),
       yearID, teamID, W, L)
```

	yearID	teamID	W	L
1	2004	NYN	71	91
2	2005	NYN	83	79
3	2006	NYN	97	65
4	2007	NYN	88	74
5	2008	NYN	89	73
6	2009	NYN	70	92
7	2010	NYN	79	83
8	2011	NYN	77	85
9	2012	NYN	74	88

This way, no additional data frames were created. However, it is easy to see that as we nest more and more of these operations together, this code could become difficult to read. To maintain readability, we instead chain these operations, rather than nest them (and get the same exact results).

```
Teams %>%
  filter(teamID == "NYN" & yearID %in% 2004:2012) %>%
  select(yearID, teamID, W, L)
```

This *piping* syntax (introduced in [Section 4.1.1](#)) is provided by the `dplyr` package. It retains the step-by-step logic of our original code, while being easily readable, and efficient with respect to memory and the creation of temporary data frames. In fact, there are also performance enhancements under the hood that make this the most efficient way to do these kinds of computations. For these reasons we will use this syntax whenever possible throughout the book. Note that we only have to type `Teams` once—it is implied by the pipe operator (`%>%`) that the subsequent command takes the previous data frame as its first argument. Thus, `df %>% f(y)` is equivalent to `f(df, y)`.

We've answered the simple question of how the Mets performed during the time that Ben was there, but since we are data scientists, we are interested in deeper questions. For example, some of these seasons were subpar—the Mets had more losses than wins. Did the team just get unlucky in those seasons? Or did they actually play as badly as their record indicates?

³The `teamID` value of `NYN` stands for the **New York National League** club.

In order to answer this question, we need a model for expected winning percentage. It turns out that one of the most widely used contributions to the field of baseball analytics (courtesy of Bill James) is exactly that. This model translates the number of runs⁴ that a team scores and allows over the course of an entire season into an expectation for how many games they should have won. The simplest version of this model is this:

$$\widehat{WPct} = \frac{1}{1 + \left(\frac{RA}{RS}\right)^2},$$

where RA is the number of runs the team allows to be scored, RS is the number of runs that the team scores, and \widehat{WPct} is the team's expected winning percentage. Luckily for us, the runs scored and allowed are present in the `Teams` table, so let's grab them and save them in a new data frame.

```
mets_ben <- Teams %>%
  select(yearID, teamID, W, L, R, RA) %>%
  filter(teamID == "NYN" & yearID %in% 2004:2012)
mets_ben
```

	yearID	teamID	W	L	R	RA
1	2004	NYN	71	91	684	731
2	2005	NYN	83	79	722	648
3	2006	NYN	97	65	834	731
4	2007	NYN	88	74	804	750
5	2008	NYN	89	73	799	715
6	2009	NYN	70	92	671	757
7	2010	NYN	79	83	656	652
8	2011	NYN	77	85	718	742
9	2012	NYN	74	88	650	709

First, note that the runs-scored variable is called `R` in the `Teams` table, but to stick with our notation we want to rename it `RS`.

```
mets_ben <- mets_ben %>%
  rename(RS = R)      # new name = old name
mets_ben
```

	yearID	teamID	W	L	RS	RA
1	2004	NYN	71	91	684	731
2	2005	NYN	83	79	722	648
3	2006	NYN	97	65	834	731
4	2007	NYN	88	74	804	750
5	2008	NYN	89	73	799	715
6	2009	NYN	70	92	671	757
7	2010	NYN	79	83	656	652
8	2011	NYN	77	85	718	742
9	2012	NYN	74	88	650	709

Next, we need to compute the team's actual winning percentage in each of these seasons. Thus, we need to add a new column to our data frame, and we do this with the `mutate()` command.

⁴In baseball, a team scores a run when a player traverses the bases and return to home plate. The team with the most runs in each game wins, and no ties are allowed.

```
mets_ben <- mets_ben %>%
  mutate(WPct = W / (W + L))
mets_ben
```

	yearID	teamID	W	L	RS	RA	WPct
1	2004	NYN	71	91	684	731	0.438
2	2005	NYN	83	79	722	648	0.512
3	2006	NYN	97	65	834	731	0.599
4	2007	NYN	88	74	804	750	0.543
5	2008	NYN	89	73	799	715	0.549
6	2009	NYN	70	92	671	757	0.432
7	2010	NYN	79	83	656	652	0.488
8	2011	NYN	77	85	718	742	0.475
9	2012	NYN	74	88	650	709	0.457

We also need to compute the model estimates for winning percentage.

```
mets_ben <- mets_ben %>%
  mutate(WPct_hat = 1 / (1 + (RA/RS)^2))
mets_ben
```

	yearID	teamID	W	L	RS	RA	WPct	WPct_hat
1	2004	NYN	71	91	684	731	0.438	0.467
2	2005	NYN	83	79	722	648	0.512	0.554
3	2006	NYN	97	65	834	731	0.599	0.566
4	2007	NYN	88	74	804	750	0.543	0.535
5	2008	NYN	89	73	799	715	0.549	0.555
6	2009	NYN	70	92	671	757	0.432	0.440
7	2010	NYN	79	83	656	652	0.488	0.503
8	2011	NYN	77	85	718	742	0.475	0.484
9	2012	NYN	74	88	650	709	0.457	0.457

The expected number of wins is then equal to the product of the expected winning percentage times the number of games.

```
mets_ben <- mets_ben %>%
  mutate(W_hat = WPct_hat * (W + L))
mets_ben
```

	yearID	teamID	W	L	RS	RA	WPct	WPct_hat	W_hat
1	2004	NYN	71	91	684	731	0.438	0.467	75.6
2	2005	NYN	83	79	722	648	0.512	0.554	89.7
3	2006	NYN	97	65	834	731	0.599	0.566	91.6
4	2007	NYN	88	74	804	750	0.543	0.535	86.6
5	2008	NYN	89	73	799	715	0.549	0.555	90.0
6	2009	NYN	70	92	671	757	0.432	0.440	71.3
7	2010	NYN	79	83	656	652	0.488	0.503	81.5
8	2011	NYN	77	85	718	742	0.475	0.484	78.3
9	2012	NYN	74	88	650	709	0.457	0.457	74.0

In this case, the Mets' fortunes were better than expected in three of these seasons, and worse than expected in the other six.

```
filter(mets_ben, W >= W_hat)
```

	yearID	teamID	W	L	RS	RA	WPct	WPct_hat	W_hat
1	2006	NYN	97	65	834	731	0.599	0.566	91.6
2	2007	NYN	88	74	804	750	0.543	0.535	86.6
3	2012	NYN	74	88	650	709	0.457	0.457	74.0

```
filter(mets_ben, W < W_hat)
```

	yearID	teamID	W	L	RS	RA	WPct	WPct_hat	W_hat
1	2004	NYN	71	91	684	731	0.438	0.467	75.6
2	2005	NYN	83	79	722	648	0.512	0.554	89.7
3	2008	NYN	89	73	799	715	0.549	0.555	90.0
4	2009	NYN	70	92	671	757	0.432	0.440	71.3
5	2010	NYN	79	83	656	652	0.488	0.503	81.5
6	2011	NYN	77	85	718	742	0.475	0.484	78.3

Naturally, the Mets experienced ups and downs during Ben's time with the team. Which seasons were best? To figure this out, we can simply sort the rows of the data frame.

```
arrange(mets_ben, desc(WPct))
```

	yearID	teamID	W	L	RS	RA	WPct	WPct_hat	W_hat
1	2006	NYN	97	65	834	731	0.599	0.566	91.6
2	2008	NYN	89	73	799	715	0.549	0.555	90.0
3	2007	NYN	88	74	804	750	0.543	0.535	86.6
4	2005	NYN	83	79	722	648	0.512	0.554	89.7
5	2010	NYN	79	83	656	652	0.488	0.503	81.5
6	2011	NYN	77	85	718	742	0.475	0.484	78.3
7	2012	NYN	74	88	650	709	0.457	0.457	74.0
8	2004	NYN	71	91	684	731	0.438	0.467	75.6
9	2009	NYN	70	92	671	757	0.432	0.440	71.3

In 2006, the Mets had the best record in baseball during the regular season and nearly made the *World Series*. How do these seasons rank in terms of the team's performance relative to our model?

```
mets_ben %>%
  mutate(Diff = W - W_hat) %>%
  arrange(desc(Diff))
```

	yearID	teamID	W	L	RS	RA	WPct	WPct_hat	W_hat	Diff
1	2006	NYN	97	65	834	731	0.599	0.566	91.6	5.3840
2	2007	NYN	88	74	804	750	0.543	0.535	86.6	1.3774
3	2012	NYN	74	88	650	709	0.457	0.457	74.0	0.0199
4	2008	NYN	89	73	799	715	0.549	0.555	90.0	-0.9605
5	2009	NYN	70	92	671	757	0.432	0.440	71.3	-1.2790
6	2011	NYN	77	85	718	742	0.475	0.484	78.3	-1.3377
7	2010	NYN	79	83	656	652	0.488	0.503	81.5	-2.4954
8	2004	NYN	71	91	684	731	0.438	0.467	75.6	-4.6250
9	2005	NYN	83	79	722	648	0.512	0.554	89.7	-6.7249

It appears that 2006 was the Mets' most fortunate year—since they won five more games than our model predicts—but 2005 was the least fortunate—since they won almost seven games fewer than our model predicts. This type of analysis helps us understand how the Mets

performed in individual seasons, but we know that any randomness that occurs in individual years is likely to average out over time. So while it is clear that the Mets performed well in some seasons and poorly in others, what can we say about their overall performance?

We can easily summarize a single variable with the `skim()` command from the **mdsr** package.

```
mets_ben %>%
  skim(W)

-- Variable type: numeric -----
var      n     na   mean    sd    p0    p25    p50    p75    p100
1 W       9      0  80.9  9.10    70     74     79     88     97
```

This tells us that the Mets won nearly 81 games on average during Ben's tenure, which corresponds almost exactly to a 0.500 winning percentage, since there are 162 games in a regular season. But we may be interested in aggregating more than one variable at a time. To do this, we use `summarize()`.

```
mets_ben %>%
  summarize(
    num_years = n(),
    total_W = sum(W),
    total_L = sum(L),
    total_WPct = sum(W) / sum(W + L),
    sum_resid = sum(W - W_hat)
  )

  num_years total_W total_L total_WPct sum_resid
1           9      728      730      0.499      -10.6
```

In these nine years, the Mets had a combined record of 728 wins and 730 losses, for an overall winning percentage of .499. Just one extra win would have made them exactly 0.500! (If we could pick which game, we would definitely pick the final game of the 2007 season. A win there would have resulted in a playoff berth.) However, we've also learned that the team under-performed relative to our model by a total of 10.6 games over those nine seasons.

Usually, when we are summarizing a data frame like we did above, it is interesting to consider different groups. In this case, we can discretize these years into three chunks: one for each of the three general managers under whom Ben worked. Jim Duquette was the Mets' *general manager* in 2004, Omar Minaya from 2005 to 2010, and Sandy Alderson from 2011 to 2012. We can define these eras using two nested `ifelse()` functions.

```
mets_ben <- mets_ben %>%
  mutate(
    gm = ifelse(
      yearID == 2004,
      "Duquette",
      ifelse(
        yearID >= 2011,
        "Alderson",
        "Minaya")
    )
  )
```

Another, more scalable approach to accomplishing this same task is to use a `case_when()` expression.

```
mets_ben <- mets_ben %>%
  mutate(
    gm = case_when(
      yearID == 2004 ~ "Duquette",
      yearID >= 2011 ~ "Alderson",
      TRUE ~ "Minaya"
    )
  )
```

Pro Tip 14. *Don't use nested `ifelse()` statements: `case_when()` is far simpler.*

Next, we use the `gm` variable to define these groups with the `group_by()` operator. The combination of summarizing data by groups can be very powerful. Note that while the Mets were far more successful during Minaya's regime (i.e., many more wins than losses), they did not meet expectations in any of the three periods.

```
mets_ben %>%
  group_by(gm) %>%
  summarize(
    num_years = n(),
    total_W = sum(W),
    total_L = sum(L),
    total_WPct = sum(W) / sum(W + L),
    sum_resid = sum(W - W_hat)
  ) %>%
  arrange(desc(sum_resid))

# A tibble: 3 x 6
#> #>   gm     num_years total_W total_L total_WPct sum_resid
#> #>   <chr>     <int>    <int>    <int>      <dbl>      <dbl>
#> 1 Alderson       2      151      173      0.466     -1.32
#> 2 Duquette        1       71       91      0.438     -4.63
#> 3 Minaya         6      506      466      0.521     -4.70
```

The full power of the chaining operator is revealed below, where we do all the analysis at once, but retain the step-by-step logic.

```
Teams %>%
  select(yearID, teamID, W, L, R, RA) %>%
  filter(teamID == "NYN" & yearID %in% 2004:2012) %>%
  rename(RS = R) %>%
  mutate(
    WPct = W / (W + L),
    WPct_hat = 1 / (1 + (RA/RS)^2),
    W_hat = WPct_hat * (W + L),
    gm = case_when(
      yearID == 2004 ~ "Duquette",
      yearID >= 2011 ~ "Alderson",
      TRUE ~ "Minaya"
    )
  )
```

```

) %>%
group_by(gm) %>%
summarize(
  num_years = n(),
  total_W = sum(W),
  total_L = sum(L),
  total_WPct = sum(W) / sum(W + L),
  sum_resid = sum(W - W_hat)
) %>%
arrange(desc(sum_resid))

# A tibble: 3 x 6
gm      num_years total_W total_L total_WPct sum_resid
<chr>    <int>    <int>    <int>      <dbl>     <dbl>
1 Alderson        2      151      173      0.466    -1.32
2 Duquette         1       71       91      0.438    -4.63
3 Minaya          6      506      466      0.521    -4.70

```

Even more generally, we might be more interested in how the Mets performed relative to our model, in the context of all teams during that 9-year period. All we need to do is remove the `teamID` filter and group by franchise (`franchID`) instead.

```

Teams %>%
  select(yearID, teamID, franchID, W, L, R, RA) %>%
  filter(yearID %in% 2004:2012) %>%
  rename(RS = R) %>%
  mutate(
    WPct = W / (W + L),
    WPct_hat = 1 / (1 + (RA/RS)^2),
    W_hat = WPct_hat * (W + L)
  ) %>%
  group_by(franchID) %>%
  summarize(
    num_years = n(),
    total_W = sum(W),
    total_L = sum(L),
    total_WPct = sum(W) / sum(W + L),
    sum_resid = sum(W - W_hat)
  ) %>%
  arrange(sum_resid) %>%
  head(6)

```

```

# A tibble: 6 x 6
franchID num_years total_W total_L total_WPct sum_resid
<fct>    <int>    <int>    <int>      <dbl>     <dbl>
1 TOR          9      717      740      0.492    -29.2
2 ATL          9      781      677      0.536    -24.0
3 COL          9      687      772      0.471    -22.7
4 CHC          9      706      750      0.485    -14.5
5 CLE          9      710      748      0.487    -13.9
6 NYM          9      728      730      0.499    -10.6

```

We can see now that only five other teams fared worse than the Mets,⁵ relative to our model, during this time period. Perhaps they are cursed!

4.3 Further resources

Hadley Wickham is an influential innovator in the field of statistical computing. Along with his colleagues at **RStudio** and other organizations, he has made significant contributions to improve data wrangling in **R**. These packages are called the *tidyverse*, and are now manageable through a single **tidyverse** (Wickham, 2019h) package. His papers and vignettes describing widely-used packages such as **dplyr** (Wickham and Francois, 2020) and **tidyr** (Wickham, 2020f) are highly recommended reading. Finzer (2013) writes of a “data habit of mind” that needs to be inculcated among data scientists. The **RStudio** data wrangling cheat sheet is a useful reference.

4.4 Exercises

Problem 1 (Easy): Here is a random subset of the **babynames** data frame in the **babynames** package:

Random_subset

```
# A tibble: 10 × 5
  year sex   name      n    prop
  <dbl> <chr> <chr> <int>    <dbl>
1 2003 M   Bilal     146 0.0000695
2 1999 F   Terria    23  0.0000118
3 2010 F   Nazyiah   45  0.0000230
4 1989 F   Shawana   41  0.0000206
5 1989 F   Jessi     210 0.000105
6 1928 M   Tillman   43  0.0000377
7 1981 F   Leslee    83  0.0000464
8 1981 F   Sherise   27  0.0000151
9 1920 F   Marquerite 26  0.0000209
10 1941 M  Lorraine  24  0.0000191
```

For each of the following tables wrangled from `Random_subset`, figure out what **dplyr** wrangling statement will produce the result.

- Hint: Both rows and variables are missing from the original

```
# A tibble: 4 × 4
  year sex   name      n
  <dbl> <chr> <chr> <int>
1 2010 F   Nazyiah   45
```

⁵Note that whereas the `teamID` that corresponds to the Mets is `NYN`, the value of the `franchID` variable is `NYM`.

```
2 1989 F     Shawana    41
3 1928 M     Tillman    43
4 1981 F     Leslee     83
```

b. Hint: the `nchar()` function is used in the statement.

```
# A tibble: 2 x 5
  year sex   name      n      prop
  <dbl> <chr> <chr> <int>     <dbl>
1 1999 F   Terria    23 0.0000118
2 1981 F   Leslee    83 0.0000464
```

c. Hint: Note the new column, which is constructed from `n` and `prop`.

```
# A tibble: 2 x 6
  year sex   name      n      prop    total
  <dbl> <chr> <chr> <int>     <dbl>    <dbl>
1 1989 F   Shawana    41 0.0000206 1992225.
2 1989 F   Jessi      210 0.000105  1991843.
```

d. Hint: All the years are still there, but there are only 8 rows as opposed to the original 10 rows.

```
# A tibble: 8 x 2
  year total
  <dbl> <int>
1 1920    26
2 1928    43
3 1941    24
4 1981   110
5 1989   251
6 1999    23
7 2003   146
8 2010    45
```

Problem 2 (Easy): We'll be working with the `babynames` data frame in the `babynames` package. To remind you what `babynames` looks like, here are a few rows.

```
# A tibble: 3 x 5
  year sex   name      n      prop
  <dbl> <chr> <chr> <int>     <dbl>
1 2004 M   Arjun    250 0.000118
2 1894 F   Fedora    5 0.0000212
3 1952 F   Donaldda 10 0.00000526
```

Say what's wrong (if anything) with each of the following wrangling commands.

- `babynames %>% select(n > 100)`
- `babynames %>% select(- year)`
- `babynames %>% mutate(name_length == nchar(name))`
- `babynames %>% sex == M %>% select(-prop)`
- `babynames %>% select(year, year, sex)`
- `babynames %>% group_by(n) %>% summarize(ave = mean(n))`
- `babynames %>% group_by(n > 100) %>% summarize(total = sum(n))`

Problem 3 (Easy): Consider the following pipeline:

```
library(tidyverse)
mtcars %>%
  group_by(cyl) %>%
  summarize(avg_mpg = mean(mpg)) %>%
  filter(am == 1)
```

What is the problem with this pipeline?

Problem 4 (Easy): Define two new variables in the `Teams` data frame in the `Lahman` package.

- batting average (*BA*). Batting average is the ratio of hits (`H`) to at-bats (`AB`).
- slugging percentage (*SLG*). Slugging percentage is total bases divided by at-bats. To compute total bases, you get 1 for a single, 2 for a double, 3 for a triple, and 4 for a home run.
- Plot out the *SLG* versus `yearID`, showing the individual teams and a smooth curve.
- Same as (c), but plot *BA* versus year.

Problem 5 (Easy): Consider the following pipeline:

```
mtcars %>%
  group_by(cyl) %>%
  summarize(
    N = n(),
    avg_mpg = mean(mpg)
  )
```

```
# A tibble: 3 x 3
  cyl      N  avg_mpg
  <dbl> <int>   <dbl>
1     4     11    26.7
2     6      7    19.7
3     8     14    15.1
```

What is the real-world meaning of the variable `N` in the result set?

Problem 6 (Easy): Each of these tasks can be performed using a single data verb. For each task, say which verb it is:

- Find the average of one of the variables.
- Add a new column that is the ratio between two variables.
- Sort the cases in descending order of a variable.
- Create a new data table that includes only those cases that meet a criterion.
- From a data table with three categorical variables A, B, and C, and a quantitative variable X, produce a data frame that has the same cases but only the variables A and X.

Problem 7 (Medium): Using the `Teams` data frame in the `Lahman` package, display the top-5 teams ranked in terms of slugging percentage (*SLG*) in Major League Baseball history. Repeat this using teams since 1969. Slugging percentage is total bases divided by at-bats.

To compute total bases, you get 1 for a single, 2 for a double, 3 for a triple, and 4 for a home run.

Problem 8 (Medium): Using the `Teams` data frame in the `Lahman` package:

- a. Plot `SLG` versus `yearID` since 1954 conditioned by league (American vs. National, see `lgID`). Slugging percentage is total bases divided by at-bats. To compute total bases, you get 1 for a single, 2 for a double, 3 for a triple, and 4 for a home run.
- b. Is slugging percentage typically higher in the American League (AL) or the National League (NL)? Can you think of why this might be the case?

Problem 9 (Medium): Use the `nycflights13` package and the `flights` data frame to answer the following questions: What month had the highest proportion of cancelled flights? What month had the lowest? Interpret any seasonal patterns.

Problem 10 (Medium): Using the `Teams` data frame in the `Lahman` package:

- a. Create a factor called `election` that divides the `yearID` into 4-year blocks that correspond to U.S. presidential terms. The first presidential term started in 1788. They each last 4 years and are still on the schedule set in 1788.
- b. During which term have the most home runs been hit?

Problem 11 (Medium): The `violations` data set in the `mdsr` package contains information regarding the outcome of health inspections of restaurants in New York City. Use these data to calculate the median violation score by zip code for zip codes in Manhattan with 50 or more inspections. What pattern do you see between the number of inspections and the median score?

Problem 12 (Medium): The `nycflights13` package includes a table (`weather`) that describes the weather during 2013. Use that table to answer the following questions:

- a. What is the distribution of temperature in July, 2013? Identify any important outliers in terms of the `wind_speed` variable.
- b. What is the relationship between `dewp` and `humid`?
- c. What is the relationship between `precip` and `visib`?

Problem 13 (Medium): The Major League Baseball Angels have at times been called the California Angels (*CAL*), the Anaheim Angels (*ANA*), and the Los Angeles Angels of Anaheim (*LAA*). Using the `Teams` data frame in the `Lahman` package:

- a. Find the 10 most successful seasons in Angels history, defining “successful” as the fraction of regular-season games won in the year. In the table you create, include the `yearID`, `teamID`, `lgID`, `W`, `L`, and `wsWin`. See the documentation for `Teams` for the definition of these variables.
- b. Have the Angels ever won the World Series?

Problem 14 (Medium): Use the `nycflights13` package and the `flights` data frame to answer the following question: What plane (specified by the `tailnum` variable) traveled the most times from New York City airports in 2013? Plot the number of trips per week over the year.

4.5 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-dataI.html#dataI-online-exercises>

5

Data wrangling on multiple tables

In the previous chapter, we illustrated how the five data wrangling verbs can be chained to perform operations on a single table. A single table is reminiscent of a single well-organized spreadsheet. But in the same way that a workbook can contain multiple spreadsheets, we will often work with multiple tables. In [Chapter 15](#), we will describe how multiple tables related by unique identifiers called *keys* can be organized into a *relational database management system*.

It is more efficient for the computer to store and search tables in which “like is stored with like.” Thus, a database maintained by the *Bureau of Transportation Statistics* on the arrival times of U.S. commercial flights will consist of multiple tables, each of which contains data about different things. For example, the `nycflights13` package contains one table about `flights`—each row in this table is a single flight. As there are many flights, you can imagine that this table will get very long—hundreds of thousands of rows per year.

There are other related kinds of information that we will want to know about these flights. We would certainly be interested in the particular airline to which each flight belonged. It would be inefficient to store the complete name of the airline (e.g., *American Airlines Inc.*) in every row of the `flights` table. A simple code (e.g., `AA`) would take up less space on disk. For small tables, the savings of storing two characters instead of 25 is insignificant, but for large tables, it can add up to noticeable savings both in terms of the size of data on disk, and the speed with which we can search it. However, we still want to have the full names of the airlines available if we need them. The solution is to store the data *about airlines* in a separate table called `airlines`, and to provide a *key* that links the data in the two tables together.

5.1 `inner_join()`

If we examine the first few rows of the `flights` table, we observe that the `carrier` column contains a two-character string corresponding to the airline.

```
library(tidyverse)
library(mdsr)
library(nycflights13)
glimpse(flights)

Rows: 336,776
Columns: 19
$ year      <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ...
$ month     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
$ day       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
```

```
$ dep_time      <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558...
$ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600...
$ dep_delay     <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, ...
$ arr_time      <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 75...
$ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 74...
$ arr_delay     <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -2, -3...
$ carrier       <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", ...
$ flight        <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, ...
$ tailnum       <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN"...
$ origin        <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", ...
$ dest          <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", ...
$ air_time      <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, ...
$ distance      <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944...
$ hour          <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 5, ...
$ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
$ time_hour    <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-...
```

In the `airlines` table, we have those same two-character strings, but also the full names of the airline.

```
head(airlines, 3)
```

```
# A tibble: 3 x 2
  carrier name
  <chr>   <chr>
1 9E      Endeavor Air Inc.
2 AA      American Airlines Inc.
3 AS      Alaska Airlines Inc.
```

In order to retrieve a list of flights and the full names of the airlines that managed each flight, we need to match up the rows in the `flights` table with those rows in the `airlines` table that have the corresponding values for the `carrier` column in *both* tables. This is achieved with the function `inner_join()`.

```
flights_joined <- flights %>%
  inner_join(airlines, by = c("carrier" = "carrier"))
glimpse(flights_joined)
```

Rows: 336,776

Columns: 20

```
$ year        <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ...
$ month       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
$ day         <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
$ dep_time    <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558...
$ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600...
$ dep_delay   <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, ...
$ arr_time    <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 75...
$ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 74...
$ arr_delay   <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -2, -3...
$ carrier     <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", ...
$ flight       <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, ...
$ tailnum     <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN"...
$ origin       <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", ...
```

```
$ dest          <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL",...
$ air_time     <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138,...
$ distance     <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944...
$ hour         <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 5, ...
$ minute        <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0...
$ time_hour    <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-...
$ name          <chr> "United Air Lines Inc.", "United Air Lines Inc."...
```

Notice that the `flights_joined` data frame now has an additional variable called `name`. This is the column from `airlines` that is now included in the combined data frame. We can view the full names of the airlines instead of the cryptic two-character codes.

```
flights_joined %>%
  select(carrier, name, flight, origin, dest) %>%
  head(3)
```

```
# A tibble: 3 x 5
  carrier name           flight origin dest
  <chr>   <chr>       <int> <chr>  <chr>
1 UA      United Air Lines Inc. 1545 EWR   IAH
2 UA      United Air Lines Inc. 1714 LGA   IAH
3 AA      American Airlines Inc. 1141 JFK   MIA
```

In an `inner_join()`, the result set contains only those rows that have matches in both tables. In this case, all of the rows in `flights` have exactly one corresponding entry in `airlines`, so the number of rows in `flights_joined` is the same as the number of rows in `flights` (this will not always be the case).

```
nrow(flights)
```

```
[1] 336776
```

```
nrow(flights_joined)
```

```
[1] 336776
```

Pro Tip 15. *It is always a good idea to carefully check that the number of rows returned by a join operation is what you expected. In particular, you should carefully check for rows in one table that matched to more than one row in the other table.*

5.2 left_join()

Another commonly-used type of join is a `left_join()`. Here the rows of the first table are *always* returned, regardless of whether there is a match in the second table.

Suppose that we are only interested in flights from the *New York City* airports on the *West Coast*. To restrict ourselves to airports in the *Pacific Time Zone* (UTC -8) we can filter the `airports` data frame to only include those airports.

```
airports_pt <- airports %>%
  filter(tz == -8)
nrow(airports_pt)
```

```
[1] 178
```

Now, if we perform an `inner_join()` on `flights` and `airports_pt`, matching the destinations in `flights` to the *FAA* codes in `airports`, we retrieve only those flights that flew to our airports in the Pacific Time Zone.

```
nyc_dests_pt <- flights %>%
  inner_join(airports_pt, by = c("dest" = "faa"))
nrow(nyc_dests_pt)
```

```
[1] 46324
```

However, if we use a `left_join()` with the same conditions, we retrieve all of the rows of `flights`. NA's are inserted into the columns where no matched data was found.

```
nyc_dests <- flights %>%
  left_join(airports_pt, by = c("dest" = "faa"))
```

```
nyc_dests %>%
  summarize(
    num_flights = n(),
    num_flights_pt = sum(!is.na(name)),
    num_flights_not_pt = sum(is.na(name))
  )
```

```
# A tibble: 1 x 3
  num_flights num_flights_pt num_flights_not_pt
        <int>          <int>            <int>
1       336776           46324            290452
```

Left joins are particularly useful in databases in which *referential integrity* is broken (not all of the *keys* are present—see [Chapter 15](#)).

5.3 Extended example: Manny Ramirez

In the context of baseball and the **Lahman** package, multiple tables are used to store information. The batting statistics of players are stored in one table (`Batting`), while information about people (most of whom are players) is in a different table (`Master`).

Every row in the `Batting` table contains the statistics accumulated by a single player during a single stint for a single team in a single year. Thus, a player like Manny Ramirez has many rows in the `Batting` table (21, in fact).

```
library(Lahman)
manny <- Batting %>%
  filter(playerID == "ramirma02")
nrow(manny)
```

```
[1] 21
```

Using what we've learned, we can quickly tabulate Ramirez's most common career offensive statistics. For those new to baseball, some additional background may be helpful. A hit (`H`) occurs when a batter reaches base safely. A *home run* (`HR`) occurs when the ball is hit

out of the park or the runner advances through all of the bases during that play. Barry Bonds has the record for most home runs (762) hit in a career. A player's batting average (BA) is the ratio of the number of hits to the number of eligible at-bats. The highest career batting average in *Major League Baseball* history of 0.366 was achieved by Ty Cobb—season averages above 0.300 are impressive. Finally, runs batted in (RBI) is the number of runners (including the batter in the case of a home run) that score during that batter's at-bat. Hank Aaron has the record for most career RBIs with 2,297.

```
manny %>%
  summarize(
    span = paste(min(yearID), max(yearID), sep = "-"),
    num_years = n_distinct(yearID),
    num_teams = n_distinct(teamID),
    BA = sum(H)/sum(AB),
    tH = sum(H),
    tHR = sum(HR),
    tRBI = sum(RBI)
  )
```

	span	num_years	num_teams	BA	tH	tHR	tRBI
1	1993-2011	19	5	0.312	2574	555	1831

Notice how we have used the `paste()` function to combine results from multiple variables into a new variable, and how we have used the `n_distinct()` function to count the number of distinct rows. In his 19-year career, Ramirez hit 555 home runs, which puts him in the top 20 among all Major League players.

However, we also see that Ramirez played for five teams during his career. Did he perform equally well for each of them? Breaking his statistics down by team, or by league, is as easy as adding an appropriate `group_by()` command.

```
manny %>%
  group_by(teamID) %>%
  summarize(
    span = paste(min(yearID), max(yearID), sep = "-"),
    num_years = n_distinct(yearID),
    num_teams = n_distinct(teamID),
    BA = sum(H)/sum(AB),
    tH = sum(H),
    tHR = sum(HR),
    tRBI = sum(RBI)
  ) %>%
  arrange(span)
```

	teamID	span	num_years	num_teams	BA	tH	tHR	tRBI
1	CLE	1993-2000	8	1	0.313	1086	236	804
2	BOS	2001-2008	8	1	0.312	1232	274	868
3	LAN	2008-2010	3	1	0.322	237	44	156
4	CHA	2010-2010	1	1	0.261	18	1	2
5	TBA	2011-2011	1	1	0.0588	1	0	1

While Ramirez was very productive for Cleveland, Boston, and the *Los Angeles Dodgers*,

his brief tours with the *Chicago White Sox* and *Tampa Bay Rays* were less than stellar. In the pipeline below, we can see that Ramirez spent the bulk of his career in the *American League*.

```
manny %>%
  group_by(lgID) %>%
  summarize(
    span = paste(min(yearID), max(yearID), sep = "-"),
    num_years = n_distinct(yearID),
    num_teams = n_distinct(teamID),
    BA = sum(H)/sum(AB),
    tH = sum(H),
    tHR = sum(HR),
    tRBI = sum(RBI)
  ) %>%
  arrange(span)
```

```
# A tibble: 2 x 8
  lgID   span     num_years num_teams     BA     tH     tHR     tRBI
  <fct> <chr>      <int>     <int> <dbl> <int> <int> <int>
1 AL    1993-2011     18        4 0.311  2337    511   1675
2 NL    2008-2010      3        1 0.322   237     44    156
```

If Ramirez played in only 19 different seasons, why were there 21 rows attributed to him? Notice that in 2008, he was traded from the *Boston Red Sox* to the *Los Angeles Dodgers*, and thus played for both teams. Similarly, in 2010 he played for both the Dodgers and the *Chicago White Sox*.

Pro Tip 16. When summarizing data, it is critically important to understand exactly how the rows of your data frame are organized.

To see what can go wrong here, suppose we were interested in tabulating the number of seasons in which Ramirez hit at least 30 home runs. The simplest solution is:

```
manny %>%
  filter(HR >= 30) %>%
  nrow()
```

```
[1] 11
```

But this answer is wrong, because in 2008, Ramirez hit 20 home runs for Boston before being traded and then 17 more for the Dodgers afterwards. Neither of those rows were counted, since they were *both* filtered out. Thus, the year 2008 does not appear among the 11 that we counted in the previous pipeline. Recall that each row in the `manny` data frame corresponds to one stint with one team in one year. On the other hand, the question asks us to consider each year, *regardless of team*. In order to get the right answer, we have to aggregate the rows by team. Thus, the correct solution is:

```
manny %>%
  group_by(yearID) %>%
  summarize(tHR = sum(HR)) %>%
  filter(tHR >= 30) %>%
  nrow()
```

```
[1] 12
```

Note that the `filter()` operation is applied to `tHR`, the total number of home runs in a season, and not `HR`, the number of home runs in a single stint for a single team in a single season. (This distinction between filtering the rows of the original data versus the rows of the aggregated results will appear again in [Chapter 15](#).)

We began this example by filtering the `Batting` table for the player with `playerID` equal to `ramirma02`. How did we know to use this identifier? This player ID is known as a *key*, and in fact, `playerID` is the *primary key* defined in the `Master` table. That is, every row in the `Master` table is uniquely identified by the value of `playerID`. There is exactly one row in that table for which `playerID` is equal to `ramirma02`.

But how did we know that this ID corresponds to Manny Ramirez? We can search the `Master` table. The data in this table include characteristics about Manny Ramirez that do not change across multiple seasons (with the possible exception of his weight).

```
Master %>%
  filter(nameLast == "Ramirez" & nameFirst == "Manny")
```

	playerID	birthYear	birthMonth	birthDay	birthCountry	birthState
1	ramirma02	1972	5	30	D.R.	Distrito Nacional
					birthCity	deathYear
					deathMonth	deathDay
					deathCountry	deathState
1	Santo Domingo	NA	NA	NA	<NA>	<NA>
					deathCity	nameFirst
					nameLast	nameGiven
					weight	height
1	<NA>	Manny	Ramirez	Manuel Aristides	225	72
					R	R
					debut	finalGame
					retroID	bbrefID
					deathDate	birthDate
1	1993-09-02	2011-04-06	ramim002	ramirma02	<NA>	1972-05-30

The `playerID` column forms a primary key in the `Master` table, but it does not in the `Batting` table, since as we saw previously, there were 21 rows with that `playerID`. In the `Batting` table, the `playerID` column is known as a *foreign key*, in that it references a primary key in another table. For our purposes, the presence of this column in both tables allows us to link them together. This way, we can combine data from the `Batting` table with data in the `Master` table. We do this with `inner_join()` by specifying the two tables that we want to join, and the corresponding columns in each table that provide the link. Thus, if we want to display Ramirez's name in our previous result, as well as his age, we must join the `Batting` and `Master` tables together.

Pro Tip 17. Always specify the `by` argument that defines the join condition. Don't rely on the defaults.

```
Batting %>%
  filter(playerID == "ramirma02") %>%
  inner_join(Master, by = c("playerID" = "playerID")) %>%
  group_by(yearID) %>%
  summarize(
    Age = max(yearID - birthYear),
    num_teams = n_distinct(teamID),
    BA = sum(H)/sum(AB),
    tH = sum(H),
    tHR = sum(HR),
    tRBI = sum(RBI)
  ) %>%
  arrange(yearID)
```

```
# A tibble: 19 x 7
  yearID  Age num_teams     BA    tH    tHR   tRBI
  <int> <int>      <int> <dbl> <int> <int> <int>
1 1993    21        1 0.170     9     2     5
2 1994    22        1 0.269    78    17    60
3 1995    23        1 0.308   149    31   107
4 1996    24        1 0.309   170    33   112
5 1997    25        1 0.328   184    26    88
6 1998    26        1 0.294   168    45   145
7 1999    27        1 0.333   174    44   165
8 2000    28        1 0.351   154    38   122
9 2001    29        1 0.306   162    41   125
10 2002   30        1 0.349   152    33   107
11 2003   31        1 0.325   185    37   104
12 2004   32        1 0.308   175    43   130
13 2005   33        1 0.292   162    45   144
14 2006   34        1 0.321   144    35   102
15 2007   35        1 0.296   143    20    88
16 2008   36        2 0.332   183    37   121
17 2009   37        1 0.290   102    19    63
18 2010   38        2 0.298    79     9    42
19 2011   39        1 0.0588    1     0     1
```

Notice that even though Ramirez's age is a constant for each season, we have to use a vector operation (i.e., `max()` or `first()`) in order to reduce any potential vector to a single number.

Which season was Ramirez's best as a hitter? One relatively simple measurement of batting prowess is OPS, or *On-Base Plus Slugging Percentage*, which is the simple sum of two other statistics: *On-Base Percentage* (OBP) and *Slugging Percentage* (SLG). The former basically measures the proportion of time that a batter reaches base safely, whether it comes via a hit (`H`), a base on balls (`BB`), or from being hit by the pitch (`HPB`). The latter measures the average number of bases advanced per at-bat (`AB`), where a single is worth one base, a double (`X2B`) is worth two, a triple (`X3B`) is worth three, and a home run (`HR`) is worth four. (Note that every hit is exactly one of a single, double, triple, or home run.) Let's add these statistics to our results and use it to rank the seasons.

```
manny_by_season <- Batting %>%
  filter(playerID == "ramirma02") %>%
  inner_join(Master, by = c("playerID" = "playerID")) %>%
  group_by(yearID) %>%
  summarize(
    Age = max(yearID - birthYear),
    num_teams = n_distinct(teamID),
    BA = sum(H)/sum(AB),
    tH = sum(H),
    tHR = sum(HR),
    tRBI = sum(RBI),
    OBP = sum(H + BB + HBP) / sum(AB + BB + SF + HBP),
    SLG = sum(H + X2B + 2 * X3B + 3 * HR) / sum(AB)
  ) %>%
  mutate(OPS = OBP + SLG) %>%
```

```
arrange(desc(OPS))
manny_by_season
```

```
# A tibble: 19 x 10
  yearID  Age num_teams     BA    tH    tHR   tRBI    OBP    SLG    OPS
  <int> <int>     <int> <dbl> <int> <int> <int> <dbl> <dbl> <dbl>
1  2000    28      1 0.351   154     38    122 0.457  0.697  1.15
2  1999    27      1 0.333   174     44    165 0.442  0.663  1.11
3  2002    30      1 0.349   152     33    107 0.450  0.647  1.10
4  2006    34      1 0.321   144     35    102 0.439  0.619  1.06
5  2008    36      2 0.332   183     37    121 0.430  0.601  1.03
6  2003    31      1 0.325   185     37    104 0.427  0.587  1.01
7  2001    29      1 0.306   162     41    125 0.405  0.609  1.01
8  2004    32      1 0.308   175     43    130 0.397  0.613  1.01
9  2005    33      1 0.292   162     45    144 0.388  0.594  0.982
10 1996    24      1 0.309   170     33    112 0.399  0.582  0.981
11 1998    26      1 0.294   168     45    145 0.377  0.599  0.976
12 1995    23      1 0.308   149     31    107 0.402  0.558  0.960
13 1997    25      1 0.328   184     26     88 0.415  0.538  0.953
14 2009    37      1 0.290   102     19     63 0.418  0.531  0.949
15 2007    35      1 0.296   143     20     88 0.388  0.493  0.881
16 1994    22      1 0.269    78     17     60 0.357  0.521  0.878
17 2010    38      2 0.298    79      9     42 0.409  0.460  0.870
18 1993    21      1 0.170     9      2      5 0.2    0.302  0.502
19 2011    39      1 0.0588    1      0      1 0.0588 0.0588 0.118
```

We see that Ramirez's OPS was highest in 2000. But 2000 was the height of the *steroid era*, when many sluggers were putting up tremendous offensive numbers. As data scientists, we know that it would be more instructive to put Ramirez's OPS in context by comparing it to the league average OPS in each season—the resulting ratio is often called *OPS+*. To do this, we will need to compute those averages. Because there is missing data in some of these columns in some of these years, we need to invoke the `na.rm` argument to ignore that data.

```
mlb <- Batting %>%
  filter(yearID %in% 1993:2011) %>%
  group_by(yearID) %>%
  summarize(
    lg_OBP = sum(H + BB + HBP, na.rm = TRUE) /
      sum(AB + BB + SF + HBP, na.rm = TRUE),
    lg_SLG = sum(H + X2B + 2*X3B + 3*HR, na.rm = TRUE) /
      sum(AB, na.rm = TRUE)
  ) %>%
  mutate(lgOPS = lg_OBP + lg_SLG)
```

Next, we need to match these league average OPS values to the corresponding entries for Ramirez. We can do this by joining these tables together, and computing the ratio of Ramirez's OPS to that of the league average.

```
manny_ratio <- manny_by_season %>%
  inner_join(mlb, by = c("yearID" = "yearID")) %>%
  mutate(OPS_plus = OPS / lgOPS) %>%
  select(yearID, Age, OPS, lgOPS, OPS_plus) %>%
```

```

arrange(desc(OPS_plus))
manny_ratio

# A tibble: 19 x 5
  yearID  Age   OPS lgOPS OPS_plus
  <int> <int> <dbl> <dbl>     <dbl>
1 2000    28  1.15  0.782    1.48
2 2002    30  1.10  0.748    1.47
3 1999    27  1.11  0.778    1.42
4 2006    34  1.06  0.768    1.38
5 2008    36  1.03  0.749    1.38
6 2003    31  1.01  0.755    1.34
7 2001    29  1.01  0.759    1.34
8 2004    32  1.01  0.763    1.32
9 2005    33  0.982  0.749    1.31
10 1998   26  0.976  0.755    1.29
11 1996   24  0.981  0.767    1.28
12 1995   23  0.960  0.755    1.27
13 2009   37  0.949  0.751    1.26
14 1997   25  0.953  0.756    1.26
15 2010   38  0.870  0.728    1.19
16 2007   35  0.881  0.758    1.16
17 1994   22  0.878  0.763    1.15
18 1993   21  0.502  0.736    0.682
19 2011   39  0.118  0.720    0.163

```

In this case, 2000 still ranks as Ramirez's best season relative to his peers, but notice that his 1999 season has fallen from 2nd to 3rd. Since by definition a league average batter has an OPS+ of 1, Ramirez posted 17 consecutive seasons with an OPS that was at least 15% better than the average across the major leagues—a truly impressive feat.

Finally, not all joins are the same. An `inner_join()` requires corresponding entries in *both* tables. Conversely, a `left_join()` returns at least as many rows as there are in the first table, regardless of whether there are matches in the second table. An `inner_join()` is bidirectional, whereas in a `left_join()`, the order in which you specify the tables matters.

Consider the career of Cal Ripken, who played in 21 seasons from 1981 to 2001. His career overlapped with Ramirez's in the nine seasons from 1993 to 2001, so for those, the league averages we computed before are useful.

```

ripken <- Batting %>%
  filter(playerID == "ripkeca01")
ripken %>%
  inner_join(mlb, by = c("yearID" = "yearID")) %>%
  nrow()

```

```

[1] 9
# same
mlb %>%
  inner_join(ripken, by = c("yearID" = "yearID")) %>%
  nrow()

```

```
[1] 9
```

For seasons when Ramirez did not play, NA's will be returned.

```
ripken %>%
  left_join(mlb, by = c("yearID" = "yearID")) %>%
  select(yearID, playerID, lg_OPS) %>%
  head(3)
```

```
yearID  playerID lg_OPS
1    1981 ripkeca01     NA
2    1982 ripkeca01     NA
3    1983 ripkeca01     NA
```

Conversely, by reversing the order of the tables in the join, we return the 19 seasons for which we have already computed the league averages, regardless of whether there is a match for Ripken (results not displayed).

```
mlb %>%
  left_join(ripken, by = c("yearID" = "yearID")) %>%
  select(yearID, playerID, lg_OPS)
```

5.4 Further resources

Sean Lahman has long curated his baseball data set, which feeds the popular website baseball-reference.com. Michael Friendly maintains the **Lahman R** package (Friendly et al., 2020). For the baseball enthusiast, *Cleveland Indians* analyst Max Marchi and Jim Albert have written an excellent book on analyzing baseball data in **R** (Marchi and Albert, 2013). A second edition updates the code for the **tidyverse** (Albert et al., 2018). Albert has also written a book describing how baseball can be used as a motivating example for teaching statistics (Albert, 2003).

5.5 Exercises

Problem 1 (Easy): Consider the following data frames with information about U.S. states from 1977.

```
statenames <- tibble(names = state.name, twoletter = state.abb)
glimpse(statenames)

Rows: 50
Columns: 2
$ names      <chr> "Alabama", "Alaska", "Arizona", "Arkansas", "California", ...
$ twoletter  <chr> "AL", "AK", "AZ", "AR", "CA", "CO", "CT", "DE", "FL", ...
```

```
statedata <- tibble(
  names = state.name,
  income = state.x77[, 2],
  illiteracy = state.x77[, 3]
```

```
)
glimpse(statedata)
```

Rows: 50

Columns: 3

\$ names <chr> "Alabama", "Alaska", "Arizona", "Arkansas", "Califor...

\$ income <dbl> 3624, 6315, 4530, 3378, 5114, 4884, 5348, 4809, 4815...

\$ illiteracy <dbl> 2.1, 1.5, 1.8, 1.9, 1.1, 0.7, 1.1, 0.9, 1.3, 2.0, 1....

Create a scatterplot of illiteracy (as a percent of population) and per capita income (in U.S. dollars) with points plus labels of the two letter state abbreviations. Add a smoother. Use the `ggrepel` package to offset the names that overlap. What pattern do you observe? Are there any outlying observations?

Problem 2 (Medium): Use the `Batting`, `Pitching`, and `Master` tables in the `Lahman` package to answer the following questions.

- Name every player in baseball history who has accumulated at least 300 home runs (`HR`) and at least 300 stolen bases (`SB`). You can find the first and last name of the player in the `Master` data frame. Join this to your result along with the total home runs and total bases stolen for each of these elite players.
- Similarly, name every pitcher in baseball history who has accumulated at least 300 wins (`w`) and at least 3,000 strikeouts (`so`).
- Identify the name and year of every player who has hit at least 50 home runs in a single season. Which player had the lowest batting average in that season?

Problem 3 (Medium): Use the `nycflights13` package and the `flights` and `planes` tables to answer the following questions:

- How many planes have a missing date of manufacture?
- What are the five most common manufacturers?
- Has the distribution of manufacturer changed over time as reflected by the airplanes flying from NYC in 2013? (Hint: you may need to use `case_when()` to recode the manufacturer name and collapse rare vendors into a category called `Other`.)

Problem 4 (Medium): Use the `nycflights13` package and the `flights` and `planes` tables to answer the following questions:

- What is the oldest plane (specified by the `tailnum` variable) that flew from New York City airports in 2013?
- How many airplanes that flew from New York City are included in the `planes` table?

Problem 5 (Medium): The Relative Age Effect is an attempt to explain anomalies in the distribution of birth month among athletes. Briefly, the idea is that children born just after the age cut-off to enroll in school will be as much as 11 months older than their fellow athletes, which is enough of a disparity to give them an advantage. That advantage will

then be compounded over the years, resulting in notably more professional athletes born in these months.

- a. Display the distribution of birth months of baseball players who batted during the decade of the 2000s.
- b. How are they distributed over the calendar year? Does this support the notion of a relative age effect? Use the `Births78` data set from the `mosaicData` package as a reference.

Problem 6 (Hard): Use the `fec12` package to download the Federal Election Commission data for 2012. Re-create [Figures 2.1](#) and [2.2](#) from the text using `ggplot2`.

5.6 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-join.html#join-online-exercises>



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

6

Tidy data

In this chapter, we will continue to develop data wrangling skills. In particular, we will discuss *tidy data*, common file formats, and techniques for scraping and cleaning data, especially dates. Together with the material from [Chapters 4](#) and [5](#), these skills will provide facility with wrangling data that is foundational for data science.

6.1 Tidy data

6.1.1 Motivation

Gapminder (Rosling et al., 2005) is the brainchild of the late Swedish physician and public health researcher Hans Rosling. Gapminder contains data about countries over time for a variety of different variables such as the prevalence of *HIV* (human immunodeficiency virus) among adults aged 15 to 49 and other health and economic indicators. These data are stored in *Google Sheets*, or one can download them as *Microsoft Excel* workbooks. The typical presentation of a small subset of such data is shown below, where we have used the `googlesheets4` package to pull these data directly into **R**. (See [Section 6.2.4](#) for a description of the `unnest()` function.)

```
library(tidyverse)
library(mdsr)
library(googlesheets4)
hiv_key <- "1kWH_xdJDM4SMFT_Kzpkk-1yuxWChfurZuWYjfmv51EA"
hiv <- read_sheet(hiv_key) %>%
  rename(Country = 1) %>%
  filter(
    Country %in% c("United States", "France", "South Africa")
  ) %>%
  select(Country, `1979`, `1989`, `1999`, `2009`) %>%
  unnest(cols = c(`2009`)) %>%
  mutate(across(matches("[0-9]"), as.double))
hiv

# A tibble: 3 x 5
  Country      `1979` `1989` `1999` `2009`
  <chr>       <dbl>   <dbl>   <dbl>   <dbl>
1 France        NA      NA     0.3     0.4
2 South Africa  NA      NA    14.8    17.2
3 United States 0.0318  NA     0.5     0.6
```

The data set has the form of a two-dimensional array where each of the $n = 3$ rows represents

a country and each of the $p = 4$ columns is a year. Each entry represents the percentage of adults aged 15 to 49 living with HIV in the i^{th} country in the j^{th} year. This presentation of the data has some advantages. First, it is possible (with a big enough display) to *see* all of the data. One can quickly follow the trend over time for a particular country, and one can also estimate quite easily the percentage of data that is missing (e.g., `NA`). If visual inspection is the primary analytical technique, this *spreadsheet-style* presentation can be convenient.

Alternatively, consider this presentation of those same data.

```
hiv %>%
  pivot_longer(-Country, names_to = "Year", values_to = "hiv_rate")
```

	Country	Year	hiv_rate
	<chr>	<chr>	<dbl>
1	France	1979	NA
2	France	1989	NA
3	France	1999	0.3
4	France	2009	0.4
5	South Africa	1979	NA
6	South Africa	1989	NA
7	South Africa	1999	14.8
8	South Africa	2009	17.2
9	United States	1979	0.0318
10	United States	1989	NA
11	United States	1999	0.5
12	United States	2009	0.6

While our data can still be represented by a two-dimensional array, it now has $np = 12$ rows and just three columns. Visual inspection of the data is now more difficult, since our data are long and very narrow—the aspect ratio is not similar to that of our screen.

It turns out that there are substantive reasons to prefer the long (or tall), narrow version of these data. With multiple tables (see [Chapter 15](#)), it is a more efficient way for the computer to store and retrieve the data. It is more convenient for the purpose of data analysis. And it is more scalable, in that the addition of a second variable simply contributes another column, whereas to add another variable to the spreadsheet presentation would require a confusing three-dimensional view, multiple tabs in the spreadsheet, or worse, *merged cells*.

These gains come at a cost: we have relinquished our ability to *see all the data at once*. When data sets are small, being able to see them all at once can be useful, and even comforting. But in this era of big data, a quest to see all the data at once in a spreadsheet layout is a *fool's errand*. Learning to manage data via programming frees us from the *click-and-drag* paradigm popularized by spreadsheet applications, allows us to work with data of arbitrary size, and reduces errors. Recording our data management operations in code also makes them reproducible (see [Appendix D](#))—an increasingly necessary trait in this era of collaboration. It enables us to fully separate the raw data from our analysis, which is difficult to achieve using a spreadsheet.

Pro Tip 18. *Always keep your raw data and your analysis in separate files. Store the uncorrected data file (with errors and problems) and make corrections with a script (see [Appendix D](#)) file that transforms the raw data into the data that will actually be analyzed.*

Table 6.1: A data table showing how many babies were given each name in each year in the United States, for a few names.

year	sex	name	n
1999	M	Kavon	104
1984	F	Somaly	6
2017	F	Dnylah	8
1918	F	Eron	6
1992	F	Arleene	5
1977	F	Alissia	5
1919	F	Bular	10

This process will maintain the provenance of your data and allow analyses to be updated with new data without having to start data wrangling from scratch.

The long, narrow format for the *Gapminder* data that we have outlined above is called *tidy data* (Wickham, 2014). In what follows, we will further expand upon this notion and develop more sophisticated techniques for wrangling data.

6.1.2 What are tidy data?

Data can be as simple as a column of numbers in a spreadsheet file or as complex as the electronic medical records collected by a hospital. A newcomer to working with data may expect each source of data to be organized in a unique way and to require unique techniques. The expert, however, has learned to operate with a small set of standard tools. As you'll see, each of the standard tools performs a comparatively simple task. Combining those simple tasks in appropriate ways is the key to dealing with complex data.

One reason the individual tools can be simple is that each tool gets applied to data arranged in a simple but precisely defined pattern called *tidy data*. Tidy data exists in systematically defined *data tables* (e.g., the rectangular arrays of data seen previously). Note that not all data tables are tidy.

To illustrate, Table 6.1 shows a handful of entries from a large *United States Social Security Administration* tabulation of names given to babies. In particular, the table shows how many babies of each sex were given each name in each year.

Table 6.1 shows that there were 104 boys named Kavon born in the U.S. in 1999 and 6 girls named Somaly born in 1984. As a whole, the `babynames` data table covers the years 1880 through 2017 and includes a total of 348,120,517 individuals, somewhat larger than the current population of the U.S.

The data in Table 6.1 are *tidy* because they are organized according to two simple rules.

1. The rows, called *cases* or observations, each refer to a specific, unique, and similar sort of thing, e.g., girls named Somaly in 1984.
2. The columns, called variables, each have the same sort of value recorded for each row. For instance, `n` gives the number of babies for each case; `sex` tells which gender was assigned at birth.

Table 6.2: The most popular baby names across all years.

sex	name	total_births
M	James	5150472
M	John	5115466
M	Robert	4814815
M	Michael	4350824
F	Mary	4123200
M	William	4102604
M	David	3611329
M	Joseph	2603445
M	Richard	2563082
M	Charles	2386048

When data are in tidy form, it is relatively straightforward to transform the data into arrangements that are more useful for answering interesting questions. For instance, you might wish to know which were the most popular baby names over all the years. Even though [Table 6.1](#) contains the popularity information implicitly, we need to rearrange these data by adding up the counts for a name across all the years before the popularity becomes obvious, as in [Table 6.2](#).

```
popular_names <- babynames %>%
  group_by(sex, name) %>%
  summarise(total_births = sum(n)) %>%
  arrange(desc(total_births))
```

The process of transforming information that is implicit in a data table into another data table that gives the information explicitly is called *data wrangling*. The wrangling itself is accomplished by using *data verbs* that take a tidy data table and transform it into another tidy data table in a different form. In [Chapters 4 and 5](#), you were introduced to several *data verbs*.

[Figure 6.1](#) displays results from the *Minneapolis* mayoral election. Unlike *babynames*, it is not in tidy form, though the display is attractive and neatly laid out. There are helpful labels and summaries that make it easy for a person to read and draw conclusions. (For instance, Ward 1 had a higher voter turnout than Ward 2, and both wards were lower than the city total.)

However, being neat is not what makes data *tidy*. [Figure 6.1](#) violates the first rule for tidy data.

- **Rule 1:** The rows, called *cases*, each must represent the same underlying attribute, that is, the same kind of thing. That's not true in [Figure 6.1](#). For most of the table, the rows represent a single precinct. But other rows give ward or city-wide totals. The first two rows are captions describing the data, not cases.
- **Rule 2:** Each column is a variable containing the same type of value for each case. That's mostly true in [Figure 6.1](#), but the tidy pattern is interrupted by labels that are not variables. For instance, the first two cells in row 15 are the label "Ward 1 Subtotal," which is different from the ward/precinct identifiers that are the values in most of the first column.

Conforming to the rules for tidy data simplifies summarizing and analyzing data. For in-

	A	B	E	F	G	H	I	J
1			City of Minneapolis Statistics General Election November 5, 2013					
3	Ward	Precinct	Voters Registering by Absentee	Total Registrations	Voters at Polls	Absentee Voters	Total Ballots Cast	Total Turnout
4	City-Wide Total		708	6,634	75,145	4,954	80,099	33.38%
5								
6	1	1	3	28	492	27	519	27.23%
7	1	2	1	44	836	56	892	31.71%
8	1	3	0	40	905	19	924	38.87%
9	1	4	5	29	768	26	794	36.62%
10	1	5	0	31	683	31	714	37.46%
11	1	6	0	69	739	20	759	32.62%
12	1	7	0	47	291	8	299	15.79%
13	1	8	0	43	415	5	420	30.55%
14	1	9	0	42	596	25	621	25.42%
15	Ward 1 Subtotal		9	373	5,725	217	5,942	30.93%
16								
17	2	1	1	63	1,011	39	1,050	36.42%
18	2	2	5	44	679	37	716	50.39%
19	2	3	4	48	324	18	342	18.88%
20	2	4	0	53	117	3	120	7.34%
21	2	5	2	50	495	26	521	25.49%
22	2	6	1	36	433	19	452	39.10%
23	2	7	0	39	138	7	145	13.78%
24	2	8	1	50	1,206	36	1,242	47.90%
25	2	9	2	39	351	16	367	30.56%
26	2	10	0	87	196	5	201	6.91%
27	Ward 2 Subtotal		16	509	4,950	206	5,156	27.56%
28								
29	3	1	0	52	165	1	166	7.04%

Figure 6.1: Ward and precinct votes cast in the 2013 Minneapolis mayoral election.

Table 6.3: A selection from the Minneapolis election data in tidy form.

ward	precinct	registered	voters	absentee	total_turnout
1	1	28	492	27	0.272
1	4	29	768	26	0.366
1	7	47	291	8	0.158
2	1	63	1011	39	0.364
2	4	53	117	3	0.073
2	7	39	138	7	0.138
2	10	87	196	5	0.069
3	3	71	893	101	0.374
3	6	102	927	71	0.353

stance, in the tidy `babynames` table, it is easy (for a computer) to find the total number of babies: just add up all the numbers in the `n` variable. It is similarly easy to find the number of cases: just count the rows. And if you want to know the total number of Ahmeds or Sherinas across the years, there is an easy way to do that.

In contrast, it would be more difficult in the *Minneapolis* election data to find, say, the total number of ballots cast. If you take the seemingly obvious approach and add up the numbers in column I of [Figure 6.1](#) (labeled “Total Ballots Cast”), the result will be *three times* the true number of ballots, because some of the rows contain summaries, not cases.

Indeed, if you wanted to do calculations based on the Minneapolis election data, you would be far better off to put it in a tidy form.

The tidy form in [Table 6.3](#) is, admittedly, not as attractive as the form published by the *Minneapolis* government. But it is much easier to use for the purpose of generating summaries and analyses.

Once data are in a tidy form, you can present them in ways that can be more effective than a formatted spreadsheet. For example, the data graphic in [Figure 6.2](#) presents the turnout within each precinct for each ward in a way that makes it easy to see how much variation there is within and among wards and precincts.

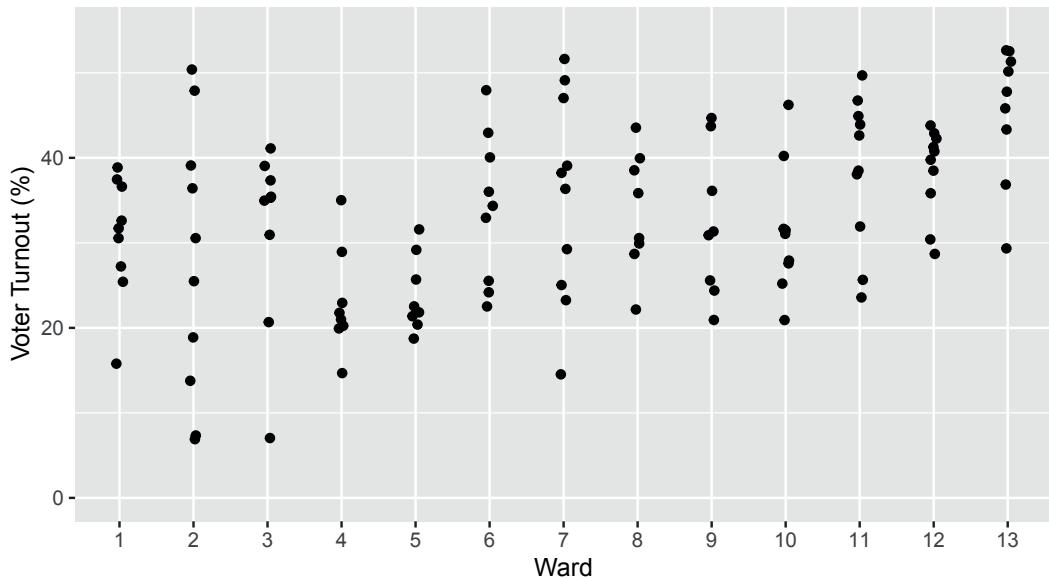


Figure 6.2: A graphical depiction of voter turnout by precinct in the different wards.

The tidy format also makes it easier to bring together data from different sources. For instance, to explain the variation in voter turnout, you might want to consider variables such as party affiliation, age, income, etc. Such data might be available on a ward-by-ward basis from other records, such as public voter registration logs and census records. Tidy data can be wrangled into forms that can be connected to one another (i.e., using the `inner_join()` function from [Chapter 5](#)). This task would be difficult if you had to deal with an idiosyncratic format for each different source of data.

6.1.2.1 Variables

In data science, the word *variable* has a different meaning than in mathematics. In *algebra*, a variable is an unknown quantity. In data, a variable is known—it has been measured. Rather, the word *variable* refers to a specific quantity or quality that can vary from case to case. There are two major types of variables:

- **Categorical variables:** record type or category and often take the form of a word.
- **Quantitative variables:** record a numerical attribute. A quantitative variable is just what it sounds like: a number.

A *categorical variable* tells you into which category or group a case falls. For instance, in the baby names data table, `sex` is a categorical variable with two levels `F` and `M`, standing for female and male. Similarly, the `name` variable is categorical. It happens that there are

Table 6.4: Individual ballots in the Minneapolis election. Each voter votes in one precinct within one ward. The ballot marks the voter’s first three choices for mayor.

Precinct	First	Second	Third	Ward
P-04	undervote	undervote	undervote	W-6
P-06	BOB FINE	MARK ANDREW	undervote	W-10
P-02D	NEAL BAXTER	BETSY HODGES	DON SAMUELS	W-7
P-01	DON SAMUELS	undervote	undervote	W-5
P-03	CAM WINTON	DON SAMUELS	OLE SAVIOR	W-1

97,310 different levels for `name`, ranging from Aaron, Ab, and Abbie to Zyhaire, Zylis, and Zymya.

6.1.2.2 Cases and what they represent

As noted previously, a row of a tidy data table refers to a case. To this point, you may have little reason to prefer the word *case* to *row*. When working with a data table, it is important to keep in mind what a case stands for in the real world. Sometimes the meaning is obvious. For instance, [Table 6.4](#) is a tidy data table showing the ballots in the Minneapolis mayoral election in 2013. Each case is an individual voter’s ballot. (The voters were directed to mark their ballot with their first choice, second choice, and third choice among the candidates. This is part of a procedure called *rank choice voting*.)

The case in [Table 6.4](#) is a different sort of thing than the case in [Table 6.3](#). In [Table 6.3](#), a case is a ward in a precinct. But in [Table 6.4](#), the case is an individual ballot. Similarly, in the baby names data ([Table 6.1](#)), a case is a name and sex and year while in [Table 6.2](#) the case is a name and sex.

When thinking about cases, ask this question: What description would make every case unique? In the vote summary data, a precinct does not uniquely identify a case. Each individual precinct appears in several rows. But each precinct and ward combination appears once and only once. Similarly, in [Table 6.1](#), name and sex do not specify a unique case. Rather, you need the combination of `name-sex-year` to identify a unique row.

6.1.2.3 Runners and races

[Table 6.5](#) displays some of the results from a 10-mile running race held each year in Washington, D.C.

What is the meaning of a case here? It is tempting to think that a case is a person. After all, it is people who run road races. But notice that individuals appear more than once: Jane Poole ran each year from 2003 to 2007. (Her times improved consistently as she got older!) Jane Schultz ran in the races from 1999 to 2006, missing only the year 2000 race. This suggests that the case is a runner in one year’s race.

6.1.2.4 Codebooks

Data tables do not necessarily display all the variables needed to figure out what makes each row unique. For such information, you sometimes need to look at the documentation of how the data were collected and what the variables mean.

The codebook is a document—separate from the data table—that describes various aspects of how the data were collected, what the variables mean and what the different levels

Table 6.5: An excerpt of runners' performance over time in a 10-mile race.

name.yob	sex	age	year	gun
jane polanek 1974	F	32	2006	114.5
jane poole 1948	F	55	2003	92.7
jane poole 1948	F	56	2004	87.3
jane poole 1948	F	57	2005	85.0
jane poole 1948	F	58	2006	80.8
jane poole 1948	F	59	2007	78.5
jane schultz 1964	F	35	1999	91.4
jane schultz 1964	F	37	2001	79.1
jane schultz 1964	F	38	2002	76.8
jane schultz 1964	F	39	2003	82.7
jane schultz 1964	F	40	2004	87.9
jane schultz 1964	F	41	2005	91.5
jane schultz 1964	F	42	2006	88.4
jane smith 1952	F	47	1999	90.6
jane smith 1952	F	49	2001	97.9

of categorical variables refer to. The word *codebook* comes from the days when data was encoded for the computer in ways that make it hard for a human to read. A codebook should include information about how the data were collected and what constitutes a case. [Figure 6.3](#) shows the codebook for the `HELPrcf` data in the `mosaicData` package. In R, codebooks for data tables in packages are available from the `help()` function.

```
help(HELPrcf)
```

For the runners data in [Table 6.5](#), a codebook should tell you that the meaning of the `gun` variable is the time from when the start gun went off to when the runner crosses the finish line and that the unit of measurement is *minutes*. It should also state what might be obvious: that `age` is the person's age in years and `sex` has two levels, male and female, represented by `M` and `F`.

6.1.2.5 Multiple tables

It is often the case that creating a meaningful display of data involves combining data from different sources and about different kinds of things. For instance, you might want your analysis of the runners' performance data in [Table 6.5](#) to include temperature and precipitation data for each year's race. Such weather data is likely contained in a table of daily weather measurements.

In many circumstances, there will be multiple tidy tables, each of which contains information relative to your analysis, but which has a different kind of thing as a case. We saw in [Chapter 5](#) how the `inner_join()` and `left_join()` functions can be used to combine multiple tables, and in [Chapter 15](#) we will further develop skills for working with relational databases. For now, keep in mind that being tidy is not about shoving everything into one table.

Percentile

Health Evaluation And Linkage To Primary Care

The HELP study was a clinical trial for adult inpatients recruited from a detoxification unit. Patients with no primary care physician were randomized to receive a multidisciplinary assessment and a brief motivational intervention or usual care, with the goal of linking them to primary medical care.

Keywords [datasets](#)

Usage

```
data(HELPrc)
```

Details

Eligible subjects were adults, who spoke Spanish or English, reported alcohol, heroin or cocaine as their first or second drug of choice, resided in proximity to the primary care clinic to which they would be referred or were homeless. Patients with established primary care relationships they planned to continue, significant dementia, specific plans to leave the Boston area that would prevent research participation, failure to provide contact information for tracking purposes, or pregnancy were excluded.

Subjects were interviewed at baseline during their detoxification stay and follow-up interviews were undertaken every 6 months for 2 years. A variety of continuous, count, discrete, and survival time predictors and outcomes were collected at each of these five occasions.

This data set is a subset of the `HELPmiss` data set restricted to the 453 subjects who were fully observed on the `age`, `cesd`, `d1`, `female`, `sex`, `g1b`, `homeless`, `i1`, `i2`, `indtot`, `mcs`, `pcs`, `pss_fr`, `racegrp`, `satreat`, `substance`, `treat`, and `sexrisk` variables. (There is some missingness in the other variables.) `HELPmiss` contains 17 additional subjects with partially observed data on some of these baseline variables. This is also a subset of the `HELPfull` data which includes 5 timepoints and many additional variables.

Note

The `\code{HELPrc}` data set was originally named `\code{HELP}` but has been renamed to avoid confusion with the `\code{help}` function.

Format

Data frame with 453 observations on the following variables.

- `age` subject age at baseline (in years)
- `anysub` use of any substance post-detox: a factor with levels `no` `yes`
- `cesd` Center for Epidemiologic Studies Depression measure at baseline (high scores indicate more depressive symptoms)
- `d1` lifetime number of hospitalizations for medical problems (measured at baseline)
- `daysanysub` time (in days) to first use of any substance post-detox
- `dayslink` time (in days) to linkage to primary care
- `drugrisk` Risk Assessment Battery drug risk scale at baseline
- `e2b` number of times in past 6 months entered a detox program (measured at baseline)

Figure 6.3: Part of the codebook for the `HELPrc` data table from the **mosaicData** package.

Table 6.6: A blood pressure data table in a wide format.

subject	before	after
BHO	160	115
GWB	120	135
WJC	105	145

Table 6.7: A tidy blood pressure data table in a narrow format.

subject	when	sbp
BHO	before	160
GWB	before	120
WJC	before	105
BHO	after	115
GWB	after	135
WJC	after	145

6.2 Reshaping data

Each row of a tidy data table is an individual case. It is often useful to re-organize the same data in a such a way that a case has a different meaning. This can make it easier to perform wrangling tasks such as comparisons, joins, and the inclusion of new data.

Consider the format of `BP_wide` shown in [Table 6.6](#), in which each case is a research study subject and there are separate variables for the measurement of *systolic blood pressure* (SBP) before and after exposure to a stressful environment. Exactly the same data can be presented in the format of the `BP_narrow` data table ([Table 6.7](#)), where the case is an individual occasion for blood pressure measurement.

Each of the formats `BP_wide` and `BP_narrow` has its advantages and its disadvantages. For example, it is easy to find the before-and-after change in blood pressure using `BP_wide`.

```
BP_wide %>%
  mutate(change = after - before)

# A tibble: 3 x 4
  subject before after change
  <chr>    <dbl> <dbl>  <dbl>
1 BHO      160    115   -45
2 GWB      120    135    15
3 WJC      105    145    40
```

On the other hand, a narrow format is more flexible for including additional variables, for example the date of the measurement or the diastolic blood pressure as in [Table 6.8](#). The narrow format also makes it feasible to add in additional measurement occasions. For instance, [Table 6.8](#) shows several “after” measurements for subject “WJC.” (Such *repeated measures* are a common feature of scientific studies.) A simple strategy allows you to get the benefits of either format: convert from wide to narrow or from narrow to wide as suits your purpose.

Table 6.8: A data table extending the information in the previous two to include additional variables and repeated measurements. The narrow format facilitates including new cases or variables.

subject	when	sbp	dbp	date
BHO	before	160	69	2007-06-19
GWB	before	120	54	1998-04-21
BHO	before	155	65	2005-11-08
WJC	after	145	75	2002-11-15
WJC	after	NA	65	2010-03-26
WJC	after	130	60	2013-09-15
GWB	after	135	NA	2009-05-08
WJC	before	105	60	1990-08-17
BHO	after	115	78	2017-06-04

6.2.1 Data verbs for converting wide to narrow and *vice versa*

Transforming a data table from wide to narrow is the action of the `pivot_longer()` data verb: A wide data table is the input and a narrow data table is the output. The reverse task, transforming from narrow to wide, involves the data verb `pivot_wider()`. Both functions are implemented in the `tidyverse` package.

6.2.2 Pivoting wider

The `pivot_wider()` function converts a data table from narrow to wide. Carrying out this operation involves specifying some information in the arguments to the function. The `values_from` argument is the name of the variable in the narrow format that is to be divided up into multiple variables in the resulting wide format. The `names_from` argument is the name of the variable in the narrow format that identifies for each case individually which column in the wide format will receive the value.

For instance, in the narrow form of `BP_narrow` (Table 6.7) the `values_from` variable is `sbp`. In the corresponding wide form, `BP_wide` (Table 6.6), the information in `sbp` will be spread between two variables: `before` and `after`. The `names_from` variable in `BP_narrow` is `when`. Note that the different categorical levels in `when` specify which variable in `BP_wide` will be the destination for the `sbp` value of each case. Only the `names_from` and `values_from` variables are involved in the transformation from narrow to wide. Other variables in the narrow table, such as `subject` in `BP_narrow`, are used to define the cases. Thus, to translate from `BP_narrow` to `BP_wide` we would write this code:

```
BP_narrow %>%
  pivot_wider(names_from = when, values_from = sbp)

# A tibble: 3 x 3
  subject before after
  <chr>    <dbl> <dbl>
1 BHO      160   115
2 GWB      120   135
3 WJC      105   145
```

6.2.3 Pivoting longer

Now consider how to transform `BP_wide` into `BP_narrow`. The names of the variables to be gathered together, `before` and `after`, will become the categorical levels in the narrow form. That is, they will make up the `names_to` variable in the narrow form. The data analyst has to invent a name for this variable. There are all sorts of sensible possibilities, for instance `before_or_after`. In gathering `BP_wide` into `BP_narrow`, we chose the concise variable name `when`.

Similarly, a name must be specified for the variable that is to hold the values in the variables being gathered. There are many reasonable possibilities. It is sensible to choose a name that reflects the kind of thing those values are, in this case systolic blood pressure. So, `sbp` is a good choice.

Finally, we need to specify which variables are to be gathered. For instance, it hardly makes sense to gather `subject` with the other variables; it will remain as a separate variable in the narrow result. Values in `subject` will be repeated as necessary to give each case in the narrow format its own correct value of `subject`. In summary, to convert `BP_wide` into `BP_narrow`, we make the following call to `pivot_longer()`.

```
BP_wide %>%
  pivot_longer(-subject, names_to = "when", values_to = "sbp")

# A tibble: 6 x 3
  subject when     sbp
  <chr>   <chr>   <dbl>
1 BHO     before   160
2 BHO     after    115
3 GWB     before   120
4 GWB     after    135
5 WJC     before   105
6 WJC     after    145
```

6.2.4 List-columns

Consider the following simple summarization of the blood pressure data. Using the techniques developed in [Section 4.1.4](#), we can compute the mean systolic blood pressure for each subject both before and after exposure.

```
BP_full %>%
  group_by(subject, when) %>%
  summarize(mean_sbp = mean(sbp, na.rm = TRUE))

# A tibble: 6 x 3
# Groups:   subject [3]
  subject when     mean_sbp
  <chr>   <chr>   <dbl>
1 BHO     after    115
2 BHO     before   158.
3 GWB     after    135
4 GWB     before   120
5 WJC     after    138.
6 WJC     before   105
```

But what if we want to do additional analysis on the blood pressure data? The individual observations are not retained in the summarized output. Can we create a summary of the data that still contains *all* of the observations?

One simplistic approach would be to use `paste()` with the `collapse` argument to condense the individual operations into a single vector.

```
BP_summary <- BP_full %>%
  group_by(subject, when) %>%
  summarize(
    sbps = paste(sbp, collapse = ", "),
    dbps = paste(dbp, collapse = ", ")
  )
```

This can be useful for seeing the data, but you can't do much computing on it, because the variables `sbps` and `dbps` are character vectors. As a result, trying to compute, say, the mean of the systolic blood pressures won't work as you hope it might. Note that the means computed below are wrong.

```
BP_summary %>%
  mutate(mean_sbp = mean(parse_number(sbps)))
```

```
# A tibble: 6 x 5
# Groups:   subject [3]
  subject when   sbps      dbps    mean_sbp
  <chr>   <chr> <chr>     <chr>    <dbl>
1 BHO     after  115       78      138.
2 BHO     before 160, 155   69, 65  138.
3 GWB     after  135       NA      128.
4 GWB     before 120       54      128.
5 WJC     after  145, NA, 130 75, 65, 60 125
6 WJC     before 105       60      125
```

Additionally, you would have to write the code to do the summarization for every variable in your data set, which could get cumbersome.

Instead, the `nest()` function will collapse *all* of the ungrouped variables in a data frame into a `tibble` (a simple data frame). This creates a new variable of type `list`, which by default has the name `data`. Each element of that list has the type `tibble`. Although you can't see all of the data in the output printed here, it's all in there. Variables in data frames that have type `list` are called *list-columns*.

```
BP_nested <- BP_full %>%
  group_by(subject, when) %>%
  nest()
BP_nested
```

```
# A tibble: 6 x 3
# Groups:   subject, when [6]
  subject when   data
  <chr>   <chr> <list>
1 BHO     before <tibble [2 x 3]>
2 GWB     before <tibble [1 x 3]>
3 WJC     after  <tibble [3 x 3]>
4 GWB     after  <tibble [1 x 3]>
```

```
5 WJC      before <tibble [1 x 3]>
6 BHO      after  <tibble [1 x 3]>
```

This construction works because a data frame is just a list of vectors of the same length, and the type of those vectors is arbitrary. Thus, the `data` variable is a vector of type `list` that consists of `tibbles`. Note also that the dimensions of each tibble (items in the `data` list) can be different.

The ability to collapse a long data frame into its nested form is particularly useful in the context of model fitting, which we illustrate in [Chapter 11](#).

While every list-column has the type `list`, the type of the data contained within that list can be anything. Thus, while the `data` variable contains a list of tibbles, we can extract only the systolic blood pressures, and put them in their own list-column. It's tempting to try to `pull()` the `sbp` variable out like this:

```
BP_nested %>%
  mutate(sbp_list = pull(data, sbp))
```

```
Error: Problem with `mutate()`' input `sbp_list`.
x no applicable method for 'pull' applied to an object of class "list"
i Input `sbp_list` is `pull(data, sbp)`.
i The error occurred in group 1: subject = "BHO", when = "after".
```

The problem is that `data` is not a `tibble`. Rather, it's a `list` of `tibbles`. To get around this, we need to use the `map()` function, which is described in [Chapter 7](#). For now, it's enough to understand that we need to apply the `pull()` function to each item in the `data` list. The `map()` function allows us to do just that, and further, it always returns a `list`, and thus creates a new list-column.

```
BP_nested <- BP_nested %>%
  mutate(sbp_list = map(data, pull, sbp))
BP_nested
```

```
# A tibble: 6 x 4
# Groups:   subject, when [6]
  subject when   data           sbp_list
  <chr>   <chr>  <list>        <list>
1 BHO     before <tibble [2 x 3]> <dbl [2]>
2 GWB     before <tibble [1 x 3]> <dbl [1]>
3 WJC     after  <tibble [3 x 3]> <dbl [3]>
4 GWB     after  <tibble [1 x 3]> <dbl [1]>
5 WJC     before <tibble [1 x 3]> <dbl [1]>
6 BHO     after  <tibble [1 x 3]> <dbl [1]>
```

Again, note that `sbp_list` is a `list`, with each item in the list being a vector of type `double`. These vectors need *not* have the same length! We can verify this by isolating the `sbp_list` variable with the `pluck()` function.

```
BP_nested %>%
  pluck("sbp_list")
```

```
[[1]]
[1] 160 155
```

```
[[2]]
```

```
[1] 120
[[3]]
[1] 145 NA 130
[[4]]
[1] 135
[[5]]
[1] 105
[[6]]
[1] 115
```

Because all of the systolic blood pressure readings are contained within this `list`, a further application of `map()` will allow us to compute the mean.

```
BP_nested <- BP_nested %>%
  mutate(sbp_mean = map(sbp_list, mean, na.rm = TRUE))
BP_nested
```

```
# A tibble: 6 x 5
# Groups:   subject, when [6]
  subject when   data      sbp_list   sbp_mean
  <chr>   <chr>  <list>    <list>     <list>
1 BHO     before <tibble [2 x 3]> <dbl [2]> <dbl [1]>
2 GWB     before <tibble [1 x 3]> <dbl [1]> <dbl [1]>
3 WJC     after  <tibble [3 x 3]> <dbl [3]> <dbl [1]>
4 GWB     after  <tibble [1 x 3]> <dbl [1]> <dbl [1]>
5 WJC     before <tibble [1 x 3]> <dbl [1]> <dbl [1]>
6 BHO     after  <tibble [1 x 3]> <dbl [1]> <dbl [1]>
```

`BP_nested` still has a nested structure. However, the column `sbp_mean` is a `list` of double vectors, each of which has a single element. We can use `unnest()` to undo the nesting structure of that column. In this case, we retain the same 6 rows, each corresponding to one subject either before or after intervention.

```
BP_nested %>%
  unnest(cols = c(sbp_mean))
```

```
# A tibble: 6 x 5
# Groups:   subject, when [6]
  subject when   data      sbp_list   sbp_mean
  <chr>   <chr>  <list>    <list>     <dbl>
1 BHO     before <tibble [2 x 3]> <dbl [2]> 158.
2 GWB     before <tibble [1 x 3]> <dbl [1]> 120
3 WJC     after  <tibble [3 x 3]> <dbl [3]> 138.
4 GWB     after  <tibble [1 x 3]> <dbl [1]> 135
5 WJC     before <tibble [1 x 3]> <dbl [1]> 105
6 BHO     after  <tibble [1 x 3]> <dbl [1]> 115
```

This computation gives the correct mean blood pressure for each subject at each time point.

On the other hand, an application of `unnest()` to the `sbp_list` variable, which has more

than one observation for each row, results in a data frame with one row for each observed subject on a specific date. This transforms the data back into the same unit of observation as `BP_full`.

```
BP_nested %>%
  unnest(cols = c(sbp_list))

# A tibble: 9 x 5
# Groups:   subject, when [6]
  subject when   data      sbp_list sbp_mean
  <chr>   <chr>  <list>      <dbl>   <list>
1 BHO     before <tibble [2 x 3]>    160 <dbl [1]>
2 BHO     before <tibble [2 x 3]>    155 <dbl [1]>
3 GWB     before <tibble [1 x 3]>    120 <dbl [1]>
4 WJC     after  <tibble [3 x 3]>    145 <dbl [1]>
5 WJC     after  <tibble [3 x 3]>     NA <dbl [1]>
6 WJC     after  <tibble [3 x 3]>    130 <dbl [1]>
7 GWB     after  <tibble [1 x 3]>    135 <dbl [1]>
8 WJC     before <tibble [1 x 3]>    105 <dbl [1]>
9 BHO     after  <tibble [1 x 3]>    115 <dbl [1]>
```

We use `nest()` or `unnest()` in [Chapters 11, 14, and 20](#).

6.2.5 Example: Gender-neutral names

In “A Boy Named Sue” country singer Johnny Cash famously told the story of a boy toughened in life—eventually reaching gratitude—by being given a traditional girl’s name. The conceit is of course the rarity of being a boy with the name `Sue`, and indeed, `Sue` is given to about 300 times as many girls as boys (at least being recorded in this manner: data entry errors may account for some of these names).

```
babynames %>%
  filter(name == "Sue") %>%
  group_by(name, sex) %>%
  summarize(total = sum(n))
```

```
# A tibble: 2 x 3
# Groups:   name [1]
  name  sex    total
  <chr> <chr> <int>
1 Sue    F      144465
2 Sue    M       519
```

On the other hand, some names that are predominantly given to girls are also commonly given to boys. Although only 15% of people named `Robin` are male, it is easy to think of a few famous men with that name: the actor Robin Williams, the singer Robin Gibb, and the basketball player Robin Lopez (not to mention *Batman*’s sidekick).

```
babynames %>%
  filter(name == "Robin") %>%
  group_by(name, sex) %>%
  summarize(total = sum(n))
```

```
# A tibble: 2 x 3
```

```
# Groups:   name [1]
#       name   sex   total
#       <chr> <chr> <int>
1 Robin F     289395
2 Robin M     44616
```

This computational paradigm (e.g., filtering) works well if you want to look at gender balance in one name at a time, but suppose you want to find the most gender-neutral names from all 97,310 names in `babynames`? For this, it would be useful to have the results in a wide format, like the one shown below.

```
babynames %>%
  filter(name %in% c("Sue", "Robin", "Leslie")) %>%
  group_by(name, sex) %>%
  summarize(total = sum(n)) %>%
  pivot_wider(
    names_from = sex,
    values_from = total
  )
```

```
# A tibble: 3 x 3
# Groups:   name [3]
#       name   F     M
#       <chr> <int> <int>
1 Leslie 266474 112689
2 Robin  289395 44616
3 Sue    144465  519
```

The `pivot_wider()` function can help us generate the wide format. Note that the `sex` variable is the `names_from` used in the conversion. A fill of zero is appropriate here: For a name like Aaban or Aadam, where there are no females, the entry for `F` should be zero.

```
baby_wide <- babynames %>%
  group_by(sex, name) %>%
  summarize(total = sum(n)) %>%
  pivot_wider(
    names_from = sex,
    values_from = total,
    values_fill = 0
  )
head(baby_wide, 3)
```

```
# A tibble: 3 x 3
#       name   F     M
#       <chr> <int> <int>
1 Aabha      35     0
2 Aabriella  32     0
3 Aada       5      0
```

One way to define “approximately the same” is to take the smaller of the ratios M/F and F/M . If females greatly outnumber males, then F/M will be large, but M/F will be small. If the sexes are about equal, then both ratios will be near one. The smaller will never be greater than one, so the most balanced names are those with the smaller of the ratios near one.

The code to identify the most balanced gender-neutral names out of the names with more than 50,000 babies of each sex is shown below. Remember, a ratio of 1 means exactly balanced; a ratio of 0.5 means two to one in favor of one sex; 0.33 means three to one. (The `pmin()` transformation function returns the smaller of the two arguments for each individual case.)

```
baby_wide %>%
  filter(M > 50000, F > 50000) %>%
  mutate(ratio = pmin(M / F, F / M)) %>%
  arrange(desc(ratio)) %>%
  head(3)

# A tibble: 3 x 4
  name      F      M ratio
  <chr>   <int> <int> <dbl>
1 Riley    100881  92789 0.920
2 Jackie   90604   78405 0.865
3 Casey    76020   110165 0.690
```

Riley has been the most gender-balanced name, followed by Jackie. Where does your name fall on this list?

6.3 Naming conventions

Like any language, **R** has some rules that you cannot break, but also many conventions that you can—but should not—break. There are a few simple rules that apply when creating a *name* for an object:

- The name cannot start with a digit. So you cannot assign the name `100NCHS` to a data frame, but `NCHS100` is fine. This rule is to make it easy for **R** to distinguish between object names and numbers. It also helps you avoid mistakes such as writing `2pi` when you mean `2*pi`.
- The name cannot contain any punctuation symbols other than `.` and `_`. So `?NCHS` or `N*Hanes` are not legitimate names. However, you can use `.` and `_` in a name. For reasons that will be explained later, the use of `.` in function names has a specific meaning, but should otherwise be avoided. The use of `_` is preferred.
- The case of the letters in the name matters. So `NCHS`, `nchs`, `Nchs`, and `nChs`, etc., are all different names that only look similar to a human reader, not to **R**.

Pro Tip 19. *Do not use `.` in function names, to avoid conflicting with internal functions.*

One of **R**'s strengths is its modularity—many people have contributed many packages that do many different things. However, this decentralized paradigm has resulted in many *different* people writing code using many *different* conventions. The resulting lack of uniformity can make code harder to read. We suggest adopting a style guide and sticking to it—we have attempted to do that in this book. However, the inescapable use of other people's code results in inevitable deviations from that style.

In this book and in our teaching, we follow the tidyverse style guide—which is public, widely adopted, and influential—as closely as possible. It provides guidance about how and why

to adopt a particular style. Other groups (e.g., Google) have adopted variants of this guide. This means:

- We use underscores (_) in variable and function names. The use of periods (.) in function names is restricted to S3 methods.
- We use spaces liberally and prefer multiline, narrow blocks of code to single lines of wide code (although we occasionally relax this to save space on the printed page).
- We use *snake_case* for the names of things. This means that each “word” is lowercase, and there are no spaces, only underscores. (The **janitor** package provides a function called `clean_names()` that by default turns variable names into snake case (other styles are also supported.)

The **styler** package can be used to reformat code into a format that implements the tidyverse style guide.

Pro Tip 20. *Faithfully adopting a consistent style for code can help to improve readability and reduce errors.*

6.4 Data intake

“Every easy data format is alike. Every difficult data format is difficult in its own way.”

—inspired by Leo Tolstoy and Hadley Wickham

The tools that we develop in this book allow one to work with data in **R**. However, most data sets are not available in **R** to begin with—they are often stored in a different file format. While **R** has sophisticated abilities for reading data in a variety of formats, it is not without limits. For data that are not in a file, one common form of data intake is *Web scraping*, in which data from the internet are processed as (structured) text and converted into data. Such data often have errors that stem from blunders in data entry or from deficiencies in the way data are stored or coded. Correcting such errors is called *data cleaning*.

The native file format for **R** is usually given the suffix `.rda` (or sometimes, `.RData`). Any object in your **R** environment can be written to this file format using the `saveRDS()` command. Using the `compress` argument will make these files smaller.

```
saveRDS(mtcars, file = "mtcars.rda", compress = TRUE)
```

This file format is usually an efficient means for storing data, but it is not the most portable. To load a stored object into your **R** environment, use the `readRDS()` command.

```
mtcars <- readRDS("mtcars.rda")
```

Pro Tip 21. *Maintaining the provenance of data from beginning to the end of an analysis*

is an important part of a reproducible workflow. This can be facilitated by creating one Markdown file or notebook that undertakes the data wrangling and generates an analytic data set (using `saveRDS()`) that can be read (using `readRDS()`) into a second Markdown file.

6.4.1 Data-table friendly formats

Many formats for data are essentially equivalent to data tables. When you come across data in a format that you don't recognize, it is worth checking whether it is one of the data-table-friendly formats. Sometimes the *filename extension* provides an indication. Here are several, each with a brief description:

- **CSV:** a non-proprietary comma-separated text format that is widely used for data exchange between different software packages. *CSVs* are easy to understand, but are not compressed, and therefore can take up more space on disk than other formats.
- **Software-package specific format:** some common examples include:
 - *Octave* (and through that, *MATLAB*): widely used in engineering and physics
 - *Stata*: commonly used for economic research
 - *SPSS*: commonly used for social science research
 - *Minitab*: often used in business applications
 - *SAS*: often used for large data sets
 - *Epi*: used by the *Centers for Disease Control* (CDC) for health and epidemiology data
- **Relational databases:** the form that much of institutional, actively-updated data are stored in. This includes business transaction records, government records, Web logs, and so on. (See [Chapter 15](#) for a discussion of relational database management systems.)
- **Excel:** a set of proprietary spreadsheet formats heavily used in business. Watch out, though. Just because something is stored in an *Excel format* doesn't mean it is a data table. Excel is sometimes used as a kind of tablecloth for writing down data with no particular scheme in mind.
- **Web-related:** For example:
 - *HTML* (hypertext markup language): `<table>` format
 - *XML* (extensible markup language) format, a tree-based document structure
 - *JSON* (JavaScript Object Notation) is a common data format that breaks the “rows-and-columns” paradigm (see [Section 21.2.4.2](#))
 - *Google Sheets*: published as HTML
 - *application programming interface* (API)

The procedure for reading data in one of these formats varies depending on the format. For Excel or Google Sheets data, it is sometimes easiest to use the application software to export the data as a CSV file. There are also **R** packages for reading directly from either (`readxl` and `googlesheets4`, respectively), which are useful if the spreadsheet is being updated frequently. For the technical software package formats, the `haven` package provides useful reading and writing functions. For relational databases, even if they are on a remote server, there are several useful **R** packages that allow you to connect to these databases directly, most notably `dbplyr` and `DBI`. CSV and HTML `<table>` formats are frequently encountered sources for data scraping, and can be read by the `readr` and `rvest` packages, respectively. The next subsections give a bit more detail about how to read them into **R**.

6.4.1.1 CSV (comma separated value) files

This text format can be read with a huge variety of software. It has a data table format, with the values of variables in each case separated by commas. Here is an example of the first several lines of a CSV file:

```
"year","sex","name","n","prop"
1880,"F","Mary",7065,0.07238359
1880,"F","Anna",2604,0.02667896
1880,"F","Emma",2003,0.02052149
1880,"F","Elizabeth",1939,0.01986579
1880,"F","Minnie",1746,0.01788843
1880,"F","Margaret",1578,0.0161672
```

The top row usually (but not always) contains the variable names. Quotation marks are often used at the start and end of character strings—these quotation marks are not part of the content of the string, but are useful if, say, you want to include a comma in the text of a field. CSV files are often named with the `.csv` suffix; it is also common for them to be named with `.txt`, `.dat`, or other things. You will also see characters other than commas being used to delimit the fields: tabs and vertical bars (or pipes, i.e., `|`) are particularly common.

Pro Tip 22. Be careful with date and time variables in CSV format: these can sometimes be formatted in inconsistent ways that make it more challenging to ingest.

Since reading from a CSV file is so common, several implementations are available. The `read.csv()` function in the `base` package is perhaps the most widely used, but the more recent `read_csv()` function in the `readr` package is noticeably faster for large CSVs. CSV files need not exist on your local hard drive. For example, here is a way to access a `.csv` file over the internet using a URL (*universal resource locator*).

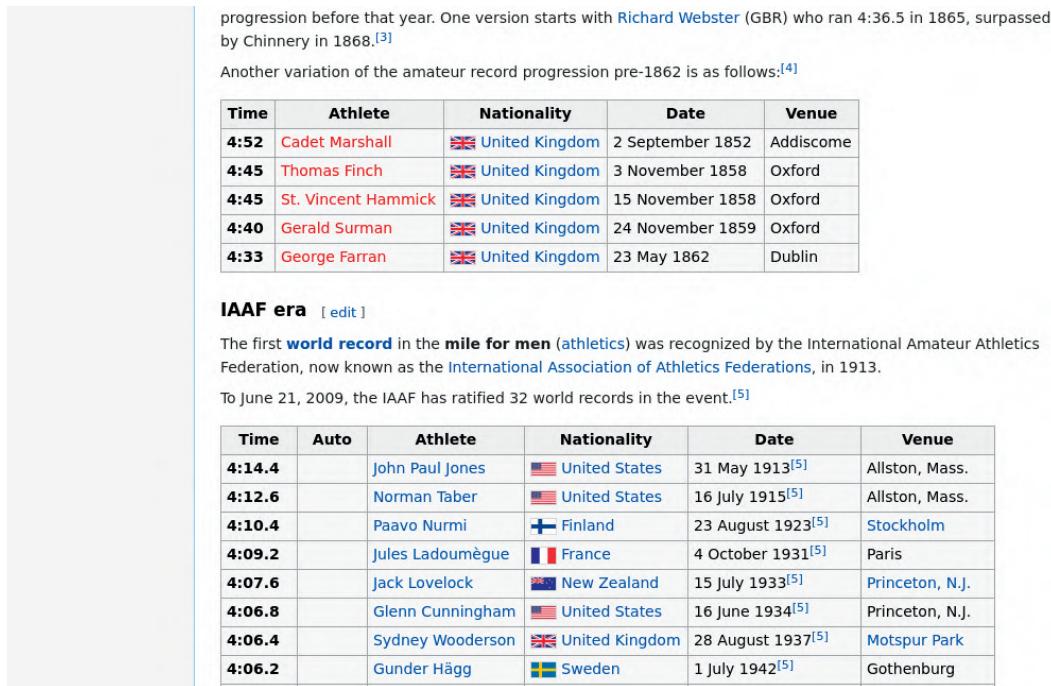
```
mdsr_url <- "https://raw.githubusercontent.com/mdsr-book/mdsr/master/data-raw/"
houses <- mdsr_url %>%
  paste0("houses-for-sale.csv") %>%
  read_csv()
head(houses, 3)
```

```
# A tibble: 3 x 16
  price lot_size waterfront age land_value construction air_cond fuel
  <dbl>    <dbl>      <dbl> <dbl>     <dbl>        <dbl>    <dbl> <dbl>
1 132500     0.09       0    42     50000         0       0     3
2 181115     0.92       0     0     22300         0       0     2
3 109000     0.19       0   133     7300         0       0     2
# ... with 8 more variables: heat <dbl>, sewer <dbl>, living_area <dbl>,
#   pct_college <dbl>, bedrooms <dbl>, fireplaces <dbl>, bathrooms <dbl>,
#   rooms <dbl>
```

Just as reading a data file from the internet uses a URL, reading a file on your computer uses a complete name, called a *path* to the file. Although many people are used to using a mouse-based selector to access their files, being specific about the path to your files is important to ensure the reproducibility of your code (see [Appendix D](#)).

6.4.1.2 HTML tables

Web pages are HTML documents, which are then translated by a browser to the formatted content that users see. HTML includes facilities for presenting tabular content. The HTML <table> markup is often the way human-readable data is arranged.



The screenshot shows a portion of a Wikipedia page about mile-run world records. At the top, there is a note about the progression before 1862. Below it, another note provides a variation of the amateur record progression pre-1862. Two tables are displayed side-by-side. The first table, titled 'IAAF era [edit]', lists records from 1913 to 1942. The second table, also titled 'IAAF era [edit]', lists records from 1913 to 1942. Both tables have columns for Time, Auto, Athlete, Nationality, Date, and Venue.

Time	Auto	Athlete	Nationality	Date	Venue
4:14.4		John Paul Jones	United States	31 May 1913 ^[5]	Allston, Mass.
4:12.6		Norman Taber	United States	16 July 1915 ^[5]	Allston, Mass.
4:10.4		Paavo Nurmi	Finland	23 August 1923 ^[5]	Stockholm
4:09.2		Jules Ladoumègue	France	4 October 1931 ^[5]	Paris
4:07.6		Jack Lovelock	New Zealand	15 July 1933 ^[5]	Princeton, N.J.
4:06.8		Glenn Cunningham	United States	16 June 1934 ^[5]	Princeton, N.J.
4:06.4		Sydney Wooderson	United Kingdom	28 August 1937 ^[5]	Motspur Park
4:06.2		Gunder Hägg	Sweden	1 July 1942 ^[5]	Gothenburg

Figure 6.4: Part of a page on mile-run world records from Wikipedia. Two separate data tables are visible. You can't tell from this small part of the page, but there are many tables on the page. These two tables are the third and fourth in the page.

When you have the URL of a page containing one or more tables, it is sometimes easy to read them into **R** as data tables. Since they are not CSVs, we can't use `read_csv()`. Instead, we use functionality in the **rvest** package to ingest the HTML as a data structure in **R**. Once you have the content of the Web page, you can translate any tables in the page from HTML to data table format.

In this brief example, we will investigate the progression of the world record time in the mile run, as detailed on Wikipedia. This page (see [Figure 6.4](#)) contains several tables, each of which contains a list of new world records for a different class of athlete (e.g., men, women, amateur, professional, etc.).

```
library(rvest)
url <- "http://en.wikipedia.org/wiki/Mile_run_world_record_progression"
tables <- url %>%
  read_html() %>%
  html_nodes("table")
```

The result, `tables`, is not a data table. Instead, it is a `list` (see [Appendix B](#)) of the tables found in the Web page. Use `length()` to find how many items there are in the list of tables.

Table 6.9: The third table embedded in the Wikipedia page on running records.

Time	Athlete	Nationality	Date	Venue
4:52	Cadet Marshall	United Kingdom	2 September 1852	Addiscombe
4:45	Thomas Finch	United Kingdom	3 November 1858	Oxford
4:45	St. Vincent Hammick	United Kingdom	15 November 1858	Oxford
4:40	Gerald Surman	United Kingdom	24 November 1859	Oxford
4:33	George Farran	United Kingdom	23 May 1862	Dublin

Table 6.10: The fourth table embedded in the Wikipedia page on running records.

Time	Athlete	Nationality	Date	Venue
4:14.4	John Paul Jones	United States	31 May 1913[6]	Allston, Mass.
4:12.6	Norman Taber	United States	16 July 1915[6]	Allston, Mass.
4:10.4	Paavo Nurmi	Finland	23 August 1923[6]	Stockholm
4:09.2	Jules Ladoumègue	France	4 October 1931[6]	Paris
4:07.6	Jack Lovelock	New Zealand	15 July 1933[6]	Princeton, N.J.
4:06.8	Glenn Cunningham	United States	16 June 1934[6]	Princeton, N.J.

```
length(tables)
```

```
[1] 12
```

You can access any of those tables using the `pluck()` function from the **purrr** package, which extracts items from a `list`. Unfortunately, as of this writing the `rvest::pluck()` function masks the more useful `purrr::pluck()` function, so we will be specific by using the double-colon operator. The first table is `pluck(tables, 1)`, the second table is `pluck(tables, 2)`, and so on. The third table—which corresponds to amateur men up until 1862—is shown in [Table 6.9](#).

```
amateur <- tables %>%
  purrr::pluck(3) %>%
  html_table()
```

Likely of greater interest is the information in the fourth table, which corresponds to the current era of *International Amateur Athletics Federation* world records. The first few rows of that table are shown in [Table 6.10](#). The last row of that table (not shown) contains the current world record of 3:43.13, which was set by Hicham El Guerrouj of *Morocco* in *Rome* on July 7th, 1999.

```
records <- tables %>%
  purrr::pluck(4) %>%
  html_table() %>%
  select(-Auto) # remove unwanted column
```

6.4.2 APIs

An *application programming interface* (API) is a protocol for interacting with a computer program that you can't control. It is a set of agreed-upon instructions for using a “*black-box*”—not unlike the manual for a television's remote control. APIs provide access to massive

Table 6.11: Four of the variables from the tables giving features of the Saratoga houses stored as integer codes. Each case is a different house.

fuel	heat	sewer	construction
3	4	2	0
2	3	2	0
2	3	3	0
2	2	2	0
2	2	3	1

troves of public data on the Web, from a vast array of different sources. Not all APIs are the same, but by learning how to use them, you can dramatically increase your ability to pull data into **R** without having to manually “scrape” it.

If you want to obtain data from a public source, it is a good idea to check to see whether: a) the organization has a public API; b) someone has already written an **R** package to said interface. These packages don’t provide the actual data—they simply provide a series of **R** functions that allow you to access the actual data. The documentation for each package should explain how to use it to collect data from the original source.

6.4.3 Cleaning data

A person somewhat knowledgeable about running would have little trouble interpreting Tables 6.9 and 6.10 correctly. The `time` is in minutes and seconds. The `Date` gives the day on which the record was set. When the data table is read into **R**, both `time` and `Date` are stored as character strings. Before they can be used, they have to be converted into a format that the computer can process like a date and time. Among other things, this requires dealing with the footnote (listed as [5]) at the end of the date information.

Data cleaning refers to taking the information contained in a variable and transforming it to a form in which that information can be used.

6.4.3.1 Recoding

Table 6.11 displays a few variables from the `houses` data table we downloaded earlier. It describes 1,728 houses for sale in *Saratoga, NY*.¹ The full table includes additional variables such as `living_area`, `price`, `bedrooms`, and `bathrooms`. The data on house systems such as `sewer_type` and `heat_type` have been stored as numbers, even though they are really categorical.

There is nothing fundamentally wrong with using integers to encode, say, fuel type, though it may be confusing to interpret results. What is worse is that the numbers imply a meaningful order to the categories when there is none.

To translate the integers to a more informative coding, you first have to find out what the various codes mean. Often, this information comes from the codebook, but sometimes you will need to contact the person who collected the data. Once you know the translation, you can use spreadsheet software (or the `tribble()` function) to enter them into a data table, like this one for the houses:

¹The example comes from Richard de Veaux at *Williams College*.

Table 6.12: The Translations data table rendered in a wide format.

	code	new_const	sewer_type	central_air	fuel_type	heat_type
0	no		invalid	no	invalid	invalid
1	yes		none	yes	invalid	invalid
2	invalid		private	invalid	gas	hot air
3	invalid		public	invalid	electric	hot water
4	invalid		invalid	invalid	oil	electric

```
translations <- mdsr_url %>%
  paste0("house_codes.csv") %>%
  read_csv()
translations %>% head(5)
```

```
# A tibble: 5 x 3
  code system_type meaning
  <dbl> <chr>      <chr>
1     0 new_const   no
2     1 new_const   yes
3     1 sewer_type  none
4     2 sewer_type  private
5     3 sewer_type  public
```

Translations describes the codes in a format that makes it easy to add new code values as the need arises. The same information can also be presented a wide format as in [Table 6.12](#).

```
codes <- translations %>%
  pivot_wider(
    names_from = system_type,
    values_from = meaning,
    values_fill = "invalid"
  )
```

In codes, there is a column for each system type that translates the integer code to a meaningful term. In cases where the integer has no corresponding term, invalid has been entered. This provides a quick way to distinguish between incorrect entries and missing entries. To carry out the translation, we join each variable, one at a time, to the data table of interest. Note how the by value changes for each variable:

```
houses <- houses %>%
  left_join(
    codes %>% select(code, fuel_type),
    by = c(fuel = "code")
  ) %>%
  left_join(
    codes %>% select(code, heat_type),
    by = c(heat = "code")
  ) %>%
  left_join(
    codes %>% select(code, sewer_type),
```

Table 6.13: The Saratoga houses data with re-coded categorical variables.

fuel_type	heat_type	sewer_type
electric	electric	private
gas	hot water	private
gas	hot water	public
gas	hot air	private
gas	hot air	public
gas	hot air	private

```
  by = c(sewer = "code")
)
```

Table 6.13 shows the re-coded data. We can compare this to the previous display in Table 6.11.

6.4.3.2 From strings to numbers

You have seen two major types of variables: quantitative and categorical. You are used to using quoted character strings as the levels of categorical variables, and numbers for quantitative variables.

Often, you will encounter data tables that have variables whose meaning is numeric but whose representation is a character string. This can occur when one or more cases is given a non-numeric value, e.g., *not available*.

The `parse_number()` function will translate character strings with numerical content into numbers. The `parse_character()` function goes the other way. For example, in the `ordway_birds` data, the `Month`, `Day`, and `Year` variables are all being stored as character vectors, even though their evident meaning is numeric.

```
ordway_birds %>%
  select(Timestamp, Year, Month, Day) %>%
  glimpse()
```

```
Rows: 15,829
Columns: 4
$ Timestamp <chr> "4/14/2010 13:20:56", "", "5/13/2010 16:00:30", "5/13...
$ Year      <chr> "1972", "", "1972", "1972", "1972", "1972", "1972", ...
$ Month     <chr> "7", "", "7", "7", "7", "7", "7", "7", "7", ...
$ Day       <chr> "16", "", "16", "16", "16", "16", "16", "16", "16", ...
```

We can convert the strings to numbers using `mutate()` and `parse_number()`. Note how the empty strings (i.e., "") in those fields are automatically converted into `NA`'s, since they cannot be converted into valid numbers.

```
library(readr)
ordway_birds <- ordway_birds %>%
  mutate(
    Month = parse_number(Month),
    Year = parse_number(Year),
    Day = parse_number(Day)
  )
```

```
ordway_birds %>%
  select(Timestamp, Year, Month, Day) %>%
  glimpse()

Rows: 15,829
Columns: 4
$ Timestamp <chr> "4/14/2010 13:20:56", "", "5/13/2010 16:00:30", "5/13...
$ Year      <dbl> 1972, NA, 1972, 1972, 1972, 1972, 1972, 1972, 1...
$ Month     <dbl> 7, NA, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7...
$ Day       <dbl> 16, NA, 16, 16, 16, 16, 16, 16, 16, 16, 16, 17, 18, 18, 1...
```

6.4.3.3 Dates

Dates are often recorded as character strings (e.g., 29 October 2014). Among other important properties, dates have a natural order. When you plot values such as 16 December 2015 and 29 October 2016, you expect the December date to come after the October date, even though this is not true alphabetically of the string itself.

When plotting a value that is numeric, you expect the axis to be marked with a few round numbers. A plot from 0 to 100 might have ticks at 0, 20, 40, 60, 100. It is similar for dates. When you are plotting dates within one month, you expect the day of the month to be shown on the axis. If you are plotting a range of several years, it would be appropriate to show only the years on the axis.

When you are given dates stored as a character vector, it is usually necessary to convert them to a data type designed specifically for dates. For instance, in the `ordway_birds` data, the `Timestamp` variable refers to the time the data were transcribed from the original lab notebook to the computer file. This variable is currently stored as a `character` string, but we can translate it into a more usable date format using functions from the **`lubridate`** package.

These dates are written in a format showing `month/day/year hour:minute:second`. The `mdy_hms()` function from the **`lubridate`** package converts strings in this format to a date. Note that the data type of the `When` variable is now `dttm`.

```
library(lubridate)
birds <- ordway_birds %>%
  mutate(When = mdy_hms(Timestamp)) %>%
  select(Timestamp, Year, Month, Day, When, DataEntryPerson)
birds %>%
  glimpse()
```

```
Rows: 15,829
Columns: 6
$ Timestamp      <chr> "4/14/2010 13:20:56", "", "5/13/2010 16:00:30",...
$ Year          <dbl> 1972, NA, 1972, 1972, 1972, 1972, 1972, 1...
$ Month         <dbl> 7, NA, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7...
$ Day           <dbl> 16, NA, 16, 16, 16, 16, 16, 16, 16, 16, 16, 17, 18, ...
$ When          <dttm> 2010-04-14 13:20:56, NA, 2010-05-13 16:00:30, ...
$ DataEntryPerson <chr> "Jerald Dosch", "Caitlin Baker", "Caitlin Baker..."
```

With the `When` variable now recorded as a timestamp, we can create a sensible plot showing when each of the transcribers completed their work, as in [Figure 6.5](#).

```
birds %>%
  ggplot(aes(x = When, y = DataEntryPerson)) +
  geom_point(alpha = 0.1, position = "jitter")
```

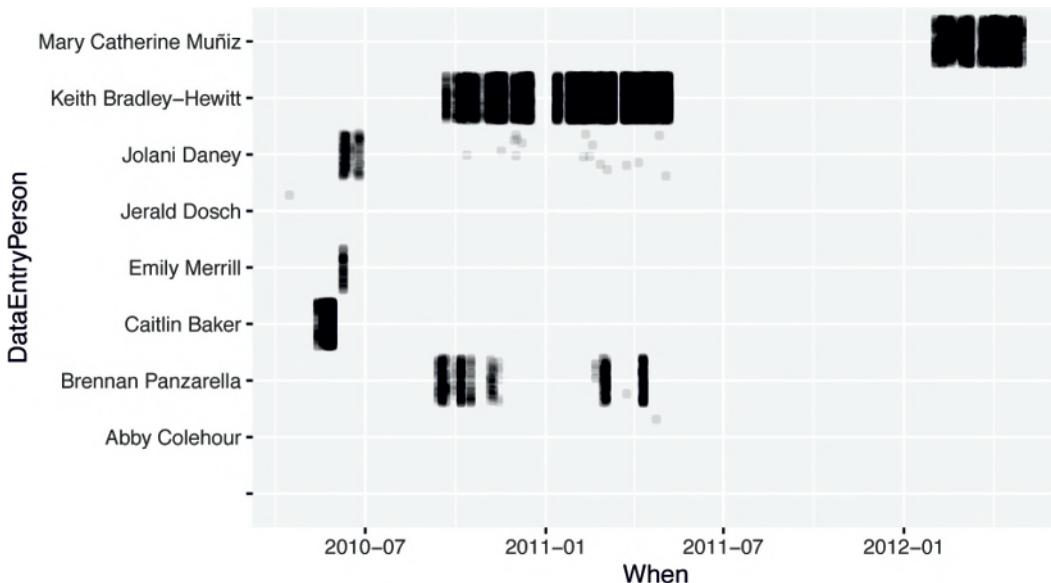


Figure 6.5: The transcribers of the Ordway Birds from lab notebooks worked during different time intervals.

Many of the same operations that apply to numbers can be used on dates. For example, the range of dates that each transcriber worked can be calculated as a difference in times (i.e., an `interval()`), and shown in Table 6.14. This makes it clear that Jolani worked on the project for nearly a year (329 days), while Abby's first transcription was also her last.

```
bird_summary <- birds %>%
  group_by(DataEntryPerson) %>%
  summarize(
    start = first(When),
    finish = last(When)
  ) %>%
  mutate(duration = interval(start, finish) / ddays(1))
```

There are many similar **lubridate** functions for converting strings in different formats into dates, e.g., `ymd()`, `dmy()`, and so on. There are also functions like `hour()`, `yday()`, etc. for extracting certain pieces of variables encoded as dates.

Internally, **R** uses several different classes to represent dates and times. For timestamps (also referred to as *datetimes*), these classes are `POSIXct` and `POSIXlt`. For most purposes, you can treat these as being the same, but internally, they are stored differently. A `POSIXct` object is stored as the number of seconds since the *UNIX epoch* (1970-01-01), whereas `POSIXlt` objects are stored as a list of year, month, day, etc., character strings.

```
now()
```

```
[1] "2021-01-09 16:26:24 EST"
```

Table 6.14: Starting and ending dates for each transcriber involved in the Ordway Birds project.

DataEntryPerson	start	finish	duration
Abby Colehour	2011-04-23 15:50:24	2011-04-23 15:50:24	0.000
Brennan Panzarella	2010-09-13 10:48:12	2011-04-10 21:58:56	209.466
Emily Merrill	2010-06-08 09:10:01	2010-06-08 14:47:21	0.234
Jerald Dosch	2010-04-14 13:20:56	2010-04-14 13:20:56	0.000
Jolani Daney	2010-06-08 09:03:00	2011-05-03 10:12:59	329.049
Keith Bradley-Hewitt	2010-09-21 11:31:02	2011-05-06 17:36:38	227.254
Mary Catherine Muñiz	2012-02-02 08:57:37	2012-04-30 14:06:27	88.214

```
class(now())
[1] "POSIXct" "POSIXt"
class(as.POSIXlt(now()))
[1] "POSIXlt" "POSIXt"
```

For dates that do not include times, the `Date` class is most commonly used.

```
as.Date(now())
[1] "2021-01-09"
```

6.4.3.4 Factors or strings?

A *factor* is a special data type used to represent categorical data. Factors store categorical data efficiently and provide a means to put the categorical levels in whatever order is desired. Unfortunately, factors also make cleaning data more confusing. The problem is that it is easy to mistake a factor for a character string, and they have different properties when it comes to converting a numeric or date form. This is especially problematic when using the character processing techniques in [Chapter 19](#).

By default, `readr::read_csv()` will interpret character strings as strings and not as factors. Other functions, such as `read.csv()` prior to version 4.0 of **R**, convert character strings into factors by default. Cleaning such data often requires converting them back to a character format using `parse_character()`. Failing to do this when needed can result in completely erroneous results without any warning. The **forcats** package was written to improve support for wrangling factor variables.

For this reason, the data tables used in this book have been stored with categorical or text data in character format. Be aware that data provided by other packages do not necessarily follow this convention. If you get mysterious results when working with such data, consider the possibility that you are working with factors rather than character vectors. Recall that `summary()`, `glimpse()`, and `str()` will all reveal the data types of each variable in a data frame.

Pro Tip 23. *It's always a good idea to carefully check all variables and data wrangling operations to ensure that correct values are generated. Such data auditing and the use of automated data consistency checking can decrease the likelihood of data integrity errors.*

6.4.4 Example: Japanese nuclear reactors

Dates and times are an important aspect of many analyses. In the example below, the vector `example` contains human-readable datetimes stored as character by **R**. The `ymd_hms()` function from **lubridate** will convert this into `POSIXct`—a datetime format. This makes it possible for **R** to do date arithmetic.

```
library(lubridate)
example <- c("2021-04-29 06:00:00", "2021-12-31 12:00:00")
str(example)
```

```
chr [1:2] "2021-04-29 06:00:00" "2021-12-31 12:00:00"
```

```
converted <- ymd_hms(example)
str(converted)
```

```
POSIXct[1:2], format: "2021-04-29 06:00:00" "2021-12-31 12:00:00"
```

```
converted
```

```
[1] "2021-04-29 06:00:00 UTC" "2021-12-31 12:00:00 UTC"
```

```
converted[2] - converted[1]
```

Time difference of 246 days

We will use this functionality to analyze data on nuclear reactors in Japan. [Figure 6.6](#) displays the first part of this table as of the summer of 2016.

Japan [edit]

See also: [Nuclear power in Japan](#)

Power station reactors [edit]

Name	Reactor No.	Reactor		Status	Capacity in MW		Construction Start Date	Commercial Operation Date	Closure
		Type	Model		Net	Gross			
Fukushima Daiichi	1	BWR	BWR-3	Inoperable	439	460	25 July 1967	26 March 1971	19 May 2011
Fukushima Daiichi	2	BWR	BWR-4	Inoperable	760	784	9 June 1969	18 July 1974	19 May 2011
Fukushima Daiichi	3	BWR	BWR-4	Inoperable	760	784	28 December 1970	27 March 1976	19 May 2011
Fukushima Daiichi	4	BWR	BWR-4	Shut down/Inoperable	760	784	12 February 1973	12 October 1978	19 May 2011
Fukushima Daiichi	5	BWR	BWR-4	Shut down	760	784	22 May 1972	18 April 1978	17 December 2013
Fukushima Daiichi	6	BWR	BWR-5	Shut down	1067	1100	26 October 1973	24 October 1979	17 December 2013
Fukushima Daini	1	BWR	BWR-5	Operation suspended	1067	1100	16 March 1976	20 April 1982	

Figure 6.6: Screenshot of Wikipedia’s list of Japanese nuclear reactors.

```
tables <- "http://en.wikipedia.org/wiki/List_of_nuclear_reactors" %>%
  read_html() %>%
  html_nodes(css = "table")

idx <- tables %>%
  html_text() %>%
  str_detect("Fukushima Daiichi") %>%
  which()
```

We see that among the first entries are the ill-fated *Fukushima Daiichi* reactors. The `mutate()` function can be used in conjunction with the `dmy()` function from the **lubridate** package to wrangle these data into a better form.

```
reactors <- reactors %>%
  mutate(
    plant_status = ifelse(
      str_detect(status, "Shut down"),
      "Shut down", "Not formally shut down"
    ),
    capacity_net = parse_number(capacity_net),
    construct_date = dmy(construction_start),
    operation_date = dmy(commercial_operation),
    closure_date = dmy(closure)
  )
glimpse(reactors)
```

```
$ capacity_net      <dbl> 148, 439, 760, 760, 760, 1067, NA, 10...
$ capacity_gross    <chr> "165", "460", "784", "784", "784", ...
$ construction_start <chr> "10 May 1972", "25 July 1967", "9 June 196...
$ commercial_operation <chr> "20 March 1979", "26 March 1971", "18 July...
$ closure           <chr> "29 March 2003", "19 May 2011", "19 May 20...
$ plant_status       <chr> "Shut down", "Not formally shut down", "No...
$ construct_date     <date> 1972-05-10, 1967-07-25, 1969-06-09, 1970-...
$ operation_date     <date> 1979-03-20, 1971-03-26, 1974-07-18, 1976-...
$ closure_date       <date> 2003-03-29, 2011-05-19, 2011-05-19, 2011-...
```

How have these plants evolved over time? It seems likely that as nuclear technology has progressed, plants should see an increase in capacity. A number of these reactors have been shut down in recent years. Are there changes in capacity related to the age of the plant? [Figure 6.7](#) displays the data.

```
ggplot(
  data = reactors,
  aes(x = construct_date, y = capacity_net, color = plant_status
  )
) +
  geom_point() +
  geom_smooth() +
  xlab("Date of Plant Construction") +
  ylab("Net Plant Capacity (MW)")
```

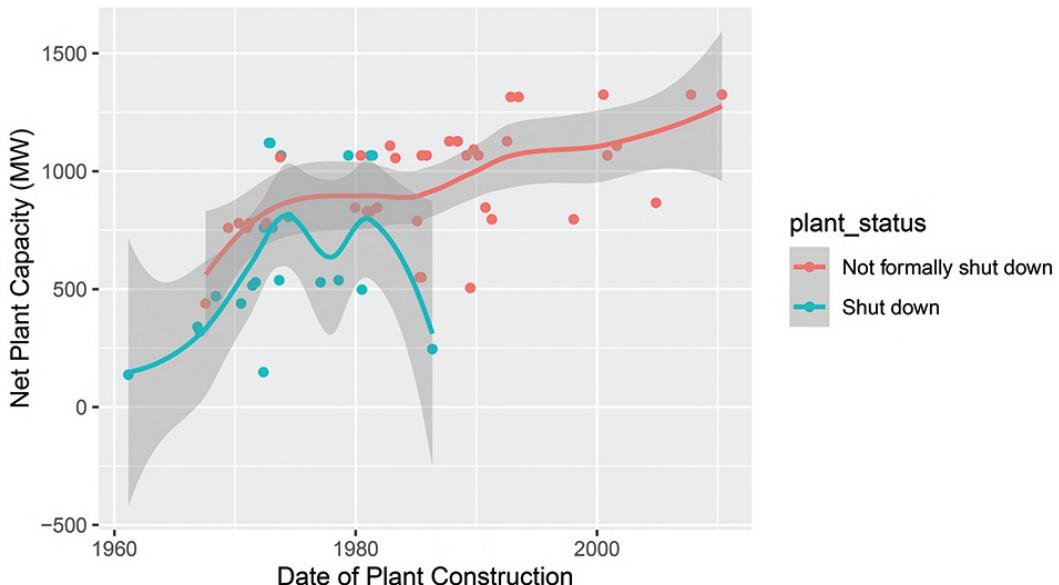


Figure 6.7: Distribution of capacity of Japanese nuclear power plants over time.

Indeed, reactor capacity has tended to increase over time, while the older reactors were more likely to have been formally shut down. While it would have been straightforward to code these data by hand, automating data ingestion for larger and more complex tables is more efficient and less error-prone.

6.5 Further resources

The tidyverse style guide (<https://style.tidyverse.org>) merits a close read by all **R** users. Broman and Woo (2018) describe helpful tips for data organization in spreadsheets. The **tidyr** package, and in particular, Wickham (2020f) provide principles for tidy data. The corresponding paper on tidy data (Wickham, 2014) builds upon notions of normal forms—common to database designers from computer science—to describe a process of thinking about how data should be stored and formatted.

There are many **R** packages that do nothing other than provide access to a public API from within **R**. There are far too many API packages to list here, but a fair number of them are maintained by the rOpenSci group. In fact, several of the packages referenced in this book, including the **twitteR** and **aRxiv** packages in Chapter 19, and the **plotly** package in Chapter 14, are APIs. The CRAN task view on Web Technologies lists hundreds more packages, including **Rfacebook**, **instaR**, **FlickrAPI**, **tumblR**, and **Rlinkedin**. The **RSocrata** package facilitates the use of *Socrata*, which is itself an API for querying—among other things—the NYC Open Data platform.

6.6 Exercises

Problem 1 (Easy): In the `Marriage` data set included in `mosaic`, the `appdate`, `ceremonydate`, and `dob` variables are encoded as factors, even though they are dates. Use `lubridate` to convert those three columns into a date format.

```
library(mosaic)
Marriage %>%
  select(appdate, ceremonydate, dob) %>%
  glimpse(width = 50)
```

```
Rows: 98
Columns: 3
$ appdate      <date> 1996-10-29, 1996-11-12, ...
$ ceremonydate <date> 1996-11-09, 1996-11-12, ...
$ dob          <date> 2064-04-11, 2064-08-06, ...
```

Problem 2 (Easy): Consider the following pipeline:

```
library(tidyverse)
mtcars %>%
  filter(cyl == 4) %>%
  select(mpg, cyl)
```

	mpg	cyl
Datsun 710	22.8	4
Merc 240D	24.4	4
Merc 230	22.8	4
Fiat 128	32.4	4
Honda Civic	30.4	4

Toyota Corolla	33.9	4
Toyota Corona	21.5	4
Fiat X1-9	27.3	4
Porsche 914-2	26.0	4
Lotus Europa	30.4	4
Volvo 142E	21.4	4

Rewrite this in nested form on a single line. Which set of commands do you prefer and why?

Problem 3 (Easy): Consider the values returned by the `as.numeric()` and `parse_number()` functions when applied to the following vectors. Describe the results and their implication.

```
x1 <- c("1900.45", "$1900.45", "1,900.45", "nearly $2000")
x2 <- as.factor(x1)
```

Problem 4 (Medium): Find an interesting Wikipedia page with a table, scrape the data from it, and generate a figure that tells an interesting story. Include an interpretation of the figure.

Problem 5 (Medium): Generate the code to convert the following data frame to wide format.

grp	sex	meanL	sdL	meanR	sdR
A	F	0.225	0.106	0.340	0.085
A	M	0.470	0.325	0.570	0.325
B	F	0.325	0.106	0.400	0.071
B	M	0.547	0.308	0.647	0.274

The result should look like the following display.

grp	F.meanL	F.meanR	F.sdL	F.sdR	M.meanL	M.meanR	M.sdL	M.sdR
1 A	0.22	0.34	0.11	0.08	0.47	0.57	0.33	0.33
2 B	0.33	0.40	0.11	0.07	0.55	0.65	0.31	0.27

Hint: use `pivot_longer()` in conjunction with `pivot_wider()`.

Problem 6 (Medium): The `HELPfull` data within the `mosaicData` package contains information about the Health Evaluation and Linkage to Primary Care (HELP) randomized trial in *tall* format.

- Generate a table of the data for subjects (`ID`) 1, 2, and 3 that includes the `ID` variable, the `TIME` variable, and the `DRUGRISK` and `SEXRISK` variables (measures of drug and sex risk-taking behaviors, respectively).
- The HELP trial was designed to collect information at 0, 6, 12, 18, and 24 month intervals. At which timepoints were measurements available on the `*RISK` variables for subject 3?
- Let's restrict our attention to the data from the baseline (`TIME = 0`) and 6-month data. Use the `pivot_wider()` function from the `dplyr` package to create a table that looks like the following:

```
# A tibble: 3 x 5
  ID DRUGRISK_0 DRUGRISK_6 SEXRISK_0 SEXRISK_6
  <int>      <int>      <int>      <int>      <int>
1     1          0          0          4          1
```

2	2	0	0	7	0
3	3	20	13	2	4

- d) Repeat this process using all subjects. What is the Pearson correlation between the baseline (`TIME = 0`) and 6-month `DRUGRISK` scores? Repeat this for the `SEXRRISK` scores. (Hint: use the `use = "complete.obs"` option from the `cor()` function.)

Problem 7 (Medium): An analyst wants to calculate the pairwise differences between the Treatment and Control values for a small data set from a crossover trial (all subjects received both treatments) that consists of the following observations.

`ds1`

```
# A tibble: 6 x 3
  id group  vals
  <int> <chr> <dbl>
1 1     T      4
2 2     T      6
3 3     T      8
4 1     C      5
5 2     C      6
6 3     C     10
```

Then use the following code to create the new `diff` variable.

```
Treat <- filter(ds1, group == "T")
Control <- filter(ds1, group == "C")
all <- mutate(Treat, diff = Treat$vals - Control$vals)
all
```

Verify that this code works for this example and generates the correct values of -1 , 0 , and -2 . Describe two problems that might arise if the data set is not sorted in a particular order or if one of the observations is missing for one of the subjects. Provide an alternative approach to generate this variable that is more robust (hint: use `pivot_wider`).

Problem 8 (Medium): Write a function called `count_seasons` that, when given a `teamID`, will count the number of seasons the team played in the `Teams` data frame from the `Lahman` package.

Problem 9 (Medium): Replicate the functionality of `make_babynames_dist()` from the `mdsr` package to wrangle the original tables from the `babynames` package.

Problem 10 (Medium): Consider the number of home runs hit (`HR`) and home runs allowed (`HRA`) for the Chicago Cubs (`CHN`) baseball team. Reshape the `Teams` data from the `Lahman` package into “long” format and plot a time series conditioned on whether the HRs that involved the Cubs were hit by them or allowed by them.

Problem 11 (Medium): Using the approach described in Section 5.5.4 of the text, find another table in Wikipedia that can be scraped and visualized. Be sure to interpret your graphical display.

6.7 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-dataII.html#dataII-online-exercises>

Iteration

Calculators free human beings from having to perform arithmetic computations *by hand*. Similarly, programming languages free humans from having to perform iterative computations by re-running chunks of code, or worse, copying-and-pasting a chunk of code many times, while changing just one or two things in each chunk.

For example, in *Major League Baseball* there are 30 teams, and the game has been played for over 100 years. There are a number of natural questions that we might want to ask about *each team* (e.g., which player has accrued the most hits for that team?) or about each season (e.g., which seasons had the highest levels of scoring?). If we can write a chunk of code that will answer these questions for a single team or a single season, then we should be able to generalize that chunk of code to work for *all* teams or seasons. Furthermore, we should be able to do this without having to re-type that chunk of code. In this section, we present a variety of techniques for automating these types of iterative operations.

7.1 Vectorized operations

In every programming language that we can think of, there is a way to write a *loop*. For example, you can write a `for()` loop in **R** the same way you can with most programming languages. Recall that the `Teams` data frame contains one row for each team in each MLB season.

```
library(tidyverse)
library(mdsr)
library(Lahman)
names(Teams)

[1] "yearID"          "lgID"           "teamID"          "franchID"
[5] "divID"           "Rank"            "G"               "Ghome"
[9] "W"               "L"               "DivWin"          "WCWin"
[13] "LgWin"           "WSWin"          "R"               "AB"
[17] "H"               "X2B"            "X3B"            "HR"
[21] "BB"              "SO"              "SB"              "CS"
[25] "HBP"             "SF"              "RA"              "ER"
[29] "ERA"             "CG"              "SHO"             "SV"
[33] "IPouts"          "HA"              "HRA"             "BBA"
[37] "SOA"             "E"               "DP"              "FP"
[41] "name"            "park"            "attendance"      "BPF"
[45] "PPF"             "teamIDBR"        "teamIDlahman45"  "teamIDretro"
```

What might not be immediately obvious is that columns 15 through 40 of this data frame

contain numerical data about how each team performed in that season. To see this, you can execute the `str()` command to see the `structure` of the data frame, but we suppress that output here. For data frames, a similar alternative that is a little cleaner is `glimpse()`.

```
str(Teams)
glimpse(Teams)
```

Suppose you are interested in computing the averages of these 26 numeric columns. You don't want to have to type the names of each of them, or re-type the `mean()` command 26 times. Seasoned programmers would identify this as a situation in which a *loop* is a natural and efficient solution. A `for()` loop will iterate over the selected column indices.

```
averages <- NULL
for (i in 15:40) {
  averages[i - 14] <- mean(Teams[, i], na.rm = TRUE)
}
names(averages) <- names(Teams)[15:40]
averages
```

	R	AB	H	X2B	X3B	HR	BB	SO
684.623	5158.655	1348.461	229.707	46.298	105.218	475.790	758.008	
SB	CS	HBP	SF	RA	ER	ERA	CG	
110.647	47.405	45.717	44.902	684.622	575.663	3.830	48.494	
SHO	SV	IPouts	HA	HRA	BBA	SOA	E	
9.678	24.371	4035.986	1348.236	105.218	475.983	757.466	183.241	
DP	FP							
133.541	0.966							

This certainly works. However, there are a number of problematic aspects of this code (e.g., the use of multiple *magic numbers* like 14, 15, and 40). The use of a `for()` loop may not be ideal. For problems of this type, it is almost always possible (and usually preferable) to iterate without explicitly defining a loop. **R** programmers prefer to solve this type of problem by applying an operation to each element in a vector. This often requires only one line of code, with no appeal to indices.

It is important to understand that the fundamental architecture of **R** is based on *vectors*. That is, in contrast to *general-purpose programming languages* like C++ or Python that distinguish between single items—like strings and integers—and arrays of those items, in **R** a “string” is just a character vector of length 1. There is no special kind of atomic object. Thus, if you assign a single “string” to an object, **R** still stores it as a vector.

```
a <- "a string"
class(a)
```

```
[1] "character"
is.vector(a)
```

```
[1] TRUE
length(a)
```

```
[1] 1
```

As a consequence of this construction, **R** is highly optimized for vectorized operations (see [Appendix B](#) for more detailed information about **R** internals). Loops, by their nature, do

not take advantage of this optimization. Thus, **R** provides several tools for performing loop-like operations without actually writing a loop. This can be a challenging conceptual hurdle for those who are used to more general-purpose programming languages.

Pro Tip 24. Try to avoid writing `for()` loops, even when it seems like the easiest solution.

Many functions in **R** are *vectorized*. This means that they will perform an operation on every element of a vector by default. For example, many mathematical functions (e.g., `exp()`) work this way.

```
exp(1:3)
```

```
[1] 2.72 7.39 20.09
```

Note that vectorized functions like `exp()` take a vector as an input, and return a vector of the same length as an output.

This is importantly different behavior than so-called *summary functions*, which take a vector as an input, and return a single value. Summary functions (e.g., `mean()`) are commonly useful within a call to `summarize()`. Note that when we call `mean()` on a vector, it only returns a single value no matter how many elements there are in the input vector.

```
mean(1:3)
```

```
[1] 2
```

Other functions in **R** are not vectorized. They may assume an input that is a vector of length one, and fail or exhibit strange behavior if given a longer vector. For example, `if()` throws a warning if given a vector of length more than one.

```
if (c(TRUE, FALSE)) {
  cat("This is a great book!")
}
```

```
Warning in if (c(TRUE, FALSE)) {}: the condition has length > 1 and only the
first element will be used
```

```
This is a great book!
```

As you get more comfortable with **R**, you will develop intuition about which functions are vectorized. If a function is vectorized, you should make use of that fact and not iterate over it. The code below shows that computing the exponential of the first 10,000 integers by appealing to `exp()` as a vectorized function is much, much faster than using `map_dbl()` to iterate over the same vector. The results are identical.

```
x <- 1:1e5
bench::mark(
  exp(x),
  map_dbl(x, exp)
)

# A tibble: 2 x 6
  expression      min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>    <bch:tm> <bch:tm>    <dbl> <bch:byt>    <dbl>
1 exp(x)        264.1us  336.9us    2508.     1.53MB     83.2
2 map_dbl(x, exp) 48.6ms   48.6ms     20.6  788.62KB    185.
```

Pro Tip 25. Try to always make use of vectorized functions before iterating an operation.

7.2 Using `across()` with dplyr functions

The `mutate()` and `summarize()` verbs described in Chapter 4 can take advantage of an *adverb* called `across()` that applies operations programmatically. In the example above, we had to observe that columns 15 through 40 of the `Teams` data frame were numeric, and hard-code those observations as magic numbers. Rather than relying on these observations to identify which variables are numeric—and therefore which variables it makes sense to compute the average of—we can use the `across()` function to do this work for us. The chunk below will compute the average values of only those variables in `Teams` that are numeric.

```
Teams %>%
  summarize(across(where(is.numeric), mean, na.rm = TRUE))
```

	yearID	Rank	G	Ghome	W	L	R	AB	H	X2B	X3B	HR	BB	SO	SB	CS
1	1958	4.06	151	78.6	75	75	685	5159	1348	230	46.3	105	476	758	111	47.4
	HBP	SF	RA	ER	ERA	CG	SHO	SV	IPouts	HA	HRA	BBA	SOA	E	DP	
1	45.7	44.9	685	576	3.83	48.5	9.68	24.4	4036	1348	105	476	757	183	134	
	FP	attendance	BPF	PPF												
1	0.966		1390692	100	100											

Note that this result included several variables (e.g., `yearID`, `attendance`) that were outside of the range we defined previously.

The `across()` adverb allows us to specify the set of variables that `summarize()` includes in different ways. In the example above, we used the *predicate function* `is.numeric()` to identify the variables for which we wanted to compute the mean. In the following example, we compute the mean of `yearID`, a series of variables from `R` (runs scored) to `SF` (*sacrifice flies*) that apply only to offensive players, and the batting *park factor* (`BPF`). Since we are specifying these columns without the use of a predicate function, we don't need to use `where()`.

```
Teams %>%
  summarize(across(c(yearID, R:SF, BPF), mean, na.rm = TRUE))
```

	yearID	R	AB	H	X2B	X3B	HR	BB	SO	SB	CS	HBP	SF	BPF
1	1958	685	5159	1348	230	46.3	105	476	758	111	47.4	45.7	44.9	100

The `across()` function behaves analogously with `mutate()`. It provides an easy way to perform an operation on a set of variables without having to type or copy-and-paste the name of each variable.

7.3 The `map()` family of functions

More generally, to apply a function to each item in a list or vector, or the columns of a data frame¹, use `map()` (or one of its type-specific variants). This is the main function from the **purrr** package. In this example, we calculate the mean of each of the statistics defined above, all at once. Compare this to the `for()` loop written above. Which is syntactically simpler? Which expresses the ideas behind the code more succinctly?

```
Teams %>%
  select(15:40) %>%
  map_dbl(mean, na.rm = TRUE)
```

	R	AB	H	X2B	X3B	HR	BB	SO
684.623	5158.655	1348.461	229.707	46.298	105.218	475.790	758.008	
SB	CS	HBP	SF	RA	ER	ERA	CG	
110.647	47.405	45.717	44.902	684.622	575.663	3.830	48.494	
SHO	SV	IPouts	HA	HRA	BBA	SOA	E	
9.678	24.371	4035.986	1348.236	105.218	475.983	757.466	183.241	
DP	FP							
133.541	0.966							

The first argument to `map_dbl()` is the thing that you want to do something to (in this case, a data frame). The second argument specifies the name of a function (the argument is named `.f`). Any further arguments are passed as options to `.f`. Thus, this command applies the `mean()` function to the 15th through the 40th columns of the `Teams` data frame, while removing any `NAs` that might be present in any of those columns. The use of the variant `map_dbl()` simply forces the output to be a vector of type `double`.²

Of course, we began by taking the subset of the columns that were all `numeric` values. If you tried to take the `mean()` of a non-numeric vector, you would get a *warning* (and a value of `NA`).

```
Teams %>%
  select(teamID) %>%
  map_dbl(mean, na.rm = TRUE)
```

```
Warning in mean.default(.x[[i]], ...): argument is not numeric or logical:
returning NA
```

```
teamID
NA
```

If you can solve your problem using `across()` and/or `where()` as described in [Section 7.2](#), that is probably the cleanest solution. However, we will show that the `map()` family of functions provides a much more general set of capabilities.

¹This works because a data frame is stored as a `list` of vectors of the same length. When you supply a data frame as the first argument to `map()`, you are giving `map()` a list of vectors, and it iterates a function over that list.

²The plain function `map()` will always return a list.

7.4 Iterating over a one-dimensional vector

7.4.1 Iterating a known function

Often you will want to apply a function to each element of a vector or list. For example, the baseball franchise now known as the *Los Angeles Angels of Anaheim* has gone by several names during its history.

```
angels <- Teams %>%
  filter(franchID == "ANA") %>%
  group_by(teamID, name) %>%
  summarize(began = first(yearID), ended = last(yearID)) %>%
  arrange(began)
angels
```

```
# A tibble: 4 x 4
# Groups:   teamID [3]
  teamID name                began ended
  <fct>  <chr>              <int> <int>
1 LAA     Los Angeles Angels 1961   1964
2 CAL     California Angels  1965   1996
3 ANA     Anaheim Angels    1997   2004
4 LAA     Los Angeles Angels of Anaheim 2005   2019
```

The franchise began as the *Los Angeles Angels* (LAA) in 1961, then became the *California Angels* (CAL) in 1965, the *Anaheim Angels* (ANA) in 1997, before taking their current name (LAA again) in 2005. This situation is complicated by the fact that the `teamID` LAA was re-used. This sort of schizophrenic behavior is unfortunately common in many data sets.

Now, suppose we want to find the length, in number of characters, of each of those team names. We could check each one manually using the function `nchar()`:

```
angels_names <- angels %>%
  pull(name)
nchar(angels_names[1])
```

```
[1] 18
nchar(angels_names[2])
```

```
[1] 17
nchar(angels_names[3])
```

```
[1] 14
nchar(angels_names[4])
```

```
[1] 29
```

But this would grow tiresome if we had many names. It would be simpler, more efficient, more elegant, and scalable to apply the function `nchar()` to each element of the vector `angels_names`. We can accomplish this using `map_int()`. `map_int()` is like `map()` or `map_dbl()`, but it always returns an `integer` vector.

```
map_int(angels_names, nchar)
```

```
[1] 18 17 14 29
```

The key difference between `map_int()` and `map()` is that the former will always return an integer vector, whereas the latter will always return a `list`. Recall that the main difference between `lists` and `data.frames` is that the elements (columns) of a `data.frame` have to have the same length, whereas the elements of a list are arbitrary. So while `map()` is more versatile, we usually find `map_int()` or one of the other variants to be more convenient when appropriate.

Pro Tip 26. *It's often helpful to use `map()` to figure out what the return type will be, and then switch to the appropriate type-specific `map()` variant.*

This section was designed to illustrate how `map()` can be used to iterate a function over a vector of values. However, the choice of the `nchar()` function was a bit silly, because `nchar()` is already vectorized. Thus, we can use it directly!

```
nchar(angels_names)
```

```
[1] 18 17 14 29
```

7.4.2 Iterating an arbitrary function

One of the most powerful uses of iteration is that you can apply *any* function, including a function that you have defined (see [Appendix C](#) for a discussion of how to write user-defined functions). For example, suppose we want to display the top-5 seasons in terms of wins for each of the Angels teams.

```
top5 <- function(data, team_name) {
  data %>%
    filter(name == team_name) %>%
    select(teamID, yearID, W, L, name) %>%
    arrange(desc(W)) %>%
    head(n = 5)
}
```

We can now do this for each element of our vector with a single call to `map()`. Note how we named the `data` argument to ensure that the `team_name` argument was the one that accepted the value over which we iterated.

```
angels_names %>%
  map(top5, data = Teams)
```

```
[[1]]
#> #>   teamID yearID  W   L           name
#> #>   1     LAA   1962 86 76 Los Angeles Angels
#> #>   2     LAA   1964 82 80 Los Angeles Angels
#> #>   3     LAA   1961 70 91 Los Angeles Angels
#> #>   4     LAA   1963 70 91 Los Angeles Angels
```

```
[[2]]
#> #>   teamID yearID  W   L           name
#> #>   1     CAL   1982 93 69 California Angels
```

```

2 CAL 1986 92 70 California Angels
3 CAL 1989 91 71 California Angels
4 CAL 1985 90 72 California Angels
5 CAL 1979 88 74 California Angels

```

```
[[3]]
#> #> #> #> #>
#> #> #> #> #>
#> #> #> #> #>
#> #> #> #> #>
#> #> #> #> #>
```

	teamID	yearID	W	L	name
1	ANA	2002	99	63	Anaheim Angels
2	ANA	2004	92	70	Anaheim Angels
3	ANA	1998	85	77	Anaheim Angels
4	ANA	1997	84	78	Anaheim Angels
5	ANA	2000	82	80	Anaheim Angels

```
[[4]]
#> #> #> #> #>
#> #> #> #> #>
#> #> #> #> #>
#> #> #> #> #>
#> #> #> #> #>
```

	teamID	yearID	W	L	name
1	LAA	2008	100	62	Los Angeles Angels of Anaheim
2	LAA	2014	98	64	Los Angeles Angels of Anaheim
3	LAA	2009	97	65	Los Angeles Angels of Anaheim
4	LAA	2005	95	67	Los Angeles Angels of Anaheim
5	LAA	2007	94	68	Los Angeles Angels of Anaheim

Alternatively, we can collect the results into a single data frame by using the `map_dfr()` function, which combines the data frames by row. Below, we do this and then compute the average number of wins in a top-5 season for each Angels team name. Based on these data, the Los Angeles Angels of Anaheim has been the most successful incarnation of the franchise, when judged by average performance in the best five seasons.

```
angels_names %>%
  map_dfr(top5, data = Teams) %>%
  group_by(teamID, name) %>%
  summarize(N = n(), mean_wins = mean(W)) %>%
  arrange(desc(mean_wins))
```

```
#> #> #> #> #>
#> #> #> #> #>
#> #> #> #> #>
#> #> #> #> #>
#> #> #> #> #>
```

	teamID	name	N	mean_wins
1	LAA	Los Angeles Angels of Anaheim	5	96.8
2	CAL	California Angels	5	90.8
3	ANA	Anaheim Angels	5	88.4
4	LAA	Los Angeles Angels	4	77

Once you've read [Chapter 15](#), think about how you might do this operation in SQL. It is not that easy!

7.5 Iteration over subgroups

In [Chapter 4](#), we introduced data *verbs* that could be chained to perform very powerful data wrangling operations. These functions—which come from the `dplyr` package—operate on data frames and return data frames. The `group_modify()` function in `purrr` allows you

to apply an arbitrary function that returns a data frame to the *groups* of a data frame. That is, you will first define a grouping using the `group_by()` function, and then apply a function to each of those groups. Note that this is similar to `map_dfr()`, in that you are mapping a function that returns a data frame over a collection of values, and returning a data frame. But whereas the values used in `map_dfr()` are individual elements of a vector, in `group_modify()` they are groups defined on a data frame.

7.5.1 Example: Expected winning percentage

As noted in [Section 4.2](#), one of the more enduring models in *sabermetrics* is Bill James's formula for estimating a team's expected *winning percentage*, given knowledge only of the team's runs scored and runs allowed to date (recall that the team that scores the most runs wins a given game). This statistic is known—unfortunately—as Pythagorean Winning Percentage, even though it has nothing to do with Pythagoras. The formula is simple, but non-linear:

$$\widehat{WPct} = \frac{RS^2}{RS^2 + RA^2} = \frac{1}{1 + (RA/RS)^2},$$

where RS and RA are the number of runs the team has scored and allowed, respectively. If we define $x = RS/RA$ to be the team's *run ratio*, then this is a function of one variable having the form $f(x) = \frac{1}{1+(1/x)^2}$.

This model seems to fit quite well upon visual inspection—in [Figure 7.1](#) we show the data since 1954, along with a line representing the model. Indeed, this model has also been successful in other sports, albeit with wholly different exponents.

```
exp_wpct <- function(x) {
  return(1/(1 + (1/x)^2))
}

TeamRuns <- Teams %>%
  filter(yearID >= 1954) %>%
  rename(RS = R) %>%
  mutate(WPct = W / (W + L), run_ratio = RS/RA) %>%
  select(yearID, teamID, lgID, WPct, run_ratio)

ggplot(data = TeamRuns, aes(x = run_ratio, y = WPct)) +
  geom_vline(xintercept = 1, color = "darkgray", linetype = 2) +
  geom_hline(yintercept = 0.5, color = "darkgray", linetype = 2) +
  geom_point(alpha = 0.2) +
  stat_function(fun = exp_wpct, size = 2, color = "blue") +
  xlab("Ratio of Runs Scored to Runs Allowed") +
  ylab("Winning Percentage")
```

However, the exponent of 2 was posited by James. One can imagine having the exponent become a parameter k , and trying to find the optimal fit. Indeed, researchers have found that in baseball, the optimal value of k is not 2, but something closer to 1.85 (Wang, 2006). It is easy enough for us to find the optimal value using the `nls()` function. We specify the formula of the nonlinear model, the data used to fit the model, and a starting value for the search.

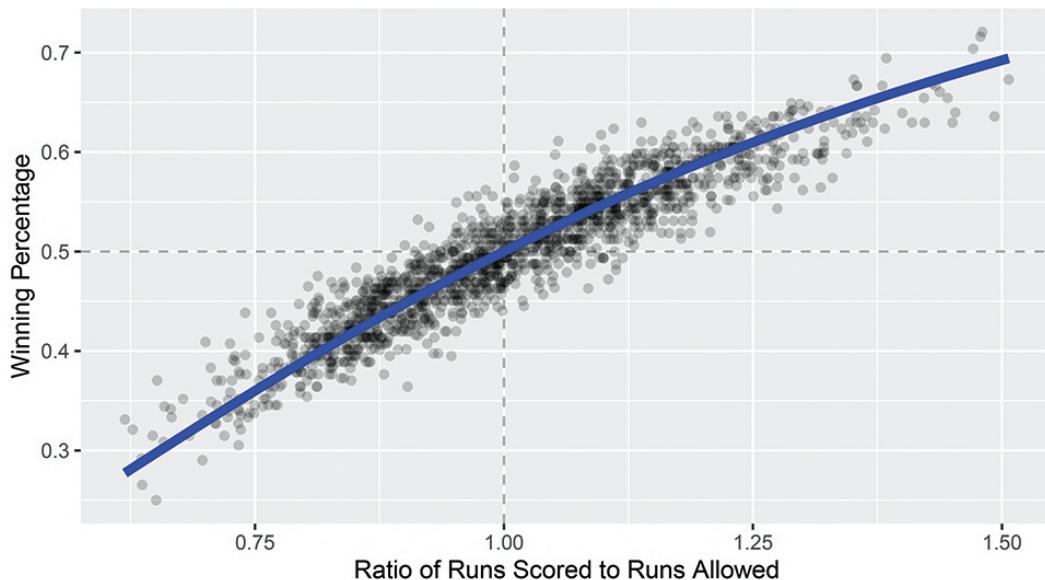


Figure 7.1: Fit for the Pythagorean Winning Percentage model for all teams since 1954.

```
TeamRuns %>%
  nls(
    formula = WPct ~ 1/(1 + (1/run_ratio)^k),
    start = list(k = 2)
  ) %>%
  coef()
```

```
k  
1.84
```

Furthermore, researchers investigating this model have found that the optimal value of the exponent varies based on the era during which the model is fit. We can use the `group_modify()` function to do this for all decades in baseball history. First, we must write a short function (see Appendix C) that will return a data frame containing the optimal exponent, and for good measure, the number of observations during that decade.

```
fit_k <- function(x) {
  mod <- nls(
    formula = WPct ~ 1/(1 + (1/run_ratio)^k),
    data = x,
    start = list(k = 2)
  )
  return(tibble(k = coef(mod), n = nrow(x)))
}
```

Note that this function will return the optimal value of the exponent over any time period.

```
fit_k(TeamRuns)
```

```
# A tibble: 1 x 2
  k      n
  <dbl> <int>
```

```
<dbl> <int>
1 1.84 1678
```

Finally, we compute the decade for each year using `mutate()`, define the group using `group_by()`, and apply `fit_k()` to those decades. The use of the `~` tells R to interpret the expression in parentheses as a `formula`, rather than the name of a function. The `.x` is a placeholder for the data frame for a particular decade.

```
TeamRuns %>%
  mutate(decade = yearID %/% 10 * 10) %>%
  group_by(decade) %>%
  group_modify(~fit_k(.x))
```

```
# A tibble: 7 x 3
# Groups:   decade [7]
  decade     k     n
  <dbl> <dbl> <int>
1 1950    1.69    96
2 1960    1.90   198
3 1970    1.74   246
4 1980    1.93   260
5 1990    1.88   278
6 2000    1.94   300
7 2010    1.77   300
```

Note the variation in the optimal value of k . Even though the exponent is not the same in each decade, it varies within a fairly narrow range between 1.69 and 1.94.

7.5.2 Example: Annual leaders

As a second example, consider the problem of identifying the team in each season that led their league in home runs. We can easily write a function that will, for a specific year and league, return a data frame with one row that contains the team with the most home runs.

```
hr_leader <- function(x) {
  # x is a subset of Teams for a single year and league
  x %>%
    select(teamID, HR) %>%
    arrange(desc(HR)) %>%
    head(1)
}
```

We can verify that in 1961, the *New York Yankees* led the *American League* in home runs.

```
Teams %>%
  filter(yearID == 1961 & lgID == "AL") %>%
  hr_leader()
```

```
teamID  HR
1  NYA 240
```

We can use `group_modify()` to quickly find all the teams that led their league in home runs. Here, we employ the `.keep` argument so that the grouping variables appear in the computation.

```
hr_leaders <- Teams %>%
  group_by(yearID, lgID) %>%
  group_modify(~hr_leader(.x), .keep = TRUE)

tail(hr_leaders, 4)
```

```
# A tibble: 4 x 4
# Groups: yearID, lgID [4]
  yearID lgID  teamID    HR
  <int> <fct> <fct> <int>
1 2018  AL    NYA     267
2 2018  NL    LAN     235
3 2019  AL    MIN     307
4 2019  NL    LAN     279
```

In this manner, we can compute the average number of home runs hit in a season by the team that hit the most.

```
hr_leaders %>%
  group_by(lgID) %>%
  summarize(mean_hr = mean(HR))

# A tibble: 7 x 2
  lgID  mean_hr
  <fct>   <dbl>
1 AA      40.5
2 AL      157.
3 FL      51
4 NA      13.8
5 NL      129.
6 PL      66
7 UA      32
```

We restrict our attention to the years since 1916, during which only the AL and NL leagues have existed.

```
hr_leaders %>%
  filter(yearID >= 1916) %>%
  group_by(lgID) %>%
  summarize(mean_hr = mean(HR))

# A tibble: 2 x 2
  lgID  mean_hr
  <fct>   <dbl>
1 AL      175.
2 NL      161.
```

In Figure 7.2, we show how this number has changed over time. We note that while the top HR hitting teams were comparable across the two leagues until the mid-1970s, the AL teams have dominated since their league adopted the *designated hitter* rule in 1973.

```
hr_leaders %>%
  filter(yearID >= 1916) %>%
  ggplot(aes(x = yearID, y = HR, color = lgID)) +
```

```
geom_line() +
  geom_point() +
  geom_smooth(se = FALSE) +
  geom_vline(xintercept = 1973) +
  annotate(
    "text", x = 1974, y = 25,
    label = "AL adopts DH", hjust = "left"
  ) +
  labs(x = "Year", y = "Home runs", color = "League")
```

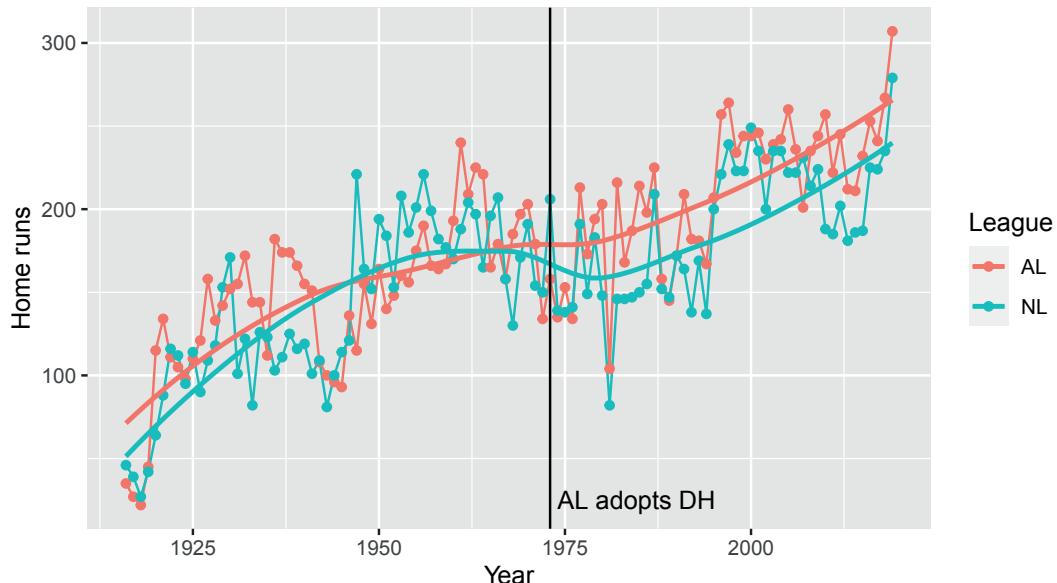


Figure 7.2: Number of home runs hit by the team with the most home runs, 1916–2019. Note how the AL has consistently bested the NL since the introduction of the designated hitter (DH) in 1973.

7.6 Simulation

In the previous section, we learned how to repeat operations while iterating over the elements of a vector. It can also be useful to simply repeat an operation many times and collect the results. Obviously, if the result of the operation is *deterministic* (i.e., you get the same answer every time) then this is pointless. On the other hand, if this operation involves randomness, then you won't get the same answer every time, and understanding the distribution of values that your random operation produces can be useful. We will flesh out these ideas further in Chapter 13.

For example, in our investigation into the expected winning percentage in baseball (Section 7.5.1), we determined that the optimal exponent fit to the 66 seasons worth of data from 1954 to 2019 was 1.84. However, we also found that if we fit this same model separately for each decade, that optimal exponent varies from 1.69 to 1.94. This gives us a rough sense

of the variability in this exponent—we observed values between 1.6 and 2, which may give some insights as to plausible values for the exponent.

Nevertheless, our choice to stratify by decade was somewhat arbitrary. A more natural question might be: What is the distribution of optimal exponents fit to a *single-season’s* worth of data? How confident should we be in that estimate of 1.84?

We can use `group_modify()` and the function we wrote previously to compute the 66 actual values. The resulting distribution is summarized in [Figure 7.3](#).

```
k_actual <- TeamRuns %>%
  group_by(yearID) %>%
  group_modify(~fit_k(.x))
k_actual %>%
  ungroup() %>%
  skim(k)

-- Variable type: numeric -----
var      n     na   mean    sd    p0    p25    p50    p75    p100
1 k       66      0  1.84  0.188  1.31  1.68  1.89  1.96  2.33

ggplot(data = k_actual, aes(x = k)) +
  geom_density() +
  xlab("Best fit exponent for a single season")
```

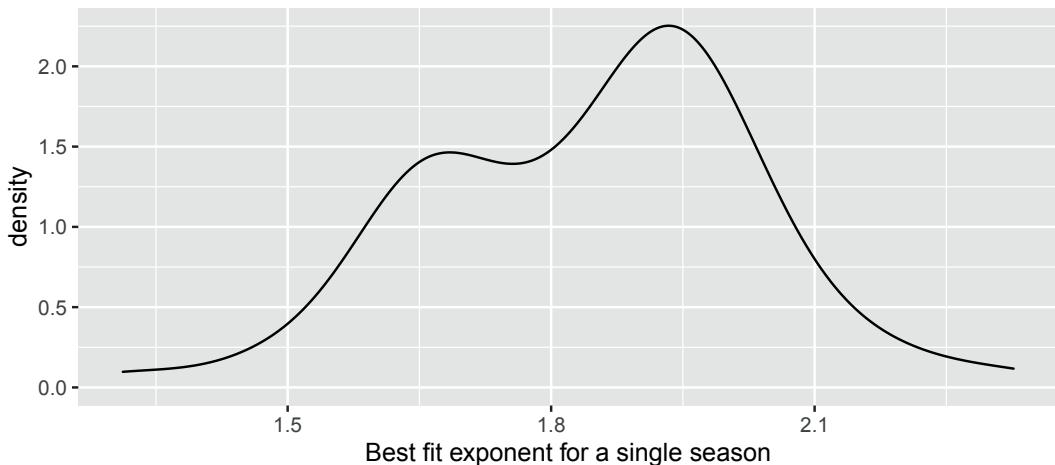


Figure 7.3: Distribution of best-fitting exponent across single seasons from 1954–2019.

Since we only have 66 samples, we might obtain a better understanding of the sampling distribution of the mean k by *resampling*—sampling with replacement—from these 66 values. (This is a statistical technique known as the *bootstrap*, which we describe further in [Chapter 9](#).) A simple way to do this is by mapping a sampling expression over an index of values. That is, we define n to be the number of iterations we want to perform, write an expression to compute the mean of a single resample, and then use `map_dbl()` to perform the iterations.

```
n <- 10000

bstrap <- 1:n %>%
  map_dbl(
```

```

~k_actual %>%
  pull(k) %>%
  sample(replace = TRUE) %>%
  mean()
)

civals <- bstrap %>%
  quantile(probs = c(0.025, .975))
civals

2.5% 97.5%
1.80 1.89

```

After repeating the resampling 10,000 times, we found that 95% of the resampled exponents were between 1.8 and 1.89, with our original estimate of 1.84 lying somewhere near the center of that distribution. This distribution, along the boundaries of the middle 95%, is depicted in Figure 7.4.

```

ggplot(data = enframe(bstrap, value = "k"), aes(x = k)) +
  geom_density() +
  xlab("Distribution of resampled means") +
  geom_vline(
    data = enframe(civals), aes(xintercept = value),
    color = "red", linetype = 3
  )

```

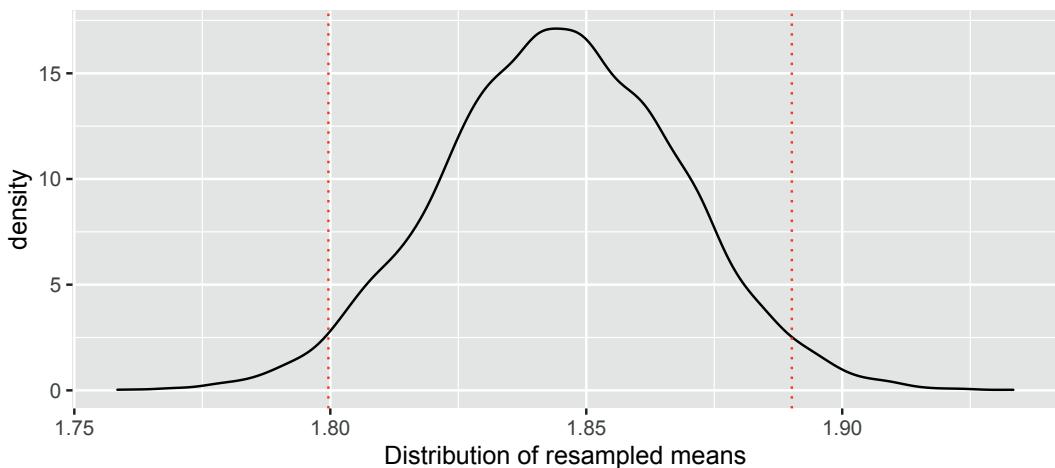


Figure 7.4: Bootstrap distribution of mean optimal Pythagorean exponent.

7.7 Extended example: Factors associated with BMI

Body Mass Index (BMI) is a common measure of a person's size, expressed as a ratio of their body's mass to the square of their height. What factors are associated with high BMI?

For answers, we turn to survey data collected by the National Center for Health Statistics.

tics (NCHS) and packaged as the *National Health and Nutrition Examination Survey* (NHANES). These data available in R through the **NHANES** package.

```
library(NHANES)
```

An exhaustive approach to understanding the relationship between BMI and some of the other variables is complicated by the fact that there are 75 potential explanatory variables for any model for BMI. In [Chapter 11](#), we develop several modeling techniques that might be useful for this purpose, but here, we focus on examining the *bivariate* relationships between BMI and the other explanatory variables. For example, we might start by simply producing a bivariate scatterplot between BMI and age, and adding a *local regression* line to show the general trend. [Figure 7.5](#) shows the result.

```
ggplot(NHANES, aes(x = Age, y = BMI)) +
  geom_point() +
  geom_smooth()
```

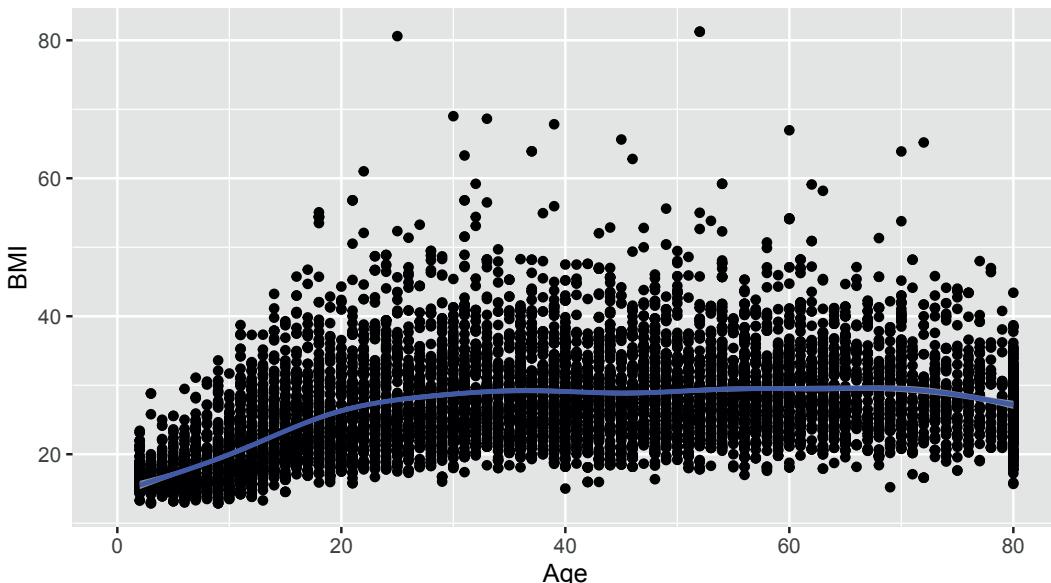


Figure 7.5: Relationship between body mass index (BMI) and age among participants in the **NHANES** study.

How can we programmatically produce an analogous image for *all* of the variables in **NHANES**? First, we'll write a function that takes the name of a variable as an input, and returns the plot. Second, we'll define a set of variables, and use `map()` to iterate our function over that list.

The following function will take a data set, and an argument called `x_var` that will be the name of a variable. It produces a slightly jazzed-up version of [Figure 7.5](#) that contains variable-specific titles, as well as information about the source.

```
bmi_plot <- function(.data, x_var) {
  ggplot(.data, aes(y = BMI)) +
    aes_string(x = x_var) +
    geom_jitter(alpha = 0.3) +
    geom_smooth() +
```

```

  labs(
    title = paste("BMI by", x_var),
    subtitle = "NHANES",
    caption = "US National Center for Health Statistics (NCHS)"
  )
}

```

The use of the `aes_string()` function is necessary for `ggplot2` to understand that we want to bind the `x` aesthetic to the variable whose name is stored in the `x_var` object, and not a variable that is named `x_var`³

We can then call our function on a specific variable.

```
bmi_plot(NHANES, "Age")
```

Or, we can specify a set of variables and then `map()` over that set. Since `map()` always returns a list, and a list of plots is not that useful, we use the `wrap_plots()` function from the `patchwork` package to combine the resulting list of plots into one image.

```

c("Age", "HHIncomeMid", "PhysActiveDays",
  "TVHrsDay", "AlcoholDay", "Pulse") %>%
  map(bmi_plot, .data = NHANES) %>%
  patchwork::wrap_plots(ncol = 2)

```

Figure 7.6 displays the results for six variables. We won't show the results of our ultimate goal to produce all 75 plots here, but you can try it for yourself by using the `names()` function to retrieve the full list of variable names. Or, you could use `across()` to retrieve only those variables that meet a certain condition.

7.8 Further resources

The chapter on *functionals* in Wickham (2019a) is the definitive source for understanding `purrr`. The name “functionals” reflects the use of a programming paradigm called *functional programming*.

For those who are already familiar with the `*apply()` family of functions popular in base R, Jenny Bryan wrote a helpful tutorial that maps these functions to their `purrr` equivalents.

The `rlang` package lays the groundwork for *tidy evaluation*, which allows you to work programmatically with unquoted variable names. The programming with `dplyr` vignette is the best place to start learning about tidy evaluation. Section C.4 provides a brief introduction to the principles.

³Note that we use the quoted variable names here. To get this to work with unquoted variable names, see the Programming with `dplyr` vignette.

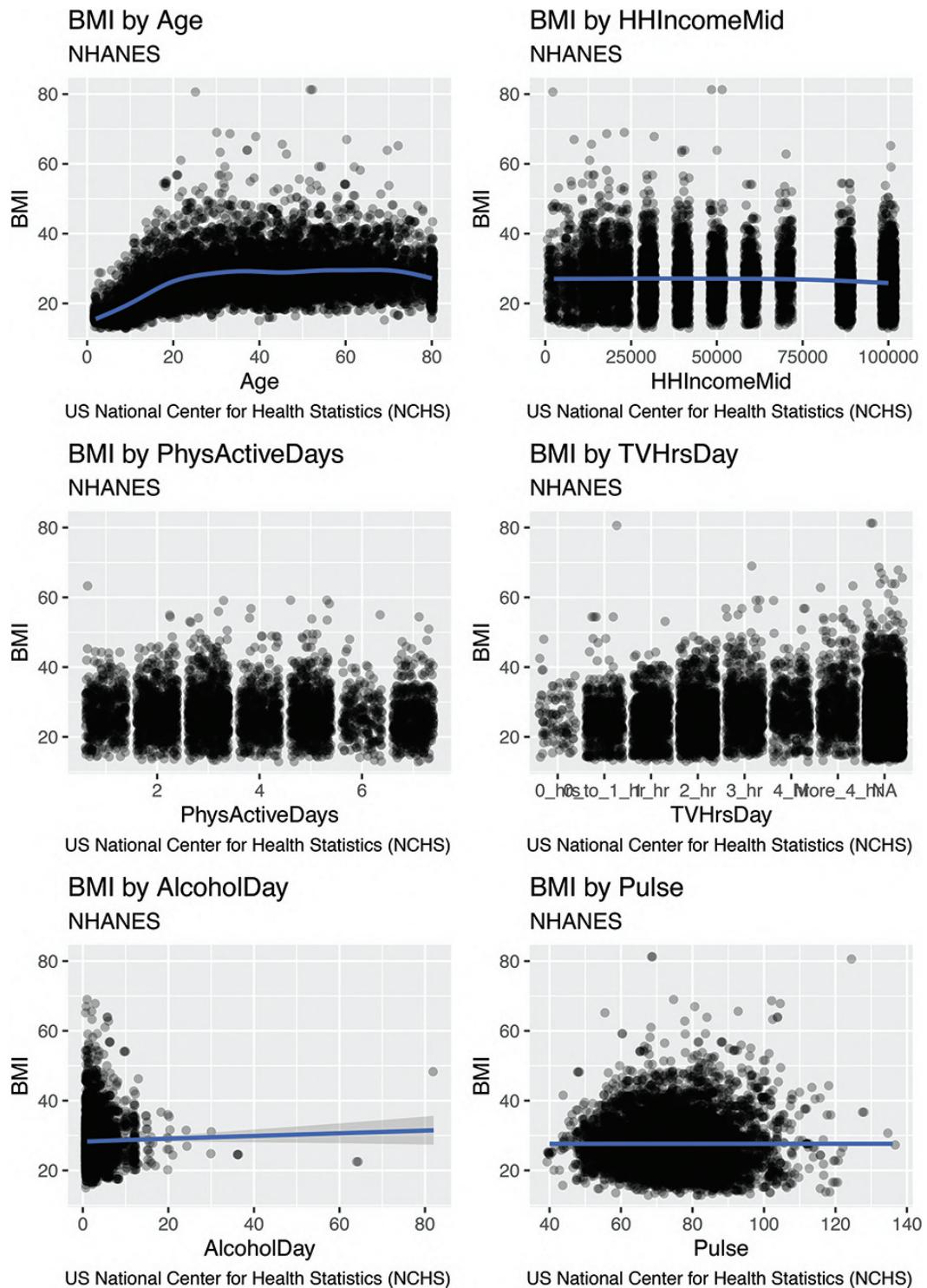


Figure 7.6: Relationship between body mass index (BMI) and a series of other variables, for participants in the **NHANES** study.

7.9 Exercises

Problem 1 (Easy): Use the `HELPrcf` data from the `mosaicData` to calculate the mean of all numeric variables (be sure to exclude missing values).

Problem 2 (Easy): Suppose you want to visit airports in Boston (`bos`), New York (`JFK`, `LGA`), San Francisco (`sfo`), Chicago (`ORD`, `MDW`), and Los Angeles (`LAX`). You have data about flight delays in a `tibble` called `flights`. You have written a pipeline that, for any given airport code (e.g., `LGA`), will return a `tibble` with two columns, the airport code, and the average arrival delay time.

Suggest a workflow that would be most efficient for computing the average arrival delay time for all seven airports.

Problem 3 (Medium): Use the `purrr::map()` function and the `HELPrcf` data frame from the `mosaicData` package to fit a regression model predicting `cesd` as a function of `age` separately for each of the levels of the `substance` variable. Generate a table of results (estimates and confidence intervals) for the slope parameter for each level of the grouping variable.

Problem 4 (Medium): The team IDs corresponding to Brooklyn baseball teams from the `Teams` data frame from the `Lahman` package are listed below. Use `map_int()` to find the number of seasons in which each of those teams played by calling a function called `count_seasons`.

```
library(Lahman)
bk_teams <- c("BR1", "BR2", "BR3", "BR4", "BRO", "BRP", "BRF")
```

Problem 5 (Medium): Use data from the `NHANES` package to create a set of scatterplots of `Pulse` as a function of `Age`, `BMI`, `TVHrsDay`, and `BPSysAve` to create a figure like the last one in the chapter. Be sure to create appropriate annotations (source, survey name, variables being displayed). What do you conclude?

Problem 6 (Hard): Use the `group_modify()` function and the `Lahman` data to replicate one of the baseball records plots (<http://tinyurl.com/nytimes-records>) from the *The New York Times*.

7.10 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-iteration.html#iteration-online-exercises>



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

Data science ethics

8.1 Introduction

Work in data analytics involves expert knowledge, understanding, and skill. In much of your work, you will be relying on the trust and confidence that your clients place in you. The term *professional ethics* describes the special responsibilities not to take unfair advantage of that trust. This involves more than being thoughtful and using common sense; there are specific professional standards that should guide your actions. Moreover, due to the potential that their work may be deployed at scale, data scientists must anticipate how their work could be used by others and wrestle with any ethical implications.

The best-known professional standards are those in the *Hippocratic Oath* for physicians, which were originally written in the 5th century B.C. Three of the eight principles in the modern version of the oath (Wikipedia, 2016) are presented here because of similarity to standards for data analytics.

1. “I will not be ashamed to say ‘I know not,’ nor will I fail to call in my colleagues when the skills of another are needed for a patient’s recovery.”
2. “I will respect the privacy of my patients, for their problems are not disclosed to me that the world may know.”
3. “I will remember that I remain a member of society, with special obligations to all my fellow human beings, those sound of mind and body as well as the infirm.”

Depending on the jurisdiction, these principles are extended and qualified by law. For instance, notwithstanding the need to “respect the privacy of my patients,” health-care providers in the United States are required by law to report to appropriate government authorities evidence of child abuse or infectious diseases such as botulism, chicken pox, and cholera.

This chapter introduces principles of professional ethics for data science and gives examples of legal obligations, as well as guidelines issued by professional societies. There is no official data scientist’s oath—although attempts to forge one exist (National Academies of Science, Engineering, and Medicine, 2018). Reasonable people can disagree about what actions are best, but the existing guidelines provide a description of the ethical expectations on which your clients can reasonably rely. As a consensus statement of professional ethics, the guidelines also establish standards of accountability.

8.2 Truthful falsehoods

The single best-selling book with “statistics” in the title is *How to Lie with Statistics* by Darrell Huff (Huff, 1954). Written in the 1950s, the book shows graphical ploys to fool people even with accurate data. A general method is to violate conventions and tacit expectations that readers rely on when interpreting graphs. One way to think of *How to Lie* is a text to show the general public what these tacit expectations are and give tips for detecting when the trick is being played on them. The book’s title, while compelling, has wrongly tarred the field of statistics. The “statistics” of the title are really just “numbers.” The misleading graphical techniques are employed by politicians, journalists, and businessmen: not statisticians. More accurate titles would be “How to Lie with Numbers,” or “Don’t be misled by graphics.”

Some of the graphical tricks in *How to Lie* are still in use. Consider these three recent examples.

8.2.1 Stand your ground

In 2005, the *Florida legislature* passed the controversial “Stand Your Ground” law that broadened the situations in which citizens can use lethal force to protect themselves against perceived threats. Advocates believed that the new law would ultimately reduce crime; opponents feared an increase in the use of lethal force. What was the actual outcome?

The graphic in [Figure 8.1](#) is a reproduction of one published by the news service Reuters on February 16, 2014 showing the number of firearm murders in Florida over the years. Upon first glance, the graphic gives the visual impression that right after the passage of the 2005 law, the number of murders decreased substantially. However, the numbers tell a different story.

The convention in data graphics is that up corresponds to increasing values. This is not an obscure convention—rather, it’s a standard part of the secondary school curriculum. Close inspection reveals that the *y*-axis in [Figure 8.1](#) has been flipped upside down—the number of gun deaths increased sharply after 2005.

8.2.2 Global temperature

[Figure 8.2](#) shows another example of misleading graphics: a tweet by the news magazine *National Review* on the subject of climate change. The dominant visual impression of the graphic is that global temperature has hardly changed at all.

There is a tacit graphical convention that the coordinate scales on which the data are plotted are relevant to an informed interpretation of the data. The *x*-axis follows the convention—1880 to 2015 is a reasonable choice when considering the relationship between human industrial activity and climate. The *y*-axis, however, is utterly misleading. The scale goes from −10 to 110 degrees *Fahrenheit*. While this is a relevant scale for showing *season-to-season* variation in temperature, that is not the salient issue with respect to climate change. The concern with climate change is about rising ocean levels, intensification of storms, ecological and agricultural disruption, etc. These are the anticipated results of a change in global *average* temperature on the order of 5 degrees Fahrenheit. The *National Review* graphic has obscured the data by showing them on an irrelevant scale where the actual changes in temperature are practically invisible. By graying out the numbers on the *y*-axis, the

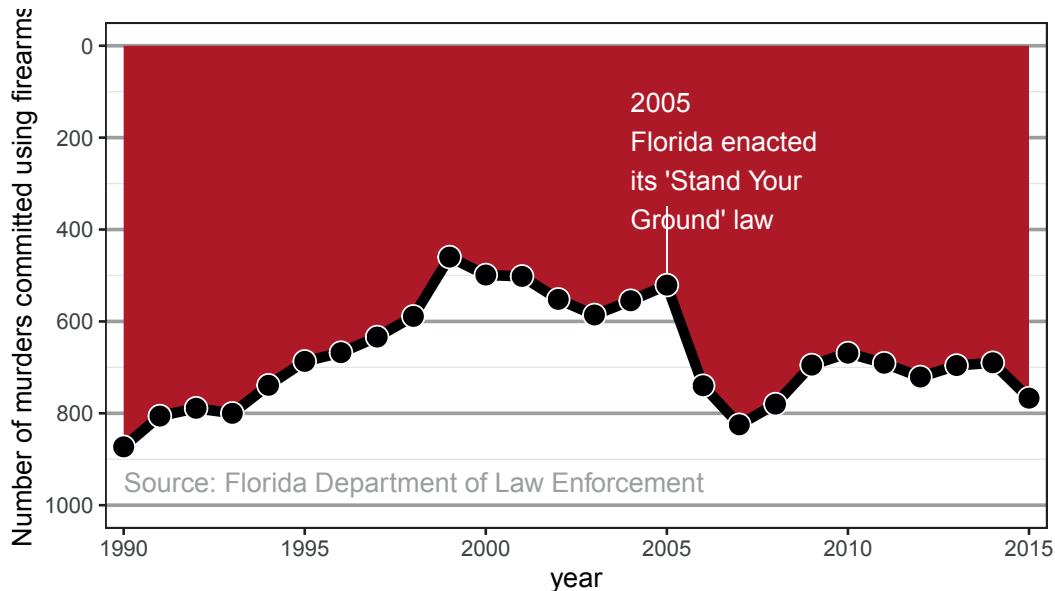


Figure 8.1: Reproduction of a data graphic reporting the number of gun deaths in Florida over time. The original image was published by Reuters.

National Review makes it even harder to see the trick that's being played. The tweet was subsequently deleted.

8.2.3 COVID-19 reporting

In May 2020, the state of *Georgia* published a highly misleading graphical display of COVID-19 cases (see Figure 8.3). Note that the results for April 17th appear to the right of April 19th, and that the counties are ordered such that all of the results are monotonically decreasing for each reporting period. The net effect of the graph is to demonstrate that confirmed COVID cases are decreasing, but it does so in a misleading fashion. Public outcry led to a statement from the governor's office that moving forward, chronological order would be used to display time.

8.3 Role of data science in society

The examples in Figures 8.1, 8.2, and 8.3 are not about lying with statistics. Statistical methodology doesn't enter into them. It's the professional ethics of journalism that the graphics violate, aided and abetted by an irresponsible ignorance of statistical methodology. Insofar as the graphics concern matters of political controversy, they can be seen as part of the political process. While politics is a profession, it's a profession without any comprehensive standard of professional ethics.

As data scientists, what role do we play in shaping public discourse? What responsibilities do we have? The stakes are high, and context matters.

The misleading data graphic about the "Stand Your Ground" law was published about six

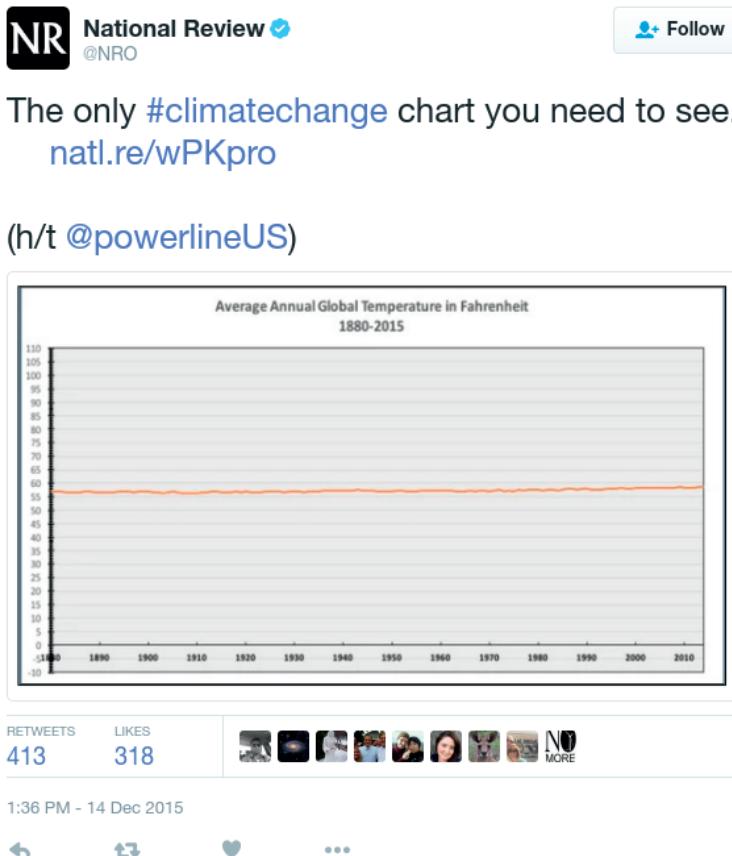


Figure 8.2: A tweet by *National Review* on December 14, 2015 showing the change in global temperature over time. The tweet was later deleted.

months after George Zimmerman was acquitted for killing Trayvon Martin. Did the data graphic affect public perception in the wake of this tragedy?

The *National Review* tweet was published during the thick of the presidential primaries leading up the 2016 election, and the publication is a leading voice in conservative political thought. *Pew Research* reports that while concern about climate change has increased steadily among those who lean Democratic since 2013 (88% said climate change is “a major threat to the United States” in 2020, up from 58% in 2013), it did not increase at all among those who lean Republican from 2010 to mid-2019, holding steady at 25%. Did the *National Review* persuade their readers to dismiss the *scientific consensus on climate change*?

The misleading data graphic about COVID-19 cases in Georgia was published during a time that Governor Brian Kemp’s reopening plan was facing stiff criticism from Atlanta mayor Keisha Lance Bottoms, his former opponent in the governor’s race Stacey Abrams, and even President Donald Trump. Journalists called attention to the data graphic on May 10th. The Georgia Department of Health itself reports that the 7-day moving average for COVID-19 cases increased by more than 125 cases per day during the two weeks following May 10th. Did the Georgia’s governor’s office convince people to ignore the risk of COVID-19?

Top 5 Counties with the Greatest Number of Confirmed COVID-19 (Reproduction of Figure)

The chart below represents the most impacted counties over the past 15 days and the number of cases over time. The table below also represents the number of deaths and hospitalizations in each of those impacted counties.

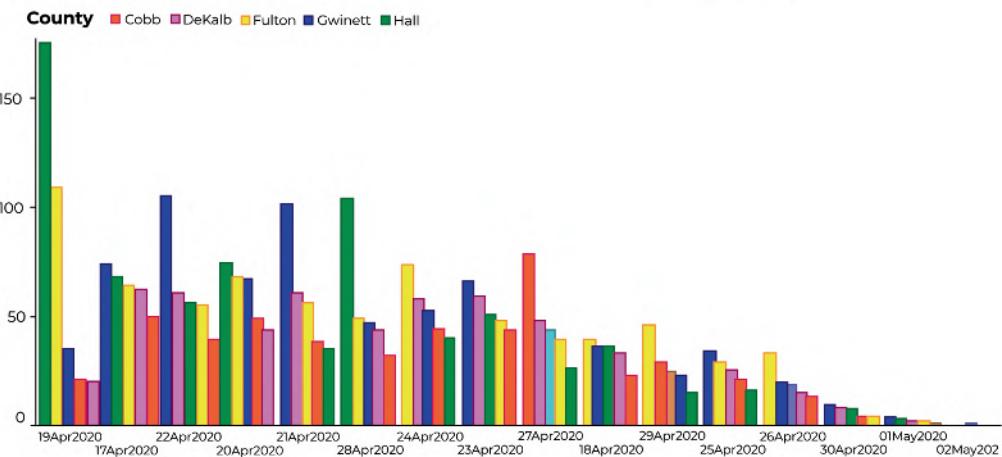


Figure 8.3: A recreation of a misleading display of confirmed COVID-19 cases in Georgia.

These unanswered (and intentionally provocative) questions are meant to encourage you to see the deep and not always obvious ways in which data science work connects to society at-large.

8.4 Some settings for professional ethics

Common sense is a good starting point for evaluating the ethics of a situation. Tell the truth. Don't steal. Don't harm innocent people. But professional ethics also require an informed assessment. A dramatic illustration of this comes from legal ethics: a situation where the lawyers for an accused murderer found the bodies of two victims whose deaths were unknown to authorities and to the victims' families. The responsibility to confidentiality for their client precluded the lawyers from following their hearts and reporting the discovery. The lawyers' careers were destroyed by the public and political recriminations that followed, yet courts and legal scholars have confirmed that the lawyers were right to do what they did, and have even held them up as heroes for their ethical behavior.

Such extreme drama is rare. This section describes in brief six situations that raise questions of the ethical course of action. Some are drawn from the authors' personal experience, others from court cases and other reports. The purpose of these short case reports is to raise questions. Principles for addressing those questions are the subject of the next section.

8.4.1 The chief executive officer

One of us once worked as a statistical consultant for a client who wanted a proprietary model to predict commercial outcomes. After reviewing the literature, an existing multiple

linear regression model was found that matched the scenario well, and available public data were used to fit the parameters of the model. The client's staff were pleased with the result, but the CEO wanted a model that would give a competitive advantage. After all, their competitors could easily follow the same process to the same model, so what advantage would the client's company have? The CEO asked the statistical consultant whether the coefficients in the model could be "tweaked" to reflect the specific values of his company. The consultant suggested that this would not be appropriate, that the fitted coefficients best match the data and to change them arbitrarily would be "playing God." In response, the CEO rose from his chair and asserted, "I want to play God."

How should the consultant respond?

8.4.2 Employment discrimination

One of us works on legal cases arising from audits of employers, conducted by the *United States Office of Federal Contract Compliance Programs* (OFCCP). In a typical case, the OFCCP asks for hiring and salary data from a company that has a contract with the United States government. The company usually complies, sometimes unaware that the OFCCP applies a method to identify "discrimination" through a two-standard-deviation test outlined in the Uniform Guidelines on Employee Selection Procedures (UGESP). A company that does not discriminate has some risk of being labeled as discriminating by the OFCCP method (Bridgeford, 2014). By using a questionable statistical method, is the OFCCP acting unethically?

8.4.3 "Gaydar"

Wang and Kosinski (2018) used a deep neural network (see [Section 11.1.5](#)) and logistic regression to build a classifier (see [Chapter 10](#)) for sexual orientation based on pictures of people's faces. The authors claim that if given five images of a person's face, their model would correctly predict the sexual orientation of 91% of men and 83% of women. The authors highlight the potential harm that their work could do in their abstract:

"Additionally, given that companies and governments are increasingly using computer vision algorithms to detect people's intimate traits, our findings expose a threat to the privacy and safety of gay men and women."

A subsequent article in *The New Yorker* also notes that:

"the study consisted entirely of white faces, but only because the dating site had served up too few faces of color to provide for meaningful analysis."

Was this research ethical? Were the authors justified in creating and publishing this model?

8.4.4 Race prediction

Imai and Khanna (2016) built a racial prediction algorithm using a Bayes classifier (see [Section 11.1.4](#)) trained on voter registration records from Florida and the U.S. Census Bureau's name list. In addition to the publishing the paper detailing the methodology, the authors published the software for the classifier on *Github* under an open-source license. The **wru** package is available on CRAN and will return predicted probabilities for a person's race based on either their last name alone, or their last name and their address.

```
library(tidyverse)
library(wru)
predict_race(voter.file = voters, surname.only = TRUE) %>%
  select(surname, pred.whi, pred.bla, pred.his, pred.asi, pred.oth)

[1] "Proceeding with surname-only predictions..."

  surname pred.whi pred.bla pred.his pred.asi pred.oth
4      Khanna  0.0676  0.0043  0.00820  0.8668  0.05310
2      Imai   0.0812  0.0024  0.06890  0.7375  0.11000
8     Velasco  0.0594  0.0026  0.82270  0.1051  0.01020
1     Fifield  0.9356  0.0022  0.02850  0.0078  0.02590
10    Zhou    0.0098  0.0018  0.00065  0.9820  0.00575
7     Ratkovic 0.9187  0.0108  0.01083  0.0108  0.04880
3     Johnson  0.5897  0.3463  0.02360  0.0054  0.03500
5     Lopez   0.0486  0.0057  0.92920  0.0102  0.00630
11   Wantchekon 0.6665  0.0853  0.13670  0.0797  0.03180
6     Morse   0.9054  0.0431  0.02060  0.0072  0.02370
```

Given the long history of *systemic racism* in the United States, it is clear how this software could be used to discriminate against people of color. One of us once partnered with a progressive voting rights organization that wanted to use racial prediction to target members of an ethnic group to *help them register to vote*.

Was the publication of this model ethical? Does the open-source nature of the code affect your answer? Is it ethical to use this software? Does your answer change depending on the intended use?

8.4.5 Data scraping

In May 2016, the online OpenPsych Forum published a paper by Kirkegaard and Bjerrekær (2016) titled “The OkCupid data set: A very large public data set of dating site users.” The resulting data set contained 2,620 variables—including usernames, gender, and dating preferences—from 68,371 people scraped from the OkCupid dating website. The ostensible purpose of the data dump was to provide an interesting open public data set to fellow researchers. These data might be used to answer questions such as this one suggested in the abstract of the paper: whether the *zodiac sign* of each user was associated with any of the other variables (spoiler alert: it wasn’t).

The data scraping did not involve any illicit technology such as breaking passwords. Nonetheless, the author received many comments on the OpenPsych Forum challenging the work as

an ethical breach and accusing him of *doxing* people by releasing personal data. Does the work raise ethical issues?

8.4.6 Reproducible spreadsheet analysis

In 2010, *Harvard University* economists Carmen Reinhart and Kenneth Rogoff published a report entitled “Growth in a Time of Debt” (Rogoff and Reinhart, 2010), which argued that countries which pursued austerity measures did not necessarily suffer from slow economic growth. These ideas influenced the thinking of policymakers—notably United States Congressman Paul Ryan—during the time of the *European debt crisis*.

University of Massachusetts graduate student Thomas Herndon requested access to the data and analysis contained in the paper. After receiving the original spreadsheet from Reinhart, Herndon found several errors.

“I clicked on cell L51, and saw that they had only averaged rows 30 through 44, instead of rows 30 through 49.” —Thomas Herndon (Roose, 2013)

In a critique of the paper, Herndon et al. (2014) point out coding errors, selective inclusion of data, and odd weighting of summary statistics that shaped the conclusions of the Reinhart/Rogoff paper.

What ethical questions does publishing a flawed analysis raise?

8.4.7 Drug dangers

In September 2004, the drug company *Merck* withdrew the popular product *Vioxx* from the market because of evidence that the drug increases the risk of *myocardial infarction* (MI), a major type of heart attack. Approximately 20 million Americans had taken Vioxx up to that point. The leading medical journal *Lancet* later reported an estimate that Vioxx use resulted in 88,000 Americans having heart attacks, of whom 38,000 died.

Vioxx had been approved in May 1999 by the *United States Food and Drug Administration* based on tests involving 5,400 subjects. Slightly more than a year after the FDA approval, a study (Bombardier et al., 2000) of 8,076 patients published in another leading medical journal, *The New England Journal of Medicine*, established that Vioxx reduced the incidence of severe gastrointestinal events substantially compared to the standard treatment, *naproxen*. That’s good for Vioxx. In addition, the abstract reports these findings regarding heart attacks:

The incidence of myocardial infarction was lower among patients in the naproxen group than among those in the [Vioxx] group (0.1 percent vs. 0.4 percent; relative risk, 0.2; 95% confidence interval, 0.1 to 0.7); the overall mortality rate and the rate of death from cardiovascular causes were similar in the two groups.”

Read the abstract again carefully. The Vioxx group had a much *higher* rate of MI than the group taking the standard treatment. This influential report identified the high risk soon after the drug was approved for use. Yet Vioxx was not withdrawn for another three years. Something clearly went wrong here. Did it involve an ethical lapse?

8.4.8 Legal negotiations

Lawyers sometimes retain statistical experts to help plan negotiations. In a common scenario, the defense lawyer will be negotiating the amount of damages in a case with the plaintiff's attorney. Plaintiffs will ask the statistician to estimate the amount of damages, with a clear but implicit directive that the estimate should reflect the plaintiff's interests. Similarly, the defense will ask their own expert to construct a framework that produces an estimate at a lower level.

Is this a game statisticians should play?

8.5 Some principles to guide ethical action

In [Section 8.1](#), we listed three principles from the Hippocratic Oath that has been administered to doctors for hundreds of years. Below, we reprint the three corresponding principles as outlined in the Data Science Oath published by the *National Academy of Sciences* (National Academies of Science, Engineering, and Medicine, 2018).

1. I will not be ashamed to say, “I know not,” nor will I fail to call in my colleagues when the skills of another are needed for solving a problem.
2. I will respect the privacy of my data subjects, for their data are not disclosed to me that the world may know, so I will tread with care in matters of privacy and security.
3. I will remember that my data are not just numbers without meaning or context, but represent real people and situations, and that my work may lead to unintended societal consequences, such as inequality, poverty, and disparities due to algorithmic bias.

To date, the Data Science Oath has not achieved the widespread adoption or formal acceptance of its inspiration. We hope this will change in the coming years.

Another set of ethical guidelines for data science is the Data Values and Principles manifesto published by DataPractices.org. This document espouses four values (inclusion, experimentation, accountability, and impact) and 12 principles that provide a guide for the ethical practice of data science:

1. Use data to improve life for our users, customers, organizations, and communities.
2. Create reproducible and extensible work.
3. Build teams with diverse ideas, backgrounds, and strengths.
4. Prioritize the continuous collection and availability of discussions and metadata.

5. Clearly identify the questions and objectives that drive each project and use to guide both planning and refinement.
6. Be open to changing our methods and conclusions in response to new knowledge.
7. Recognize and mitigate bias in ourselves and in the data we use.
8. Present our work in ways that empower others to make better-informed decisions.
9. Consider carefully the ethical implications of choices we make when using data, and the impacts of our work on individuals and society.
10. Respect and invite fair criticism while promoting the identification and open discussion of errors, risks, and unintended consequences of our work.
11. Protect the privacy and security of individuals represented in our data.
12. Help others to understand the most useful and appropriate applications of data to solve real-world problems.

In October 2020, this document had over 2,000 signatories (including two of the authors of this book).

In what follows we explore how these principles can be applied to guide ethical thinking in the several scenarios outlined in the previous section.

8.5.1 The CEO

You've been asked by a company CEO to modify model coefficients from the correct values, that is, from the values found by a generally accepted method. The stakeholder in this setting is the company. If your work will involve a method that's not generally accepted by the professional community, you're obliged to point this out to the company.

Principles 8 and 12 are germane. Have you presented your work in a way that empowers others to make better-informed decisions (principle 8)? Certainly your client also has substantial knowledge of how their business works. It's important to realize that your client's needs may not map well onto a particular statistical methodology. The consultant should work genuinely to understand the client's whole set of interests (principle 12). Often the problem that clients identify is not really the problem that needs to be solved when seen from an expert statistical perspective.

8.5.2 Employment discrimination

The procedures adopted by the OFCCP are stated using statistical terms like "standard deviation" that themselves suggest that they are part of a legitimate statistical method. Yet the methods raise significant questions, since by construction they will sometimes label a company that is not discriminating as a discriminator. Principle 10 suggests the OFCCP should "invite fair criticism" of their methodology. OFCCP and others might argue that they are not a statistical organization. They are enforcing a law, not participating in research. The OFCCP has a responsibility to the courts. The courts themselves, including the United States Supreme Court, have not developed or even called for a coherent approach to the use of statistics (although in 1977 the Supreme Court labeled differences greater than two or three standard deviations as too large to attribute solely to chance).

8.5.3 "Gaydar"

Principles 1, 3, 7, 9, and 11 are relevant here. Does the prediction of sexual orientation based on facial recognition improve life for communities (principle 1)? As noted in the abstract,

the researchers *did* consider the ethical implications of their work (principle 9), but did they protect the privacy and security of the individuals presented in their data (principle 11)? The exclusion of non-white faces from the study casts doubt on whether the standard outlined in principle 7 was met.

8.5.4 Race prediction

Clearly, using this software to discriminate against historically marginalized people would violate some combination of principles 3, 7, and 9. On the other hand, is it ethical to use this software to try and help underrepresented groups if those same principles are not violated? The authors of the **wru** package admirably met principle 2, but they may not have fully adhered to principle 9.

8.5.5 Data scraping

OkCupid provides public access to data. A researcher uses legitimate means to acquire those data. What could be wrong?

There is the matter of the stakeholders. The collection of data was intended to support psychological research. The ethics of research involving humans requires that the human not be exposed to any risk for which consent has not been explicitly given. The OkCupid members did not provide such consent. Since the data contain information that makes it possible to identify individual humans, there is a realistic risk of the release of potentially embarrassing information, or worse, information that jeopardizes the physical safety of certain users. Principles 1 and 11 were clearly violated by the authors. Ultimately, the Danish Data Protection Agency decided not to file any charges against the authors.

Another stakeholder is OkCupid itself. Many information providers, like OkCupid, have *terms of use* that restrict how the data may be legitimately used. Such terms of use (see [Section 8.7.3](#)) form an explicit agreement between the service and the users of that service. They cannot ethically be disregarded.

8.5.6 Reproducible spreadsheet analysis

The scientific community as a whole is a stakeholder in public research. Insofar as the research is used to inform public policy, the public as a whole is a stakeholder. Researchers have an obligation to be truthful in their reporting of research. This is not just a matter of being honest but also of participating in the process by which scientific work is challenged or confirmed. Reinhart and Rogoff honored this professional obligation by providing reasonable access to their software and data. In this regard, they complied with principle 10.

Seen from the perspective of data science, Microsoft Excel, the tool used by Reinhart and Rogoff, is an unfortunate choice. It mixes the data with the analysis. It works at a low level of abstraction, so it's difficult to program in a concise and readable way. Commands are customized to a particular size and organization of data, so it's hard to apply to a new or modified data set. One of the major strategies in debugging is to work on a data set where the answer is known; this is impractical in Excel. Programming and revision in Excel generally involves lots of click-and-drag copying, which is itself an error-prone operation.

Data science professionals have an ethical obligation to use tools that are reliable, verifiable, and conducive to reproducible data analysis (see [Appendix D](#)). Reinhart and Rogoff did not meet the standard implied by principle 2.

8.5.7 Drug dangers

When something goes wrong on a large scale, it's tempting to look for a breach of ethics. This may indeed identify an offender, but we must also beware of creating scapegoats. With Vioxx, there were many claims, counterclaims, and lawsuits. The researchers failed to incorporate some data that were available and provided a misleading summary of results. The journal editors also failed to highlight the very substantial problem of the increased rate of myocardial infarction with Vioxx.

To be sure, it's unethical not to include data that undermines the conclusion presented in a paper. The Vioxx researchers were acting according to their original research protocol—a solid professional practice.

What seems to have happened with Vioxx is that the researchers had a theory that the higher rate of infarction was not due to Vioxx, *per se*, but to an aspect of the study protocol that excluded subjects who were being treated with aspirin to reduce the risk of heart attacks. The researchers believed with some justification that the drug to which Vioxx was being compared, naproxen, was acting as a substitute for aspirin. They were wrong, as subsequent research showed. Their failure was in not honoring principle 6 and publishing their results in a misleading way.

Professional ethics dictate that professional standards be applied in work. Incidents like Vioxx should remind us to work with appropriate humility and to be vigilant to the possibility that our own explanations are misleading us.

8.5.8 Legal negotiations

In legal cases such as the one described earlier in the chapter, the data scientist has ethical obligations to their client. Depending on the circumstances, they may also have obligations to the court.

As always, you should be forthright with your client. Usually you will be using methods that you deem appropriate, but on occasion you will be directed to use a method that you think is inappropriate. For instance, we've seen occasions when the client requested that the time period of data included in the analysis be limited in some way to produce a "better" result. We've had clients ask us to subdivide the data (in employment discrimination cases, say, by job title) in order to change p-values. Although such subdivision may be entirely legitimate, the decision about subdividing—seen from a purely statistical point of view—ought to be based on the situation, not the desired outcome (see the discussion of the "garden of forking paths" in [Section 9.7](#)).

Your client is entitled to make such requests. Whether or not you think the method being asked for is the right one doesn't enter into it. Your professional obligation is to inform the client what the flaws in the proposed method are and how and why you think another method would be better (principle 8). (See the major exception that follows.)

The legal system in countries such as the U.S. is an *adversarial* system. Lawyers are allowed to frame legal arguments that may be dismissed: They are entitled to enter some facts and not others into evidence. Of course, the opposing legal team is entitled to create their own legal arguments and to cross-examine the evidence to show how it is incomplete and misleading. When you are working with a legal team as a data scientist, you are part of the team. The lawyers on the team are the experts about what negotiation strategies and legal theories to use, how to define the limits of the case (such as damages), and how to present their case or negotiate with the other party.

It is a different matter when you are presenting to the court. This might take the form of filing an expert report to the court, testifying as an expert witness, or being deposed. A deposition is when you are questioned, under oath, outside of the courtroom. You are obliged to answer all questions honestly. (Your lawyer may, however, direct you not to answer a question about privileged communications.)

If you are an expert witness or filing an expert report, the word “expert” is significant. A court will certify you as an expert in a case giving you permission to express your opinions. Now you have professional ethical obligations to apply your expertise honestly and openly in forming those opinions.

When working on a legal case, you should get advice from a legal authority, which might be your client. Remember that if you do shoddy work, or fail to reply honestly to the other side’s criticisms of your work, your credibility as an expert will be imperiled.

8.6 Algorithmic bias

Algorithms are at the core of many data science models (see [Chapter 11](#) for a comprehensive introduction). These models are being used to automate decision-making in settings as diverse as navigation for self-driving cars and determinations of risk for recidivism (return to criminal behavior) in the criminal justice system. The potential for bias to be reinforced when these models are implemented is dramatic.

Biased data may lead to algorithmic bias. As an example, some groups may be underrepresented or systematically excluded from data collection efforts. D’Ignazio and Klein (2020) highlight issues with data collection related to undocumented immigrants. O’Neil (2016) details several settings in which algorithmic bias has harmful consequences, whether intended or not.

Consider a criminal recidivism algorithm used in several states and detailed in a *ProPublica* story titled “Machine Bias” (Angwin et al., 2016). The algorithm returns predictions about how likely a criminal is to commit another crime based on a survey of 137 questions. *ProPublica* claims that the algorithm is biased:

“Black defendants were still 77 percent more likely to be pegged as at higher risk of committing a future violent crime and 45 percent more likely to be predicted to commit a future crime of any kind.”

How could the predictions be biased, when the race of the defendants is not included in the model? Consider that one of the survey questions is “was one of your parents ever sent to jail or prison?” Because of the longstanding relationship between *race and crime in the United States*, Black people are much more likely to have a parent who was sent to prison. In this manner, the question about the defendant’s parents acts as a *proxy* for race. Thus, even though the recidivism algorithm doesn’t take race into account directly, it learns about race from the data that reflects the centuries-old inequities in the criminal justice system.

For another example, suppose that this model for recidivism included interactions with the police as an important feature. It may seem logical to assume that people who have had more interactions with the police are more likely to commit crimes in the future. However, including this variable would likely lead to bias, since Black people are more likely to have interactions with police, even among those whose underlying probability of criminal behavior is the same (Gelman et al., 2007).

Data scientists need to ensure that model assessment, testing, accountability, and transparency are integrated into their analysis to identify and counteract bias and maximize fairness.

8.7 Data and disclosure

8.7.1 Reidentification and disclosure avoidance

The ability to link multiple data sets and to use public information to identify individuals is a growing problem. A glaring example of this occurred in 1996 when then-Governor of Massachusetts William Weld collapsed while attending a graduation ceremony at *Bentley College*. An *MIT* graduate student used information from a public data release by the *Massachusetts Group Insurance Commission* to identify Weld's subsequent hospitalization records.

The disclosure of this information was highly publicized and led to many changes in data releases. This was a situation where the right balance was not struck between disclosure (to help improve health care and control costs) and nondisclosure (to help ensure private information is not made public). There are many challenges to ensure disclosure avoidance (Zaslavsky and Horton, 1998; Ohm, 2010). This remains an active and important area of research.

The *Health Insurance Portability and Accountability Act* (HIPAA) was passed by the United States Congress in 1996—the same year as Weld's illness. The law augmented and clarified the role that researchers and medical care providers had in maintaining protected health information (PHI). The HIPAA regulations developed since then specify procedures to ensure that individually identifiable PHI is protected when it is transferred, received, handled, analyzed, or shared. As an example, detailed geographic information (e.g., home or office location) is not allowed to be shared unless there is an overriding need. For research purposes, geographic information might be limited to state or territory, though for certain rare diseases or characteristics even this level of detail may lead to disclosure. Those whose PHI is not protected can file a complaint with the Office of Civil Rights.

The HIPAA structure, while limited to medical information, provides a useful model for disclosure avoidance that is relevant to other data scientists. Parties accessing PHI need to have privacy policies and procedures. They must identify a privacy official and undertake training of their employees. If there is a disclosure they must mitigate the effects to the extent practical. There must be reasonable data safeguards to prevent intentional or unintentional use. Covered entities may not retaliate against someone for assisting in investigations of disclosures. Organizations must maintain records and documentation for six years after their last use of the data. Similar regulations protect information collected by the statistical agencies of the United States.

8.7.2 Safe data storage

Inadvertent disclosures of data can be even more damaging than planned disclosures. Stories abound of protected data being made available on the internet with subsequent harm to those whose information is made accessible. Such releases may be due to misconfigured databases, malware, theft, or by posting on a public forum. Each individual and organization needs to practice safe computing, to regularly audit their systems, and to implement plans to address computer and data security. Such policies need to ensure that protections remain even when equipment is transferred or disposed of.

8.7.3 Data scraping and terms of use

A different issue arises relating to legal status of material on the Web. Consider *Zillow.com*, an online real-estate database company that combines data from a number of public and private sources to generate house price and rental information on more than 100 million homes across the United States. Zillow has made access to their database available through an API (see [Section 6.4.2](#)) under certain restrictions. The terms of use for Zillow are provided in a legal document. They require that users of the API consider the data on an “as is” basis, not replicate functionality of the Zillow website or mobile app, not retain any copies of the Zillow data, not separately extract data elements to enhance other data files, and not use the data for direct marketing.

Another common form for terms of use is a limit to the amount or frequency of access. Zillow’s API is limited to 1,000 calls per day to the home valuations or property details. Another example: *The Weather Underground* maintains an API focused on weather information. They provide no-cost access limited to 500 calls per day and 10 calls per minute and with no access to historical information. They have a for-pay system with multiple tiers for accessing more extensive data.

Data points are not just content in tabular form. Text is also data. Many websites have restrictions on text mining. *Slate.com*, for example, states that users may not:

“Engage in unauthorized spidering, scraping, or harvesting of content or information, or use any other unauthorized automated means to compile information.”

Apparently, it violates the *Slate.com* terms of use to compile a compendium of *Slate* articles (even for personal use) without their authorization.

To get authorization, you need to ask for it. Albert Y. Kim of *Smith College* published data with information for 59,946 *San Francisco* OkCupid users (a free online dating website) with the permission of the president of OkCupid (Kim and Escobedo-Land, 2015). To help minimize possible damage, he also removed certain variables (e.g., username) that would make it more straightforward to reidentify the profiles. Contrast the concern for privacy taken here to the careless doxing of OkCupid users mentioned above.

8.8 Reproducibility

Disappointingly often, even the original researchers are unable to reproduce their own results upon revisit. This failure arises naturally enough when researchers use menu-driven software that does not keep an audit trail of each step in the process. For instance, in *Excel*, the process of sorting data is not recorded. You can't look at a spreadsheet and determine what range of data was sorted, so mistakes in selecting cases or variables for a sort are propagated untraceably through the subsequent analysis.

Researchers commonly use tools like word processors that do not mandate an explicit tie between the result presented in a publication and the analysis that produced the result. These seemingly innocuous practices contribute to the loss of reproducibility: numbers may be copied by hand into a document and graphics are cut-and-pasted into the report. (Imagine that you have inserted a graphic into a report in this way. How could you, or anyone else, easily demonstrate that the correct graphic was selected for inclusion?)

We describe *reproducible analysis* as the practice of recording each and every step, no matter how trivial seeming, in a data analysis. The main elements of a reproducible analysis plan (as described by Project TIER include:

- **Data:** all original data files in the form in which they originated,
- **Metadata:** codebooks and other information needed to understand the data,
- **Commands:** the computer code needed to extract, transform, and load the data—then run analyses, fit models, generate graphical displays, and
- **Map:** a file that maps between the output and the results in the report.

The *American Statistical Association* (ASA) notes the importance of reproducible analysis in its curricular guidelines. The development of new tools such as **R** Markdown and **knitr** have dramatically improved the usability of these methods in practice. See [Appendix D](#) for an introduction to these tools.

Individuals and organizations have been working to develop protocols to facilitate making the data analysis process more transparent and to integrate this into the workflow of practitioners and students. One of us has worked as part of a research project team at the *Channing Laboratory* at *Harvard University*. As part of the vetting process for all manuscripts, an analyst outside of the research team is required to review all programs used to generate results. In addition, another individual is responsible for checking each number in the paper to ensure that it was correctly transcribed from the results. Similar practice is underway at The Odum Institute for Research in Social Science at the *University of North Carolina*. This organization performs third-party code and data verification for several political science journals.

8.8.1 Example: Erroneous data merging

In [Chapter 5](#), we discuss how the *join* operation can be used to merge two data tables together. Incorrect merges can be very difficult to unravel unless the exact details of the merge have been recorded. The **dplyr inner_join()** function simplifies this process.

In a 2013 paper published in the journal *Brain, Behavior, and Immunity*, Kern et al. reported a link between immune response and depression. To their credit, the authors later noticed that the results were the artifact of a faulty data merge between the lab results and other

survey data. A retraction (Kern et al., 2013), as well as a corrected paper reporting negative results (Kern et al., 2014), was published in the same journal.

In some ways, this is science done well—ultimately the correct negative result was published, and the authors acted ethically by alerting the journal editor to their mistake. However, the error likely would have been caught earlier had the authors adhered to stricter standards of reproducibility (see [Appendix D](#)) in the first place.

8.9 Ethics, collectively

Although science is carried out by individuals and teams, the scientific community as a whole is a stakeholder. Some of the ethical responsibilities faced by data scientists are created by the collective nature of the enterprise.

A team of *Columbia University* scientists discovered that a former post-doc in the group, unbeknownst to the others, had fabricated and falsified research reported in articles in the journals *Cell* and *Nature*. Needless to say, the post-doc had violated his ethical obligations both with respect to his colleagues and to the scientific enterprise as a whole. When the misconduct was discovered, the other members of the team incurred an ethical obligation to the scientific community. In fulfillment of this obligation, they notified the journals and retracted the papers, which had been highly cited. To be sure, such episodes can tarnish the reputation of even the innocent team members, but the ethical obligation outweighs the desire to protect one's reputation.

Perhaps surprisingly, there are situations where it is not ethical *not* to publish one's work. *Publication bias* (or the “file-drawer problem”) refers to the situation where reports of statistically significant (i.e., $p < 0.05$) results are much more likely to be published than reports where the results are not statistically significant. In many settings, this bias is for the good; a lot of scientific work is in the pursuit of hypotheses that turn out to be wrong or ideas that turn out not to be productive.

But with many research teams investigating similar ideas, or even with a single research team that goes down many parallel paths, the meaning of “statistically significant” becomes clouded and corrupt. Imagine 100 parallel research efforts to investigate the effect of a drug that in reality has no effect at all. Roughly five of those efforts are expected to culminate in a misleadingly “statistically significant” ($p < 0.05$) result. Combine this with publication bias and the scientific literature might consist of reports on just the five projects that happened to be significant. In isolation, five such reports would be considered substantial evidence about the (non-null) effect of the drug. It might seem unlikely that there would be 100 parallel research efforts on the same drug, but at any given time there are tens of thousands of research efforts, any one of which has a 5% chance of producing a significant result even if there were no genuine effect.

The *American Statistical Association*'s ethical guidelines state, “Selecting the one ‘significant’ result from a multiplicity of parallel tests poses a grave risk of an incorrect conclusion. Failure to disclose the full extent of tests and their results in such a case would be highly misleading.” So, if you're examining the effect on five different measures of health by five different foods, and you find that broccoli consumption has a statistically significant relationship with the development of colon cancer, not only should you be skeptical but you should include in your report the null result for the other 24 tests or perform an appropriate

statistical correction to account for the multiple tests. Often, there may be several different outcome measures, several different food types, and several potential covariates (age, sex, whether breastfed as an infant, smoking, the geographical area of residence or upbringing, etc.), so it's easy to be performing dozens or hundreds of different tests without realizing it.

For clinical health trials, there are efforts to address this problem through trial registries. In such registries (e.g., <https://clinicaltrials.gov>), researchers provide their study design and analysis protocol in advance and post results.

8.10 Professional guidelines for ethical conduct

This chapter has outlined basic principles of professional ethics. Usefully, several organizations have developed detailed statements on topics such as professionalism, integrity of data and methods, responsibilities to stakeholders, conflicts of interest, and the response to allegations of misconduct. One good source is the framework for professional ethics endorsed by the *American Statistical Association* (ASA) (Committee on Professional Ethics, 1999).

The Committee on Science, Engineering, and Public Policy of the National Academy of Sciences, National Academy of Engineering, and Institute of Medicine has published the third edition of *On Being a Scientist: A Guide to Responsible Conduct in Research*. The guide is structured into a number of chapters, many of which are highly relevant for data scientists (including “the Treatment of Data,” “Mistakes and Negligence,” “Sharing of Results,” “Competing Interests, Commitment, and Values,” and “The Researcher in Society”).

The *Association for Computing Machinery* (ACM)—the world’s largest computing society, with more than 100,000 members—adopted a code of ethics in 1992 that was revised in 2018 (see <https://www.acm.org/about/code-of-ethics>). Other relevant statements and codes of conduct have been promulgated by the Data Science Association, the International Statistical Institute, and the United Nations Statistics Division. The Belmont Report outlines ethical principles and guidelines for the protection of human research subjects.

8.11 Further resources

For a book-length treatment of ethical issues in statistics, see Hubert and Wainer (2012). The National Academies report on data science for undergraduates National Academies of Science, Engineering, and Medicine (2018) included data ethics as a key component of data acumen. The report also included a draft oath for data scientists.

A historical perspective on the ASA’s Ethical Guidelines for Statistical Practice can be found in Ellenberg (1983). The University of Michigan provides an EdX course on “Data Science Ethics.” Carl Bergstrom and Jevin West developed a course “Calling Bullshit: Data Reasoning in a Digital World”. Course materials and related resources can be found at <https://callingbullshit.org>.

Andrew Gelman has written a column on ethics in statistics in *CHANCE* for the past several years (see, for example Gelman (2011); Gelman and Loken (2012); Gelman (2012); Gelman

(2020)). *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy* describes a number of frightening misuses of big data and algorithms (O’Neil, 2016).

The *Teach Data Science* blog has a series of entries focused on data ethics (<https://teachdatascience.com>). D’Ignazio and Klein (2020) provide a comprehensive introduction to data feminism (in contrast to data ethics). The ACM Conference on Fairness, Accountability, and Transparency (FAccT) provides a cross-disciplinary focus on data ethics issues (<https://facctconference.org/2020>).

The Center for Open Science—which develops the Open Science Framework (OSF)—is an organization that promotes openness, integrity, and reproducibility in scientific research. The OSF provides an online platform for researchers to publish their scientific projects. Emil Kirkegaard used OSF to publish his OkCupid data set.

The Institute for Quantitative Social Science at Harvard and the Berkeley Initiative for Transparency in the Social Sciences are two other organizations working to promote reproducibility in social science research. The *American Political Association* has incorporated the Data Access and Research Transparency (DA-RT) principles into its ethics guide. The Consolidated Standards of Reporting Trials (CONSORT) statement at (<http://www.consort-statement.org>) provides detailed guidance on the analysis and reporting of clinical trials.

Many more examples of how irreproducibility has led to scientific errors are available at <http://retractionwatch.com/>. For example, a study linking severe illness and divorce rates was retracted due to a coding mistake.

8.12 Exercises

Problem 1 (Easy): A researcher is interested in the relationship of weather to sentiment (positivity or negativity of posts) on Twitter. They want to scrape data from <https://www.wunderground.com> and join that to Tweets in that geographic area at a particular time. One complication is that Weather Underground limits the number of data points that can be downloaded for free using their API (application program interface). The researcher sets up six free accounts to allow them to collect the data they want in a shorter time-frame. What ethical guidelines are violated by this approach to data scraping?

Problem 2 (Medium): A data scientist compiled data from several public sources (voter registration, political contributions, tax records) that were used to predict sexual orientation of individuals in a community. What ethical considerations arise that should guide use of such data sets?

Problem 3 (Medium): A statistical analyst carried out an investigation of the association of gender and teaching evaluations at a university. They undertook exploratory analysis of the data and carried out a number of bivariate comparisons. The multiple items on the teaching evaluation were consolidated to a single measure based on these exploratory analyses. They used this information to construct a multivariable regression model that found evidence for biases. What issues might arise based on such an analytic approach?

Problem 4 (Medium): In 2006, AOL released a database of search terms that users had used in the prior month (see <http://www.nytimes.com/2006/08/09/technology/09aol.html>).

Research this disclosure and the reaction that ensued. What ethical issues are involved? What potential impact has this disclosure had?

Problem 5 (Medium): A reporter carried out a clinical trial of chocolate where a small number of overweight subjects who had received medical clearance were randomized to either eat dark chocolate or not to eat dark chocolate. They were followed for a period and their change in weight was recorded from baseline until the end of the study. More than a dozen outcomes were recorded and one proved to be significantly different in the treatment group than the outcome. This study was publicized and received coverage from a number of magazines and television programs. Outline the ethical considerations that arise in this situation.

Problem 6 (Medium): A *Slate* article (<http://tinyurl.com/slate-ethics>) discussed whether race/ethnicity should be included in a predictive model for how long a homeless family would stay in homeless services. Discuss the ethical considerations involved in whether race/ethnicity should be included as a predictor in the model.

Problem 7 (Medium): In the United States, the Confidential Information Protection and Statistical Efficiency Act (CIPSEA) governs the confidentiality of data collected by agencies such as the Bureau of Labor Statistics and the Census Bureau. What are the penalties for willful and knowing disclosure of protected information to unauthorized persons?

Problem 8 (Medium): A data analyst received permission to post a data set that was scraped from a social media site. The full data set included name, screen name, email address, geographic location, IP (internet protocol) address, demographic profiles, and preferences for relationships. Why might it be problematic to post a deidentified form of this data set where name and email address were removed?

Problem 9 (Medium): A company uses a machine-learning algorithm to determine which job advertisement to display for users searching for technology jobs. Based on past results, the algorithm tends to display lower-paying jobs for women than for men (after controlling for other characteristics than gender). What ethical considerations might be considered when reviewing this algorithm?

Problem 10 (Hard): An investigative team wants to winnow the set of variables to include in their final multiple regression model. They have 100 variables and one outcome measured for $n = 250$ observations).

They use the following procedure:

1. Fit each of the 100 bivariate models for the outcome as a function of a single predictor, then
2. Include all of the significant predictors in the overall model.

What does the distribution of the p-value for the overall test look like, assuming that there are no associations between any of the predictors and the outcome (all are assumed to be multivariate normal and independent). Carry out a simulation to check your answer.

8.13 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-ethics.html#ethics-online-exercises>



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

Part II

Part II: Statistics and Modeling



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

Statistical foundations

The ultimate objective in data science is to extract meaning from data. Data wrangling and visualization are tools to this end. Wrangling re-organizes cases and variables to make data easier to interpret. Visualization is a primary tool for connecting our minds with the data, so that we humans can search for meaning.

Visualizations are powerful because human visual cognitive skills are strong. We are very good at seeing patterns even when partially obscured by random noise. On the other hand, we are also very good at seeing patterns even when they are not there. People can easily be misled by the accidental, evanescent patterns that appear in random noise. It's important, therefore, to be able to discern when the patterns we see are so strong and robust that we can be confident they are not mere accidents.

Statistical methods quantify patterns and their strength. They are essential tools for interpreting data. As we'll see later in this book, the methods are also crucial for finding patterns that are too complex or multi-faceted to be seen visually.

Some people think that *big data* has made statistics obsolete. The argument is that with lots of data, the data can speak clearly for themselves. This is wrong, as we shall see. The discipline for making efficient use of data that is a core of statistical methodology leads to deeper thinking about how to make use of data—that thinking applies to large data sets as well.

In this chapter, we will introduce key ideas from statistics that permeate data science and that will be reinforced later in the book. At the same time, the extended example used in this chapter will illustrate a data science *workflow* that uses a cycle of wrangling, exploring, visualizing, and modeling.

9.1 Samples and populations

In previous chapters, we've considered data as being fixed. Indeed, the word “data” stems from the Latin word for “given”—any set of data is treated as given.

Statistical methodology is governed by a broader point of view. Yes, the data we have in hand are fixed, but the methodology assumes that the cases are drawn from a much larger set of potential cases. The given data are a *sample* of a larger *population* of potential cases. In statistical methodology, we view our sample of cases in the context of this population. We imagine other samples that might have been drawn from the population.

At the same time, we imagine that there might have been additional variables that could have been measured from the population. We permit ourselves to construct new variables that have a special feature: any patterns that appear involving the new variables are guar-

anted to be random and accidental. The tools we will use to gain access to the imagined cases from the population and the contrived no-pattern variables involve the mathematics of probability or (more simply) random selection from a set.

In the next section, we'll elucidate some of the connections between the sample—the data we've got—and the population. To do this, we'll use an artifice: constructing a playground that contains the entire population. Then, we can work with data consisting of a smaller set of cases selected at random from this population. This lets us demonstrate and justify the statistical methods in a setting where we know the “correct” answer. That way, we can develop ideas about how much confidence statistical methods can give us about the patterns we see.

9.1.1 Example: Setting travel policy by sampling from the population

Suppose you were asked to help develop a travel policy for business travelers based in *New York City*. Imagine that the traveler has a meeting in *San Francisco* (airport code SFO) at a specified time t . The policy to be formulated will say how much earlier than t an acceptable flight should arrive in order to avoid being late to the meeting due to a flight delay.

For the purpose of this example, recall from the previous section that we are going to pretend that we already have on hand the complete *population* of flights. For this purpose, we're going to use the subset of 336,776 flights in 2013 in the **nycflights13** package, which gives airline delays from New York City airports in 2013. The policy we develop will be for 2013. Of course this is unrealistic in practice. If we had the complete population we could simply look up the best flight that arrived in time for the meeting!

More realistically, the problem would be to develop a policy for this year based on the sample of data that have already been collected. We're going to simulate this situation by drawing a sample from the population of flights into SFO. Playing the role of the population in our little drama, SF comprises the complete collection of such flights.

```
library(tidyverse)
library(mdsr)
library(nycflights13)
SF <- flights %>%
  filter(dest == "SFO", !is.na(arr_delay))
```

We're going to work with just a sample from this population. For now, we'll set the sample size to be $n = 25$ cases.

```
set.seed(101)
sf_25 <- SF %>%
  slice_sample(n = 25)
```

A simple (but naïve) way to set the policy is to look for the longest flight delay and insist that travel be arranged to deal with this delay.

```
sf_25 %>%
  skim(arr_delay)

-- Variable type: numeric --
#>   var      n    na  mean     sd    p0    p25    p50    p75    p100
#> 1 arr_delay  25     0  0.12  29.8   -38   -23     -5     14    103
```

The maximum delay is 103 minutes, about 2 hours. So, should our travel policy be that the

traveler should plan on arriving in SFO about 2 hours ahead? In our example world, we can look at the complete set of flights to see what was the actual worst delay in 2013.

```
SF %>%
  skim(arr_delay)

-- Variable type: numeric -----
#>   var      n    na   mean     sd    p0    p25    p50    p75   p100
#> 1 arr_delay 13173     0  2.67  47.7   -86   -23     -8     12  1007
```

Notice that the results from the sample are different from the results for the population. In the population, the longest delay was 1,007 minutes—almost 17 hours. This suggests that to avoid missing a meeting, you should travel the day before the meeting. Safe enough, but then:

- an extra travel day is expensive in terms of lodging, meals, and the traveler’s time;
- even at that, there’s no guarantee that there will never be a delay of more than 1,007 minutes.

A sensible travel policy will trade off small probabilities of being late against the savings in cost and traveler’s time. For instance, you might judge it acceptable to be late just 2% of the time—a 98% chance of being on time.

Here’s the 98th percentile of the arrival delays in our data sample:

```
sf_25 %>%
  summarize(q98 = quantile(arr_delay, p = 0.98))

# A tibble: 1 x 1
#>   q98
#>   <dbl>
#> 1 67.5
```

A delay of 68 minutes is more than an hour. The calculation is easy, but how good is the answer? This is not a question about whether the 98th percentile was calculated properly—that will always be the case for any competent data scientist. The question is really along these lines: Suppose we used a 90-minute travel policy. How well would that have worked in achieving our intention to be late for meetings only 2% of the time?

With the population data in hand, it’s easy to answer this question.

```
SF %>%
  group_by(arr_delay < 90) %>%
  count() %>%
  mutate(pct = n / nrow(SF))

# A tibble: 2 x 3
# Groups:   arr_delay < 90 [2]
#>   `arr_delay < 90`     n     pct
#>   <lgl>           <int>  <dbl>
#> 1 FALSE            640  0.0486
#> 2 TRUE             12533 0.951
```

The 90-minute policy would miss its mark 5% of the time, much worse than we intended. To correctly hit the mark 2% of the time, we will want to increase the policy from 90 minutes to what value?

With the population, it's easy to calculate the 98th percentile of the arrival delays:

```
SF %>%
  summarize(q98 = quantile(arr_delay, p = 0.98))

# A tibble: 1 × 1
  q98
  <dbl>
1 153
```

It should have been about 150 minutes.

But in most real-world settings, we do not have access to the population data. We have only our sample. How can we use our sample to judge whether the result we get from the sample is going to be good enough to meet the 98% goal? And if it's not good enough, how large should a sample be to give a result that is likely to be good enough? This is where the concepts and methods from statistics come in.

We will continue exploring this example throughout the chapter. In addition to addressing our initial question, we'll examine the extent to which the policy should depend on the airline carrier, the time of year, hour of day, and day of the week.

The basic concepts we'll build on are sample statistics such as the *mean* and *standard deviation*. These topics are covered in introductory statistics books. Readers who have not yet encountered these topics should review an introductory statistics text such as the OpenIntro Statistics books (<http://openintro.org>), Appendix E, or the materials in Section 9.8 (Further resources).

9.2 Sample statistics

Statistics (plural) is a field that overlaps with and contributes to data science. A *statistic* (singular) is a number that summarizes data. Ideally, a statistic captures all of the useful information from the individual observations.

When we calculate the 98th percentile of a sample, we are calculating one of many possible sample statistics. Among the many sample statistics are the mean of a variable, the standard deviation, the *median*, the maximum, and the minimum. It turns out that sample statistics such as the maximum and minimum are not very useful. The reason is that there is not a reliable (or *robust*) way to figure out how well the sample statistic reflects what is going on in the population. Similarly, the 98th percentile is not a reliable sample statistic for small samples (such as our 25 flights into SFO), in the sense that it will vary considerably in small samples.

On the other hand, a median is a more reliable sample statistic. Under certain conditions, the mean and standard deviation are reliable as well. In other words, there are established techniques for figuring out—from the sample itself—how well the sample statistic reflects the population.

9.2.1 The sampling distribution

Ultimately we need to figure out the reliability of a sample statistic from the sample itself. For now, though, we are going to use the population to develop some ideas about how to

define reliability. So we will still be in the playground world where we have the population in hand.

If we were to collect a new sample from the population, how similar would the sample statistic on that new sample be to the same statistic calculated on the original sample? Or, stated somewhat differently, if we draw many different samples from the population, each of size n , and calculated the sample statistic on each of those samples, how similar would the sample statistic be across all the samples?

With the population in hand, it's easy to figure this out; use `slice_sample()` many times and calculate the sample statistic on each trial. For instance, here are two trials in which we sample and calculate the mean arrival delay. (We'll explain the `replace = FALSE` in the next section. Briefly, it means to draw the sample as one would deal from a set of cards: None of the cards can appear twice in one hand.)

```
n <- 25
SF %>%
  slice_sample(n = n) %>%
  summarize(mean_arr_delay = mean(arr_delay))
```

```
# A tibble: 1 x 1
  mean_arr_delay
  <dbl>
1      8.32
SF %>%
  slice_sample(n = n) %>%
  summarize(mean_arr_delay = mean(arr_delay))
```

```
# A tibble: 1 x 1
  mean_arr_delay
  <dbl>
1      19.8
```

Perhaps it would be better to run many trials (though each one would require considerable effort in the real world). The `map()` function from the **purrr** package (see [Chapter 7](#)) lets us automate the process. Here are the results from 500 trials.

```
num_trials <- 500
sf_25_means <- 1:num_trials %>%
  map_dfr(
    ~ SF %>%
      slice_sample(n = n) %>%
      summarize(mean_arr_delay = mean(arr_delay)))
  ) %>%
  mutate(n = n)

head(sf_25_means)
```

```
# A tibble: 6 x 2
  mean_arr_delay     n
  <dbl> <dbl>
1     -3.64     25
2      1.08     25
3     16.2      25
```

```
4      -2.64    25
5       0.4     25
6      8.04    25
```

We now have 500 trials, for each of which we calculated the mean arrival delay. Let's examine how spread out the results are.

```
sf_25_means %>%
  skim(mean_arr_delay)

-- Variable type: numeric -----
var          n    na   mean     sd    p0    p25    p50    p75    p100
1 mean_arr_delay  500     0  1.78  9.22 -17.2 -4.37  0.76  7.36  57.3
```

To discuss reliability, it helps to have some standardized vocabulary.

- The *sample size* is the number of cases in the sample, usually denoted with n . In the above, the sample size is $n = 25$.
- The *sampling distribution* is the collection of the sample statistic from all of the trials. We carried out 500 trials here, but the exact number of trials is not important so long as it is large.
- The *shape* of the sampling distribution is worth noting. Here it is a little skewed to the right. We can tell because in this case the mean is more than twice the median.
- The *standard error* is the standard deviation of the sampling distribution. It describes the width of the sampling distribution. For the trials calculating the sample mean in samples with $n = 25$, the standard error is 9.22 minutes. (You can see this value in the output of `skim()` above, as the standard deviation of the sample means that we generated.)
- The 95% *confidence interval* is another way of summarizing the sampling distribution. From [Figure 9.1](#) (left panel) you can see it is about -16 to $+20$ minutes. The interval can be used to identify plausible values for the true mean arrival delay. It is calculated from the mean and standard error of the sampling distribution.

```
sf_25_means %>%
  summarize(
    x_bar = mean(mean_arr_delay),
    se = sd(mean_arr_delay)
  ) %>%
  mutate(
    ci_lower = x_bar - 2 * se, # approximately 95% of observations
    ci_upper = x_bar + 2 * se # are within two standard errors
  )
```

```
# A tibble: 1 x 4
  x_bar     se ci_lower ci_upper
  <dbl>  <dbl>    <dbl>    <dbl>
1  1.78   9.22   -16.7    20.2
```

Alternatively, it can be calculated directly using a *t-test*.

```
sf_25_means %>%
  pull(mean_arr_delay) %>%
  t.test()
```

```
One Sample t-test

data: .
t = 4, df = 499, p-value = 2e-05
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
0.969 2.590
sample estimates:
mean of x
1.78
```

Pro Tip 27. This vocabulary can be very confusing at first. Remember that “standard error” and “confidence interval” always refer to the sampling distribution, not to the population and not to a single sample. The standard error and confidence intervals are two different, but closely related, forms for describing the reliability of the calculated sample statistic.

An important question that statistical methods allow you to address is what size of sample n is needed to get a result with an acceptable reliability. What constitutes “acceptable” depends on the goal you are trying to accomplish. But measuring the reliability is a straightforward matter of finding the standard error and/or confidence interval.

Notice that the sample statistic varies considerably. For samples of size $n = 25$ they range from -17 to 57 minutes. This is important information. It illustrates the reliability of the sample mean for samples of arrival delays of size $n = 25$. Figure 9.1 (left) shows the distribution of the trials with a histogram.

In this example, we used a sample size of $n = 25$ and found a standard error of 9.2 minutes. What would happen if we used an even larger sample, say $n = 100$? The calculation is the same as before but with a different n .

```
n <- 100
sf_100_means <- 1:500 %>%
  map_dfr(
    ~ SF %>%
      slice_sample(n = n) %>%
      summarize(mean_arr_delay = mean(arr_delay)))
  ) %>%
  mutate(n = n)

sf_25_means %>%
  bind_rows(sf_100_means) %>%
  ggplot(aes(x = mean_arr_delay)) +
  geom_histogram(bins = 30) +
  facet_grid(~ n) +
  xlab("Sample mean")
```

Figure 9.1 (right panel) displays the shape of the sampling distribution for samples of size $n = 25$ and $n = 100$. Comparing the two sampling distributions, one with $n = 25$ and the other with $n = 100$ shows some patterns that are generally true for statistics such as the mean:

- Both sampling distributions are centered at the same value.
- A larger sample size produces a standard error that is smaller. That is, a larger sample

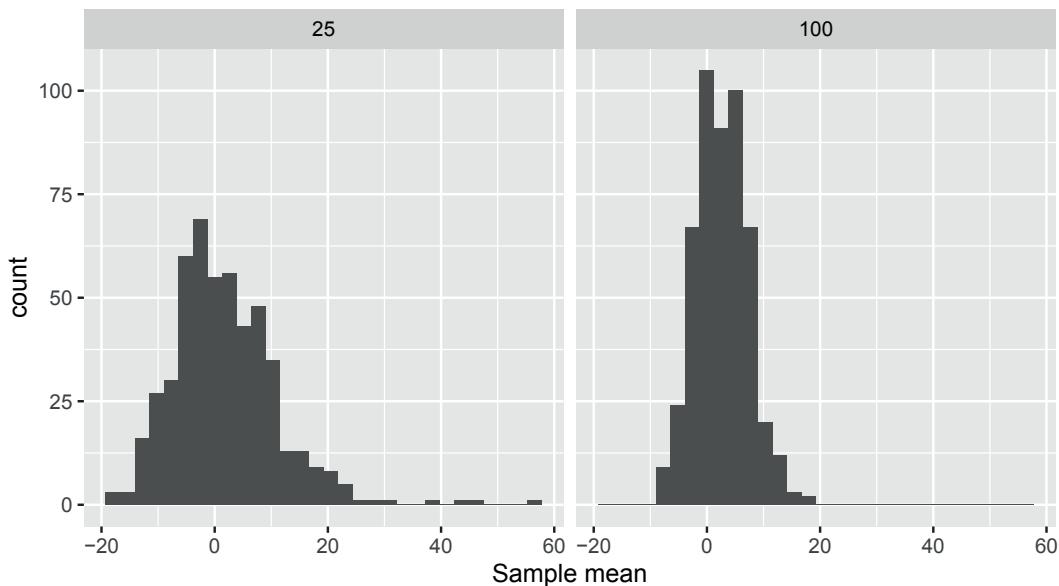


Figure 9.1: The sampling distribution of the mean arrival delay with a sample size of $n = 25$ (left) and also for a larger sample size of $n = 100$ (right). Note that the sampling distribution is less variable for a larger sample size.

size is more reliable than a smaller sample size. You can see that the standard deviation for $n = 100$ is one-half that for $n = 25$. As a rule, the standard error of a sampling distribution scales as $1/\sqrt{n}$.

- For large sample sizes, the shape of the sampling distribution tends to bell-shaped. In a bit of archaic terminology, this shape is often called the *normal distribution*. Indeed, the distribution arises very frequently in statistics, but there is nothing abnormal about any other distribution shape.

9.3 The bootstrap

In the previous examples, we had access to the population data and so we could find the sampling distribution by repeatedly sampling from the population. In practice, however, we have only one sample and not the entire population. The *bootstrap* is a statistical method that allows us to approximate the sampling distribution even without access to the population.

The logical leap involved in the bootstrap is to think of our sample itself as if it were the population. Just as in the previous examples we drew many samples from the population, now we will draw many new samples from our original sample. This process is called *resampling*: drawing a new sample from an existing sample.

When sampling from a population, we would of course make sure not to duplicate any of the cases, just as we would never deal the same playing card twice in one hand. When resampling, however, we do allow such duplication (in fact, this is what allows us to estimate the variability of the sample). Therefore, we *sample with replacement*.

To illustrate, consider `three_flights`, a very small sample ($n = 3$) from the flights data. Notice that each of the cases in `three_flights` is unique. There are no duplicates.

```
three_flights <- SF %>%
  slice_sample(n = 3, replace = FALSE) %>%
  select(year, month, day, dep_time)
three_flights
```

```
# A tibble: 3 x 4
  year month   day dep_time
  <int> <int> <int>    <int>
1 2013     11     4      726
2 2013      3    12      734
3 2013      3    25     1702
```

Resampling from `three_flights` is done by setting the `replace` argument to `TRUE`, which allows the sample to include duplicates.

```
three_flights %>% slice_sample(n = 3, replace = TRUE)
```

```
# A tibble: 3 x 4
  year month   day dep_time
  <int> <int> <int>    <int>
1 2013     3    25     1702
2 2013     11     4      726
3 2013     3    12      734
```

In this particular resample, each of the individual cases appear once (but in a different order). That's a matter of luck. Let's try again.

```
three_flights %>% slice_sample(n = 3, replace = TRUE)
```

```
# A tibble: 3 x 4
  year month   day dep_time
  <int> <int> <int>    <int>
1 2013     3    12      734
2 2013     3    12      734
3 2013     3    25     1702
```

This resample has two instances of one case and a single instance of another.

Bootstrapping does not create new cases: It isn't a way to collect data. In reality, constructing a sample involves genuine data acquisition, e.g., field work or lab work or using information technology systems to consolidate data. In this textbook example, we get to save all that effort and simply select at random from the population, `SF`. The one and only time we use the population is to draw the original sample, which, as always with a sample, we do without replacement.

Let's use bootstrapping to estimate the reliability of the mean arrival time calculated on a sample of size 200. (Ordinarily this is all we get to observe about the population.)

```
n <- 200
orig_sample <- SF %>%
  slice_sample(n = n, replace = FALSE)
```

Now, with this sample in hand, we can draw a resample (of that sample size) and calculate the mean arrival delay.

```
orig_sample %>%
  slice_sample(n = n, replace = TRUE) %>%
  summarize(mean_arr_delay = mean(arr_delay))

# A tibble: 1 x 1
  mean_arr_delay
  <dbl>
1       6.80
```

By repeating this process many times, we'll be able to see how much variation there is from sample to sample:

```
sf_200_bs <- 1:num_trials %>%
  map_dfr(
    ~orig_sample %>%
      slice_sample(n = n, replace = TRUE) %>%
      summarize(mean_arr_delay = mean(arr_delay))
  ) %>%
  mutate(n = n)

sf_200_bs %>%
  skim(mean_arr_delay)

-- Variable type: numeric -----
var          n     na   mean     sd    p0    p25    p50    p75   p100
1 mean_arr_delay  500      0  3.05  3.09 -5.03  1.01     3  5.14  13.1
```

We could estimate the standard deviation of the arrival delays to be about 3.1 minutes.

Ordinarily, we wouldn't be able to check this result. But because we have access to the population data in this example, we can. Let's compare our bootstrap estimate to a set of (hypothetical) samples of size $n = 200$ from the original SF flights (the population).

```
sf_200_pop <- 1:num_trials %>%
  map_dfr(
    ~SF %>%
      slice_sample(n = n, replace = TRUE) %>%
      summarize(mean_arr_delay = mean(arr_delay))
  ) %>%
  mutate(n = n)

sf_200_pop %>%
  skim(mean_arr_delay)

-- Variable type: numeric -----
var          n     na   mean     sd    p0    p25    p50    p75   p100
1 mean_arr_delay  500      0  2.59  3.34 -5.90  0.235  2.51  4.80  14.2
```

Notice that the population was not used in the bootstrap (`sf_200_bs`), just the original sample. What's remarkable here is that the standard error calculated using the bootstrap (3.1 minutes) is a reasonable approximation to the standard error of the sampling distribution calculated by taking repeated samples from the population (3.3 minutes).

The distribution of values in the bootstrap trials is called the *bootstrap distribution*. It's not exactly the same as the sampling distribution, but for moderate to large sample sizes

and sufficient number of bootstraps it has been proven to approximate those aspects of the sampling distribution that we care most about, such as the standard error and quantiles (Efron and Tibshirani, 1993).

9.3.1 Example: Setting travel policy

Let's return to our original example of setting a travel policy for selecting flights from New York to San Francisco. Recall that we decided to set a goal of arriving in time for the meeting 98% of the time. We can calculate the 98th percentile from our sample of size $n = 200$ flights, and use bootstrapping to see how reliable that sample statistic is.

The sample itself suggests a policy of scheduling a flight to arrive 141 minutes early.

```
orig_sample %>%
  summarize(q98 = quantile(arr_delay, p = 0.98))

# A tibble: 1 x 1
  q98
  <dbl>
1 141.
```

We can check the reliability of that estimate using bootstrapping.

```
n <- nrow(orig_sample)
sf_200_bs <- 1:num_trials %>%
  map_dfr(
    ~orig_sample %>%
      slice_sample(n = n, replace = TRUE) %>%
      summarize(q98 = quantile(arr_delay, p = 0.98))
  )

sf_200_bs %>%
  skim(q98)

-- Variable type: numeric -----
var     n     na   mean    sd    p0    p25    p50    p75    p100
1 q98     500      0 140.  29.2  53.0  123.   141   154.   196.
```

The bootstrapped standard error is about 29 minutes. The corresponding 95% confidence interval is 140 ± 58 minutes. A policy based on this would be practically a shot in the dark: unlikely to hit the target.

One way to fix things might be to collect more data, hoping to get a more reliable estimate of the 98th percentile. Imagine that we could do the work to generate a sample with $n = 10,000$ cases.

```
set.seed(1001)
n_large <- 10000
sf_10000_bs <- SF %>%
  slice_sample(n = n_large, replace = FALSE)

sf_200_bs <- 1:num_trials %>%
  map_dfr(~sf_10000_bs %>%
    slice_sample(n = n_large, replace = TRUE) %>%
    summarize(q98 = quantile(arr_delay, p = 0.98)))
```

```
)
sf_200_bs %>%
  skim(q98)

-- Variable type: numeric -----
#> var      n     na   mean    sd    p0    p25    p50    p75    p100
#> q98     500      0 154.  4.14 139.  151.  153.  156.   169
```

The standard deviation is much narrower, 154 ± 8 minutes. Having more data makes it easier to better refine estimates, particularly in the tails.

9.4 Outliers

One place where more data is helpful is in identifying unusual or extreme events: *outliers*. Suppose we consider any flight delayed by 7 hours (420 minutes) or more as an extreme event (see [Section 15.5](#)). While an arbitrary choice, 420 minutes may be valuable as a marker for seriously delayed flights.

```
SF %>%
  filter(arr_delay >= 420) %>%
  select(month, day, dep_delay, arr_delay, carrier)

# A tibble: 7 x 5
#> month  day dep_delay arr_delay carrier
#> <int> <int>    <dbl>     <dbl> <chr>
#> 1     12     7       374      422  UA
#> 2      7     6       589      561  DL
#> 3      7     7       629      676  VX
#> 4      7     7       653      632  VX
#> 5      7    10       453      445  B6
#> 6      7    10       432      433  VX
#> 7      9    20      1014     1007 AA
```

Most of the very long delays (five of seven) were in July, and *Virgin America* (vx) is the most frequent offender. Immediately, this suggests one possible route for improving the outcome of the business travel policy we have been asked to develop. We could tell people to arrive extra early in July and to avoid vx.

But let's not rush into this. The outliers themselves may be misleading. These outliers account for a tiny fraction of the flights into San Francisco from New York in 2013. That's a small component of our goal of having a failure rate of 2% in getting to meetings on time. And there was an even more extremely rare event at SFO in July 2013: the crash-landing of Asiana Airlines flight 214. We might remove these points to get a better sense of the main part of the distribution.

Pro Tip 28. *Outliers can often tell us interesting things. How they should be handled depends on their cause. Outliers due to data irregularities or errors should be fixed. Other outliers may yield important insights. Outliers should never be dropped unless there is a clear rationale. If outliers are dropped this should be clearly reported.*

Figure 9.2 displays the histogram without those outliers.

```
SF %>%
  filter(arr_delay < 420) %>%
  ggplot(aes(arr_delay)) +
  geom_histogram(binwidth = 15) +
  labs(x = "Arrival delay (in minutes)")
```

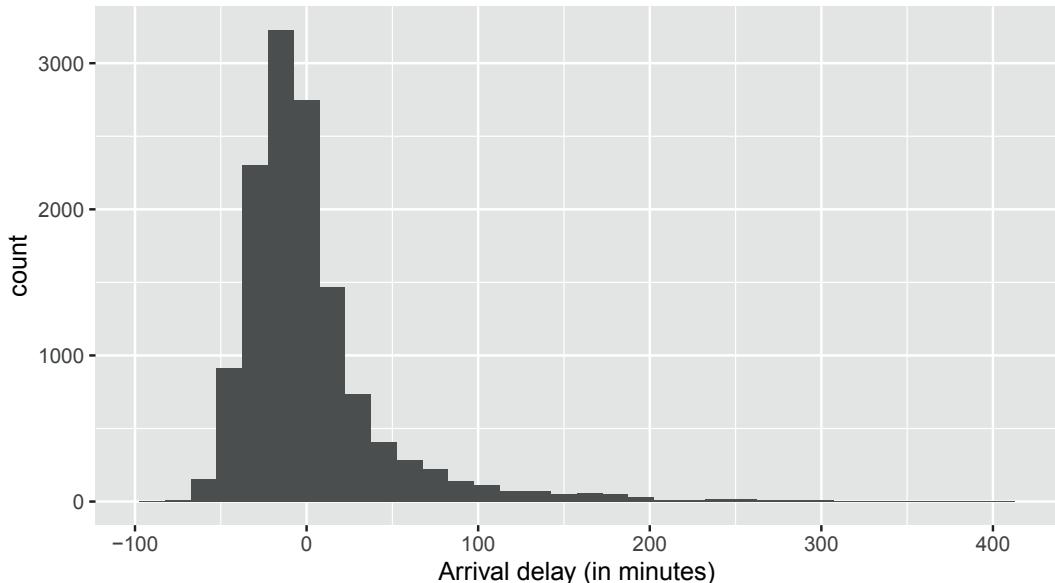


Figure 9.2: Distribution of flight arrival delays in 2013 for flights to San Francisco from NYC airports that were delayed less than 7 hours. The distribution features a long right tail (even after pruning the outliers).

Note that the large majority of flights arrive without any delay or a delay of less than 60 minutes. Might we be able to identify patterns that can presage when the longer delays are likely to occur? The outliers suggested that `month` or `carrier` may be linked to long delays. Let's see how that plays out with the large majority of data.

```
SF %>%
  mutate(long_delay = arr_delay > 60) %>%
  group_by(month, long_delay) %>%
  count() %>%
  pivot_wider(names_from = month, values_from = n) %>%
  data.frame()
```

	long_delay	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12
1	FALSE	856	741	812	993	1128	980	966	1159	1124	1177	1107	1093
2	TRUE	29	21	61	112	65	209	226	96	65	36	51	66

We see that June and July (months 6 and 7) are problem months.

```
SF %>%
  mutate(long_delay = arr_delay > 60) %>%
  group_by(carrier, long_delay) %>%
  count() %>%
```

```

pivot_wider(names_from = carrier, values_from = n) %>%
data.frame()

  long_delay   AA   B6    DL    UA    VX
1      FALSE 1250  934 1757 6236 1959
2      TRUE   148   86   91  492  220

```

Delta Airlines (DL) has reasonable performance. These two simple analyses hint at a policy that might advise travelers to plan to arrive extra early in June and July and to consider Delta as an airline for travel to SFO (see [Section 15.5](#) for a fuller discussion of which airlines seem to have fewer delays in general).

9.5 Statistical models: Explaining variation

In the previous section, we used month of the year and airline to narrow down the situations in which the risk of an unacceptable flight delay is large. Another way to think about this is that we are *explaining* part of the variation in arrival delay from flight to flight. *Statistical modeling* provides a way to relate variables to one another. Doing so helps us better understand the system we are studying.

To illustrate modeling, let's consider another question from the airline delays data set: What impact, if any, does scheduled time of departure have on expected flight delay? Many people think that earlier flights are less likely to be delayed, since flight delays tend to cascade over the course of the day. Is this theory supported by the data?

We first begin by considering time of day. In the **nycflights13** package, the **flights** data frame has a variable (**hour**) that specifies the *scheduled* hour of departure.

```

SF %>%
  group_by(hour) %>%
  count() %>%
  pivot_wider(names_from = hour, values_from = n) %>%
  data.frame()

```

```

X5   X6   X7   X8   X9   X10  X11  X12  X13  X14  X15  X16   X17  X18  X19  X20  X21
1 55 663 1696 987 429 1744 413 504 476 528 946 897 1491 1091 731 465 57

```

We see that many flights are scheduled in the early to mid-morning and from the late afternoon to early evening. None are scheduled before 5 am or after 10 pm.

Let's examine how the arrival delay depends on the hour. We'll do this in two ways: first using standard box-and-whisker plots to show the distribution of arrival delays; second with a kind of statistical model called a *linear model* that lets us track the mean arrival delay over the course of the day.

```

SF %>%
  ggplot(aes(x = hour, y = arr_delay)) +
  geom_boxplot(alpha = 0.1, aes(group = hour)) +
  geom_smooth(method = "lm") +
  xlab("Scheduled hour of departure") +
  ylab("Arrival delay (minutes)") +
  coord_cartesian(ylim = c(-30, 120))

```

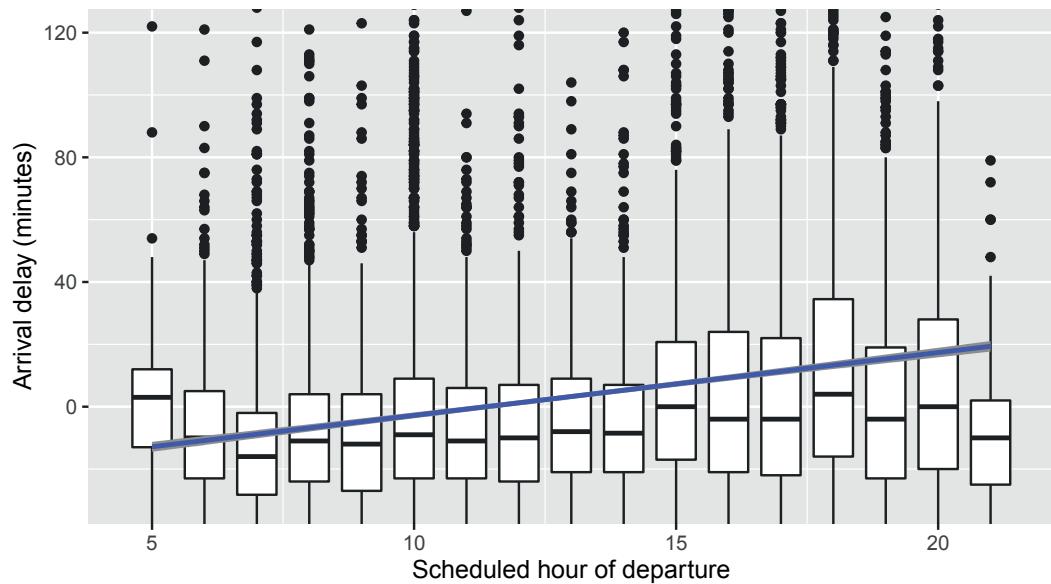


Figure 9.3: Association of flight arrival delays with scheduled departure time for flights to San Francisco from New York airports in 2013.

Figure 9.3 displays the arrival delay versus schedule departure hour. The average arrival delay increases over the course of the day. The trend line itself is created via a regression model (see Appendix E).

```
mod1 <- lm(arr_delay ~ hour, data = SF)
broom::tidy(mod1)
```

term	estimate	std.error	statistic	p.value
<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1 (Intercept)	-22.9	1.23	-18.6	2.88e- 76
2 hour	2.01	0.0915	22.0	1.78e-105

The number under the “estimate” for hour indicates that the arrival delay is predicted to be about 2 minutes higher per hour. Over the 15 hours of flights, this leads to a 30-minute increase in arrival delay comparing flights at the end of the day to flights at the beginning of the day. The `tidy()` function from the **broom** package also calculates the standard error: 0.09 minutes per hour. Stated as a 95% confidence interval, this model indicates that we are 95% confident that the true arrival delay increases by 2.0 ± 0.18 minutes per hour.

The rightmost column gives the *p-value*, a way of translating the estimate and standard error onto a scale from zero to one. By convention, p-values below 0.05 provide a kind of certificate testifying that random, accidental patterns would be unlikely to generate an estimate as large as that observed. The tiny p-value given in the report ($2e-16$ is 0.0000000000000002) is another way of saying that if there was no association between time of day and flight delays, we would be *very* unlikely to see a result this extreme or more extreme.

Re-read those last three sentences. Confusing? Despite an almost universal practice of presenting p-values, they are mostly misunderstood even by scientists and other professionals.

The p-value conveys much less information than usually supposed: The “certificate” might not be worth the paper it’s printed on (see [Section 9.7](#)).

Can we do better? What additional factors might help to explain flight delays? Let’s look at departure airport, carrier (airline), month of the year, and day of the week. Some wrangling will let us extract the day of the week (`dow`) from the year, month, and day of month. We’ll also create a variable `season` that summarizes what we already know about the month: that June and July are the months with long delays. These will be used as *explanatory variables* to account for the *response variable*: arrival delay.

```
library(lubridate)
SF <- SF %>%
  mutate(
    day = as.Date(time_hour),
    dow = as.character(wday(day, label = TRUE)),
    season = ifelse(month %in% 6:7, "summer", "other month")
  )
```

Now we can build a model that includes variables we want to use to explain arrival delay.

```
mod2 <- lm(arr_delay ~ hour + origin + carrier + season + dow, data = SF)
broom::tidy(mod2)
```

term	estimate	std.error	statistic	p.value
<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1 (Intercept)	-24.6	2.17	-11.3	1.27e- 29
2 hour	2.08	0.0898	23.2	1.44e-116
3 originJFK	4.12	1.00	4.10	4.17e- 5
4 carrierB6	-10.3	1.88	-5.49	4.07e- 8
5 carrierDL	-18.4	1.62	-11.4	5.88e- 30
6 carrierUA	-4.76	1.48	-3.21	1.31e- 3
7 carrierVX	-5.06	1.60	-3.17	1.54e- 3
8 seasonsummer	25.3	1.03	24.5	5.20e-130
9 dowMon	1.74	1.45	1.20	2.28e- 1
10 dowSat	-5.60	1.55	-3.62	2.98e- 4
11 dowSun	5.12	1.48	3.46	5.32e- 4
12 dowThu	3.16	1.45	2.18	2.90e- 2
13 dowTue	-1.65	1.45	-1.14	2.53e- 1
14 dowWed	-0.884	1.45	-0.610	5.42e- 1

The numbers in the “estimate” column tell us that we should add 4.1 minutes to the average delay if departing from `JFK` (instead of `EWR`, also known as *Newark*, which is the reference group). Delta has a better average delay than the other carriers. Delays are on average longer in June and July (by 25 minutes), and on Sundays (by 5 minutes). Recall that the Aviana crash was in July.

The model also indicates that Sundays are associated with roughly 5 minutes of additional delays; Saturdays are 6 minutes less delayed on average. (Each of the days of the week is being compared to Friday, chosen as the reference group because it comes first alphabetically.) The standard errors tell us the precision of these estimates; the p-values describe whether the individual patterns are consistent with what might be expected to occur by accident even if there were no systemic association between the variables.

In this example, we've used `lm()` to construct what are called *linear models*. Linear models describe how the mean of the response variable varies with the explanatory variables. They are the most widely used statistical modeling technique, but there are others. In particular, since our original motivation was to set a policy about business travel, we might want a modeling technique that lets us look at another question: What is the probability that a flight will be, say, greater than 100 minutes late? Without going into detail, we'll mention that a technique called *logistic regression* is appropriate for such *dichotomous* outcomes (see [Chapter 11](#) and [Section E.5](#) for more examples).

9.6 Confounding and accounting for other factors

We drill the mantra *correlation does not imply causation* into students whenever statistics are discussed. While the statement is certainly true, it may not be so helpful.

There are many times when correlations *do* imply causal relationships (beyond just in carefully conducted *randomized trials*). A major concern for observational data is whether the true associations are being distorted by *other factors* that may be the actual determinants of the observed relationship between two factors. Such other factors may *confound* the relationship being studied.

Randomized trials in scientific experiments are considered the gold standard for evidence-based research. Such trials, sometimes called *A/B tests*, are commonly undertaken to compare the effect of a treatment (e.g., two different forms of a Web page). By controlling who receives a new intervention and who receives a control (or standard treatment), the investigator ensures that, on average, all other factors are balanced between the two groups. This allows them to conclude that if there are differences in the outcomes measured at the end of the trial, they can be attributed to the application of the treatment. (It's worth noting that randomized trials can also have confounding if subjects don't comply with treatments or are lost on follow-up.)

While they are ideal, randomized trials are not practical in many settings. It is not ethical to randomize some children to smoke and the others not to smoke in order to determine whether cigarettes cause lung cancer. It is not practical to randomize adults to either drink coffee or abstain to determine whether it has long-term health impacts. Observational (or "found") data may be the only feasible way to answer important questions.

Let's consider an example of confounding using observational data on average teacher salaries (in 2010) and average total SAT scores for each of the 50 United States. The SAT (*Scholastic Aptitude Test*) is a high-stakes exam used for entry into college. Are higher teacher salaries associated with better outcomes on the test at the state level? If so, should we adjust salaries to improve test performance? [Figure 9.4](#) displays a scatterplot of these data. We also fit a linear regression model.

```
SAT_2010 <- SAT_2010 %>%
  mutate(Salary = salary/1000)
SAT_plot <- ggplot(data = SAT_2010, aes(x = Salary, y = total)) +
  geom_point() +
  geom_smooth(method = "lm") +
  ylab("Average total score on the SAT") +
```

```
xlab("Average teacher salary (thousands of USD)")
SAT_plot
```

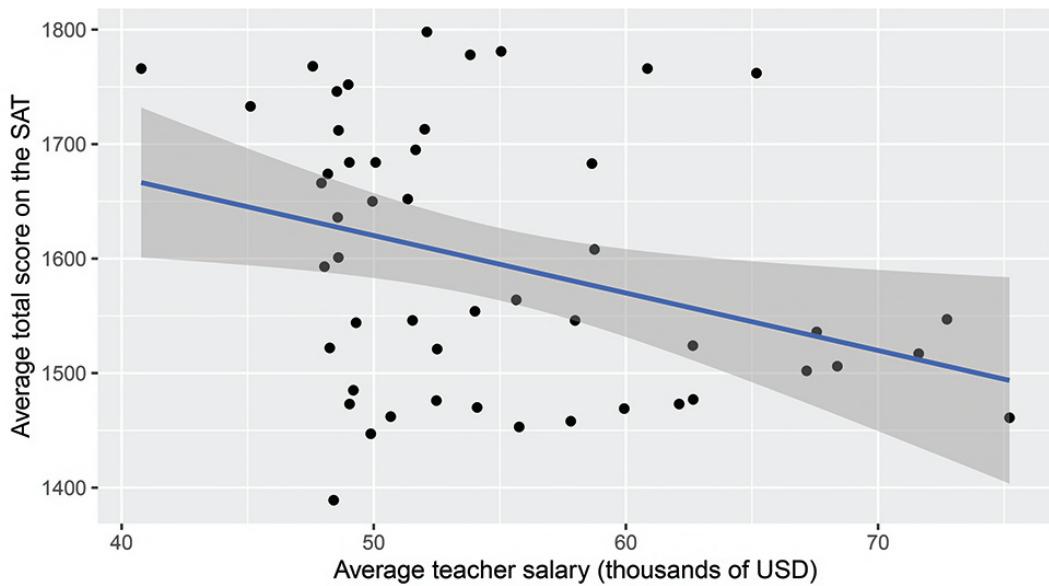


Figure 9.4: Scatterplot of average SAT scores versus average teacher salaries (in thousands of dollars) for the 50 United States in 2010.

```
SAT_mod1 <- lm(total ~ Salary, data = SAT_2010)
broom::tidy(SAT_mod1)
```

```
# A tibble: 2 x 5
  term      estimate std.error statistic p.value
  <chr>       <dbl>     <dbl>      <dbl>    <dbl>
1 (Intercept) 1871.      113.      16.5  1.86e-21
2 Salary       -5.02      2.05     -2.45  1.79e- 2
```

Lurking in the background, however, is another important factor. The percentage of students who take the SAT in each state varies dramatically (from 3% to 93% in 2010). We can create a variable called `SAT_grp` that divides the states into two groups.

```
SAT_2010 %>%
  skim(sat_pct)
```

```
-- Variable type: numeric -----
#>   var      n    na   mean    sd    p0    p25    p50    p75    p100
#> 1 sat_pct  50     0 38.5 32.0     3     6    27    68    93
```

```
SAT_2010 <- SAT_2010 %>%
  mutate(SAT_grp = ifelse(sat_pct <= 27, "Low", "High"))
SAT_2010 %>%
  group_by(SAT_grp) %>%
  count()
```

```
# A tibble: 2 x 2
# Groups:   SAT_grp [2]
```

```
SAT_grp      n
<chr>    <int>
1 High        25
2 Low         25
```

Figure 9.5 displays a scatterplot of these data stratified by the grouping of percentage taking the SAT.

```
SAT_plot %>% SAT_2010 +
  aes(color = SAT_grp) +
  scale_color_brewer("% taking\nthe SAT", palette = "Set2")
```

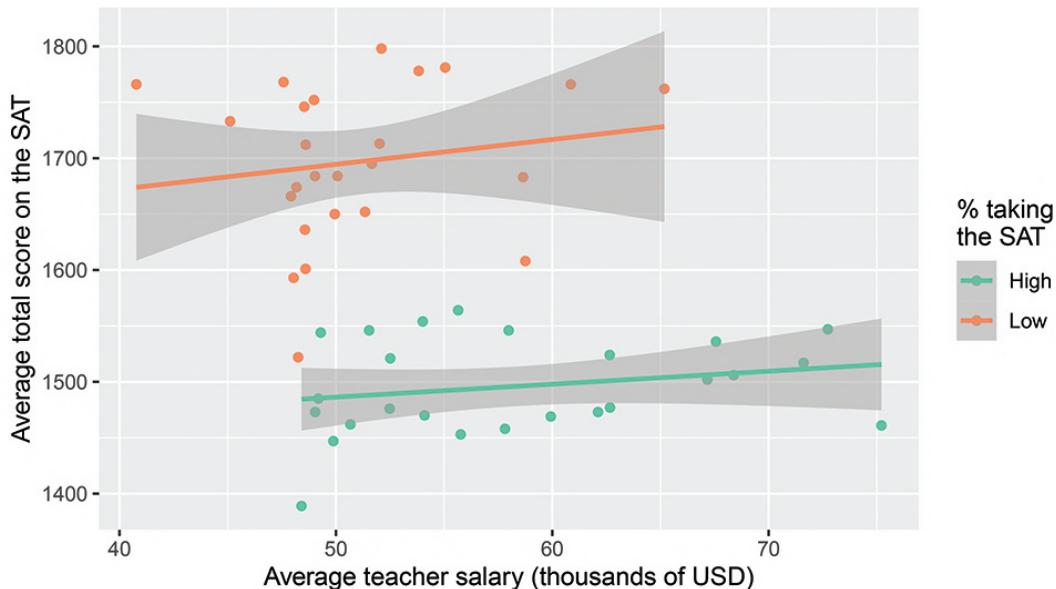


Figure 9.5: Scatterplot of average SAT scores versus average teacher salaries (in thousands of dollars) for the 50 United States in 2010, stratified by the percentage of students taking the SAT in each state.

Using techniques developed in [Section 7.5](#), we can derive the coefficients of the linear model fit to the two separate groups.

```
SAT_2010 %>%
  group_by(SAT_grp) %>%
  group_modify(~broom:::tidy(lm(total ~ Salary, data = .x)))
```

```
# A tibble: 4 x 6
# Groups:   SAT_grp [2]
  SAT_grp term      estimate std.error statistic p.value
  <chr>   <chr>      <dbl>     <dbl>      <dbl>    <dbl>
1 High    (Intercept) 1428.      62.4      22.9    2.51e-17
2 High    Salary       1.16      1.06      1.09    2.85e- 1
3 Low     (Intercept) 1583.     141.      11.2    8.52e-11
4 Low     Salary       2.22      2.75      0.809   4.27e- 1
```

For each of the groups, average teacher salary is positively associated with average SAT score. But when we collapse over this variable, average teacher salary is negatively associated with

average SAT score. This form of confounding is a quantitative version of *Simpson's paradox* and arises in many situations. It can be summarized in the following way:

- Among states with a low percentage taking the SAT, teacher salaries and SAT scores are positively associated.
- Among states with a high percentage taking the SAT, teacher salaries and SAT scores are positively associated.
- Among all states, salaries and SAT scores are negatively associated.

Addressing confounding is straightforward if the confounding variables are measured. Stratification is one approach (as seen above). Multiple regression is another technique. Let's add the `sat_pct` variable as an additional predictor into the regression model.

```
SAT_mod2 <- lm(total ~ Salary + sat_pct, data = SAT_2010)
broom::tidy(SAT_mod2)
```

# A tibble: 3 x 5	term	estimate	std.error	statistic	p.value
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1	(Intercept)	1589.	58.5	27.2	2.16e-30
2	Salary	2.64	1.15	2.30	2.62e- 2
3	sat_pct	-3.55	0.278	-12.8	7.11e-17

We now see that the slope for `Salary` is positive and statistically significant when we control for `sat_pct`. This is consistent with the results when the model was stratified by `SAT_grp`.

We still can't really conclude that teacher salaries cause improvements in SAT scores. However, the associations that we observe after accounting for the confounding are likely more reliable than those that do not take those factors into account.

Pro Tip 29. *Data scientists spend most of their time working with observational data. When seeking to find meaning from such data, it is important to look out for potential confounding factors that could distort observed associations.*

9.7 The perils of p-values

We close with a reminder of the perils of *null hypothesis statistical testing*. Recall that a p-value is defined as the probability of seeing a sample statistic as extreme (or more extreme) than the one that was observed if it were really the case that patterns in the data are a result of random chance. This hypothesis, that only randomness is in play, is called the *null hypothesis*.

For the earlier models involving the airlines data, the null hypothesis would be that there is no association between the predictors and the flight delay. For the SAT and salary example, the null hypothesis would be that the true (population) regression coefficient (slope) is zero.

Historically, when using *hypothesis testing*, analysts have declared results with a p-value of 0.05 or smaller as *statistically significant*, while values larger than 0.05 are declared non-significant. The threshold for that cutoff is called the *alpha level* of the test. If the null hypothesis is true, hypothesis testers would incorrectly reject the null hypothesis $100 \cdot \alpha\%$ of the time.

There are a number of serious issues with this form of “all or nothing thinking.” Recall that p-values are computed by simulating a world in which a null hypothesis is set to be true (see [Chapter 13](#)). The p-value indicates the quality of the concordance between the data and the simulation results. A large p-value indicates the data are concordant with the simulation. A very small p-value means otherwise: that the simulation is irrelevant to describing the mechanism behind the observed patterns. Unfortunately, that in itself tells us little about what kind of hypothesis would be relevant. Ironically, a “significant result” means that we get to reject the null hypothesis but doesn’t tell us what hypothesis to accept!

Pro Tip 30. *Always report the actual p-value (or a statement that it is less than some small value such as $p < 0.0001$) rather than just the decision (reject null vs. fail to reject the null). In addition, confidence intervals are often more interpretable and should be reported as well.*

Null hypothesis testing and p-values are a vexing topic for many analysts. To help clarify these issues, the American Statistical Association endorsed a statement on p-values (Wasserstein and Lazar, 2016) that laid out six useful principles:

- p-values can indicate how incompatible the data are with a specified statistical model.
- p-values do not measure the probability that the studied hypothesis is true, or the probability that the data were produced by random chance alone.
- Scientific conclusions and business or policy decisions should not be based only on whether a p-value passes a specific threshold.
- Proper inference requires full reporting and transparency.
- A p-value, or statistical significance, does not measure the size of an effect or the importance of a result.
- By itself, a p-value does not provide a good measure of evidence regarding a model or hypothesis.

More recent guidance (Wasserstein et al., 2019) suggested the ATOM proposal: “Accept uncertainty, be Thoughtful, Open, and Modest.” The problem with p-values is even more vexing in most real-world investigations. Analyses might involve not just a single hypothesis test but instead have dozens or more. In such a situation, even small p-values do not demonstrate discordance between the data and the null hypothesis, so the statistical analysis may tell us nothing at all.

In an attempt to restore meaning to p-values, investigators are starting to clearly delineate and pre-specify the primary and secondary outcomes for a randomized trial. Imagine that such a trial has five outcomes that are defined as being of primary interest. If the usual procedure in which a test is declared statistically significant if its p-value is less than 0.05 is used, the null hypotheses are true, and the tests are independent, we would expect that we would reject one or more of the null hypotheses more than 22% of the time (considerably more than 5% of the time we want).

```
1 - (1 - 0.05)^5
```

```
[1] 0.226
```

Clinical trialists have sometimes adapted to this problem by using more stringent statistical determinations. A simple, albeit conservative approach is use of a *Bonferroni correction*. Consider dividing our α -level by the number of tests, and only rejecting the null hypothesis when the p-value is less than this adjusted value. In our example, the new threshold would be 0.01 (and the overall experiment-wise error rate is preserved at 0.05).

```
1 - (1 - 0.01)^5
```

```
[1] 0.049
```

For observational analyses without pre-specified protocols, it is much harder to determine what (if any) Bonferroni correction is appropriate.

Pro Tip 31. *For analyses that involve many hypothesis tests it is appropriate to include a note of possible limitations that some of the results may be spurious due to multiple comparisons.*

A related problem has been called the *garden of forking paths* by Andrew Gelman of *Columbia University*. Most analyses involve many decisions about how to code data, determine important factors, and formulate and then revise models before the final analyses are set. This process involves looking at the data to construct a parsimonious representation. For example, a continuous predictor might be cut into some arbitrary groupings to assess the relationship between that predictor and the outcome. Or certain variables might be included or excluded from a regression model in an exploratory process.

This process tends to lead towards hypothesis tests that are biased against a null result, since decisions that yield more of a signal (or smaller p-value) might be chosen rather than other options. In clinical trials, the garden of forking paths problem may be less common, since analytic plans need to be prespecified and published. For most data science problems, however, this is a vexing issue that leads to questions about *reproducible results*.

9.8 Further resources

While this chapter raises many important issues related to the appropriate use of statistics in data science, it can only scratch the surface. A number of accessible books provide background in basic statistics (Diez et al., 2019) and statistical practice (van Belle, 2008; Good and Hardin, 2012). Rice (2006) provides a modern introduction to the foundations of statistics as well as a detailed derivation of the sampling distribution of the median (pages 409–410). Other resources related to theoretical statistics can be found in Nolan and Speed (1999); Horton et al. (2004); Horton (2013); Green and Blankenship (2015). Shalizi's forthcoming *Advanced Data Analysis from an Elementary Point of View* provides a technical introduction to a wide range of important topics in statistics, including causal inference.

Wasserstein and Lazar (2016) laid out principles for the appropriate use of p-values. A special issue of *The American Statistician* was devoted to issues around p-values (Wasserstein et al., 2019).

Hesterberg et al. (2005) and Hesterberg (2015) discuss the potential and perils for resampling-based inference. Efron and Hastie (2016) provide an overview of modern inference techniques.

Missing data provide job security for data scientists since it arises in almost all real-world studies. A number of principled approaches have been developed to account for missing values, most notably multiple imputation. Accessible references to the extensive literature on incomplete data include Little and Rubin (2002); Raghunathan (2004); Horton and Kleinman (2007).

While clinical trials are often considered a gold standard for evidence-based decision making, it is worth noting that they are almost always imperfect. Subjects may not comply with the intervention that they were randomized to. They may break the *blinding* and learn what treatment they have been assigned. Some subjects may drop out of the study. All of these issues complicate analysis and interpretation and have led to improvements in trial design and analysis along with the development of causal inference models. The CONSORT (Consolidated Standards of Reporting Trials) statement (<http://www.consort-statement.org>) was developed to alleviate problems with trial reporting.

Reproducibility and the perils of multiple comparisons have been the subject of much discussion in recent years. Nuzzo (2014) summarizes why p-values are not as reliable as often assumed. The STROBE (Strengthening the Reporting of Observational Studies in Epidemiology) statement discusses ways to improve the use of inferential methods (see also [Appendix D](#)).

Aspects of ethics and bias are covered in detail in [Chapter 8](#).

9.9 Exercises

Problem 1 (Easy): We saw that a 95% confidence interval for a mean was constructed by taking the estimate and adding and subtracting two standard deviations. How many standard deviations should be used if a 99% confidence interval is desired?

Problem 2 (Easy): Calculate and interpret a 95% confidence interval for the mean age of mothers from the `Gestation` data set from the `mosaicData` package.

Problem 3 (Medium): Use the bootstrap to generate and interpret a 95% confidence interval for the median age of mothers for the `Gestation` data set from the `mosaicData` package.

Problem 4 (Medium): The `NHANES` data set in the `NHANES` package includes survey data collected by the U.S. National Center for Health Statistics (NCHS), which has conducted a series of health and nutrition surveys since the early 1960s.

- a. An investigator is interested in fitting a model to predict the probability that a female subject will have a diagnosis of diabetes. Predictors for this model include age and BMI. Imagine that only 1/10 of the data are available but that these data are sampled randomly from the full set of observations (this mechanism is called “Missing Completely at Random,” or MCAR). What implications will this sampling have on the results?
- b. Imagine that only 1/10 of the data are available but that these data are sampled from the full set of observations such that missingness depends on age, with older subjects less likely to be observed than younger subjects (this mechanism is called “Covariate Dependent Missingness,” or CDM). What implications will this sampling have on the results?
- c. Imagine that only 1/10 of the data are available but that these data are sampled from the full set of observations such that missingness depends on diabetes status (this mechanism is called “Non-Ignorable Non-Response,” or NINR). What implications will this sampling have on the results?

Problem 5 (Medium): Use the bootstrap to generate a 95% confidence interval for the regression parameters in a model for weight as a function of age for the `Gestation` data frame from the `mosaicData` package.

Problem 6 (Medium): A data scientist working for a company that sells mortgages for new home purchases might be interested in determining what factors might be predictive of defaulting on the loan. Some of the mortgagees have missing income in their data set. Would it be reasonable for the analyst to drop these loans from their analytic data set? Explain.

Problem 7 (Medium): The `Whickham` data set in the `mosaicData` package includes data on age, smoking, and mortality from a one-in-six survey of the electoral roll in Whickham, a mixed urban and rural district near Newcastle upon Tyne, in the United Kingdom. The survey was conducted in 1972–1974 to study heart disease and thyroid disease. A follow-up on those in the survey was conducted 20 years later. Describe the association between smoking status and mortality in this study. Be sure to consider the role of age as a possible confounding factor.

9.10 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-foundations.html#datavizI-online-exercises>

10

Predictive modeling

Thus far, we have discussed two primary methods for investigating relationships among variables in our data: graphics and regression models. Graphics are often interpretable through intuitive inspection alone. They can be used to identify patterns and multivariate relationships in data—this is called *exploratory data analysis*. Regression models can help us quantify the magnitude and direction of relationships among variables. Thus, both are useful for helping us understand the world and then tell a coherent story about it.

However, graphics are not always the best way to explore or to present data. Graphics work well when there are two or three or even four variables involved. As we saw in [Chapter 2](#), two variables can be represented with position on paper or on screen via a scatterplot. Ultimately, that information is processed by the eye's retina. To represent a third variable, color or size can be used. In principle, more variables can be represented by other graphical aesthetics: shape, angle, color saturation, opacity, facets, etc., but doing so raises problems for human cognition—people simply struggle to integrate so many graphical modes into a coherent whole.

While regression scales well into higher dimensions, it is a limited modeling framework. Rather, it is just one type of model, and the space of all possible models is infinite. In the next three chapters we will explore this space by considering a variety of models that exist outside of a regression framework. The idea that a general specification for a model could be tuned to a specific data set automatically has led to the field of *machine learning*.

The term machine learning was coined in the late 1950s to describe a set of inter-related algorithmic techniques for extracting information from data without human intervention.

In the days before computers, the dominant modeling framework was regression, which is based heavily on the mathematical disciplines of linear algebra and calculus. Many of the important concepts in machine learning emerged from the development of regression, but models that are associated with machine learning tend to be valued more for their ability to make accurate predictions and scale to large data sets, as opposed to the mathematical simplicity, ease of interpretation of the parameters, and solid inferential setting that has made regression so widespread (Breiman, 2001, Efron (2020)). Nevertheless, regression and related statistical techniques from [Chapter 9](#) provide an important foundation for understanding machine learning. [Appendix E](#) provides a brief overview of regression modeling.

There are two main branches in machine learning: *supervised learning* (modeling a specific response variable as a function of some explanatory variables) and *unsupervised learning* (approaches to finding patterns or groupings in data where there is no clear response variable).

In unsupervised learning, the outcome is unmeasured, and thus the task is often framed as a search for otherwise *unmeasured features* of the cases. For instance, assembling DNA data into an evolutionary tree is a problem in unsupervised learning. No matter how much DNA data you have, you don't have a direct measurement of where each organism fits on the

“true” evolutionary tree. Instead, the problem is to create a representation that organizes the DNA data themselves.

By contrast, in supervised learning—which includes linear and logistic regression—the data being studied already include measurements of outcome variables. For instance, in the **NHANES** data, there is already a variable indicating whether or not a person has diabetes. These outcome variables are often referred to as *labels*. Building a model to explore or describe how other variables (often called *features* or predictors) are related to diabetes (weight? age? smoking?) is an exercise in supervised learning.

We discuss metrics for model evaluation in this chapter, several types of supervised learning models in the next, and postpone discussion of unsupervised learning to [Chapter 12](#). It is important to understand that we cannot provide an in-depth treatment of each technique in this book. Rather, our goal is to provide a high-level overview of machine learning techniques that you are likely to come across. By working through these chapters, you will understand the general goals of machine learning, the evaluation techniques that are typically employed, and the basic models that are most commonly used. For a deeper understanding of these techniques, we strongly recommend James et al. (2013) or Hastie et al. (2009).

10.1 Predictive modeling

The basic goal of predictive modeling is to find a *function* that accurately describes how different measured explanatory variables can be combined to make a prediction about a response variable.

A function represents a relationship between inputs and an output (see [Appendix C](#)). Outdoor temperature is a function of season: Season is the input; temperature is the output. Length of the day—i.e., how many hours of daylight—is a function of latitude and day of the year: Latitude and day of the year (e.g., March 22) are the inputs; day length is the output. Modeling a person’s risk of developing *diabetes* could also be a function. We might suspect that age and obesity are likely informative, but how should they be combined?

A bit of **R** syntax will help with defining functions: the *tilde*. The tilde is used to define what the output variable (or outcome, on the left-hand side) is and what the input variables (or predictors, on the right-hand side) are. You’ll see expressions like this:

```
diabetic ~ age + sex + weight + height
```

Here, the variable `diabetic` is marked as the output, simply because it is on the left of the tilde (~). The variables `age`, `sex`, `weight`, and `height` are to be the inputs to the function. You may also see the form `diabetic ~ .` in certain places. The dot to the right of the tilde is a shortcut that means: “use all the available variables (except the output).” The object above has class `formula` in **R**.

There are several different goals that might motivate constructing a function.

- Predict the output given an input. It is February, what will the temperature be? Or on June 15th in *Northampton, MA*, U.S.A. (latitude 42.3 deg N), how many hours of daylight will there be?
- Determine which variables are useful inputs. It is obvious from experience that tempera-

ture is a function of season. But in less familiar situations, e.g., predicting diabetes, the relevant inputs are uncertain or unknown.

- Generate hypotheses. For a scientist trying to figure out the causes of diabetes, it can be useful to construct a predictive model, then look to see what variables turn out to be related to the risk of developing this disorder. For instance, you might find that diet, age, and blood pressure are risk factors. Socioeconomic status is not a direct cause of diabetes, but it might be that there is an association through factors related to the accessibility of health care. That “might be” is a hypothesis, and one that you probably would not have thought of before finding a function relating risk of diabetes to those inputs.
- Understand how a system works. For instance, a reasonable function relating hours of daylight to day-of-the-year and latitude reveals that the northern and southern hemisphere have reversed patterns: Long days in the southern hemisphere will be short days in the northern hemisphere.

Depending on your motivation, the kind of model and the input variables may differ. In understanding how a system works, the variables you use should be related to the actual, causal mechanisms involved, e.g., the genetics of diabetes. For predicting an output, it hardly matters what the causal mechanisms are. Instead, all that’s required is that the inputs are known at a time *before* the prediction is to be made.

10.2 Simple classification models

Classifiers are an important complement to regression models in the fields of machine learning and predictive modeling. Whereas regression models have a quantitative response variable (and can thus often be visualized as a geometric surface), classification models have a categorical response (and are often visualized as a discrete surface, i.e., a tree).

To reduce cognitive overhead, we will restrict our attention in this chapter to classification models based on logistic regression. In the next chapter, we will introduce other types of *classifiers*.

A *logistic regression* model (see [Appendix E](#)) can take a set of explanatory variables (or features) and convert them into a predicted probability. In such a model, the analyst specifies the form of the relationship and what variables are included. If \mathbf{X} is the *matrix* of our p explanatory variables, we can think of this as a function $f : \mathbb{R}^p \rightarrow (0, 1)$ that returns a probability $\pi \in (0, 1)$. However, since the actual values of the response variable y are binary (i.e., in $\{0, 1\}$), we can implement rules $g : (0, 1) \rightarrow \{0, 1\}$ that map values of p to either 0 or 1. Thus, we can use a logistic regression model as the core of a function $h : \mathbb{R}^p \rightarrow \{0, 1\}$, such that $h(\mathbf{X}) = g(f(\mathbf{X}))$ is always either 0 or 1. Such models are known as *classifiers*. More generally, whereas regression models for quantitative response variables return real numbers, models for categorical response variables are called classifiers.

10.2.1 Example: High-earners in the 1994 United States Census

A marketing analyst might be interested in finding factors that can be used to predict whether a potential customer is a high-earner. The 1994 *United States Census* provides information that can inform such a model, with records from 32,561 adults that include a binary variable indicating whether each person makes greater or less than \$50,000 (nearly

\$90,000 in 2020 after accounting for inflation). We will use the indicator of high income as our response variable.

```
library(tidyverse)
library(mdsr)
url <-
  "http://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data"
census <- read_csv(
  url,
  col_names = c(
    "age", "workclass", "fnlwgt", "education",
    "education_1", "marital_status", "occupation", "relationship",
    "race", "sex", "capital_gain", "capital_loss", "hours_per_week",
    "native_country", "income"
  )
) %>%
  mutate(income = factor(income))
glimpse(census)
```

```
Rows: 32,561
Columns: 15
$ age            <dbl> 39, 50, 38, 53, 28, 37, 49, 52, 31, 42, 37, 30, ...
$ workclass      <chr> "State-gov", "Self-emp-not-inc", "Private", "Pri...
$ fnlwgt         <dbl> 77516, 83311, 215646, 234721, 338409, 284582, 16...
$ education       <chr> "Bachelors", "Bachelors", "HS-grad", "11th", "Ba...
$ education_1     <dbl> 13, 13, 9, 7, 13, 14, 5, 9, 14, 13, 10, 13, 13, ...
$ marital_status  <chr> "Never-married", "Married-civ-spouse", "Divorced...
$ occupation      <chr> "Adm-clerical", "Exec-managerial", "Handlers-cle...
$ relationship    <chr> "Not-in-family", "Husband", "Not-in-family", "Hu...
$ race            <chr> "White", "White", "White", "Black", "Black", "Wh...
$ sex              <chr> "Male", "Male", "Male", "Male", "Female", "Femal...
$ capital_gain    <dbl> 2174, 0, 0, 0, 0, 0, 0, 14084, 5178, 0, 0, 0, ...
$ capital_loss    <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
$ hours_per_week   <dbl> 40, 13, 40, 40, 40, 40, 16, 45, 50, 40, 80, 40, ...
$ native_country   <chr> "United-States", "United-States", "United-States...
$ income          <fct> <=50K, <=50K, <=50K, <=50K, <=50K, <=50K, ...
```

Throughout this chapter, we will use the **tidymodels** package to streamline our computations. The **tidymodels** package is really a collection of packages, similar to the **tidyverse**. The workhorse package for model fitting is called **parsnip**, while the model evaluation metrics are provided by the **yardstick** package.

For reasons that we will discuss later (in [Section 10.3.2](#)), we will first separate our data set into two pieces by sampling the rows at random. A sample of 80% of the rows will become the training data set, with the remaining 20% set aside as the testing (or “hold-out”) data set. The **initial_split()** function divides the data, while the **training()** and **testing()** functions recover the two smaller data sets.

```
library(tidymodels)
set.seed(364)
n <- nrow(census)
census_parts <- census %>%
  initial_split(prop = 0.8)
```

```

train <- census_parts %>%
  training()

test <- census_parts %>%
  testing()

list(train, test) %>%
  map_int(nrow)

[1] 26049  6512

```

We first compute the observed percentage of high earners in the training set as $\bar{\pi}$.

```

pi_bar <- train %>%
  count(income) %>%
  mutate(pct = n / sum(n)) %>%
  filter(income == ">50K") %>%
  pull(pct)

pi_bar

```

```
[1] 0.238
```

Note that only about 24% of those in the sample make more than \$50k.

10.2.1.1 The null model

Since we know $\bar{\pi}$, it follows that the *accuracy* of the *null model* is $1 - \bar{\pi}$, which is about 76%, since we can get that many right by just predicting that everyone makes less than \$50k.

```

train %>%
  count(income) %>%
  mutate(pct = n / sum(n))

# A tibble: 2 x 3
  income     n     pct
  <fct> <int> <dbl>
1 <=50K   19843  0.762
2 >50K    6206   0.238

```

While we can compute the accuracy of the null model with simple arithmetic, when we compare models later, it will be useful to have our null model stored as a model object. We can create such an object using **tidymodels** by specifying a logistic regression model with no explanatory variables. The computational engine is **glm** because **glm()** is the name of the **R** function that actually fits **vocab("generalized linear models")** (of which logistic regression is a special case).

```

mod_null <- logistic_reg(mode = "classification") %>%
  set_engine("glm") %>%
  fit(income ~ 1, data = train)

```

After using the **predict()** function to compute the predicted values, the **yardstick** package will help us compute the accuracy.

```

library(yardstick)
pred <- train %>%

```

```

select(income, capital_gain) %>%
bind_cols(
  predict(mod_null, new_data = train, type = "class")
) %>%
rename(income_null = .pred_class)
accuracy(pred, income, income_null)

# A tibble: 1 x 3
.metric  .estimator .estimate
<chr>    <chr>        <dbl>
1 accuracy binary      0.762

```

Pro Tip 32. Always benchmark your predictive models against a reasonable null model.

Another important tool in verifying a model's accuracy is called the *confusion matrix* (really). Simply put, this is a two-way table that counts how often our model made the correct prediction. Note that there are two different types of mistakes that our model can make: predicting a high income when the income was in fact low (a *Type I error*), and predicting a low income when the income was in fact high (a *Type II error*).

```

confusion_null <- pred %>%
  conf_mat(truth = income, estimate = income_null)
confusion_null

```

		Truth	
		<=50K	>50K
Prediction	<=50K	19843	6206
	>50K	0	0

Note again that the null model predicts that *everyone* is a low earner, so it makes many Type II errors (false negatives) but no Type I errors (false positives).

10.2.1.2 Logistic regression

Beating the null model shouldn't be hard. Our first attempt will be to employ a simple logistic regression model. First, we'll fit the model using only one explanatory variable: *capital_gain*. This variable measures the amount of money that each person paid in *capital gains* tax. Since capital gains are accrued on assets (e.g., stocks, houses), it stands to reason that people who pay more in capital gains are likely to have more wealth and, similarly, are likely to have high incomes. In addition, capital gains is directly related since it is a component of total income.

```

mod_log_1 <- logistic_reg(mode = "classification") %>%
  set_engine("glm") %>%
  fit(income ~ capital_gain, data = train)

```

Figure 10.1 illustrates how the predicted probability of being a high earner varies in our simple logistic regression model with respect to the amount of capital gains tax paid.

```

train_plus <- train %>%
  mutate(high_earner = as.integer(income == ">50K"))

ggplot(train_plus, aes(x = capital_gain, y = high_earner)) +
  geom_count()

```

```

position = position_jitter(width = 0, height = 0.05),
alpha = 0.5
) +
geom_smooth(
  method = "glm", method.args = list(family = "binomial"),
  color = "dodgerblue", lty = 2, se = FALSE
) +
geom_hline(aes(yintercept = 0.5), linetype = 3) +
scale_x_log10(labels = scales::dollar)

```

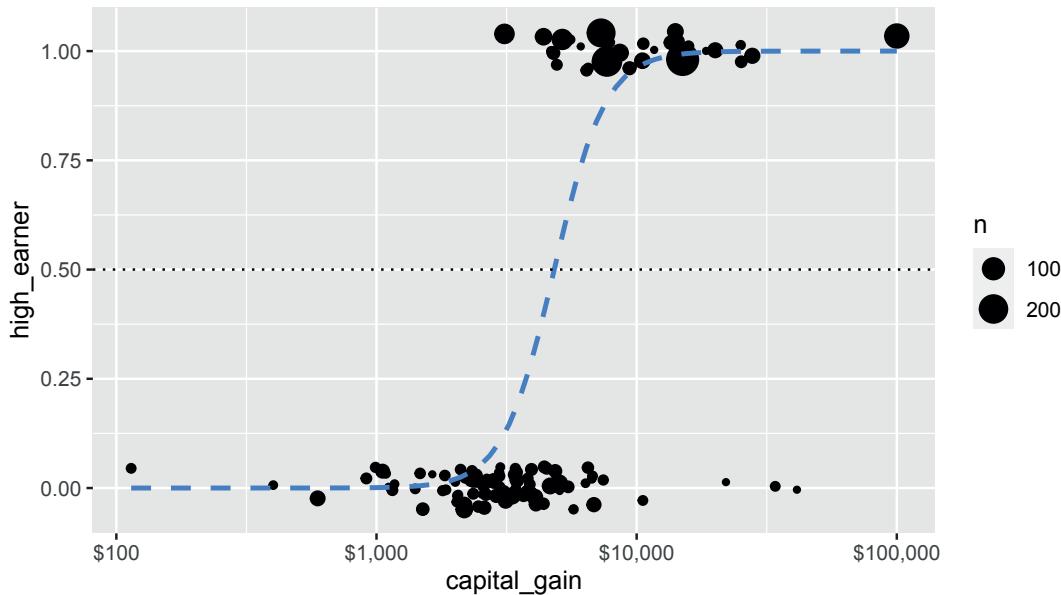


Figure 10.1: Simple logistic regression model for high-earner status based on capital gains tax paid.

How accurate is this model?

```

pred <- pred %>%
  bind_cols(
    predict(mod_log_1, new_data = train, type = "class")
  ) %>%
  rename(income_log_1 = .pred_class)

confusion_log_1 <- pred %>%
  conf_mat(truth = income, estimate = income_log_1)

confusion_log_1

```

		Truth
Prediction	$\leq 50K$	$> 50K$
$\leq 50K$	19640	4966
$> 50K$	203	1240

```
accuracy(pred, income, income_log_1)
```

```
# A tibble: 1 x 3
  .metric  .estimator .estimate
  <chr>    <chr>        <dbl>
1 accuracy binary      0.802
```

In [Figure 10.2](#), we graphically compare the confusion matrices of the null model and the simple logistic regression model. The true positives of the latter model are an important improvement.

```
autoplot(confusion_null) +
  geom_label(
    aes(
      x = (xmax + xmin) / 2,
      y = (ymax + ymin) / 2,
      label = c("TN", "FP", "FN", "TP")
    )
  )
  autoplot(confusion_log_1) +
  geom_label(
    aes(
      x = (xmax + xmin) / 2,
      y = (ymax + ymin) / 2,
      label = c("TN", "FP", "FN", "TP")
    )
  )
```

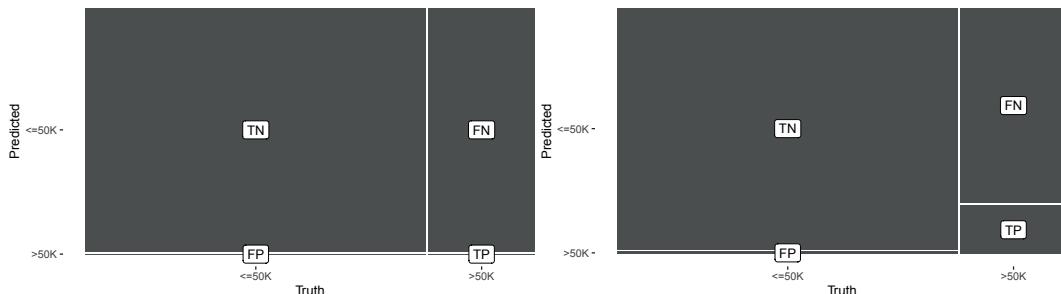


Figure 10.2: Visual summary of the predictive accuracy of the null model (left) versus the logistic regression model with one explanatory variable (right). The null model never predicts a positive.

Using `capital_gains` as a single explanatory variable improved the model's accuracy on the training data to 80.2%, a notable increase over the null model's accuracy of 76.2%.

We can easily interpret the rule generated by the logistic regression model here, since there is only a single predictor.

```
broom::tidy(mod_log_1)
```

```
# A tibble: 2 x 5
  term      estimate std.error statistic p.value
  <chr>      <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept)  0.0000000 0.0000000 0.0000000 1.0000000
2 capital_gains 0.0000000 0.0000000 0.0000000 1.0000000
```

```
1 (Intercept) -1.39      0.0160      -86.7 0.
2 capital_gain  0.000332  0.00000963     34.5 6.54e-261
```

Recall that logistic regression uses the *logit* function to map predicted probabilities to the whole real line. We can invert this function to find the value of capital gains that would yield a predicted value of 0.5.

$$\text{logit}(\hat{\pi}) = \log\left(\frac{\hat{\pi}}{1-\hat{\pi}}\right) = \beta_0 + \beta_1 \cdot \text{capital_gain}$$

We can invert this function to find the value of capital gains that would yield a predicted value of 0.5 by plugging in $\hat{\pi} = 0.5$, $\beta_0 = -1.389$, $\beta_1 = 0.000332$, and solving for *capital_gain*. The answer in this case is $-\beta_0/\beta_1$, or \$4,178.

We can confirm that when we inspect the predicted probabilities, the classification shifts from <=50K to >50K as the value of *capital_gain* jumps from \$4,101 to \$4,386. For these observations, the predicted probabilities jump from 0.494 to 0.517.

```
income_probs <- pred %>%
  select(income, income_log_1, capital_gain) %>%
  bind_cols(
    predict(mod_log_1, new_data = train, type = "prob")
  )

income_probs %>%
  rename(rich_prob = `pred_>50K`) %>%
  distinct() %>%
  filter(abs(rich_prob - 0.5) < 0.02) %>%
  arrange(desc(rich_prob))
```

```
# A tibble: 6 x 5
  income income_log_1 capital_gain `pred_<=50K` rich_prob
  <fct> <fct>          <dbl>        <dbl>      <dbl>
1 <=50K  >50K           4416       0.480      0.520
2 >50K   >50K           4386       0.483      0.517
3 <=50K  >50K           4386       0.483      0.517
4 <=50K  <=50K          4101       0.506      0.494
5 <=50K  <=50K          4064       0.509      0.491
6 <=50K  <=50K          3942       0.520      0.480
```

Thus, the model says to call a taxpayer high income if their capital gains are above \$4,178.

But why should we restrict our model to one explanatory variable? Let's fit a more sophisticated model that incorporates the other explanatory variables.

```
mod_log_all <- logistic_reg(mode = "classification") %>%
  set_engine("glm") %>%
  fit(
    income ~ age + workclass + education + marital_status +
      occupation + relationship + race + sex +
      capital_gain + capital_loss + hours_per_week,
    data = train
  )
```

```

pred <- pred %>%
  bind_cols(
    predict(mod_log_all, new_data = train, type = "class")
  ) %>%
  rename(income_log_all = .pred_class)

pred %>%
  conf_mat(truth = income, estimate = income_log_all)

      Truth
Prediction <=50K  >50K
<=50K   18497  2496
>50K     1346   3710

accuracy(pred, income, income_log_all)

# A tibble: 1 x 3
  .metric  .estimator .estimate
  <chr>    <chr>        <dbl>
1 accuracy binary      0.853

```

Not surprisingly, by including more explanatory variables, we have improved the predictive accuracy on the training set. Unfortunately, predictive modeling is not quite this easy. In the next section, we'll see where our naïve approach can fail.

10.3 Evaluating models

How do you know if your model is a good one? In this section, we outline some of the key concepts in model evaluation—a critical step in predictive analytics.

10.3.1 Bias-variance trade-off

We want to have models that minimize both *bias* and *variance*, but to some extent these are mutually exclusive goals. A complicated model will have less bias, but will in general have higher variance. A simple model can reduce variance but at the cost of increased bias. The optimal balance between bias and variance depends on the purpose for which the model is constructed (e.g., prediction vs. description of causal relationships) and the system being modeled. One helpful class of techniques—called *regularization*—provides model architectures that can balance bias and variance in a graduated way. Examples of regularization techniques are *ridge regression* and the *lasso* (see Section 11.5).

10.3.2 Cross-validation

A vexing and seductive trap that modelers sometimes fall into is *overfitting*. Every model discussed in this chapter is *fit* to a set of data. That is, given a set of *training* data and the specification for the type of model, each algorithm will determine the optimal set of parameters for that model and those data. However, if the model works well on those training data, but not so well on a set of *testing* data—that the model has never seen—then the model is said to be *overfitting*. Perhaps the most elementary mistake in predictive analytics

is to overfit your model to the training data, only to see it later perform miserably on the testing set.

In predictive analytics, data sets are often divided into two sets:

- **Training:** The set of data on which you build your model
- **Testing:** After your model is built, this is the set used to test it by evaluating it against data that it has not previously seen.

For example, in this chapter we set aside 80% of the observations to use as a training set, but held back another 20% for testing. The 80/20 scheme we have employed in this chapter is among the simplest possible schemes, but there are other possibilities. Perhaps a 90/10 or a 75/25 split would be a better option. The goal is to have as much data in the training set to allow the model to perform well while having sufficient numbers in the test set to properly assess it.

An alternative approach to combat this problem is *cross-validation*. To perform a 2-fold cross-validation:

- Randomly separate your data (by rows) into two data sets with the same number of observations. Let's call them X_1 and X_2 .
- Build your model on the data in X_1 , and then run the data in X_2 through your model. How well does it perform? Just because your model performs well on X_1 (this is known as *in-sample* testing) does not imply that it will perform as well on the data in X_2 (*out-of-sample* testing).
- Now reverse the roles of X_1 and X_2 , so that the data in X_2 is used for training, and the data in X_1 is used for testing.
- If your first model is overfit, then it will likely not perform as well on the second set of data.

More complex schemes for cross-validating are possible. k -fold cross-validation is the generalization of 2-fold cross validation, in which the data are separated into k equal-sized partitions, and each of the k partitions is chosen to be the testing set once, with the other $k - 1$ partitions used for training.

10.3.3 Confusion matrices and ROC curves

For classifiers, we have already seen the confusion matrix, which is a common way to assess the effectiveness of a classifier.

Recall that each of the classifiers we have discussed in this chapter are capable of producing not only a binary class label, but also the predicted probability of belonging to either class. Rounding the probabilities in the usual way (using 0.5 as a threshold) may not be a good idea, since the average probability might not be anywhere near 0.5, and thus we could have far too many predictions in one class.

For example, in the census data, only about 24% of the people in the training set had income above \$50,000. Thus, a properly calibrated predictive model should predict that about 24% of the people have incomes above \$50,000. Consider the raw probabilities returned by the simple logistic regression model.

```
head(income_probs)

# A tibble: 6 x 5
  income income_log_1 capital_gain `pred_<=50K` `pred_>50K`
```

		<dbl>	<dbl>	<dbl>
1	<=50K	2174	0.661	0.339
2	<=50K	0	0.800	0.200
3	<=50K	0	0.800	0.200
4	<=50K	0	0.800	0.200
5	<=50K	0	0.800	0.200
6	<=50K	0	0.800	0.200

If we round these using a threshold of 0.5, then only NA% are predicted to have high incomes. Note that here we are able to work with the unfortunate characters in the variable names by wrapping them with backticks. Of course, we could also rename them.

```
income_probs %>%
  group_by(rich = `pred_>50K` > 0.5) %>%
  count() %>%
  mutate(pct = n / nrow(income_probs))
```

```
# A tibble: 2 x 3
# Groups:   rich [2]
  rich     n     pct
  <lgl> <int>  <dbl>
1 FALSE  24606  0.945
2 TRUE    1443  0.0554
```

A better alternative would be to use the overall observed percentage (i.e., 24%) as a threshold instead:

```
income_probs %>%
  group_by(rich = `pred_>50K` > pi_bar) %>%
  count() %>%
  mutate(pct = n / nrow(income_probs))
```

```
# A tibble: 2 x 3
# Groups:   rich [2]
  rich     n     pct
  <lgl> <int>  <dbl>
1 FALSE  23937  0.919
2 TRUE    2112  0.0811
```

This is an improvement, but a more principled approach to assessing the quality of a classifier is a *receiver operating characteristic* (ROC) curve. This considers all possible threshold values for rounding, and graphically displays the trade-off between *sensitivity* (the true positive rate) and *specificity* (the true negative rate). What is actually plotted is the true positive rate as a function of the false positive rate.

ROC curves are common in machine learning and operations research as well as assessment of test characteristics and medical imaging. They can be constructed in **R** using the **yardstick** package. Note that ROC curves operate on the fitted probabilities in (0, 1).

```
roc <- pred %>%
  mutate(estimate = pull(income_probs, `pred_>50K`)) %>%
  roc_curve(truth = income, estimate, event_level = "second") %>%
  autoplot()
```

Note that while the `roc_curve()` function performs the calculations necessary to draw the ROC curve, the `autoplot()` function is the one that actually returns a `ggplot2` object.

In [Figure 10.3](#) the upper-left corner represents a perfect classifier, which would have a true positive rate of 1 and a false positive rate of 0. On the other hand, a random classifier would lie along the diagonal, since it would be equally likely to make either kind of mistake.

The simple logistic regression model that we used had the following true and false positive rates, which are indicated in [Figure 10.3](#) by the black dot. A number of other metrics are available.

```
metrics <- pred %>%
  conf_mat(income, income_log_1) %>%
  summary(event_level = "second")
metrics

# A tibble: 13 x 3
  .metric      .estimator .estimate
  <chr>        <chr>       <dbl>
1 accuracy    binary     0.802
2 kap          binary     0.257
3 sens         binary     0.200
4 spec         binary     0.990
5 ppv          binary     0.859
6 npv          binary     0.798
7 mcc          binary     0.353
8 j_index      binary     0.190
9 bal_accuracy binary     0.595
10 detection_prevalence binary  0.0554
11 precision   binary     0.859
12 recall      binary     0.200
13 f_meas      binary     0.324

roc_mod <- metrics %>%
  filter(.metric %in% c("sens", "spec")) %>%
  pivot_wider(-.estimator, names_from = .metric, values_from = .estimate)

roc +
  geom_point(
    data = roc_mod, size = 3,
    aes(x = 1 - spec, y = sens)
  )
```

Depending on our tolerance for false positives vs. false negatives, we could modify the way that our logistic regression model rounds probabilities, which would have the effect of moving the black dot in [Figure 10.3](#) along the curve.

10.3.4 Measuring prediction error for quantitative responses

For evaluating models with a quantitative response, there are a variety of criteria that are commonly used. Here we outline three of the simplest and most common. The following presumes a vector of real observations denoted y and a corresponding vector of prediction \hat{y} :

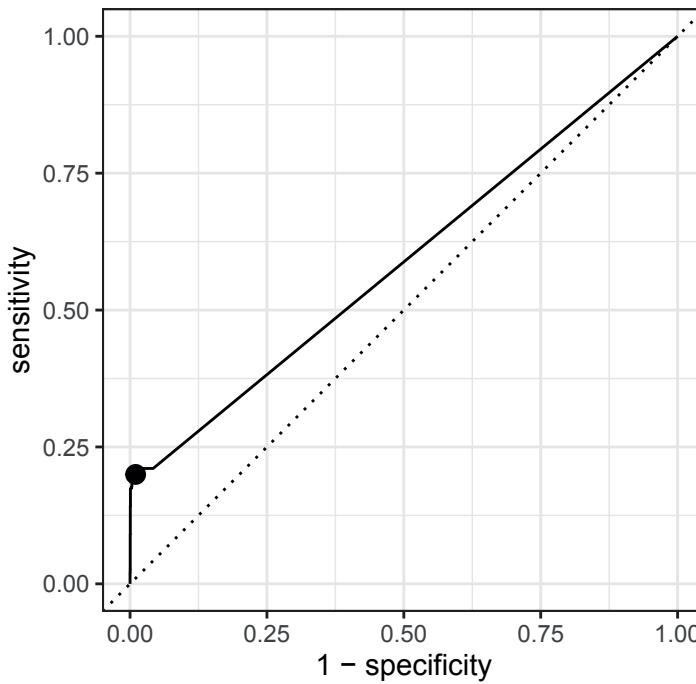


Figure 10.3: ROC curve for the simple logistic regression model.

- **RMSE:** *Root-mean-square error* is probably the most common:

$$RMSE(y, \hat{y}) = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}.$$

The RMSE has several desirable properties. Namely, it is in the same units as the response variable y , it captures both overestimates and underestimates equally, and it penalizes large misses heavily.

- **MAE:** *Mean absolute error* is similar to the RMSE, but does not penalize large misses as heavily, due to the replacement of the squared term by an absolute value:

$$MAE(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|.$$

- **Correlation:** The previous two methods require that the units and scale of the predictions \hat{y} are the same as the response variable y . While this is of course necessary for accurate predictions, some predictive models merely want to track the trends in the response. In many such cases the *correlation* between y and \hat{y} may suffice. In addition to the usual Pearson product-moment correlation (measure of linear association), statistics related to *rank correlation* may be useful. That is, instead of trying to minimize $y - \hat{y}$, it might be enough to make sure that the \hat{y}_i 's are in the same relative order as the y_i 's. Popular measures of rank correlation include Spearman's ρ and Kendall's τ .
- **Coefficient of determination:** (R^2) The *coefficient of determination* describes what proportion of variability in the outcome is explained by the model. It is measured on a scale of $[0, 1]$, with 1 indicating a perfect match between y and \hat{y} .

10.3.5 Example: Evaluation of income models

Recall that we separated the 32,561 observations in the `census` data set into a training set that contained 80% of the observations and a testing set that contained the remaining 20%. Since the separation was done by selecting rows uniformly at random, and the number of observations was fairly large, it seems likely that both the training and testing set will contain similar information. For example, the distribution of `capital_gain` is similar in both the testing and training sets. Nevertheless, it is worth formally testing the performance of our models on both sets.

```
train %>%
  skim(capital_gain)

-- Variable type: numeric -----
var          n    na   mean     sd    p0    p25    p50    p75    p100
1 capital_gain 26049      0 1079. 7451.      0      0      0      0 99999

test %>%
  skim(capital_gain)

-- Variable type: numeric -----
var          n    na   mean     sd    p0    p25    p50    p75    p100
1 capital_gain 6512       0 1071. 7115.      0      0      0      0 99999
```

We note that at least three quarters of both samples reported no capital gains.

To do this, we build a data frame that contains an identifier for each of our three models, as well as a list-column with the model objects.

```
mods <- tibble(
  type = c("null", "log_1", "log_all"),
  mod = list(mod_null, mod_log_1, mod_log_all)
)
```

We can iterate through the list of models and apply the `predict()` method to each object, using both the testing and training sets.

```
mods <- mods %>%
  mutate(
    y_train = list(pull(train, income)),
    y_test = list(pull(test, income)),
    y_hat_train = map(
      mod,
      ~pull(predict(.x, new_data = train, type = "class"), .pred_class)
    ),
    y_hat_test = map(
      mod,
      ~pull(predict(.x, new_data = test, type = "class"), .pred_class)
    )
  )
mods

# A tibble: 3 x 6
  type    mod      y_train      y_test      y_hat_train      y_hat_test
  <chr>   <list>   <list>      <list>      <list>      <list>
1 null   <fit[+]> <fct [26,049]> <fct [6,512]> <fct [26,049]> <fct [6,512]>
```

Table 10.1: Model accuracy measures for the income model.

type	accuracy_train	accuracy_test	sens_test	spec_test
log_all	0.853	0.846	0.586	0.933
log_1	0.802	0.795	0.212	0.991
null	0.762	0.749	0.000	1.000

```
2 log_1  <fit[+]> <fct [26,049]> <fct [6,512]> <fct [26,049]> <fct [6,512]>
3 log_all <fit[+]> <fct [26,049]> <fct [6,512]> <fct [26,049]> <fct [6,512]>
```

Now that we have the predictions for each model, we just need to compare them to the truth (y) and tally the results. We can do this using the `map2_dbl()` function from the **purrr** package.

```
mods <- mods %>%
  mutate(
    accuracy_train = map2_dbl(y_train, y_hat_train, accuracy_vec),
    accuracy_test = map2_dbl(y_test, y_hat_test, accuracy_vec),
    sens_test =
      map2_dbl(y_test, y_hat_test, sens_vec, event_level = "second"),
    spec_test =
      map2_dbl(y_test, y_hat_test, spec_vec, event_level = "second")
  )
```

Table 10.1 displays a number of model accuracy measures. Note that each model performs slightly worse on the testing set than it did on the training set. As expected, the null model has a sensitivity of 0 and a specificity of 1, because it always makes the same prediction. While the model that includes all of the variables is slightly less specific than the single explanatory variable model, it is much more sensitive. In this case, we should probably conclude that the `log_all` model is the most likely to be useful.

In [Figure 10.4](#), we compare the ROC curves for all census models on the testing data set. Some data wrangling is necessary before we can gather the information to make these curves.

```
mods <- mods %>%
  mutate(
    y_hat_prob_test = map(
      mod,
      ~pull(predict(.x, new_data = test, type = "prob"), `_.pred_>50K`)
    ),
    type = fct_reorder(type, sens_test, .desc = TRUE)
  )

mods %>%
  select(type, y_test, y_hat_prob_test) %>%
  unnest(cols = c(y_test, y_hat_prob_test)) %>%
  group_by(type) %>%
  roc_curve(truth = y_test, y_hat_prob_test, event_level = "second") %>%
  autoplot() +
  geom_point(
    data = mods,
```

```

aes(x = 1 - spec_test, y = sens_test, color = type),
size = 3
) +
scale_color_brewer("Model", palette = "Set2")

```

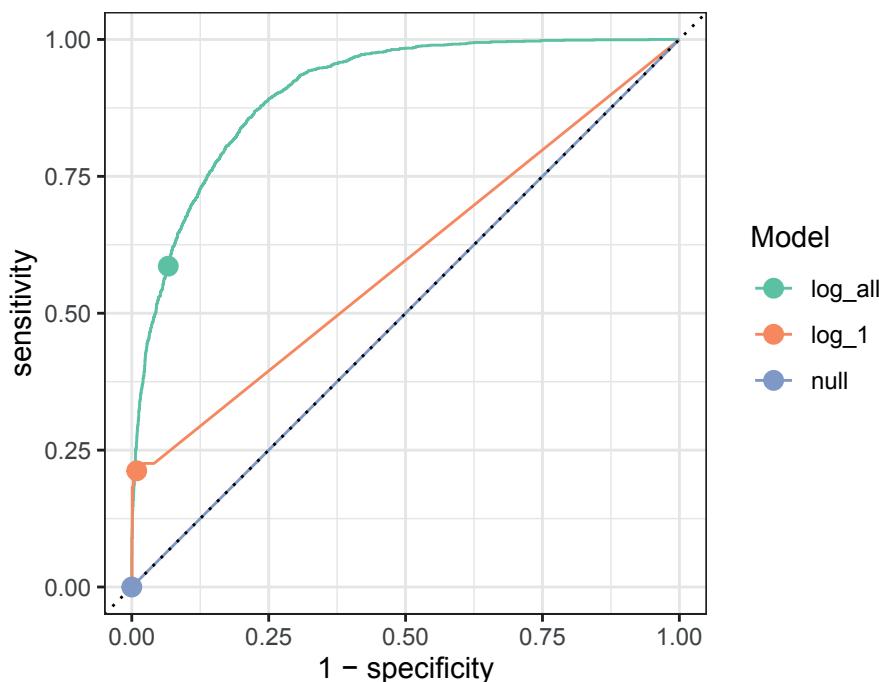


Figure 10.4: Comparison of ROC curves across three logistic regression models on the Census testing data. The null model has a true positive rate of zero and lies along the diagonal. The full model is the best overall performer, as its curve lies furthest from the diagonal.

10.4 Extended example: Who has diabetes?

Consider the relationship between age and *diabetes mellitus*, a group of metabolic diseases characterized by high blood sugar levels. As with many diseases, the risk of contracting diabetes increases with age and is associated with many other factors. Age does not suggest a way to avoid diabetes: there is no way for you to change your age. You can, however, change things like diet, physical fitness, etc. Knowing what is predictive of diabetes can be helpful in practice, for instance, to design an efficient screening program to test people for the disease.

Let's start simply. What is the relationship between age, body-mass index (BMI), and diabetes for adults surveyed in NHANES? Note that the overall rate of diabetes is relatively low.

```

library(NHANES)
people <- NHANES %>%
  select(Age, Gender, Diabetes, BMI, HHIncome, PhysActive) %>%
  drop_na()
glimpse(people)

Rows: 7,555
Columns: 6
$ Age      <int> 34, 34, 34, 49, 45, 45, 45, 66, 58, 54, 58, 50, 33, ...
$ Gender    <fct> male, male, male, female, female, female, female, ma...
$ Diabetes  <fct> No, ...
$ BMI       <dbl> 32.2, 32.2, 32.2, 30.6, 27.2, 27.2, 27.2, 23.7, 23.7...
$ HHIncome   <fct> 25000-34999, 25000-34999, 25000-34999, 35000-44999, ...
$ PhysActive <fct> No, No, No, No, Yes, Yes, Yes, Yes, Yes, Yes, Y...

people %>%
  group_by(Diabetes) %>%
  count() %>%
  mutate(pct = n / nrow(people))

# A tibble: 2 x 3
# Groups:   Diabetes [2]
  Diabetes     n     pct
  <fct>     <int>   <dbl>
1 No          6871  0.909
2 Yes         684   0.0905

```

We can visualize any model. In this case, we will tile the (Age, BMI) -plane with a fine grid of 10,000 points.

```

library(modelr)
num_points <- 100
fake_grid <- data_grid(
  people,
  Age = seq_range(Age, num_points),
  BMI = seq_range(BMI, num_points)
)

```

Next, we will evaluate each of our four models on each grid point, taking care to retrieve not the classification itself, but the probability of having diabetes. The null model considers no variable. The next two models consider only age, or BMI, while the last model considers both.

```

dmod_null <- logistic_reg(mode = "classification") %>%
  set_engine("glm") %>%
  fit(Diabetes ~ 1, data = people)
dmod_log_1 <- logistic_reg(mode = "classification") %>%
  set_engine("glm") %>%
  fit(Diabetes ~ Age, data = people)
dmod_log_2 <- logistic_reg(mode = "classification") %>%
  set_engine("glm") %>%
  fit(Diabetes ~ BMI, data = people)
dmod_log_12 <- logistic_reg(mode = "classification") %>%

```

```

set_engine("glm") %>%
  fit(Diabetes ~ Age + BMI, data = people)
bmi_mods <- tibble(
  type = factor(
    c("Null", "Logistic (Age)", "Logistic (BMI)", "Logistic (Age, BMI)")
  ),
  mod = list(dmod_null, dmod_log_1, dmod_log_2, dmod_log_12),
  y_hat = map(mod, predict, new_data = fake_grid, type = "prob")
)

```

Next, we add the grid data (x), and then use `map2()` to combine the predictions (y_hat) with the grid data.

```

bmi_mods <- bmi_mods %>%
  mutate(
    X = list(fake_grid),
    yX = map2(y_hat, X, bind_cols)
  )

```

Finally, we use `unnest()` to stretch the data frame out. We now have a prediction at each of our 10,000 grid points for each of our four models.

```

res <- bmi_mods %>%
  select(type, yX) %>%
  unnest(cols = yX)
res

# A tibble: 40,000 x 5
  type .pred_No .pred_Yes   Age   BMI
  <fct>   <dbl>    <dbl> <dbl> <dbl>
1 Null     0.909    0.0905    12  13.3
2 Null     0.909    0.0905    12  14.0
3 Null     0.909    0.0905    12  14.7
4 Null     0.909    0.0905    12  15.4
5 Null     0.909    0.0905    12  16.0
6 Null     0.909    0.0905    12  16.7
7 Null     0.909    0.0905    12  17.4
8 Null     0.909    0.0905    12  18.1
9 Null     0.909    0.0905    12  18.8
10 Null    0.909    0.0905    12  19.5
# ... with 39,990 more rows

```

Figure 10.5 illustrates each model in the data space. Whereas the null model predicts the probability of diabetes to be constant irrespective of age and BMI, including age (BMI) as an explanatory variable allows the predicted probability to vary in the horizontal (vertical) direction. Older patients and those with larger body mass have a higher probability of having diabetes. Having both variables as covariates allows the probability to vary with respect to both age and BMI.

```

ggplot(data = res, aes(x = Age, y = BMI)) +
  geom_tile(aes(fill = .pred_Yes), color = NA) +
  geom_count(
    data = people,

```

```

aes(color = Diabetes), alpha = 0.4
) +
scale_fill_gradient("Prob of\nnDiabetes", low = "white", high = "red") +
scale_color_manual(values = c("gold", "black")) +
scale_size(range = c(0, 2)) +
scale_x_continuous(expand = c(0.02, 0)) +
scale_y_continuous(expand = c(0.02, 0)) +
facet_wrap(~fct_rev(type))

```

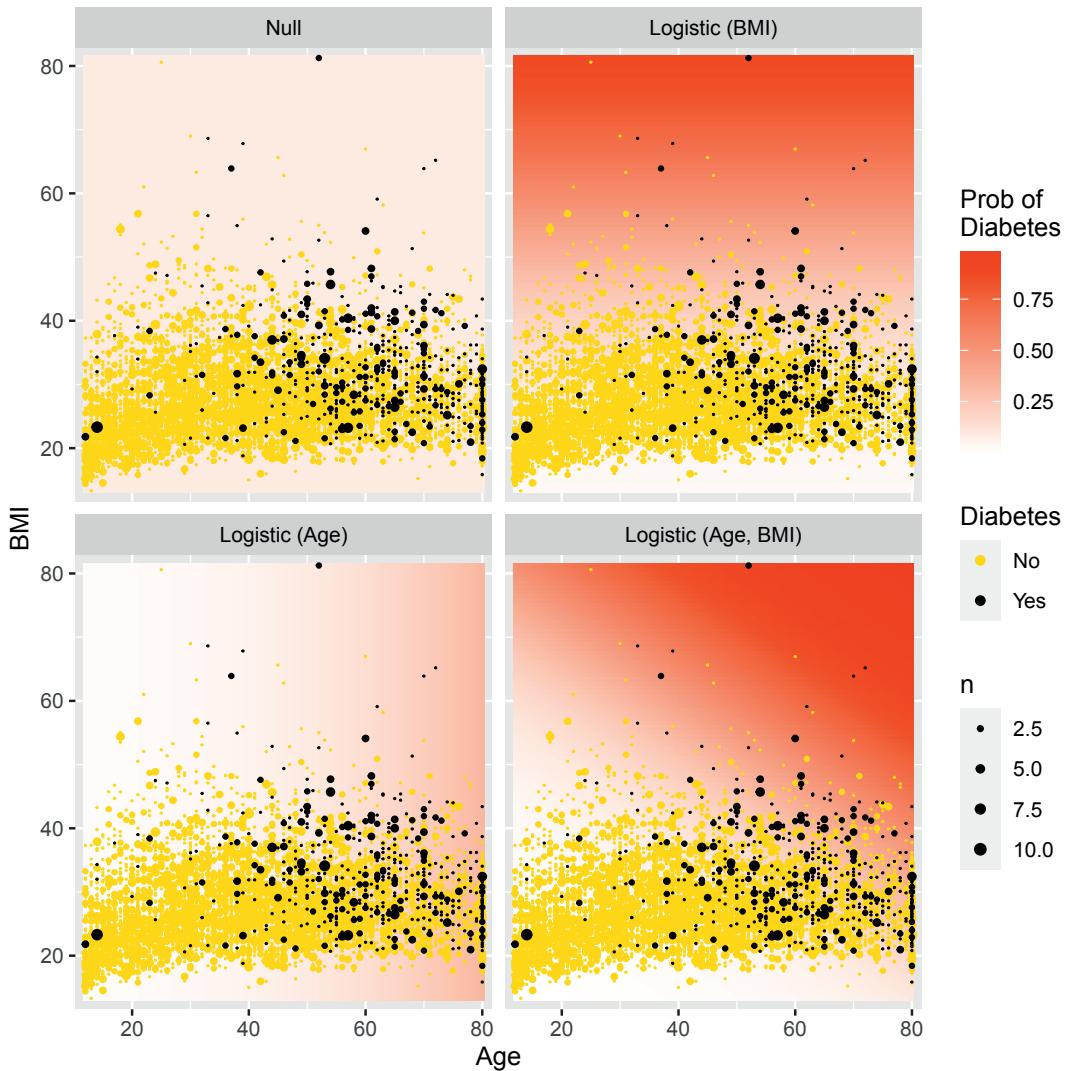


Figure 10.5: Comparison of logistic regression models in the data space. Note the greater flexibility as more variables are introduced.

10.5 Further resources

The **tidymodels** package and documentation contains many vignettes¹ that go into further detail on how the package can be used.

10.6 Exercises

Problem 1 (Easy): In the first example in the chapter, a training dataset of 80% of the rows was created for the Census data. What would be the tradeoffs of using a 90%/10% split instead?

Problem 2 (Easy): Without using jargon, describe what a receiver operating characteristic (ROC) curve is and why it is important in predictive analytics and machine learning.

Problem 3 (Medium): Investigators in the HELP (Health Evaluation and Linkage to Primary Care) study were interested in modeling the probability of being `homeless` (one or more nights spent on the street or in a shelter in the past six months vs. housed) as a function of age.

- a. Generate a confusion matrix for the null model and interpret the result.
- b. Fit and interpret logistic regression model for the probability of being `homeless` as a function of age.
- c. What is the predicted probability of being homeless for a 20 year old? For a 40 year old?
- d. Generate a confusion matrix for the second model and interpret the result.

Problem 4 (Medium): Investigators in the HELP (Health Evaluation and Linkage to Primary Care) study were interested in modeling associations between demographic factors and a baseline measure of depressive symptoms `cesd`. They fit a linear regression model using the following predictors: `age`, `sex`, and `homeless` to the `HELPrcf` data from the `mosaicData` package.

- a. Calculate and interpret the coefficient of determination (R^2) for this model and the null model.
- b. Calculate and interpret the root mean squared error for this model and for the null model.
- c. Calculate and interpret the mean absolute error (MAE) for this model and the null model.

Problem 5 (Medium): What impact does the random number seed have on our results?

- a. Repeat the Census logistic regression model that controlled only for capital gains

¹<https://www.tidymodels.org/learn/statistics/tidy-analysis>

but using a different random number seed (365 instead of 364) for the 80%/20% split. Would you expect big differences in the accuracy using the training data? Testing data?

- b. Repeat the process using a random number seed of 366. What do you conclude?

Problem 6 (Hard): Smoking is an important public health concern. Use the NHANES data from the NHANES package to develop a logistic regression model that identifies predictors of current smoking among those 20 or older. (Hint: note that the SmokeNow variable is missing for those who have never smoked: you will need to recode the variable to construct your outcome variable.)

```
library(tidyverse)
library(NHANES)
mosaic::tally(~ SmokeNow + Smoke100, data = filter(NHANES, Age >= 20))

Smoke100
SmokeNow   No    Yes
  No        0  1745
  Yes       0  1466
<NA>  4024     0
```

10.7 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-modeling.html#modeling-online-exercises>

11

Supervised learning

In this chapter, we will extend our discussion on predictive modeling to include many other models that are not based on regression. The framework for model evaluation that we developed in [Chapter 10](#) will remain useful.

We continue with the example about high earners in the 1994 United States Census.

```
library(tidyverse)
library(mdsr)
url <- 
  "http://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data"
census <- read_csv(
  url,
  col_names = c(
    "age", "workclass", "fnlwgt", "education",
    "education_1", "marital_status", "occupation", "relationship",
    "race", "sex", "capital_gain", "capital_loss", "hours_per_week",
    "native_country", "income"
  )
) %>%
  mutate(income = factor(income))

library(tidymodels)
set.seed(364)
n <- nrow(census)
census_parts <- census %>%
  initial_split(prop = 0.8)
train <- census_parts %>% training()
test <- census_parts %>% testing()
pi_bar <- train %>%
  count(income) %>%
  mutate(pct = n / sum(n)) %>%
  filter(income == ">50K") %>%
  pull(pct)
```

11.1 Non-regression classifiers

The classifiers we built in [Chapter 10](#) were fit using logistic regression. These models were smooth, in that they are based on continuous *parametric functions*. The models we explore in

this chapter are not necessarily continuous, nor are they necessarily expressed as parametric functions.

11.1.1 Decision trees

A decision tree (also known as a classification and regression tree¹ or “CART”) is a tree-like flowchart that assigns class labels to individual observations. Each branch of the tree separates the records in the data set into increasingly “pure” (i.e., homogeneous) subsets, in the sense that they are more likely to share the same class label.

How do we construct these trees? First, note that the number of possible decision trees grows exponentially with respect to the number of variables p . In fact, it has been proven that an efficient algorithm to determine the optimal decision tree almost certainly does not exist (Hyafil and Rivest, 1976).² The lack of a globally optimal algorithm means that there are several competing heuristics for building decision trees that employ greedy (i.e., locally optimal) strategies. While the differences among these algorithms can mean that they will return different results (even on the same data set), we will simplify our presentation by restricting our discussion to *recursive partitioning* decision trees. One R package that builds these decision trees is called **rpart**, which works in conjunction with **tidymodels**.

The partitioning in a decision tree follows *Hunt’s algorithm*, which is itself recursive. Suppose that we are somewhere in the decision tree, and that $D_t = (y_t, \mathbf{X}_t)$ is the set of records that are associated with node t and that $\{y_1, y_2\}$ are the available class labels for the response variable.³ Then:

- If all records in D_t belong to a single class, say, y_1 , then t is a leaf node labeled as y_1 .
- Otherwise, split the records into at least two child nodes, in such a way that the *purity* of the new set of nodes exceeds some threshold. That is, the records are separated more distinctly into groups corresponding to the response class. In practice, there are several competitive methods for optimizing the purity of the candidate child nodes, and—as noted above—we don’t know the optimal way of doing this.

A decision tree works by running Hunt’s algorithm on the full training data set.

What does it mean to say that a set of records is “purer” than another set? Two popular methods for measuring the purity of a set of candidate child nodes are the *Gini coefficient* and the *information gain*. Both are implemented in **rpart**, which uses the Gini measurement by default. If $w_i(t)$ is the fraction of records belonging to class i at node t , then

$$Gini(t) = 1 - \sum_{i=1}^2 (w_i(t))^2, \quad Entropy(t) = - \sum_{i=1}^2 w_i(t) \cdot \log_2 w_i(t)$$

The information gain is the change in entropy. The following example should help to clarify how this works in practice.

```
mod_dtree <- decision_tree(mode = "classification") %>%
  set_engine("rpart") %>%
  fit(income ~ capital_gain, data = train)
```

¹More precisely, regression trees are analogous to decision trees, but with a quantitative response variable. The acronym CART stands for “classification and regression trees.”

²Specifically, the problem of determining the optimal decision tree is NP-complete, meaning that it does not have a polynomial-time solution unless $P = NP$.

³For simplicity, we focus on a binary outcome in this chapter, but classifiers can generalize to any number of discrete response values.

```
split_val <- mod_dtreet$fit$splits %>%
  as_tibble() %>%
  pull(index)
```

Let's consider the optimal split for `income` using only the variable `capital_gain`, which measures the amount each person paid in capital gains taxes. According to our tree, the optimal split occurs for those paying more than \$5,119 in capital gains.

```
mod_dtreet
```

```
parsnip model object

Fit time: 31ms
n= 26049

node), split, n, loss, yval, (yprob)
  * denotes terminal node

1) root 26049 6210 <=50K (0.7618 0.2382)
  2) capital_gain< 5.12e+03 24805 5030 <=50K (0.7972 0.2028) *
  3) capital_gain>=5.12e+03 1244    68 >50K (0.0547 0.9453) *
```

Although nearly 80% of those who paid less than \$5,119 in capital gains tax made less than \$50k, about 95% of those who paid more than \$5,119 in capital gains tax made *more* than \$50k. Thus, splitting (partitioning) the records according to this criterion helps to divide them into relatively purer subsets. We can see this distinction geometrically as we divide the training records in [Figure 11.1](#).

```
train_plus <- train %>%
  mutate(hi_cap_gains = capital_gain >= split_val)

ggplot(data = train_plus, aes(x = capital_gain, y = income)) +
  geom_count(
    aes(color = hi_cap_gains),
    position = position_jitter(width = 0, height = 0.1),
    alpha = 0.5
  ) +
  geom_vline(xintercept = split_val, color = "dodgerblue", lty = 2) +
  scale_x_log10(labels = scales::dollar)
```

Comparing [Figure 11.1](#) to [Figure 10.1](#) reveals how the non-parametric decision tree models differs geometrically from the parametric logistic regression model. In this case, the perfectly vertical split achieved by the decision tree is a mathematical impossibility for the logistic regression model.

Thus, this decision tree uses a single variable (`capital_gain`) to partition the data set into two parts: those who paid more than \$5,119 in capital gains, and those who did not. For the former—who make up 0.952 of all observations—we get 79.7% right by predicting that they made less than \$50k. For the latter, we get 94.5% right by predicting that they made more than \$50k. Thus, our overall accuracy jumps to 80.4%, easily besting the 76.2% in the null model. Note that this performance is comparable to the performance of the single variable logistic regression model from [Chapter 10](#).

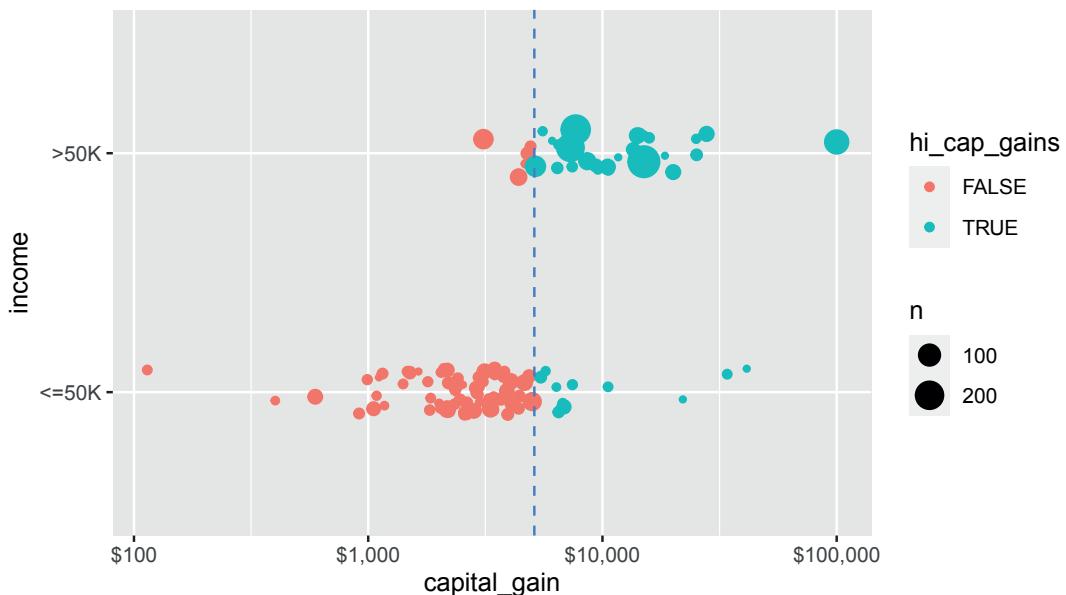


Figure 11.1: A single partition of the census data set using the capital gain variable to determine the split. Color and the vertical line at \$5,119 in capital gains tax indicate the split. If one paid more than this amount, one almost certainly made more than \$50,000 in income. On the other hand, if one paid less than this amount in capital gains, one almost certainly made less than \$50,000.

How did the algorithm know to pick \$5,119 as the threshold value? It tried all of the sensible values, and this was the one that lowered the *Gini coefficient* the most. This can be done efficiently, since thresholds will always be between actual values of the splitting variable, and thus there are only $O(n)$ possible splits to consider. (We use *Big O notation* to denote the complexity of an algorithm, where $O(n)$ means that the number of calculations scales with the sample size.)

So far, we have only used one variable, but we can build a decision tree for `income` in terms of all of the other variables in the data set. (We have left out `native_country` because it is a categorical variable with many levels, which can make some learning models computationally infeasible.)

```
form <- as.formula(
  "income ~ age + workclass + education + marital_status +
  occupation + relationship + race + sex +
  capital_gain + capital_loss + hours_per_week"
)

mod_tree <- decision_tree(mode = "classification") %>%
  set_engine("rpart") %>%
  fit(form, data = train)
mod_tree
```

parsnip model object

Fit time: 385ms
n = 26049

```

node), split, n, loss, yval, (yprob)
 * denotes terminal node

1) root 26049 6210 <=50K (0.7618 0.2382)
  2) relationship=Not-in-family,Other-relative,Own-child,Unmarried 14310
  940 <=50K (0.9343 0.0657)
   4) capital_gain< 7.07e+03 14055  694 <=50K (0.9506 0.0494) *
   5) capital_gain>=7.07e+03 255      9 >50K (0.0353 0.9647) *
  3) relationship=Husband,Wife 11739 5270 <=50K (0.5514 0.4486)
   6) education=10th,11th,12th,1st-4th,5th-6th,7th-8th,9th,Assoc-acdm,Assoc-
voc,HS-grad,Preschool,Some-college 8199 2720 <=50K (0.6686 0.3314)
  12) capital_gain< 5.1e+03 7796 2320 <=50K (0.7023 0.2977) *
  13) capital_gain>=5.1e+03 403      7 >50K (0.0174 0.9826) *
  7) education=Bachelors,Doctorate,Masters,Prof-school
  3540  991 >50K (0.2799 0.7201) *

```

In this more complicated tree, the optimal first split now does not involve `capital_gain`, but rather `relationship`. A plot (shown in [Figure 11.2](#)) that is more informative is available through the `partykit` package, which contains a series of functions for working with decision trees.

```

library(rpart)
library(partykit)
plot(as.party(mod_tree$fit))

```

[Figure 11.2](#) shows the decision tree itself, while [Figure 11.3](#) shows how the tree recursively partitions the original data. Here, the first question is whether `relationship` status is `Husband` or `Wife`. If not, then a capital gains threshold of \$7,073.50 is used to determine one's income. 96.5% of those who paid more than the threshold earned more than \$50k, but 95.1% of those who paid less than the threshold did not. For those whose `relationship` status was `Husband` or `Wife`, the next question was whether you had a college degree. If so, then the model predicts with 72% accuracy that you made more than \$50k. If not, then again we ask about capital gains tax paid, but this time the threshold is \$5,095.50. 98.3% of those who were neither a husband nor a wife, and had no college degree, but paid more than that amount in capital gains tax, made more than \$50k. On the other hand, 70.2% of those who paid below the threshold made less than \$50k.

```

train_plus <- train_plus %>%
  mutate(
    husband_or_wife = relationship %in% c("Husband", "Wife"),
    college_degree = husband_or_wife & education %in%
      c("Bachelors", "Doctorate", "Masters", "Prof-school")
  ) %>%
  bind_cols(
    predict(mod_tree, new_data = train, type = "class")
  ) %>%
  rename(income_dtreet = .pred_class)

cg_splits <- tribble(
  ~husband_or_wife, ~vals,
  TRUE, 5095.5,

```

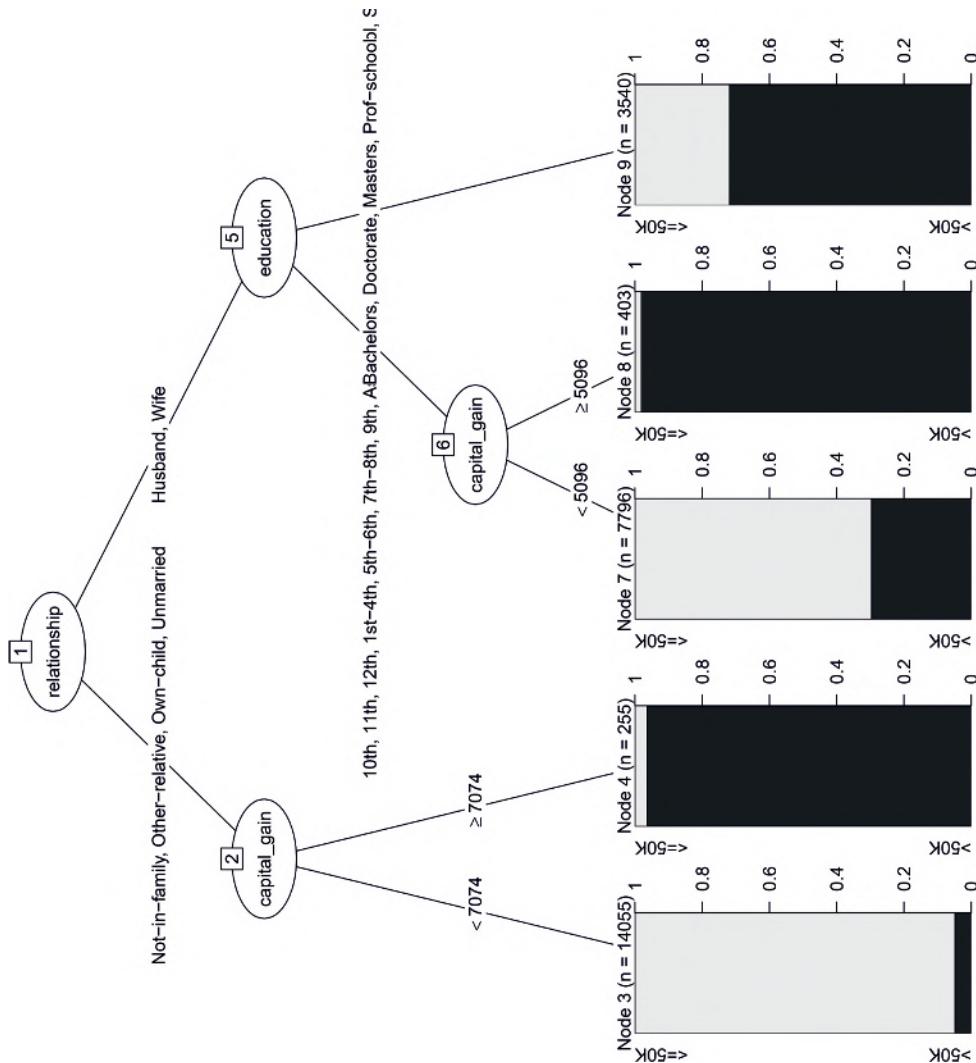


Figure 11.2: Decision tree for income using the census data.

```

FALSE, 7073.5
)

ggplot(data = train_plus, aes(x = capital_gain, y = income)) +
  geom_count(
    aes(color = income_dtreetree, shape = college_degree),
    position = position_jitter(width = 0, height = 0.1),
    alpha = 0.5
  ) +
  facet_wrap(~ husband_or_wife) +
  geom_vline(
    data = cg_splits, aes(xintercept = vals),
    color = "dodgerblue", lty = 2
  )

```

```
) +
  scale_x_log10()
```

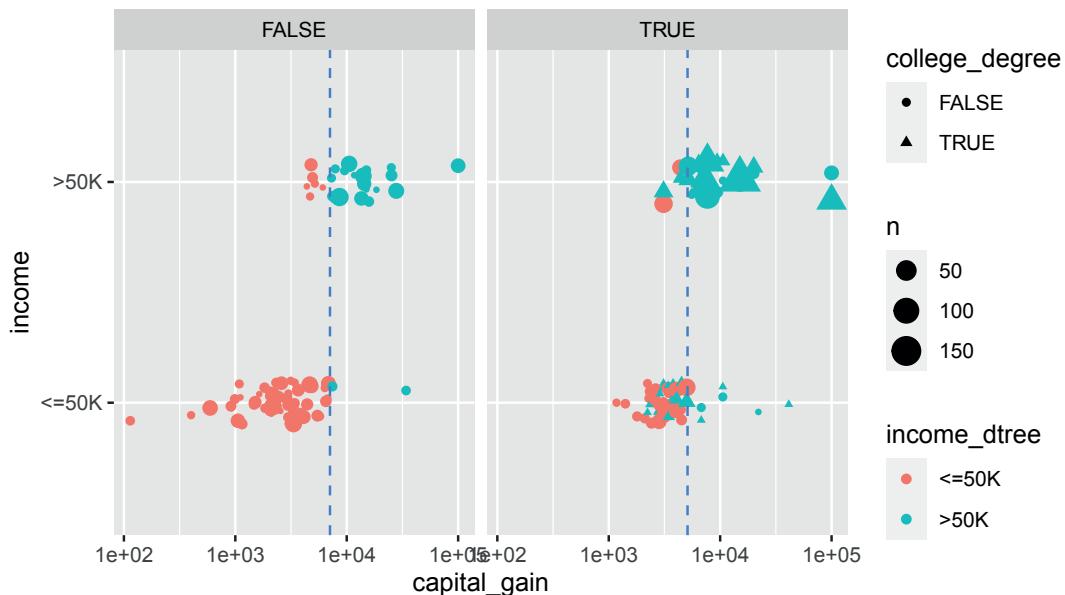


Figure 11.3: Graphical depiction of the full recursive partitioning decision classifier. On the left, those whose relationship status is neither 'Husband' nor 'Wife' are classified based on their capital gains paid. On the right, not only is the capital gains threshold different, but the decision is also predicated on whether the person has a college degree.

Since there are exponentially many trees, how did the algorithm know to pick this one? The *complexity parameter* controls whether to keep or prune possible splits. That is, the algorithm considers many possible splits (i.e., new branches on the tree), but prunes them if they do not sufficiently improve the predictive power of the model (i.e., bear fruit). By default, each split has to decrease the error by a factor of 1%. This will help to avoid *overfitting* (more on that later). Note that as we add more splits to our model, the relative error decreases.

```
printcp(mod_tree$fit)
```

```
Classification tree:
`rpart::rpart`(data = train)

Variables actually used in tree construction:
[1] capital_gain education    relationship

Root node error: 6206/26049 = 0.238

n= 26049

      CP nsplit rel error xerror     xstd
1 0.1255      0     1.000  1.000  0.01108
2 0.0627      2     0.749  0.749  0.00996
3 0.0382      3     0.686  0.686  0.00962
```

```
4 0.0100      4      0.648  0.648 0.00940
```

We can also use the model evaluation metrics we developed in [Chapter 10](#). Namely, the confusion matrix and the accuracy.

```
library(yardstick)
pred <- train %>%
  select(income) %>%
  bind_cols(
    predict(mod_tree, new_data = train, type = "class")
  ) %>%
  rename(income_dtreet = .pred_class)

confusion <- pred %>%
  conf_mat(truth = income, estimate = income_dtreet)
confusion

  Truth
Prediction <=50K >50K
  <=50K 18836 3015
  >50K   1007 3191

accuracy(pred, income, income_dtreet)
```

```
# A tibble: 1 x 3
  .metric  .estimator .estimate
  <chr>    <chr>        <dbl>
1 accuracy binary     0.846
```

In this case, the accuracy of the decision tree classifier is now 84.6%, a considerable improvement over the null model. Again, this is comparable to the analogous logistic regression model we build using this same set of variables in [Chapter 10](#). [Figure 11.4](#) displays the confusion matrix for this model.

```
autoplots(confusion) +
  geom_label(
    aes(
      x = (xmax + xmin) / 2,
      y = (ymax + ymin) / 2,
      label = c("TN", "FP", "FN", "TP")
    )
  )
```

11.1.1.1 Tuning parameters

The decision tree that we built previously was based on the default parameters. Most notably, our tree was pruned so that only splits that decreased the overall lack of fit by 1% were retained. If we lower this threshold to 0.2%, then we get a more complex tree.

```
mod_tree2 <- decision_tree(mode = "classification") %>%
  set_engine("rpart", control = rpart.control(cp = 0.002)) %>%
  fit(form, data = train)
```

Can you find the accuracy of this more complex tree. Is it more or less accurate than our original tree?

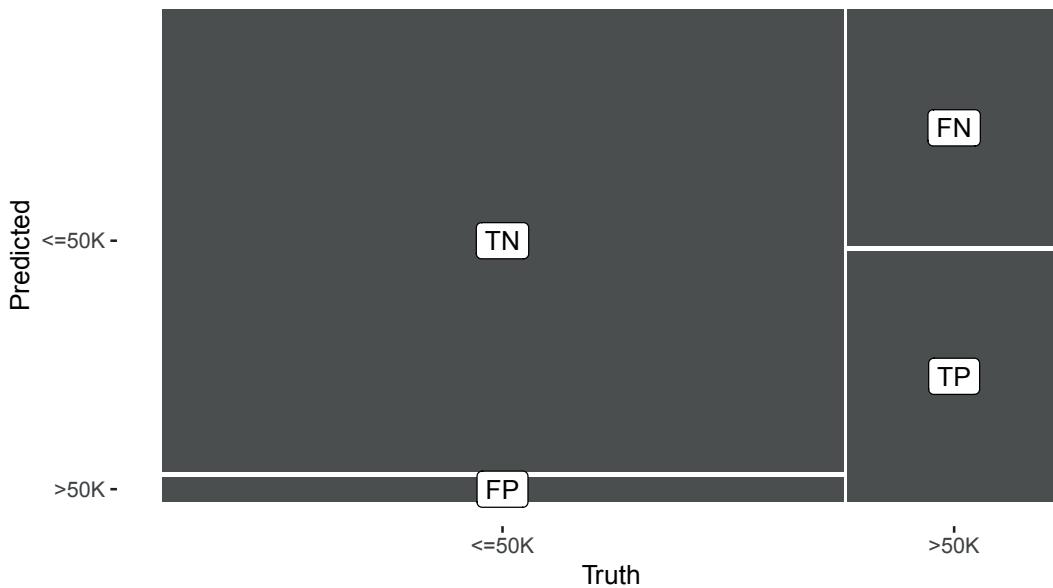


Figure 11.4: Visual summary of the predictive accuracy of our decision tree model. The largest rectangle represents the cases that are true negatives.

11.1.2 Random forests

A natural extension of a decision tree is a *random forest*. A random forest is collection of decision trees that are aggregated by majority rule. In a sense, a random forest is like a collection of bootstrapped (see Chapter 9) decision trees. A random forest is constructed by:

- Choosing the number of decision trees to grow (controlled by the `trees` argument) and the number of variables to consider in each tree (`mtry`)
- Randomly selecting the rows of the data frame *with replacement*
- Randomly selecting `mtry` variables from the data frame
- Building a decision tree on the resulting data set
- Repeating this procedure `trees` times

A prediction for a new observation is made by taking the majority rule from all of the decision trees in the forest. Random forests are available in **R** via the **randomForest** package. They can be very effective but are sometimes computationally expensive.

```
mod_forest <- rand_forest(
  mode = "classification",
  mtry = 3,
  trees = 201
) %>%
  set_engine("randomForest") %>%
  fit(form, data = train)

pred <- pred %>%
  bind_cols(
    predict(mod_forest, new_data = train, type = "class")
  ) %>%
```

```
rename(income_rf = .pred_class)

pred %>%
  conf_mat(income, income_rf)
```

		Truth	
		<=50K	>50K
Prediction	<=50K	19273	1251
	>50K	570	4955

```
pred %>%
  accuracy(income, income_rf)
```

```
# A tibble: 1 x 3
  .metric  .estimator .estimate
  <chr>    <chr>        <dbl>
1 accuracy binary     0.930
```

Because each tree in a random forest uses a different set of variables, it is possible to keep track of which variables seem to be the most consistently influential. This is captured by the notion of *importance*. While—unlike p-values in a regression model—there is no formal statistical inference here, importance plays an analogous role in that it may help to generate hypotheses. Here, we see that `capital_gain` and `age` seem to be influential, while `race` and `sex` do not.

```
randomForest::importance(mod_forest$fit) %>%
  as_tibble(rownames = "variable") %>%
  arrange(desc(MeanDecreaseGini))
```

```
# A tibble: 11 x 2
  variable      MeanDecreaseGini
  <chr>            <dbl>
1 capital_gain    1187.
2 relationship    1074.
3 age              1060.
4 education       755.
5 hours_per_week   661.
6 occupation       629.
7 marital_status    620.
8 capital_loss      410.
9 workclass         320.
10 race             132.
11 sex               86.5
```

The results are put into a `tibble` (simple data frame) to facilitate further wrangling. A model object of class `randomForest` also has a `predict()` method for making new predictions.

11.1.2.1 Tuning parameters

Hastie et al. (2009) recommend using \sqrt{p} variables in each classification tree (and $p/3$ for each regression tree), and this is the default behavior in `randomForest`. However, this is a parameter that can be tuned for a particular application. The number of trees is another parameter that can be tuned—we simply picked a reasonably large odd number.

11.1.3 Nearest neighbor

Thus far, we have focused on using data to build models that we can then use to predict outcomes on a new set of data. A slightly different approach is offered by *lazy learners*, which seek to predict outcomes without constructing a “model.” A very simple, yet widely-used approach is *k-nearest neighbor*.

Recall that data with p attributes (explanatory variables) are manifest as points in a p -dimensional space. The *Euclidean distance* between any two points in that space can be easily calculated in the usual way as the square root of the sum of the squared deviations. Thus, it makes sense to talk about the *distance* between two points in this p -dimensional space, and as a result, it makes sense to talk about the distance between two observations (rows of the data frame). Nearest-neighbor classifiers exploit this property by assuming that observations that are “close” to each other probably have similar outcomes.

Suppose we have a set of training data $(\mathbf{X}, y) \in \mathbb{R}^{n \times p} \times \mathbb{R}^n$. For some positive integer k , a k -nearest neighbor algorithm classifies a new observation x^* by:

- Finding the k observations in the training data \mathbf{X} that are closest to x^* , according to some distance metric (usually Euclidean). Let $D(x^*) \subseteq (\mathbf{X}, y)$ denote this set of observations.
- For some aggregate function f , computing $f(y)$ for the k values of y in $D(x^*)$ and assigning this value (y^*) as the predicted value of the response associated with x^* . The logic is that since x^* is similar to the k observations in $D(x^*)$, the response associated with x^* is likely to be similar to the responses in $D(x^*)$. In practice, simply taking the value shared by the majority (or a plurality) of the y 's is enough.

Note that a k -NN classifier does not need to process the training data before making new classifications—it can do this on the fly. A k -NN classifier is provided by the `kknn()` function in the `kknn` package. Note that since the distance metric only makes sense for quantitative variables, we have to restrict our data set to those first. Setting the `scale` to `TRUE` rescales the explanatory variables to have the same *standard deviation*. We choose $k = 5$ neighbors for reasons that we explain in the next section.

```
library(kknn)
# distance metric only works with quantitative variables
train_q <- train %>%
  select(income, where(is.numeric), -fnlwgt)

mod_knn <- nearest_neighbor(neighbors = 5, mode = "classification") %>%
  set_engine("kknn", scale = TRUE) %>%
  fit(income ~ ., data = train_q)

pred <- pred %>%
  bind_cols(
    predict(mod_knn, new_data = train, type = "class")
  ) %>%
  rename(income_knn = .pred_class)

pred %>%
  conf_mat(income, income_knn)
```

		Truth
		Prediction
Prediction	<=50K	>50K
<=50K	18533	2492

```
>50K    1310  3714
pred %>%
  accuracy(income, income_knn)

# A tibble: 1 x 3
  .metric   .estimator .estimate
  <chr>     <chr>        <dbl>
1 accuracy  binary      0.854
```

k-NN classifiers are widely used in part because they are easy to understand and code. They also don't require any pre-processing time. However, predictions can be slow, since the data must be processed at that time.

The usefulness of *k*-NN can depend importantly on the geometry of the data. Are the points clustered together? What is the distribution of the distances among each variable? A wider scale on one variable can dwarf a narrow scale on another variable.

11.1.3.1 Tuning parameters

An appropriate choice of *k* will depend on the application and the data. Cross-validation can be used to optimize the choice of *k*. Here, we compute the accuracy for several values of *k*.

```
knn_fit <- function(.data, k) {
  nearest_neighbor(neighbors = k, mode = "classification") %>%
    set_engine("kknn", scale = TRUE) %>%
    fit(income ~ ., data = .data)
}

knn_accuracy <- function(mod, .new_data) {
  mod %>%
    predict(new_data = .new_data) %>%
    mutate(income = .new_data$income) %>%
    accuracy(income, .pred_class) %>%
    pull(.estimate)
}

ks <- c(1:10, 15, 20, 30, 40, 50)

knn_tune <- tibble(
  k = ks,
  mod = map(k, knn_fit, .data = train_q),
  train_accuracy = map_dbl(mod, knn_accuracy, .new_data = train_q)
)
knn_tune

# A tibble: 5 x 3
  k mod      train_accuracy
  <dbl> <list>        <dbl>
1     1 <fit[+]>      0.846
2     5 <fit[+]>      0.854
3    10 <fit[+]>      0.848
4    20 <fit[+]>      0.846
5    40 <fit[+]>      0.841
```

In Figure 11.5, we show how the accuracy decreases as k increases. That is, if one seeks to maximize the accuracy rate *on this data set*, then the optimal value of k is 5.⁴ We will see why this method of optimizing the value of the parameter k is not robust when we learn about *cross-validation* below.

```
ggplot(data = knn_tune, aes(x = k, y = train_accuracy)) +
  geom_point() +
  geom_line() +
  ylab("Accuracy rate")
```

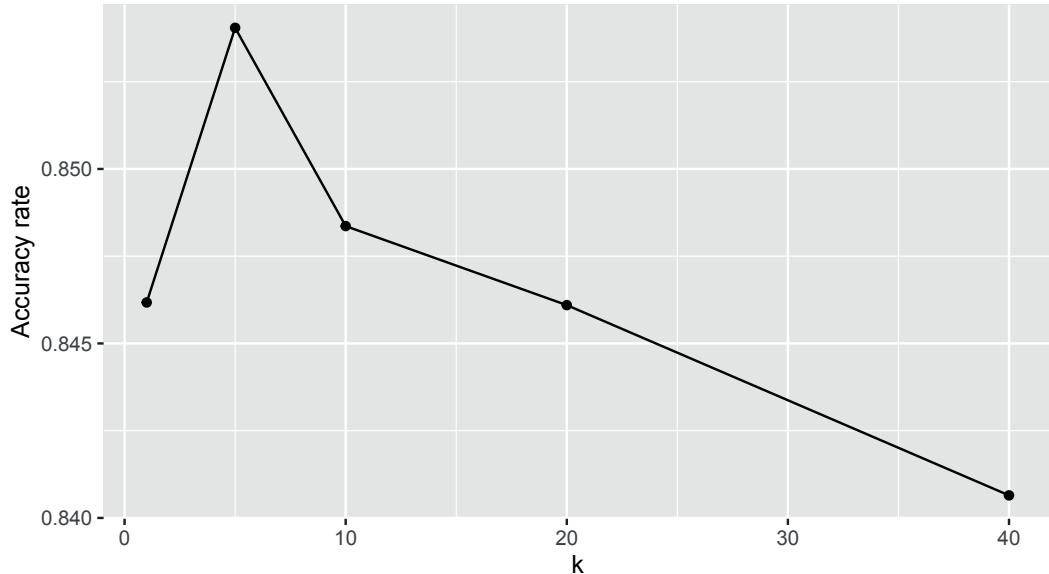


Figure 11.5: Performance of nearest-neighbor classifier for different choices of k on census training data.

11.1.4 Naïve Bayes

Another relatively simple classifier is based on Bayes Theorem. Bayes theorem is a very useful result from probability that allows conditional probabilities to be calculated from other conditional probabilities. It states:

$$\Pr(y|x) = \frac{\Pr(xy)}{\Pr(x)} = \frac{\Pr(x|y)\Pr(y)}{\Pr(x)}.$$

How does this relate to a naïve Bayes classifier? Suppose that we have a binary response variable y and we want to classify a new observation x^* (recall that x is a vector). Then if we can compute that the conditional probability $\Pr(y = 1|x^*) > \Pr(y = 0|x^*)$, we have evidence that $y = 1$ is a more likely outcome for x^* than $y = 0$. This is the crux of a naïve Bayes classifier. In practice, how we arrive at the estimates $\Pr(y = 1|x^*)$ are based on Bayes theorem and estimates of conditional probabilities derived from the training data (\mathbf{X}, y) .

Consider the first person in the training data set. This is a 39-year-old white male with a

⁴In section 11.2, we discuss why this particular optimization criterion might not be the wisest choice.

bachelor's degree working for a state government in a clerical role. In reality, this person made less than \$50,000.

```
train %>%
  as.data.frame() %>%
  head(1)
```

```
age workclass fnlwgt education education_1 marital_status occupation
1 39 State-gov 77516 Bachelors          13 Never-married Adm-clerical
  relationship race sex capital_gain capital_loss hours_per_week
1 Not-in-family White Male        2174           0             40
  native_country income
1 United-States <=50K
```

The naïve Bayes classifier would make a prediction for this person based on the probabilities observed in the data. For example, in this case the probability $\Pr(\text{male}|\text{male})$ of being male given that you had high income is 0.845, while the unconditional probability of being male is $\Pr(\text{male}) = 0.670$. We know that the overall probability of having high income is $\Pr(\text{high income}) = 0.243$. Bayes's rule tells us that the resulting probability of having high income given that one is male is:

$$\Pr(\text{high income}|\text{male}) = \frac{\Pr(\text{male}|\text{high income}) \cdot \Pr(\text{high income})}{\Pr(\text{male})} = \frac{0.845 \cdot 0.243}{0.670} = 0.306.$$

This simple example illustrates the case where we have a single explanatory variable (e.g., `sex`), but the naïve Bayes model extends to multiple variables by making the sometimes overly simplistic assumption that the explanatory variables are conditionally independent (hence the name “naïve”).

A naïve Bayes classifier is provided in **R** by the `naive_Bayes()` function from the **discrim** package. Note that like `lm()` and `glm()`, a `naive_Bayes()` object has a `predict()` method.

```
library(discrim)
mod_nb <- naive_Bayes(mode = "classification") %>%
  set_engine("klaR") %>%
  fit(form, data = train)

pred <- pred %>%
  bind_cols(
    predict(mod_nb, new_data = train, type = "class")
  ) %>%
  rename(income_nb = .pred_class)

accuracy(pred, income, income_nb)

# A tibble: 1 x 3
  .metric   .estimator .estimate
  <chr>     <chr>       <dbl>
1 accuracy  binary      0.824
```

11.1.5 Artificial neural networks

An *artificial neural network* is yet another classifier. While the impetus for the artificial neural network comes from a biological understanding of the brain, the implementation here is entirely mathematical.

```
mod_nn <- mlp(mode = "classification", hidden_units = 5) %>%
  set_engine("nnet") %>%
  fit(form, data = train)
```

A neural network is a directed graph (see [Chapter 20](#)) that proceeds in stages. First, there is one node for each input variable. In this case, because each factor level counts as its own variable, there are 57 input variables. These are shown on the left in [Figure 11.6](#). Next, there are a series of nodes specified as a *hidden layer*. In this case, we have specified five nodes for the hidden layer. These are shown in the middle of [Figure 11.6](#), and each of the input variables are connected to these hidden nodes. Each of the hidden nodes is connected to the single output variable. In addition, `nnet()` adds two control nodes, the first of which is connected to the five hidden nodes, and the latter is connected to the output node. The total number of edges is thus $pk + k + k + 1$, where k is the number of hidden nodes. In this case, there are $57 \cdot 5 + 5 + 5 + 1 = 296$ edges.

The algorithm iteratively searches for the optimal set of weights for each edge. Once the weights are computed, the neural network can make predictions for new inputs by running these values through the network.

```
pred <- pred %>%
  bind_cols(
    predict(mod_nn, new_data = train, type = "class")
  ) %>%
  rename(income_nn = .pred_class)

accuracy(pred, income, income_nn)

# A tibble: 1 x 3
  .metric   .estimator .estimate
  <chr>     <chr>        <dbl>
1 accuracy  binary      0.833
```

11.1.6 Ensemble methods

The benefit of having multiple classifiers is that they can be easily combined into a single classifier. Note that there is a real probabilistic benefit to having multiple prediction systems, especially if they are independent. For example, if you have three independent classifiers with error rates ϵ_1, ϵ_2 , and ϵ_3 , then the probability that all three are wrong is $\prod_{i=1}^3 \epsilon_i$. Since $\epsilon_i < 1$ for all i , this probability is lower than any of the individual error rates. Moreover, the probability that at least one of the classifiers is correct is $1 - \prod_{i=1}^3 \epsilon_i$, which will get closer to 1 as you add more classifiers—even if you have not improved the individual error rates!

Consider combining the five classifiers that we have built previously. Suppose that we build an ensemble classifier by taking the majority vote from each. Does this ensemble classifier outperform any of the individual classifiers? We can use the `rowwise()` and `c_across()` functions to easily compute these values.

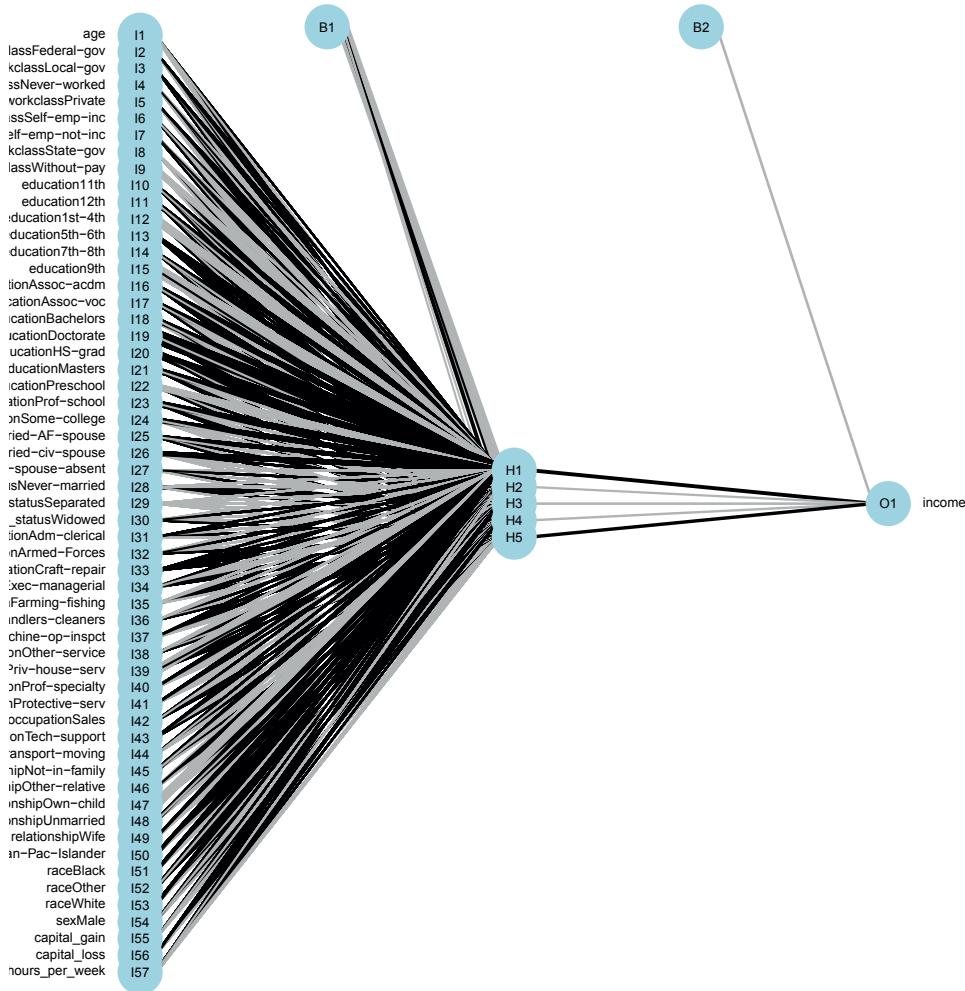


Figure 11.6: Visualization of an artificial neural network. The 57 input variables are shown on the left, with the five hidden nodes in the middle, and the single output variable on the right.

```

pred <- pred %>%
  rowwise() %>%
  mutate(
    rich_votes = sum(c_across(contains("income_")) == ">50K"),
    income_ensemble = factor(ifelse(rich_votes >= 3, ">50K", "<=50K"))
  ) %>%
  ungroup()

pred %>%
  select(-rich_votes) %>%
  pivot_longer(
    cols = -income,
    names_to = "model",
  )
  
```

```

  values_to = "prediction"
) %>%
group_by(model) %>%
summarize(accuracy = accuracy_vec(income, prediction)) %>%
arrange(desc(accuracy))

# A tibble: 6 x 2
  model      accuracy
  <chr>     <dbl>
1 income_rf 0.930
2 income_ensemble 0.878
3 income_knn 0.854
4 income_dtreet 0.846
5 income_nn 0.833
6 income_nb 0.824

```

In this case, the ensemble model achieves a 87.8% accuracy rate, which is slightly lower than our random forest. Thus, ensemble methods are a simple but effective way of hedging your bets.

11.2 Parameter tuning

In Section 11.1.3, we showed how after a certain point, the accuracy rate *on the training data* of the k -NN model increased as k increased. That is, as information from more neighbors—who are necessarily farther away from the target observation—was incorporated into the prediction for any given observation, those predictions got worse. This is not surprising, since the actual observation is in the training data set and that observation necessarily has distance 0 from the target observation. The error rate is not zero for $k = 1$ likely due to many points having the exact same coordinates in this five-dimensional space.

However, as seen in Figure 11.7, the story is different when evaluating the k -NN model *on the testing set*. Here, the truth is *not* in the training set, and so pooling information across more observations leads to *better* predictions—at least for a while. Again, this should not be surprising—we saw in Chapter 9 how means are less variable than individual observations. Generally, one hopes to minimize the misclassification rate on data that the model has not seen (i.e., the testing data) without introducing too much bias. In this case, that point occurs somewhere between $k = 5$ and $k = 10$. We can see this in Figure 11.7, since the accuracy on the testing data set improves rapidly up to $k = 5$, but then very slowly for larger values of k .

```

test_q <- test %>%
  select(income, where(is.numeric), -fnlwgt)

knn_tune <- knn_tune %>%
  mutate(test_accuracy = map_dbl(mod, knn_accuracy, .new_data = test_q))

knn_tune %>%
  select(-mod) %>%
  pivot_longer(-k, names_to = "type", values_to = "accuracy") %>%

```

```
ggplot(aes(x = k, y = accuracy, color = factor(type))) +
  geom_point() +
  geom_line() +
  ylab("Accuracy") +
  scale_color_discrete("Set")
```

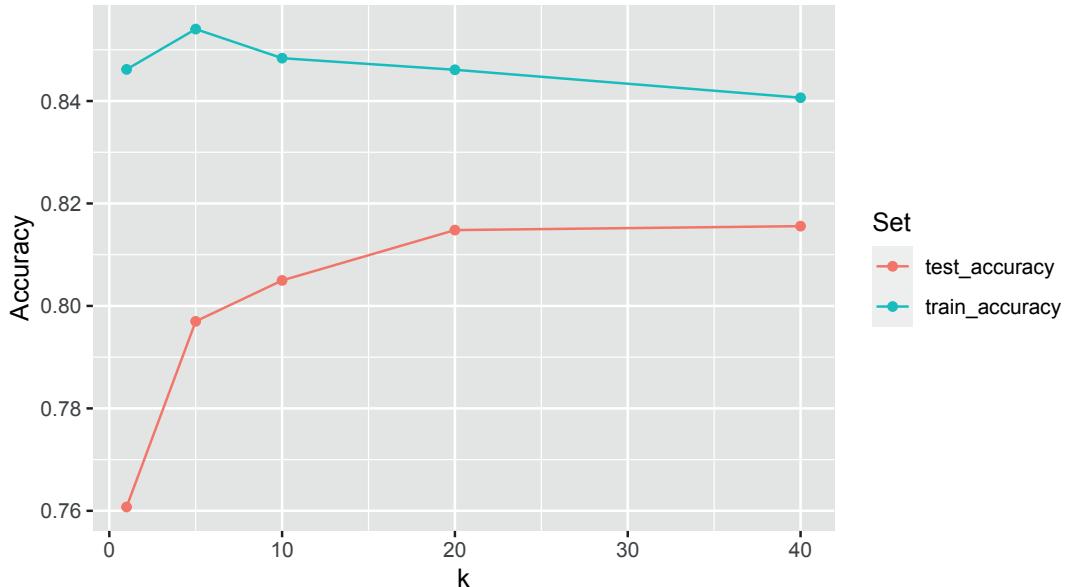


Figure 11.7: Performance of nearest-neighbor classifier for different choices of k on census training and testing data.

11.3 Example: Evaluation of income models redux

Just as we did in [Section 10.3.5](#), we should evaluate these new models on both the training and testing sets.

First, we build the null model that simply predicts that everyone makes \$50,000 with the same probability, regardless of the explanatory variables. (See [Appendix E](#) for an introduction to logistic regression.) We'll add this to the list of models that we built previously in this chapter.

```
mod_null <- logistic_reg(mode = "classification") %>%
  set_engine("glm") %>%
  fit(income ~ 1, data = train)

mod_log_all <- logistic_reg(mode = "classification") %>%
  set_engine("glm") %>%
  fit(form, data = train)

mods <- tibble(
  type = c(
```

```

    "null", "log_all", "tree", "forest",
    "knn", "neural_net", "naive_bayes"
),
mod = list(
  mod_null, mod_log_all, mod_tree, mod_forest,
  mod_knn, mod_nn, mod_nb
)
)

```

While each of the models we have fit have different classes in **R** (see B.3.6), each of those classes has a `predict()` method that will generate predictions.

```

map(mods$mod, class)

[[1]]
[1] "_glm"      "model_fit"

[[2]]
[1] "_glm"      "model_fit"

[[3]]
[1] "_rpart"    "model_fit"

[[4]]
[1] "_randomForest" "model_fit"

[[5]]
[1] "_train.kknn" "model_fit"

[[6]]
[1] "_nnet.formula" "model_fit"

[[7]]
[1] "_NaiveBayes" "model_fit"

```

Thus, we can iterate through the list of models and apply the appropriate `predict()` method to each object.

```

mods <- mods %>%
  mutate(
    y_train = list(pull(train, income)),
    y_test = list(pull(test, income)),
    y_hat_train = map(
      mod,
      ~pull(predict(.x, new_data = train, type = "class"), .pred_class)
    ),
    y_hat_test = map(
      mod,
      ~pull(predict(.x, new_data = test, type = "class"), .pred_class)
    )
  )

```

```
# A tibble: 7 x 6
  type      mod     y_train      y_test     y_hat_train   y_hat_test
  <chr>    <list>    <list>     <list>    <list>       <list>
1 null     <fct[+]> <fct [26,049~ <fct [6,512~ <fct [26,049~ <fct [6,512~
2 log_all   <fct[+]> <fct [26,049~ <fct [6,512~ <fct [26,049~ <fct [6,512~
3 tree      <fct[+]> <fct [26,049~ <fct [6,512~ <fct [26,049~ <fct [6,512~
4 forest    <fct[+]> <fct [26,049~ <fct [6,512~ <fct [26,049~ <fct [6,512~
5 knn       <fct[+]> <fct [26,049~ <fct [6,512~ <fct [26,049~ <fct [6,512~
6 neural_net <fct[+]> <fct [26,049~ <fct [6,512~ <fct [26,049~ <fct [6,512~
7 naive_bayes <fct[+]> <fct [26,049~ <fct [6,512~ <fct [26,049~ <fct [6,512~
```

We can also add our majority rule ensemble classifier. First, we write a function that will compute the majority vote when given a list of predictions.

```
predict_ensemble <- function(x) {
  majority <- ceiling(length(x) / 2)
  x %>%
    data.frame() %>%
    rowwise() %>%
    mutate(
      rich_votes = sum(c_across() == ">50K"),
      .pred_class = factor(ifelse(rich_votes >= majority , ">50K", "<=50K"))
    ) %>%
    pull(.pred_class) %>%
    fct_relevel("<=50K")
}
```

Next, we use `bind_rows()` to add an additional row to our models data frame with the relevant information for the ensemble classifier.

```
ensemble <- tibble(
  type = "ensemble",
  mod = NA,
  y_train = list(predict_ensemble(pull(mods, y_train))),
  y_test = list(predict_ensemble(pull(mods, y_test))),
  y_hat_train = list(predict_ensemble(pull(mods, y_hat_train))),
  y_hat_test = list(predict_ensemble(pull(mods, y_hat_test))),
)

mods <- mods %>%
  bind_rows(ensemble)
```

Now that we have the predictions for each model, we just need to compare them to the truth (y), and tally the results. We can do this using the `map2_dbl()` function from the `purrr` package.

```
mods <- mods %>%
  mutate(
    accuracy_train = map2_dbl(y_train, y_hat_train, accuracy_vec),
    accuracy_test = map2_dbl(y_test, y_hat_test, accuracy_vec),
    sens_test = map2_dbl(
      y_test,
      y_hat_test,
```

```

    sens_vec,
    event_level = "second"
),
spec_test = map2_dbl(y_test,
  y_hat_test,
  spec_vec,
  event_level = "second"
)
)

mods %>%
  select(-mod, -matches("^\$y")) %>%
  arrange(desc(accuracy_test))

```

	type	accuracy_train	accuracy_test	sens_test	spec_test
1	forest	0.930	0.861	0.605	0.946
2	ensemble	0.872	0.849	0.491	0.969
3	log_all	0.853	0.846	0.586	0.933
4	tree	0.846	0.840	0.510	0.951
5	neural_net	0.833	0.831	0.543	0.927
6	naive_bayes	0.824	0.814	0.319	0.980
7	knn	0.854	0.797	0.486	0.901
8	null	0.762	0.749	0	1

While the random forest performed notably better than the other models on the training set, its accuracy dropped the most on the testing set. We note that even though the k -NN model slightly outperformed the decision tree on the training set, the decision tree performed better on the testing set. The ensemble model and the logistic regression model performed quite well. In this case, however, the accuracy rates of all models were in the same ballpark on both the testing set.

In Figure 11.8, we compare the ROC curves for all census models on the testing data set.

```

mods <- mods %>%
  filter(type != "ensemble") %>%
  mutate(
    y_hat_prob_test = map(
      mod,
      ~pull(predict(.x, new_data = test, type = "prob"), `^.pred_>50K`)
    ),
    type = fct_reorder(type, sens_test, .desc = TRUE)
  )

mods %>%
  select(type, y_test, y_hat_prob_test) %>%
  unnest(cols = c(y_test, y_hat_prob_test)) %>%
  group_by(type) %>%
  roc_curve(truth = y_test, y_hat_prob_test, event_level = "second") %>%
  autoplot() +
  geom_point(
    data = mods,

```

```
aes(x = 1 - spec_test, y = sens_test, color = type),
size = 3
)
```

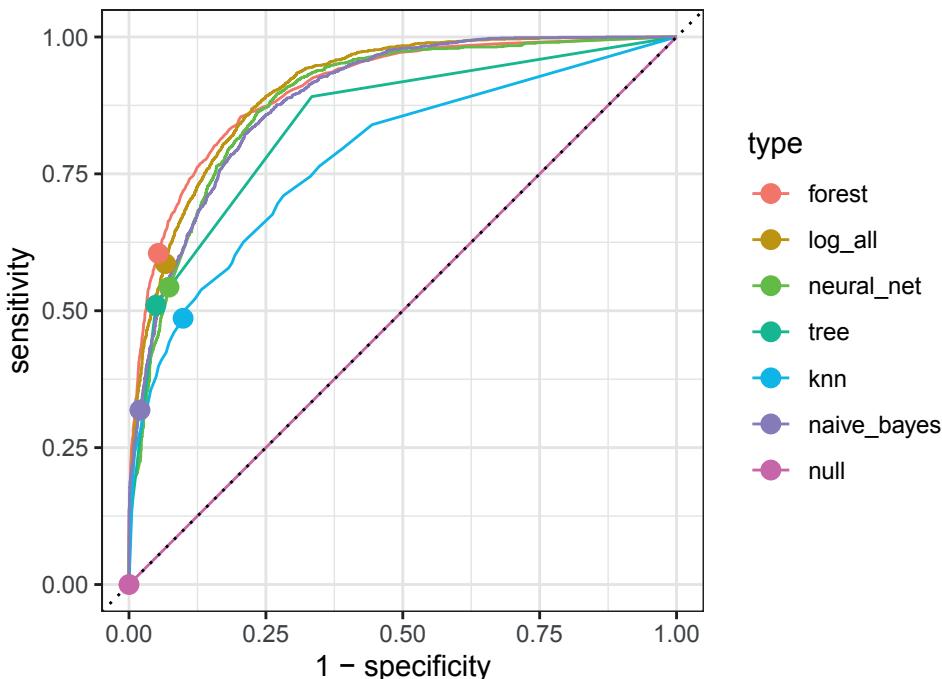


Figure 11.8: Comparison of ROC curves across five models on the Census testing data. The null model has a true positive rate of zero and lies along the diagonal. The naïve Bayes model has a lower true positive rate than the other models. The random forest may be the best overall performer, as its curve lies furthest from the diagonal.

11.4 Extended example: Who has diabetes this time?

Recall the example about diabetes in [Section 10.4](#).

```
library(NHANES)
people <- NHANES %>%
  select(Age, Gender, Diabetes, BMI, HHIncome, PhysActive) %>%
  drop_na()
glimpse(people)

Rows: 7,555
Columns: 6
$ Age      <int> 34, 34, 34, 49, 45, 45, 45, 66, 58, 54, 58, 50, 33, ...
$ Gender    <fct> male, male, male, female, female, female, female, ma...
$ Diabetes  <fct> No, ...
$ BMI       <dbl> 32.22, 32.22, 32.22, 30.57, 27.24, 27.24, 27.24, 23....
$ HHIncome  <fct> 25000-34999, 25000-34999, 25000-34999, 35000-44999, ...
```

```
$ PhysActive <fct> No, No, No, No, Yes, Yes, Yes, Yes, Yes, Yes, Yes, Y...
people %>%
  group_by(Diabetes) %>%
  count() %>%
  mutate(pct = n / nrow(people))

# A tibble: 2 x 3
# Groups:   Diabetes [2]
  Diabetes     n     pct
  <fct>    <int>  <dbl>
1 No          6871  0.909
2 Yes         684   0.0905
```

We illustrate the use of a decision tree using all of the variables except for household income in [Figure 11.9](#). From the original data shown in [Figure 11.10](#), it appears that older people, and those with higher BMIs, are more likely to have diabetes.

```
mod_diabetes <- decision_tree(mode = "classification") %>%
  set_engine(
    "rpart",
    control = rpart.control(cp = 0.005, minbucket = 30)
  ) %>%
  fit(Diabetes ~ Age + BMI + Gender + PhysActive, data = people)
mod_diabetes
```

parsnip model object

Fit time: 43ms
n= 7555

```
node), split, n, loss, yval, (yprob)
  * denotes terminal node

1) root 7555 684 No (0.909464 0.090536)
  2) Age< 52.5 5092 188 No (0.963079 0.036921) *
  3) Age>=52.5 2463 496 No (0.798620 0.201380)
    6) BMI< 39.985 2301 416 No (0.819209 0.180791) *
    7) BMI>=39.985 162  80 No (0.506173 0.493827)
      14) Age>=67.5 50  18 No (0.640000 0.360000) *
      15) Age< 67.5 112  50 Yes (0.446429 0.553571)
        30) Age< 60.5 71  30 No (0.577465 0.422535) *
        31) Age>=60.5 41  9 Yes (0.219512 0.780488) *
```

```
plot(as.party(mod_diabetes$fit))
```

If you are 52 or younger, then you very likely do not have diabetes. However, if you are 53 or older, your risk is higher. If your BMI is above 40—indicating obesity—then your risk increases again. Strangely—and this may be evidence of overfitting—your risk is highest if you are between 61 and 67 years old. This partition of the data is overlaid on [Figure 11.10](#).

```
segments <- tribble(
  ~Age, ~xend, ~BMI, ~yend,
  52.5, 100, 39.985, 39.985,
```

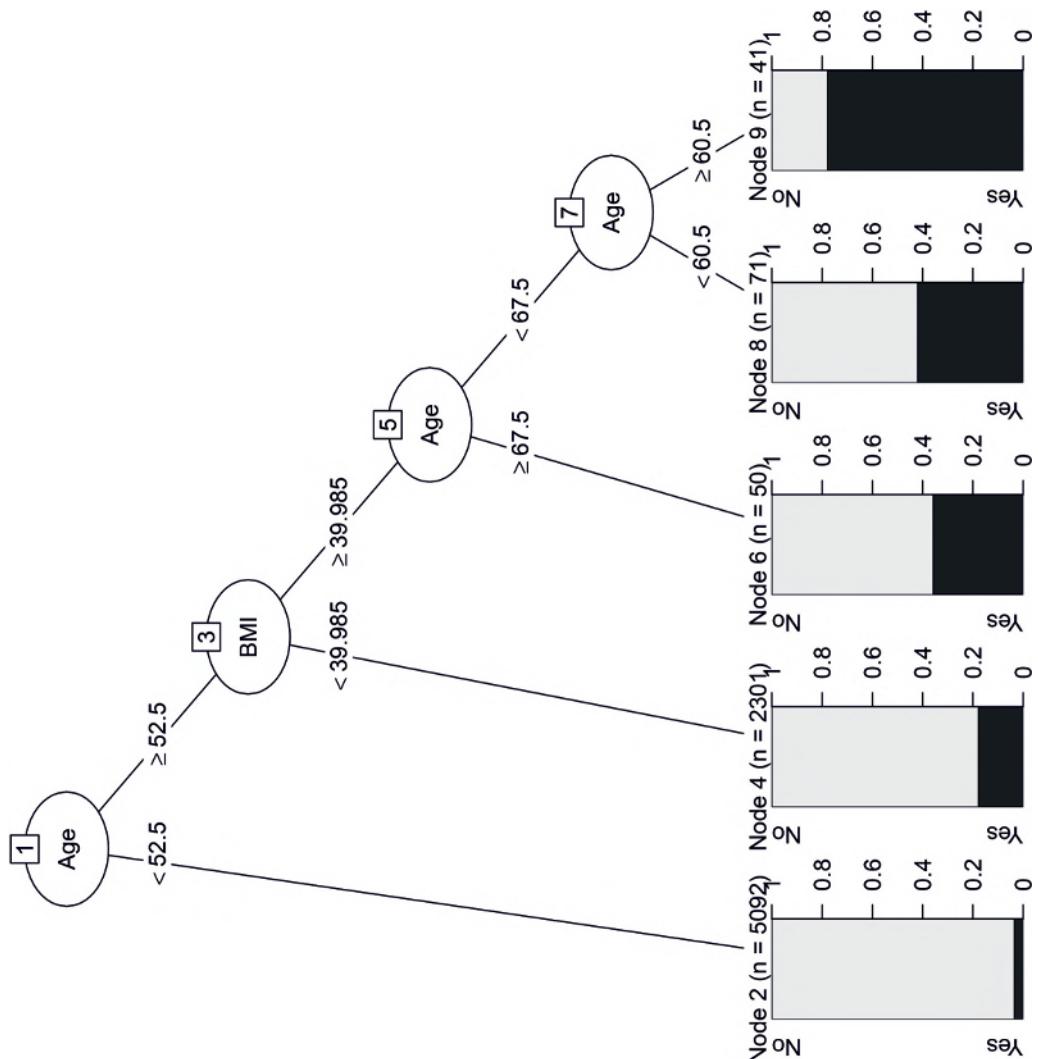


Figure 11.9: Illustration of decision tree for diabetes.

```

67.5, 67.5, 39.985, Inf,
60.5, 60.5, 39.985, Inf
)
ggplot(data = people, aes(x = Age, y = BMI)) +
  geom_count(aes(color = Diabetes), alpha = 0.5) +
  geom_vline(xintercept = 52.5) +
  geom_segment(
    data = segments,
    aes(xend = xend, yend = yend)
  ) +
  scale_fill_gradient(low = "white", high = "red") +
  scale_color_manual(values = c("gold", "black")) +
  annotate(
  
```

```
"rect", fill = "blue", alpha = 0.1,
xmin = 60.5, xmax = 67.5, ymin = 39.985, ymax = Inf
)
```

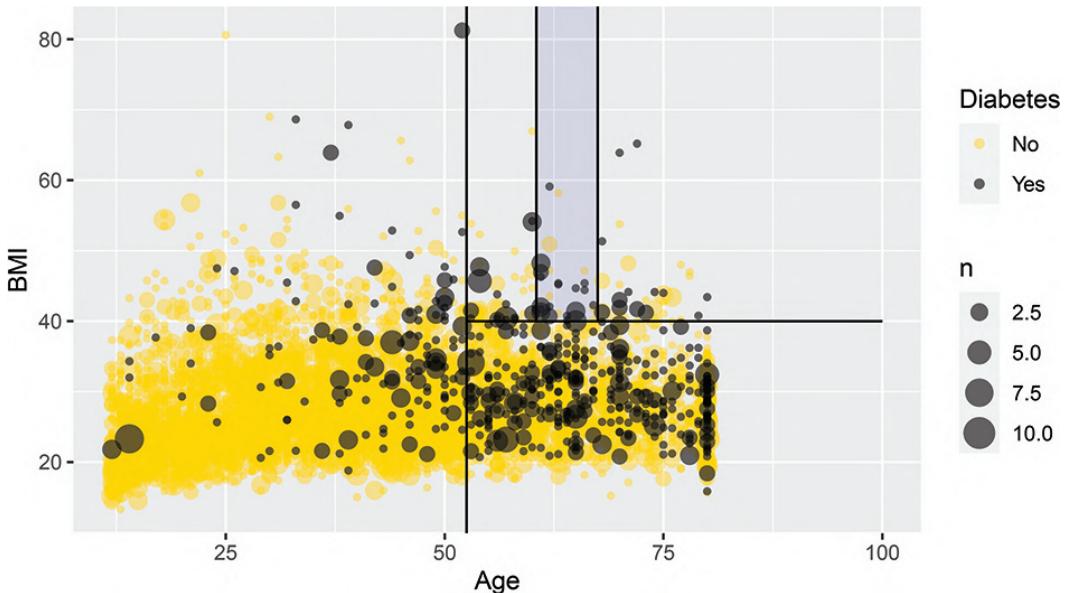


Figure 11.10: Scatterplot of age against BMI for individuals in the NHANES data set. The black dots represent a collection of people with diabetes, while the gold dots represent those without diabetes.

[Figure 11.10](#) is a nice way to visualize a complex model. We have plotted our data in two quantitative dimensions (`Age` and `BMI`) while using color to represent our binary response variable (`Diabetes`). The decision tree simply partitions this two-dimensional space into axis-parallel rectangles. The model makes the same prediction for all observations within each rectangle. It is not hard to imagine—although it is hard to draw—how this recursive partitioning will scale to higher dimensions.

Note, however, that [Figure 11.10](#) provides a clear illustration of the strengths and weaknesses of models based on recursive partitioning. These types of models can *only* produce axis-parallel rectangles in which all points in each rectangle receive the same prediction. This makes these models relatively easy to understand and apply, but it is not hard to imagine a situation in which they might perform miserably (e.g., what if the relationship was non-linear?). Here again, this underscores the importance of visualizing your model *in the data space* (Wickham et al., 2015) as demonstrated in [Figure 11.10](#).

11.4.1 Comparing all models

We close the loop by extending this model visualization exercise to all of our models.

Once again, we tile the (Age, BMI) -plane with a fine grid of 10,000 points.

```
library(modelr)
fake_grid <- data_grid(
  people,
```

```

Age = seq_range(Age, 100),
BMI = seq_range(BMI, 100)
)

```

Next, we evaluate each of our six models on each grid point, taking care to retrieve not the classification itself, but the probability of having diabetes.

```

form <- as.formula("Diabetes ~ Age + BMI")

dmod_null <- logistic_reg(mode = "classification") %>%
  set_engine("glm")

dmod_tree <- decision_tree(mode = "classification") %>%
  set_engine("rpart", control = rpart.control(cp = 0.005, minbucket = 30))

dmod_forest <- rand_forest(
  mode = "classification",
  trees = 201,
  mtry = 2
) %>%
  set_engine("randomForest")

dmod_knn <- nearest_neighbor(mode = "classification", neighbors = 5) %>%
  set_engine("knn", scale = TRUE)

dmod_nnet <- mlp(mode = "classification", hidden_units = 6) %>%
  set_engine("nnet")

dmod_nb <- naive_Bayes() %>%
  set_engine("klaR")

bmi_mods <- tibble(
  type = c(
    "Logistic Regression", "Decision Tree", "Random Forest",
    "k-Nearest-Neighbor", "Neural Network", "Naive Bayes"
  ),
  spec = list(
    dmod_null, dmod_tree, dmod_forest, dmod_knn, dmod_nnet, dmod_nb
  ),
  mod = map(spec, fit, form, data = people),
  y_hat = map(mod, predict, new_data = fake_grid, type = "prob")
)

bmi_mods <- bmi_mods %>%
  mutate(
    X = list(fake_grid),
    yX = map2(y_hat, X, bind_cols)
  )

res <- bmi_mods %>%
  select(type, yX) %>%

```

```

unnest(cols = yX)
res

# A tibble: 60,000 x 5
  type          .pred_No .pred_Yes   Age   BMI
  <chr>        <dbl>    <dbl> <dbl> <dbl>
1 Logistic Regression 0.998  0.00234   12  13.3
2 Logistic Regression 0.998  0.00249   12  14.0
3 Logistic Regression 0.997  0.00265   12  14.7
4 Logistic Regression 0.997  0.00282   12  15.4
5 Logistic Regression 0.997  0.00300   12  16.0
6 Logistic Regression 0.997  0.00319   12  16.7
7 Logistic Regression 0.997  0.00340   12  17.4
8 Logistic Regression 0.996  0.00361   12  18.1
9 Logistic Regression 0.996  0.00384   12  18.8
10 Logistic Regression 0.996  0.00409  12  19.5
# ... with 59,990 more rows

```

Figure 11.11 illustrates each model in the data space. The differences between the models are striking. The rigidity of the decision tree is apparent, especially relative to the flexibility of the k -NN model. The k -NN model and the random forest have similar flexibility, but regions in the former are based on polygons, while regions in the latter are based on rectangles. Making k larger would result in smoother k -NN predictions, while making k smaller would make the predictions more bold. The logistic regression model makes predictions with a smooth grade, while the naïve Bayes model produces a non-linear horizon. The neural network has made relatively uniform predictions in this case.

```

ggplot(data = res, aes(x = Age, y = BMI)) +
  geom_tile(aes(fill = .pred_Yes), color = NA) +
  geom_count(
    data = people,
    aes(color = Diabetes), alpha = 0.4
  ) +
  scale_fill_gradient("Prob of\nDiabetes", low = "white", high = "red") +
  scale_color_manual(values = c("gold", "black")) +
  scale_size(range = c(0, 2)) +
  scale_x_continuous(expand = c(0.02, 0)) +
  scale_y_continuous(expand = c(0.02, 0)) +
  facet_wrap(~type, ncol = 2)

```

11.5 Regularization

Regularization is a technique where constraints are added to a regression model to prevent overfitting. Two techniques for *regularization* include *ridge regression* and the *LASSO* (least absolute shrinkage and selection operator). Instead of fitting a model that minimizes $\sum_{i=1}^n (y - \hat{y})^2$ where $\hat{y} = \mathbf{X}'\beta$, ridge regression adds a constraint that $\sum_{j=1}^p \beta_j^2 \leq c_1$ and the LASSO imposes the constraint that $\sum_{j=1}^p |\beta_j| \leq c_2$, for some constants c_1 and c_2 .

These methods are considered part of statistical or machine learning since they automate

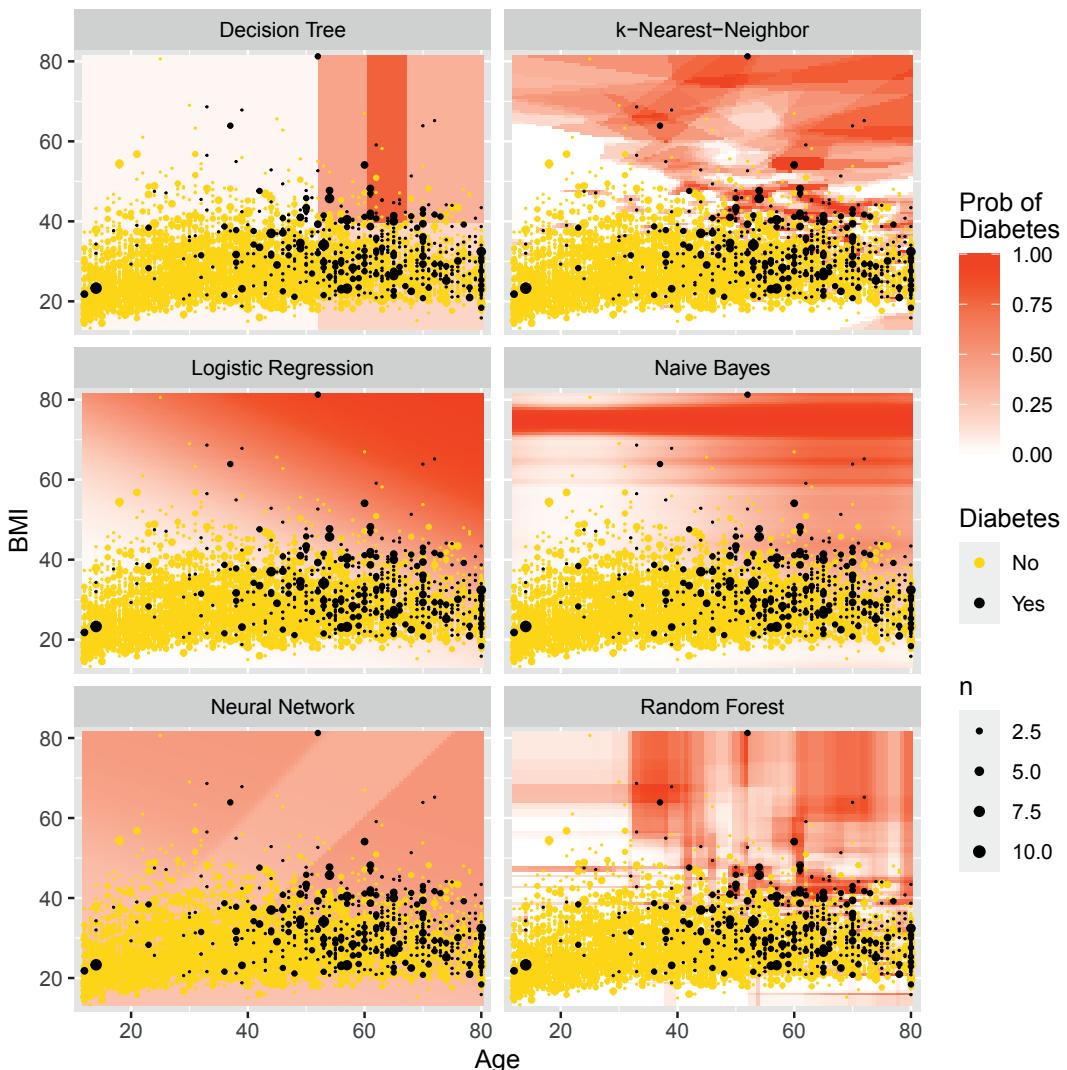


Figure 11.11: Comparison of predictive models in the data space. Note the rigidity of the decision tree, the flexibility of k -NN and the random forest, and the bold predictions of k -NN.

model selection by shrinking coefficients (for ridge regression) or retaining predictors (for the LASSO) automatically. Such *shrinkage* may induce bias but decrease variability. These regularization methods are particularly helpful when the set of predictors is large.

To help illustrate this process we consider a model for the flight delays example introduced in [Chapter 9](#). Here we are interested in arrival delays for flights from the two New York City airports that service California (EWR and JFK) to four California airports.

```
library(nycflights13)
California <- flights %>%
  filter(
    dest %in% c("LAX", "SFO", "OAK", "SJC"),
    !is.na(arr_delay)
```

```
) %>%
mutate(
  day = as.Date(time_hour),
  dow = as.character(lubridate::wday(day, label = TRUE)),
  month = as.factor(month),
  hour = as.factor(hour)
)
dim(California)
```

```
[1] 29836    20
```

We begin by splitting the data into a training set (70%) and testing set (30%).

```
library(broom)
set.seed(386)
California_split <- initial_split(California, prop = 0.7)
California_train <- training(California_split)
California_test <- testing(California_split)
```

Now we can build a model that includes variables we want to use to explain arrival delay, including hour of day, originating airport, arrival airport, carrier, month of the year, day of week, plus interactions between destination and day of week and month.

```
flight_model <- formula(
  "arr_delay ~ origin + dest + hour + carrier + month + dow")
mod_reg <- linear_reg() %>%
  set_engine("lm") %>%
  fit(flight_model, data = California_train)
tidy(mod_reg) %>%
  head(4)
```

```
# A tibble: 4 x 5
  term      estimate std.error statistic p.value
  <chr>     <dbl>     <dbl>     <dbl>     <dbl>
1 (Intercept) -10.5      5.85    -1.80  0.0719
2 originJFK      3.08     0.789     3.90  0.0000961
3 destOAK       -6.11      3.11    -1.97  0.0493
4 destSFO        1.80      0.625     2.88  0.00396
```

Our regression coefficient for `originJFK` indicates that controlling for other factors, we would anticipate an additional 3.1-minute delay flying from JFK compared to EWR (Newark), the reference airport.

```
California_test %>%
  select(arr_delay) %>%
  bind_cols(predict(mod_reg, new_data = California_test)) %>%
  metrics(truth = arr_delay, estimate = .pred)
```

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>       <dbl>
1 rmse    standard     42.0
2 rsq     standard     0.0877
3 mae     standard     26.4
```

Next we fit a LASSO model to the same data.

```
mod_lasso <- linear_reg(penalty = 0.01, mixture = 1) %>%
  set_engine("glmnet") %>%
  fit(flight_model, data = California_train)
tidy(mod_lasso) %>%
  head(4)

# A tibble: 4 x 3
  term      estimate penalty
  <chr>     <dbl>    <dbl>
1 (Intercept) -8.86    0.01
2 originJFK     2.98    0.01
3 destOAK      -5.96    0.01
4 destSFO       1.79    0.01
```

We see that the coefficients for the LASSO tend to be attenuated slightly towards 0 (e.g., `originJFK` has shifted from 3.08 to 2.98).

```
California_test %>%
  select(arr_delay) %>%
  bind_cols(predict(mod_lasso, new_data = California_test)) %>%
  metrics(truth = arr_delay, estimate = .pred)

# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>     <dbl>
1 rmse    standard    42.0
2 rsq     standard    0.0877
3 mae     standard    26.4
```

In this example, the LASSO hasn't improved the performance of our model on the test data. In situations where there are many more predictors and the model may be overfit, it will tend to do better.

11.6 Further resources

James et al. (2013) provides an accessible introduction to these topics (see <http://www-bcf.usc.edu/~gareth/ISL>). A graduate-level version of Hastie et al. (2009) is freely downloadable at <http://www-stat.stanford.edu/~tibs/ElemStatLearn>. Another helpful source is Tan et al. (2006), which has more of a computer science flavor. Breiman (2001) is a classic paper that describes two cultures in statistics: prediction and modeling. Efron (2020) offers a more recent perspective.

The `cmtree()` function from the **partykit** package builds a recursive partitioning model using conditional inference trees. The functionality is similar to `rpart()` but uses different criteria to determine the splits. The **partykit** package also includes a `cforest()` function. The **caret** package provides a number of useful functions for training and plotting classification and regression models. The **glmnet** and **lars** packages include support for regularization methods. The **RWeka** package provides an **R** interface to the comprehensive Weka machine learning library, which is written in Java.

11.7 Exercises

Problem 1 (Easy): Use the `HELPrcf` data from the `mosaicData` to fit a tree model to the following predictors: `age`, `sex`, `cesd`, and `substance`.

- Plot the resulting tree and interpret the results.
- What is the accuracy of your decision tree?

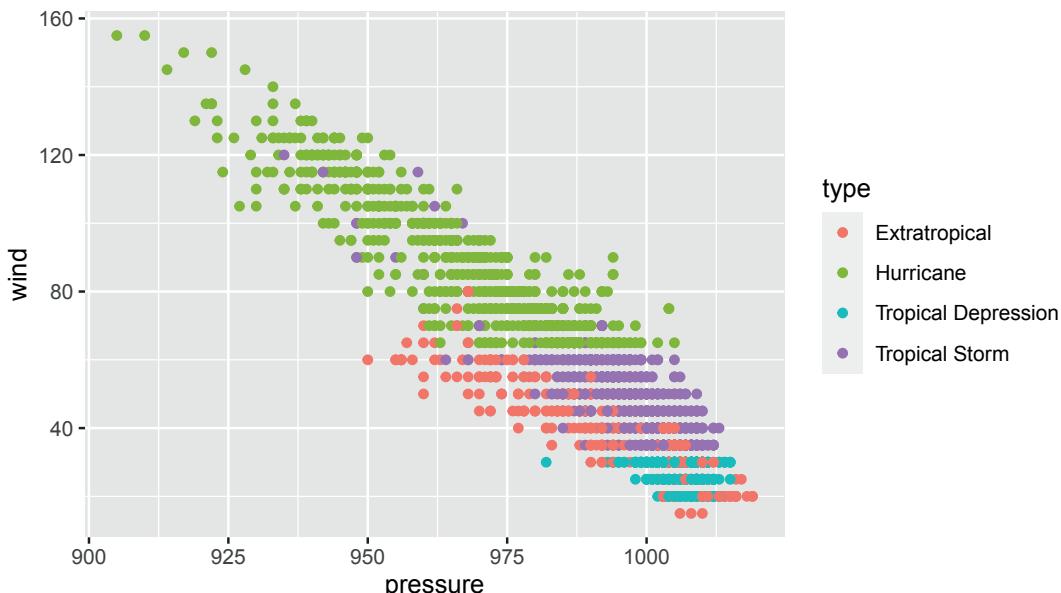
Problem 2 (Medium): Fit a series of supervised learning models to predict arrival delays for flights from New York to SFO using the `nycflights13` package. How do the conclusions change from the multiple regression model presented in the Statistical Foundations chapter?

Problem 3 (Medium): Use the College Scorecard Data from the `CollegeScorecard` package to model student debt as a function of institutional characteristics using the techniques described in this chapter. Compare and contrast results from at least three methods.

```
# remotes::install_github("Amherst-Statistics/CollegeScorecard")
library(CollegeScorecard)
```

Problem 4 (Medium): The `nasaweather` package contains data about tropical storms from 1995–2005. Consider the scatterplot between the wind speed and pressure of these storms shown below.

```
library(mdsr)
library(nasaweather)
ggplot(data = storms, aes(x = pressure, y = wind, color = type)) +
  geom_point(alpha = 0.5)
```



The type of storm is present in the data, and four types are given: extratropical, hurricane, tropical depression, and tropical storm. There are complicated and not terribly precise

definitions for storm type. Build a classifier for the `type` of each storm as a function of its `wind speed` and `pressure`.

Why would a decision tree make a particularly good classifier for these data? Visualize your classifier in the data space.

Problem 5 (Medium): Pre-natal care has been shown to be associated with better health of babies and mothers. Use the `NHANES` data set in the `NHANES` package to develop a predictive model for the `PregnantNow` variable. What did you learn about who is pregnant?

Problem 6 (Hard): The ability to get a good night's sleep is correlated with many positive health outcomes. The `NHANES` data set contains a binary variable `SleepTrouble` that indicates whether each person has trouble sleeping.

- a. For each of the following models:

- Build a classifier for `SleepTrouble`
- Report its effectiveness on the `NHANES` training data
- Make an appropriate visualization of the model
- Interpret the results. What have you learned about people's sleeping habits?

You may use whatever variable you like, except for `SleepHrsNight`.

- Null model
- Logistic regression
- Decision tree
- Random forest
- Neural network
- Naive Bayes
- k -NN

- b. Repeat the previous exercise, but now use the quantitative response variable `SleepHrsNight`. Build and interpret the following models:

- Null model
 - Multiple regression
 - Regression tree
 - Random forest
 - Ridge regression
 - LASSO
- c. Repeat either of the previous exercises, but this time first separate the `NHANES` data set uniformly at random into 75% training and 25% testing sets. Compare the effectiveness of each model on training vs. testing data.
 - d. Repeat the first exercise in part (a), but for the variable `PregnantNow`. What did you learn about who is pregnant?

11.8 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-learningI.html#learningI-online-exercises>



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

12

Unsupervised learning

In the previous chapter, we explored models for learning about a response variable y from a set of explanatory variables \mathbf{X} . This process is called *supervised learning* because the response variable provides not just a clear goal for the modeling (to improve predictions about future y 's), but also a guide (sometimes called the "ground truth"). In this chapter, we explore techniques in *unsupervised learning*, where there is no response variable y . Here, we simply have a set of observations \mathbf{X} , and we want to understand the relationships among them.

12.1 Clustering

Figure 12.1 shows an evolutionary tree of mammals. We humans (hominidae) are on the far left of the tree. The numbers at the branch points are estimates of how long ago—in millions of years—the branches separated. According to the diagram, rodents and primates diverged about 90 million years ago.

How do evolutionary biologists construct a tree like this? They study various traits of different kinds of mammals. Two mammals that have similar traits are deemed closely related. Animals with dissimilar traits are distantly related. By combining all of this information about the proximity of species, biologists can propose these kinds of evolutionary trees.

A tree—sometimes called a *dendrogram*—is an attractive organizing structure for relationships. Evolutionary biologists imagine that at each branch point there was an actual animal whose descendants split into groups that developed in different directions. In evolutionary biology, the inferences about branches come from comparing existing creatures to one another (as well as creatures from the fossil record). Creatures with similar traits are in nearby branches while creatures with different traits are in faraway branches. It takes considerable expertise in anatomy and morphology to know which similarities and differences are important. Note, however, that there is no outcome variable—just a construction of what is closely related or distantly related.

Trees can describe degrees of similarity between different things, regardless of how those relationships came to be. If you have a set of objects or cases, and you can measure how similar any two of the objects are, you can construct a tree. The tree may or may not reflect some deeper relationship among the objects, but it often provides a simple way to visualize relationships.

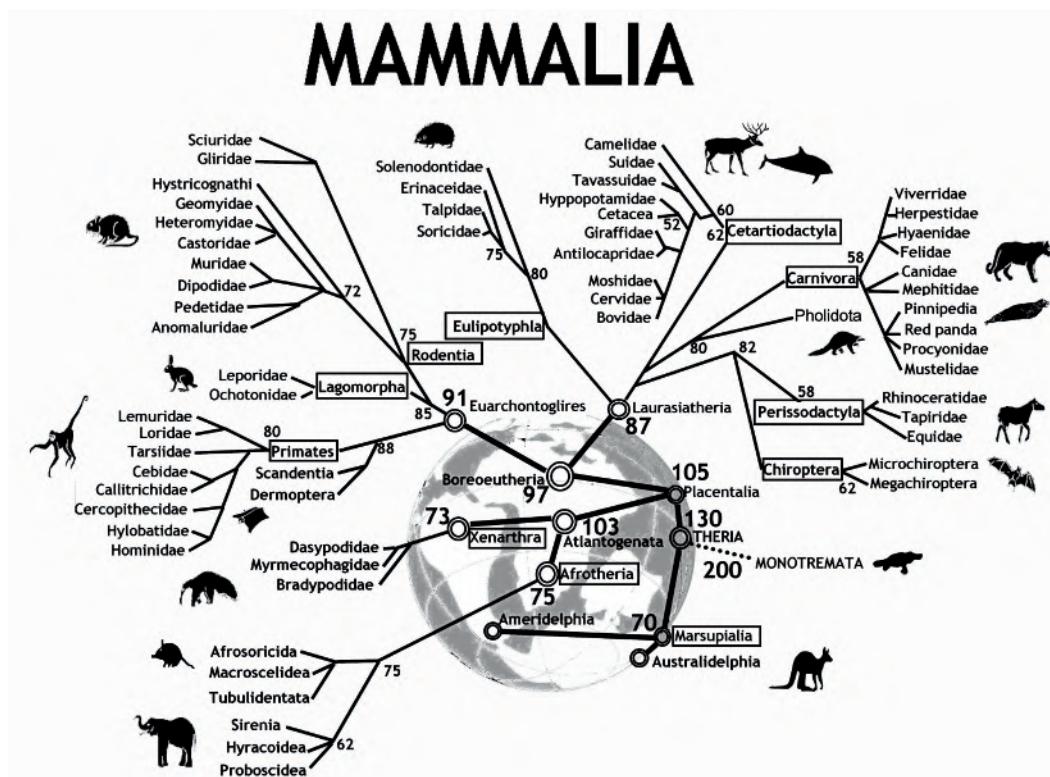


Figure 12.1: An evolutionary tree for mammals. Reprinted with permission under Creative Commons Attribution 2.0 Generic. No changes were made to this image. Source: Graphodatsky et al. (2011)

12.1.1 Hierarchical clustering

When the description of an object consists of a set of numerical variables (none of which is a *response variable*), there are two main steps in constructing a tree to describe the relationship among the cases in the data:

1. Represent each case as a point in a Cartesian space.
2. Make branching decisions based on how close together points or clouds of points are.

To illustrate, consider the unsupervised learning process of identifying different types of cars. The *United States Department of Energy* maintains automobile characteristics for thousands of cars: miles per gallon, engine size, number of cylinders, number of gears, etc. Please see their guide for more information.¹ Here, we download a ZIP file from their website that contains fuel economy rating for the 2016 model year.

```
src <- "https://www.fueleconomy.gov/feg/epadata/16data.zip"
lcl <- usethis::use_zip(src)
```

Next, we use the **readxl** package to read this file into **R**, clean up some of the resulting

¹<https://www.fueleconomy.gov/feg/pdfs/guides/FEG2016.pdf>

variable names, select a small subset of the variables, and filter for distinct models of *Toyota* vehicles. The resulting data set contains information about 75 different models that Toyota produces.

```
library(tidyverse)
library(mdsr)
library(readxl)
filename <- fs::dir_ls("data", regexp = "public\\*.xlsx") %>%
  head(1)
cars <- read_excel(filename) %>%
  janitor::clean_names() %>%
  select(
    make = mfr_name,
    model = carline,
    displacement = eng_displ,
    number_cyl,
    number_gears,
    city_mpg = city_fe_guide_conventional_fuel,
    hwy_mpg = hwy_fe_guide_conventional_fuel
  ) %>%
  distinct(model, .keep_all = TRUE) %>%
  filter(make == "Toyota")
glimpse(cars)
```

Rows: 75
 Columns: 7

	make	model	displacement	number_cyl	number_gears	city_mpg	hwy_mpg
1	Toyota	FR-S	2.0	4	6	25	34
2	Toyota	RC 200t	2.0	6	8	22	32
3	Toyota	RC 300 AWD	3.5	6	8	19	26
4	Toyota	RC 350	3.5	6	8	19	25
5	Toyota	RC 350...	5.0	4	6	33	42
6	Toyota		1.5	4	4	16	40
7	Toyota		1.8	4	4	22	24
8	Toyota		5.0	6	6	43	33
9	Toyota		2.0	6	4	19	28
10	Toyota		4.0	4	4	19	26
11	Toyota		4.0	4	4	19	26
12	Toyota		4.0	4	4	19	26
13	Toyota		4.0	4	4	19	26
14	Toyota		4.0	4	4	19	26
15	Toyota		4.0	4	4	19	26
16	Toyota		4.0	4	4	19	26
17	Toyota		4.0	4	4	19	26
18	Toyota		4.0	4	4	19	26
19	Toyota		4.0	4	4	19	26
20	Toyota		4.0	4	4	19	26
21	Toyota		4.0	4	4	19	26
22	Toyota		4.0	4	4	19	26
23	Toyota		4.0	4	4	19	26
24	Toyota		4.0	4	4	19	26
25	Toyota		4.0	4	4	19	26
26	Toyota		4.0	4	4	19	26
27	Toyota		4.0	4	4	19	26
28	Toyota		4.0	4	4	19	26
29	Toyota		4.0	4	4	19	26
30	Toyota		4.0	4	4	19	26
31	Toyota		4.0	4	4	19	26
32	Toyota		4.0	4	4	19	26
33	Toyota		4.0	4	4	19	26
34	Toyota		4.0	4	4	19	26
35	Toyota		4.0	4	4	19	26
36	Toyota		4.0	4	4	19	26
37	Toyota		4.0	4	4	19	26
38	Toyota		4.0	4	4	19	26
39	Toyota		4.0	4	4	19	26
40	Toyota		4.0	4	4	19	26
41	Toyota		4.0	4	4	19	26
42	Toyota		4.0	4	4	19	26
43	Toyota		4.0	4	4	19	26
44	Toyota		4.0	4	4	19	26
45	Toyota		4.0	4	4	19	26
46	Toyota		4.0	4	4	19	26
47	Toyota		4.0	4	4	19	26
48	Toyota		4.0	4	4	19	26
49	Toyota		4.0	4	4	19	26
50	Toyota		4.0	4	4	19	26
51	Toyota		4.0	4	4	19	26
52	Toyota		4.0	4	4	19	26
53	Toyota		4.0	4	4	19	26
54	Toyota		4.0	4	4	19	26
55	Toyota		4.0	4	4	19	26
56	Toyota		4.0	4	4	19	26
57	Toyota		4.0	4	4	19	26
58	Toyota		4.0	4	4	19	26
59	Toyota		4.0	4	4	19	26
60	Toyota		4.0	4	4	19	26
61	Toyota		4.0	4	4	19	26
62	Toyota		4.0	4	4	19	26
63	Toyota		4.0	4	4	19	26
64	Toyota		4.0	4	4	19	26
65	Toyota		4.0	4	4	19	26
66	Toyota		4.0	4	4	19	26
67	Toyota		4.0	4	4	19	26
68	Toyota		4.0	4	4	19	26
69	Toyota		4.0	4	4	19	26
70	Toyota		4.0	4	4	19	26
71	Toyota		4.0	4	4	19	26
72	Toyota		4.0	4	4	19	26
73	Toyota		4.0	4	4	19	26
74	Toyota		4.0	4	4	19	26
75	Toyota		4.0	4	4	19	26

As a large automaker, Toyota has a diverse lineup of cars, trucks, SUVs, and hybrid vehicles. Can we use unsupervised learning to categorize these vehicles in a sensible way with only the data we have been given?

For an individual quantitative variable, it is easy to measure how far apart any two cars are: Take the difference between the numerical values. The different variables are, however, on different scales and in different units. For example, `gears` ranges only from 1 to 8, while `city_mpg` goes from 13 to 58. This means that some decision needs to be made about rescaling the variables so that the differences along each variable reasonably reflect how different the respective cars are. There is more than one way to do this, and in fact, there is no universally “best” solution—the best solution will always depend on the data and your domain expertise. The `dist()` function takes a simple and pragmatic point of view: Each variable is equally important.²

The output of `dist()` gives the *distance* from each individual car to every other car.

²The default distance metric used by `dist()` is the Euclidean distance. Recall that we discussed this in Chapter 11 in our explanation of *k*-nearest-neighbor methods.

```

car_diffs <- cars %>%
  column_to_rownames(var = "model") %>%
  dist()
str(car_diffs)

'dist' num [1:2775] 4.52 11.29 9.93 11.29 15.14 ...
- attr(*, "Size")= int 75
- attr(*, "Labels")= chr [1:75] "FR-S" "RC 200t" "RC 300 AWD" "RC 350" ...
- attr(*, "Diag")= logi FALSE
- attr(*, "Upper")= logi FALSE
- attr(*, "method")= chr "euclidean"
- attr(*, "call")= language dist(x = .)

car_mat <- car_diffs %>%
  as.matrix()
car_mat[1:6, 1:6] %>%
  round(digits = 2)

```

	FR-S	RC 200t	RC 300 AWD	RC 350	RC 350 AWD	RC F
FR-S	0.00	4.52	11.29	9.93	11.29	15.14
RC 200t	4.52	0.00	8.14	6.12	8.14	11.49
RC 300 AWD	11.29	8.14	0.00	3.10	0.00	4.93
RC 350	9.93	6.12	3.10	0.00	3.10	5.39
RC 350 AWD	11.29	8.14	0.00	3.10	0.00	4.93
RC F	15.14	11.49	4.93	5.39	4.93	0.00

This point-to-point distance matrix is analogous to the tables that used to be printed on road maps giving the distance from one city to another, like [Figure 12.2](#), which states that it is 1,095 miles from Atlanta to Boston, or 715 miles from Atlanta to Chicago. Notice that the distances are symmetric: It is the same distance from Boston to Los Angeles as from Los Angeles to Boston (3,036 miles, according to the table).

	Atlanta	Boston	Chicago	Dallas	Denver	Houston	Las Vegas	Los Angeles
Atlanta	1095	715	805	1437	844	1920	2230	
Boston	1095		983	1815	1991	1886	2500	3036
Chicago	715	983		931	1050	1092	1500	2112
Dallas	805	1815	931		801	242	1150	1425
Denver	1437	1991	1050	801		1032	885	1174
Houston	844	1886	1092	242	1032		1525	1556
Las Vegas	1920	2500	1500	1150	885	1525		289
Los Angeles	2230	3036	2112	1425	1174	1556	289	

Figure 12.2: Distances between some U.S. cities.

Knowing the distances between the cities is not the same thing as knowing their locations. But the set of mutual distances is enough information to reconstruct the relative positions of the cities.

Cities, of course, lie on the surface of the earth. That need not be true for the “distance” between automobile types. Even so, the set of mutual distances provides information equivalent to knowing the relative positions of these cars in a p -dimensional space. This can be used to construct branches between nearby items, then to connect those branches, and so on until an entire tree has been constructed. The process is called *hierarchical clustering*. Figure 12.3 shows a tree constructed by hierarchical clustering that relates Toyota car models to one another.

```
library(ape)
car_diffs %>%
  hclust() %>%
  as.phylo() %>%
  plot(cex = 0.8, label.offset = 1)
```

There are many ways to graph such trees, but here we have borrowed from biology by graphing these cars as a phylogenetic tree, similar to Figure 12.1. Careful inspection of Figure 12.3 reveals some interesting insights. The first branch in the tree is evidently between hybrid vehicles and all others. This makes sense, since hybrid vehicles use a fundamentally different type of power to achieve considerably better fuel economy. Moreover, the first branch among conventional cars divides large trucks and SUVs (e.g., Sienna, Tacoma, Sequoia, Tundra, Land Cruiser) from smaller cars and cross-over SUVs (e.g., Camry, Corolla, Yaris, RAV4). We are confident that the gearheads in the readership will identify even more subtle logic to this clustering. One could imagine that this type of analysis might help a car-buyer or marketing executive quickly decipher what might otherwise be a bewildering product line.

12.1.2 k -means

Another way to group similar cases is to assign each case to one of several distinct groups, but without constructing a hierarchy. The output is not a tree but a choice of group to which each case belongs. (There can be more detail than this; for instance, a probability for each specific case that it belongs to each group.) This is like classification except that here there is no response variable. Thus, the definition of the groups must be inferred implicitly from the data.

As an example, consider the cities of the world (in `world_cities`). Cities can be different and similar in many ways: population, age structure, public transportation and roads, building space per person, etc. The choice of *features* (or variables) depends on the purpose you have for making the grouping.

Our purpose is to show you that clustering via machine learning can actually identify genuine patterns in the data. We will choose features that are utterly familiar: the latitude and longitude of each city.

You already know about the location of cities. They are on land. And you know about the organization of land on earth: most land falls in one of the large clusters called continents. But the `world_cities` data doesn’t have any notion of continents. Perhaps it is possible that this feature, which you long ago internalized, can be learned by a computer that has never even taken grade-school geography.

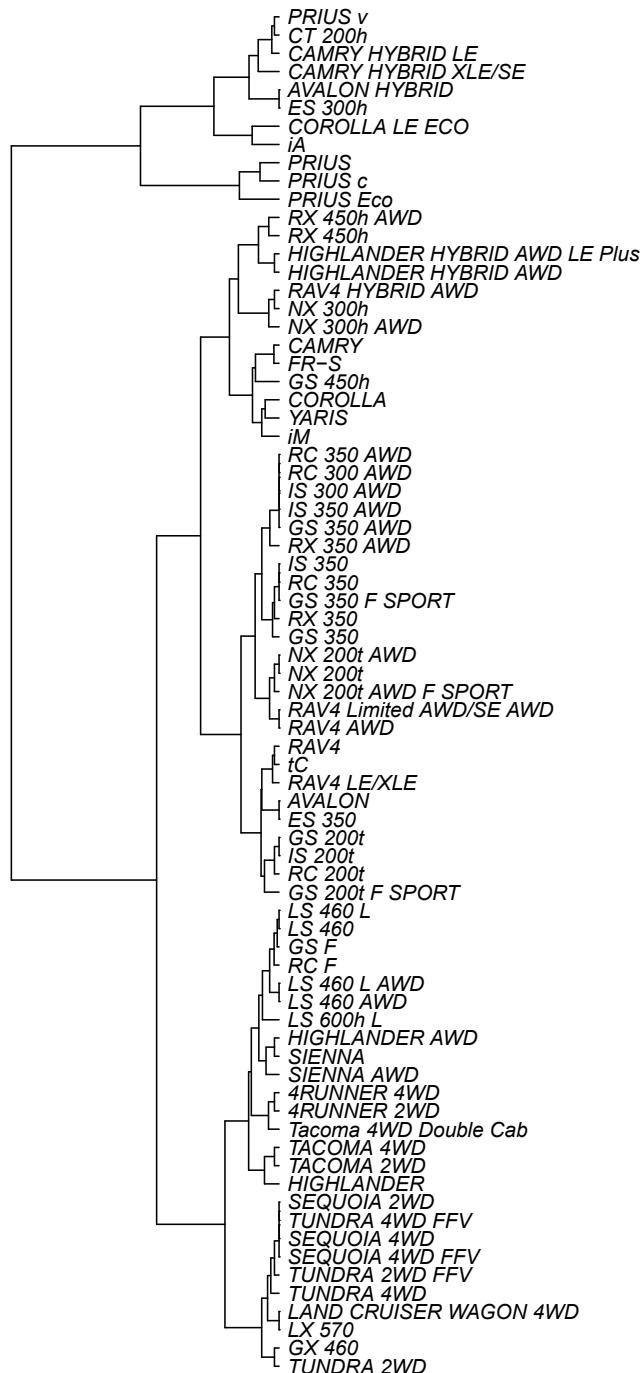


Figure 12.3: A dendrogram constructed by hierarchical clustering from car-to-car distances implied by the Toyota fuel economy data.

For simplicity, consider the 4,000 biggest cities in the world and their longitudes and latitudes.

```
big_cities <- world_cities %>%
  arrange(desc(population)) %>%
  head(4000) %>%
  select(longitude, latitude)
glimpse(big_cities)
```

```
Rows: 4,000
Columns: 2
$ longitude <dbl> 121.46, 28.95, -58.38, 72.88, -99.13, 116.40, 67.01, ...
$ latitude <dbl> 31.22, 41.01, -34.61, 19.07, 19.43, 39.91, 24.86, 39....
```

Note that in these data, there is no ancillary information—not even the name of the city. However, the *k-means* clustering algorithm will separate these 4,000 points—each of which is located in a two-dimensional plane—into k clusters based on their locations alone.

```
set.seed(15)
library(mclust)
city_clusts <- big_cities %>%
  kmeans.centers = 6) %>%
  fitted("classes") %>%
  as.character()
big_cities <- big_cities %>%
  mutate(cluster = city_clusts)
big_cities %>%
  ggplot(aes(x = longitude, y = latitude)) +
  geom_point(aes(color = cluster), alpha = 0.5) +
  scale_color_brewer(palette = "Set2")
```

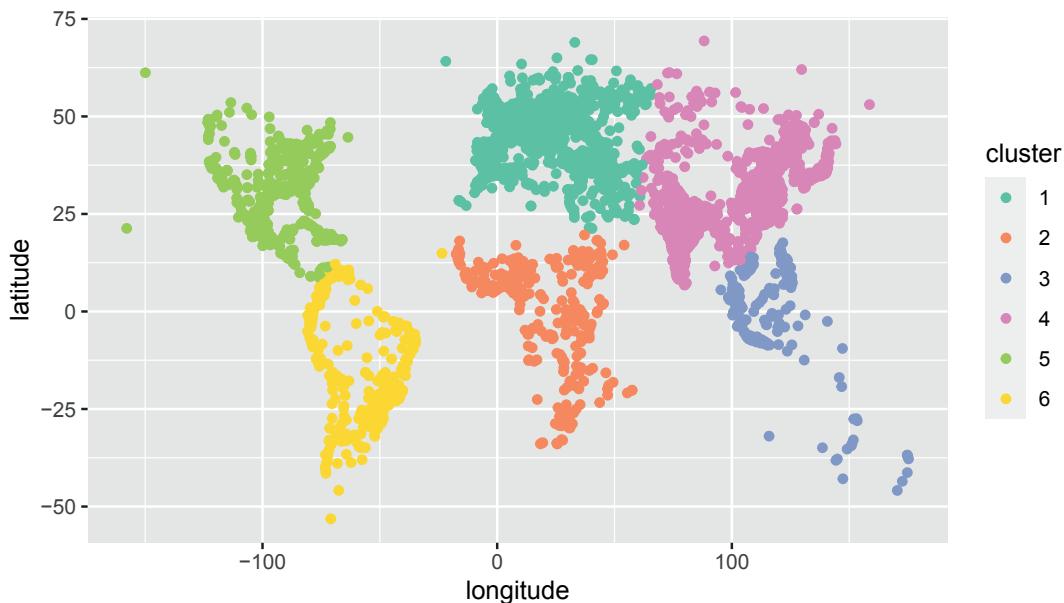


Figure 12.4: The world's 4,000 largest cities, clustered by the 6-means clustering algorithm.

Table 12.1: Sample voting records data from the Scottish Parliament.

name	S1M-1	S1M-4.1	S1M-4.3	S1M-4
Canavan, Dennis	1	1	1	-1
Aitken, Bill	1	1	0	-1
Davidson, Mr David	1	1	0	0
Douglas Hamilton, Lord James	1	1	0	0
Fergusson, Alex	1	1	0	0
Fraser, Murdo	0	0	0	0
Gallie, Phil	1	1	0	-1
Goldie, Annabel	1	1	0	0
Harding, Mr Keith	1	1	0	0
Johnston, Nick	0	1	0	-1

As shown in [Figure 12.4](#), the clustering algorithm seems to have identified the continents. North and South America are clearly distinguished, as is most of Africa. The cities in North Africa are matched to Europe, but this is consistent with history, given the European influence in places like Morocco, Tunisia, and Egypt. Similarly, while the cluster for Europe extends into what is called Asia, the distinction between Europe and Asia is essentially historic, not geographic. Note that to the algorithm, there is little difference between oceans and deserts—both represent large areas where no big cities exist.

12.2 Dimension reduction

Often, a variable carries little information that is relevant to the task at hand. Even for variables that are informative, there can be redundancy or near duplication of variables. That is, two or more variables are giving essentially the same information—they have similar patterns across the cases.

Such irrelevant or redundant variables make it harder to learn from data. The irrelevant variables are simply noise that obscures actual patterns. Similarly, when two or more variables are redundant, the differences between them may represent random noise. Furthermore, for some machine learning algorithms, a large number of variables p will present computational challenges. It is usually helpful to remove irrelevant or redundant variables so that they—and the noise they carry—don't obscure the patterns that machine learning algorithms could identify.

For example, consider votes in a parliament or congress. Specifically, we will explore the Scottish Parliament in 2008.³ Legislators often vote together in pre-organized blocs, and thus the pattern of “ayes” and “nays” on particular ballots may indicate which members are affiliated (i.e., members of the same political party). To test this idea, you might try clustering the members by their voting record.

[Table 12.1](#) shows a small part of the voting record. The names of the members of parliament are the cases. Each ballot—identified by a file number such as `S1M-4.3`—is a variable. A 1

³The Scottish Parliament example was constructed by then-student Caroline Ettinger and her faculty advisor, Andrew Beveridge, at *Macalester College*, and presented in Ms. Ettinger's senior capstone thesis.

means an “aye” vote, -1 is “nay,” and 0 is an abstention. There are $n = 134$ members and $p = 773$ ballots—note that in this data set p far exceeds n . It is impractical to show all of the more than 100,000 votes in a table, but there are only 3 possible votes, so displaying the table as an image (as in [Figure 12.5](#)) works well.

```
Votes %>%
  mutate(Vote = factor(vote, labels = c("Nay", "Abstain", "Aye"))) %>%
  ggplot(aes(x = bill, y = name, fill = Vote)) +
  geom_tile() +
  xlab("Ballot") +
  ylab("Member of Parliament") +
  scale_fill_manual(values = c("darkgray", "white", "goldenrod")) +
  scale_x_discrete(breaks = NULL, labels = NULL) +
  scale_y_discrete(breaks = NULL, labels = NULL)
```

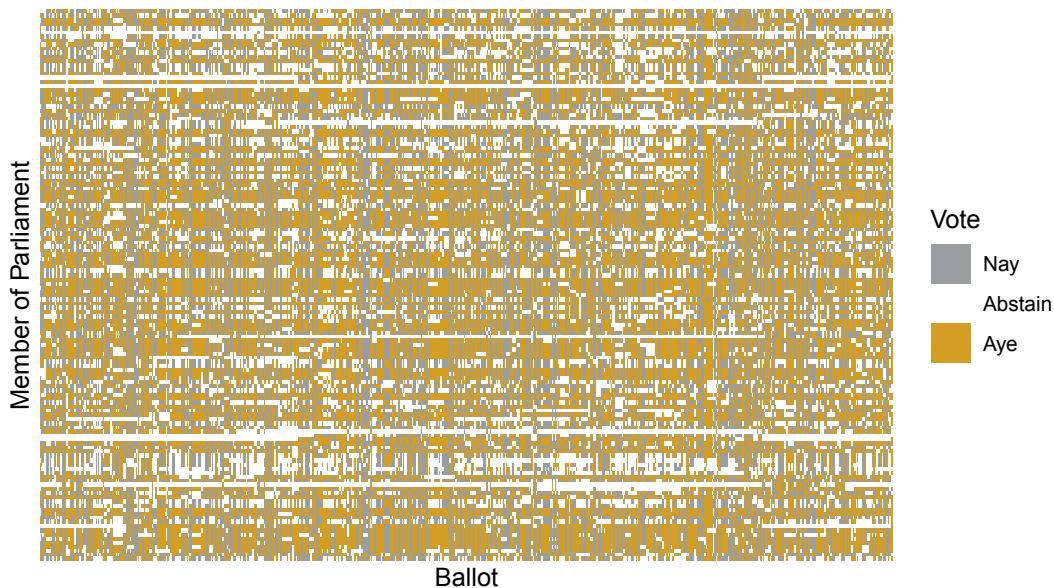


Figure 12.5: Visualization of the Scottish Parliament votes.

[Figure 12.5](#) is a 134×773 grid in which each cell is color-coded based on one member of Parliament’s vote on one ballot. It is hard to see much of a pattern here, although you may notice the Scottish tartan structure. The tartan pattern provides an indication to experts that the matrix can be approximated by a matrix of low-rank.

12.2.1 Intuitive approaches

As a start, [Figure 12.6](#) shows the ballot values for all of the members of parliament for just two arbitrarily selected ballots. To give a better idea of the point count at each position, the values are jittered by adding some random noise. The red dots are the actual positions. Each point is one member of parliament. Similarly aligned members are grouped together at one of the nine possibilities marked in red: (Aye, Nay), (Aye, Abstain), (Aye, Aye), and so on through to (Nay, Nay). In these two ballots, eight of the nine possibilities are populated. Does this mean that there are eight clusters of members?

```
Votes %>%
  filter(bill %in% c("S1M-240.2", "S1M-639.1")) %>%
  pivot_wider(names_from = bill, values_from = vote) %>%
  ggplot(aes(x = `S1M-240.2`, y = `S1M-639.1`)) +
  geom_point(
    alpha = 0.7,
    position = position_jitter(width = 0.1, height = 0.1)
  ) +
  geom_point(alpha = 0.01, size = 10, color = "red" )
```

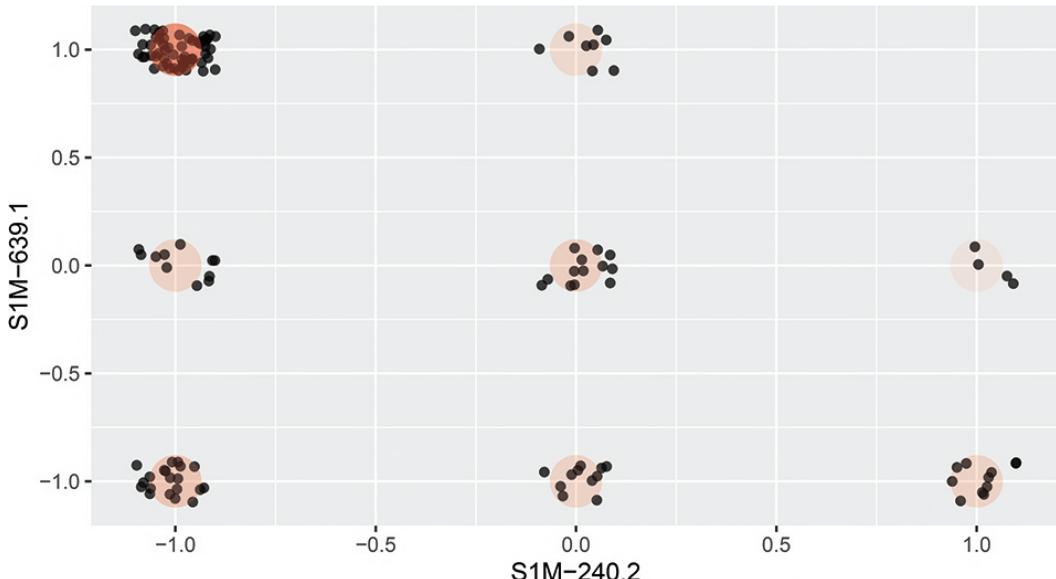


Figure 12.6: Scottish Parliament votes for two ballots.

Intuition suggests that it would be better to use *all* of the ballots, rather than just two. In [Figure 12.7](#), the first 387 ballots (half) have been added together, as have the remaining ballots. [Figure 12.7](#) suggests that there might be two clusters of members who are aligned with each other. Using all of the data seems to give more information than using just two ballots.

```
Votes %>%
  mutate(
    set_num = as.numeric(factor(bill)),
    set = ifelse(
      set_num < max(set_num) / 2, "First_Half", "Second_Half"
    )
  ) %>%
  group_by(name, set) %>%
  summarize(Ayes = sum(vote)) %>%
  pivot_wider(names_from = set, values_from = Ayes) %>%
  ggplot(aes(x = First_Half, y = Second_Half)) +
  geom_point(alpha = 0.7, size = 5)
```

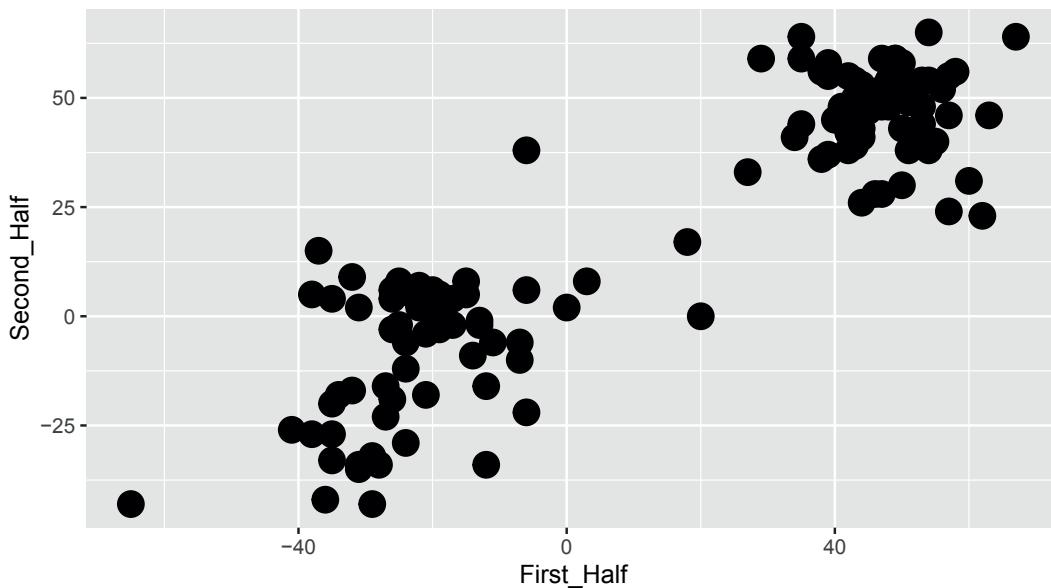


Figure 12.7: Scatterplot showing the correlation between Scottish Parliament votes in two arbitrary collections of ballots.

12.2.2 Singular value decomposition

You may ask why the choice was made to add up the first half of the ballots as x and the remaining ballots as y . Perhaps there is a better choice to display the underlying patterns. Perhaps we can think of a way to add up the ballots in a more meaningful way.

In fact, there is a mathematical approach to finding the *best* approximation to the ballot-voter matrix using simple matrices, called *singular value decomposition* (SVD). (The statistical dimension reduction technique of *principal component analysis* (PCA) can be accomplished using SVD.) The mathematics of SVD draw on a knowledge of matrix algebra, but the operation itself is accessible to anyone. Geometrically, SVD (or PCA) amounts to a rotation of the coordinate axes so that more of the variability can be explained using just a few variables. Figure 12.8 shows the position of each member on the two principal components that explain the most variability.

```
Votes_wide <- Votes %>%
  pivot_wider(names_from = bill, values_from = vote)
vote_svd <- Votes_wide %>%
  select(-name) %>%
  svd()

num_clusters <- 5    # desired number of clusters
library(broom)
vote_svd_tidy <- vote_svd %>%
  tidy(matrix = "u") %>%
  filter(PC < num_clusters) %>%
  mutate(PC = paste0("pc_", PC)) %>%
  pivot_wider(names_from = PC, values_from = value) %>%
  select(-row)
```

```

clusts <- vote_svd_tidy %>%
  kmeans(centers = num_clusters)

tidy(clusts)

# A tibble: 5 x 7
  pc_1    pc_2    pc_3    pc_4    size withinss cluster
  <dbl>   <dbl>   <dbl>   <dbl>   <int>     <dbl> <fct>
1 -0.0529  0.142  0.0840  0.0260     26    0.0118 1
2  0.0851  0.0367  0.0257 -0.182      20    0.160  2
3 -0.0435  0.109  0.0630 -0.0218      10    0.0160 3
4 -0.0306 -0.116  0.183  -0.00962     20    0.0459 4
5  0.106   0.0206  0.0323  0.0456      58    0.112  5

voters <- clusts %>%
  augment(vote_svd_tidy)

ggplot(data = voters, aes(x = pc_1, y = pc_2)) +
  geom_point(aes(x = 0, y = 0), color = "red", shape = 1, size = 7) +
  geom_point(size = 5, alpha = 0.6, aes(color = .cluster)) +
  xlab("Best Vector from SVD") +
  ylab("Second Best Vector from SVD") +
  ggtitle("Political Positions of Members of Parliament") +
  scale_color_brewer(palette = "Set2")

```

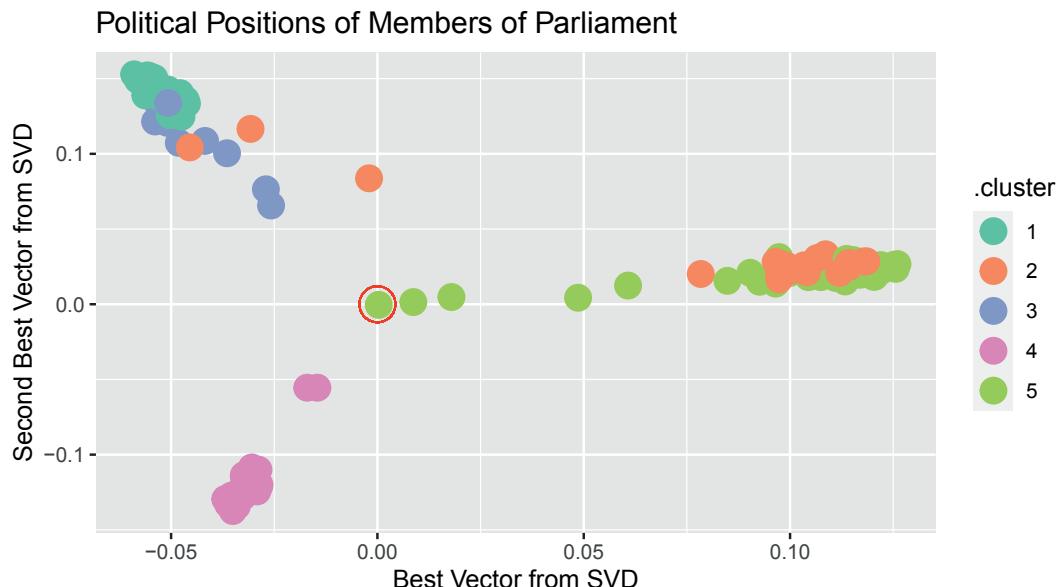


Figure 12.8: Clustering members of Scottish Parliament based on SVD along the members.

Figure 12.8 shows, at a glance, that there are three main clusters. The red circle marks the *average* member. The three clusters move away from average in different directions. There are several members whose position is in-between the average and the cluster to which they are closest. These clusters may reveal the alignment of Scottish members of parliament according to party affiliation and voting history.

For a graphic, one is limited to using two variables for position. Clustering, however, can be based on many more variables. Using more SVD sums may allow the three clusters to be split up further. The color in [Figure 12.8](#) above shows the result of asking for five clusters using the five best SVD sums. The confusion matrix below compares the actual party of each member to the cluster memberships.

```
voters <- voters %>%
  mutate(name = Votes_wide$name) %>%
  left_join(Parties, by = c("name" = "name"))
mosaic::tally(party ~ .cluster, data = voters)
```

	.cluster				
party	1	2	3	4	5
Member for Falkirk West	0	1	0	0	0
Scottish Conservative and Unionist Party	0	0	0	20	0
Scottish Green Party	0	1	0	0	0
Scottish Labour	0	1	0	0	57
Scottish Liberal Democrats	0	16	0	0	1
Scottish National Party	26	0	10	0	0
Scottish Socialist Party	0	1	0	0	0

How well did the clustering algorithm do? The party affiliation of each member of parliament is known, even though it wasn't used in finding the clusters. Cluster 1 consists of most of the members of the *Scottish National Party* (SNP). Cluster 2 includes a number of individuals plus all but one of the *Scottish Liberal Democrats*. Cluster 3 picks up the remaining 10 members of the SNP. Cluster 4 includes all of the members of the *Scottish Conservative and Unionist Party*, while Cluster 5 accounts for all but one member of the *Scottish Labour* party. For most parties, the large majority of members were placed into a unique cluster for that party with a small smattering of other like-voting colleagues. In other words, the technique has identified correctly nearly all of the members of the four different parties with significant representation (i.e., Conservative and Unionist, Labour, Liberal Democrats, and National).

```
ballots <- vote_svd %>%
  tidy(matrix = "v") %>%
  filter(PC < num_clusters) %>%
  mutate(PC = paste0("pc_", PC)) %>%
  pivot_wider(names_from = PC, values_from = value) %>%
  select(-column)
clust_ballots <- kmeans(ballots, centers = num_clusters)
ballots <- clust_ballots %>%
  augment(ballots) %>%
  mutate(bill = names(select(Votes_wide, -name)))
```

```
ggplot(data = ballots, aes(x = pc_1, y = pc_2)) +
  geom_point(aes(x = 0, y = 0), color = "red", shape = 1, size = 7) +
  geom_point(size = 5, alpha = 0.6, aes(color = .cluster)) +
  xlab("Best Vector from SVD") +
  ylab("Second Best Vector from SVD") +
  ggtitle("Influential Ballots") +
  scale_color_brewer(palette = "Set2")
```

There is more information to be extracted from the ballot data. Just as there are clusters

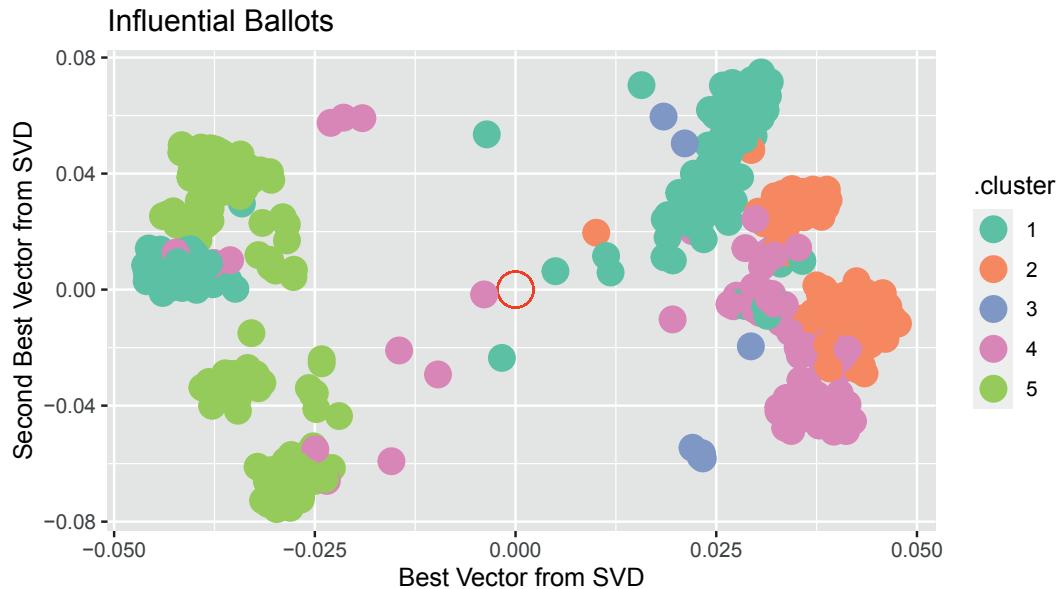


Figure 12.9: Clustering of Scottish Parliament ballots based on SVD along the ballots.

of political positions, there are clusters of ballots that might correspond to such factors as social effect, economic effect, etc. Figure 12.9 shows the position of ballots, using the first two principal components.

There are obvious clusters in this figure. Still, interpretation can be tricky. Remember that, on each issue, there are both “aye” and “nay” votes. This accounts for the symmetry of the dots around the center (indicated in red). The opposing dots along each angle from the center might be interpreted in terms of *socially liberal* versus *socially conservative* and *economically liberal* versus *economically conservative*. Deciding which is which likely involves reading the bill itself, as well as a nuanced understanding of Scottish politics.

Finally, the principal components can be used to rearrange members of parliament and separately rearrange ballots while maintaining each person’s vote. This amounts simply to re-ordering the members in a way other than alphabetical and similarly with the ballots. Such a transformation can bring dramatic clarity to the appearance of the data—as shown in Figure 12.10—where the large, nearly equally sized, and opposing voting blocs of the two major political parties (the National and Labour parties) become obvious. Alliances among the smaller political parties muddy the waters on the lower half of Figure 12.10.

```
Votes_svd <- Votes %>%
  mutate(Vote = factor(vote, labels = c("Nay", "Abstain", "Aye"))) %>%
  inner_join(ballots, by = "bill") %>%
  inner_join(voters, by = "name")

ggplot(data = Votes_svd,
  aes(x = reorder(bill, pc_1.x), y = reorder(name, pc_1.y), fill = Vote)) +
  geom_tile() +
  xlab("Ballot") +
  ylab("Member of Parliament") +
  scale_fill_manual(values = c("darkgray", "white", "goldenrod")) +
```

```
scale_x_discrete(breaks = NULL, labels = NULL) +
  scale_y_discrete(breaks = NULL, labels = NULL)
```

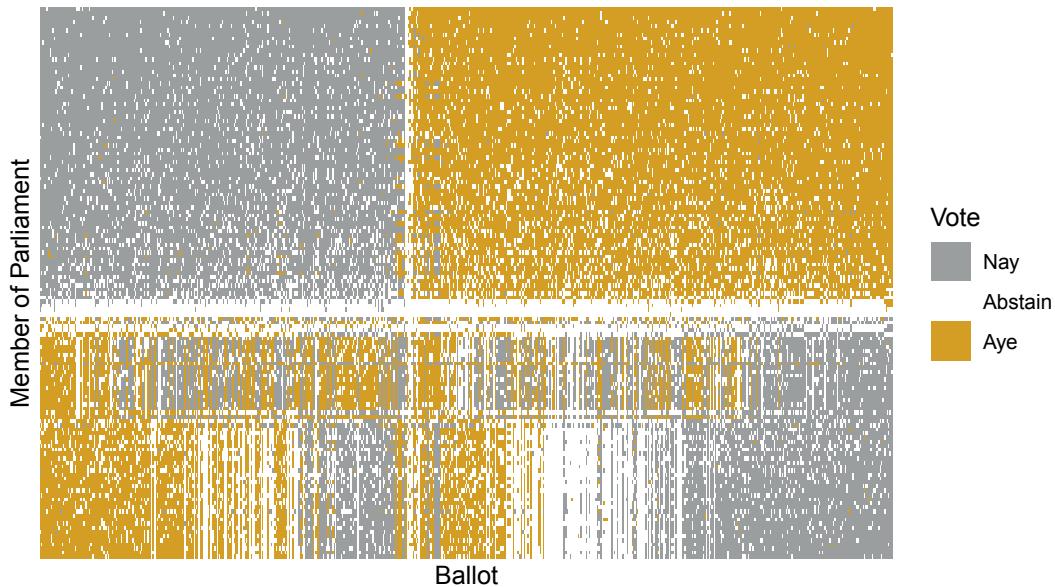


Figure 12.10: Illustration of the Scottish Parliament votes when ordered by the primary vector of the SVD.

The person represented by the top row in Figure 12.10 is Nicola Sturgeon, the leader of the Scottish National Party. Along the primary vector identified by our SVD, she is the most extreme voter. According to Wikipedia, the National Party belongs to a “mainstream European social democratic tradition.”

```
Votes_svd %>%
  arrange(pc_1.y) %>%
  head(1)
```

	bill	name	vote	Vote	pc_1.x	pc_2.x	pc_3.x	pc_4.x	.cluster.x	pc_1.y	pc_2.y	pc_3.y	pc_4.y	.cluster.y	party	
1	S1M-1	Sturgeon, Nicola	1	Aye	-0.00391	-0.00167	0.0498	-0.0734								Scottish National Party
1			4	-0.059	0.153	0.0832	0.0396									

Conversely, the person at the bottom of Figure 12.10 is Paul Martin, a member of the Scottish Labour Party. It is easy to see in Figure 12.10 that Martin opposed Sturgeon on most ballot votes.

```
Votes_svd %>%
  arrange(pc_1.y) %>%
  tail(1)
```

	bill	name	vote	Vote	pc_1.x	pc_2.x	pc_3.x	pc_4.x	.cluster.x	pc_1.y	pc_2.y	pc_3.y	pc_4.y	.cluster.y	party	
103582	S1M-4064	Martin, Paul	1	Aye	0.0322	-0.00484	0.0653	-0.0317								Scottish Labour
103582			4	0.126	0.0267	0.0425	0.056									

The beauty of Figure 12.10 is that it brings profound order to the chaos apparent in Figure

12.5. This was accomplished by simply ordering the rows (members of Parliament) and the columns (ballots) in a sensible way. In this case, the ordering was determined by the primary vector identified by the SVD of the voting matrix. This is yet another example of how machine learning techniques can identify meaningful patterns in data, but human beings are required to bring domain knowledge to bear on the problem in order to extract meaningful contextual understanding.

12.3 Further resources

The machine learning and phylogenetics CRAN task views provide guidance on functionality within **R**. Readers interested in learning more about unsupervised learning are encouraged to consult James et al. (2013) or Hastie et al. (2009). Kuiper and Sklar (2012) includes an accessible treatment of *principal component analysis*.

12.4 Exercises

Problem 1 (Medium): Re-fit the k -means algorithm on the `BigCities` data with a different value of k (i.e., not six). Experiment with different values of k and report on the sensitivity of the algorithm to changes in this parameter.

Problem 2 (Medium): Carry out and interpret a clustering of vehicles from another manufacturer using the approach outlined in the first section of the chapter.

Problem 3 (Medium): Perform the clustering on *pitchers* who have been elected to the Hall of Fame using the `Pitching` dataset in the `Lahman` package. Use wins (`w`), strikeouts (`so`), and saves (`sv`) as criteria.

Problem 4 (Medium): Consider the k -means clustering algorithm applied to the `BigCities` data. Would you expect to obtain different results if the location coordinates were *projected* (see the “Working with spatial data” chapter)?

Problem 5 (Hard): Use the College Scorecard Data from the `CollegeScorecard` package to cluster educational institutions using the techniques described in this chapter. Be sure to include variables related to student debt, number of students, graduation rate, and selectivity.

```
# remotes::install_github("Amherst-Statistics/CollegeScorecard")
```

Problem 6 (Hard): Baseball players are voted into the Hall of Fame by the members of the Baseball Writers of America Association. Quantitative criteria are used by the voters, but they are also allowed wide discretion. The following code identifies the position players who have been elected to the Hall of Fame and tabulates a few basic statistics, including their number of career hits (`h`), home runs (`HR`), and stolen bases (`SB`).

- a. Use the `kmeans` function to perform a cluster analysis on these players. Describe the properties that seem common to each cluster.

```
library(mdsr)
library(Lahman)
hof <- Batting %>%
  group_by(playerID) %>%
  inner_join(HallOfFame, by = c("playerID" = "playerID")) %>%
  filter(inducted == "Y" & votedBy == "BBWAA") %>%
  summarize(tH = sum(H), tHR = sum(HR), tRBI = sum(RBI), tSB = sum(SB)) %>%
  filter(tH > 1000)
```

- b. Building on the previous exercise, compute new statistics and run the clustering algorithm again. Can you produce clusters that you think are more pure? Justify your choices.

Problem 7 (Hard): Project the `world_cities` coordinates using the Gall-Peters projection and run the k -means algorithm again. Are the resulting clusters importantly different from those identified in the chapter?

```
library(tidyverse)
library(mdsr)
big_cities <- world_cities %>%
  arrange(desc(population)) %>%
  head(4000) %>%
  select(longitude, latitude)
```

12.5 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-learningII.html#learningII-online-exercises>



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

13

Simulation

13.1 Reasoning in reverse

In [Chapter 1](#) of this book we stated a simple truth: The purpose of data science is to turn data into usable information. Another way to think of this is that we use data to improve our understanding of the systems we live and work with: Data → Understanding.

This chapter is about computing techniques relating to the reverse way of thinking: Speculation → Data. In other words, this chapter is about “making up data.”

Many people associate “making up data” with deception. Certainly, data can be made up for exactly that purpose. Our purpose is different. We are interested in legitimate purposes for making up data, purposes that support the proper use of data science in transforming data into understanding.

How can made-up data be legitimately useful? In order to make up data, you need to build a mechanism that contains—implicitly—an idea about how the system you are interested in works. The data you make up tell you what data generated by that system would look like. There are two main (legitimate) purposes for doing this:

- Conditional inference. If our mechanism is reflective of how the real system works, the data it generates are similar to real data. You might use these to inform tweaks to the mechanism in order to produce even more representative results. This process can help you refine your understanding in ways that are relevant to the real world.
- Winnowing out hypotheses. To “winnow” means to remove from a set the less desirable choices so that what remains is useful. Traditionally, grain was winnowed to separate the edible parts from the inedible chaff. For data science, the set is composed of hypotheses, which are ideas about how the world works. Data are generated from each hypothesis and compared to the data we collect from the real world. When the hypothesis-generated data fails to resemble the real-world data, we can remove that hypothesis from the set. What remains are hypotheses that are plausible candidates for describing the real-world mechanisms.

“Making up” data is undignified, so we will leave that term to refer to fraud and trickery. In its place we’ll use the word *simulation*, which derives from “similar.” Simulations involve constructing mechanisms that are similar to how systems in the real world work—or at least to our belief and understanding of how such systems work.

13.2 Extended example: Grouping cancers

There are many different kinds of cancer. They are often given the name of the tissue in which they originate: lung cancer, ovarian cancer, prostate cancer, and so on. Different kinds of cancer are treated with different chemotherapeutic drugs. But perhaps the tissue origin of each cancer is not the best indicator of how it should be treated. Could we find a better way? Let's revisit the data introduced in [Section 3.2.4](#).

Like all cells, cancer cells have a genome containing tens of thousands of genes. Sometimes just a few genes dictate a cell's behavior. Other times there are networks of genes that regulate one another's expression in ways that shape cell features, such as the over-rapid reproduction characteristic of cancer cells. It is now possible to examine the expression of individual genes within a cell. So-called *microarrays* are routinely used for this purpose. Each microarray has tens to hundreds of thousands of *probes* for gene activity. The result of a microarray assay is a snapshot of gene activity. By comparing snapshots of cells in different states, it's possible to identify the genes that are expressed differently in the states. This can provide insight into how specific genes govern various aspects of cell activity.

A data scientist, as part of a team of biomedical researchers, might take on the job of compiling data from many microarray assays to identify whether different types of cancer are related based on their gene expression. For instance, the `NCI60` data (provided by the `etl_NCI60()` function in the `mdsr` package) contains readings from assays of $n = 60$ different cell lines of cancer of different tissue types. For each cell line, the data contain readings on over $p > 40,000$ different probes. Your job might be to find relationships between different cell lines based on the patterns of probe expression. For this purpose, you might find the techniques of statistical learning and unsupervised learning from [Chapters 10–12](#) to be helpful.

However, there is a problem. Even cancer cells have to carry out the routine actions that all cells use to maintain themselves. Presumably, the expression of most of the genes in the `NCI60` data are irrelevant to the peculiarities of cancer and the similarities and differences between different cancer types. Data interpreting methods—including those in [Chapters 10](#) and [11](#)—can be swamped by a wave of irrelevant data. They are more likely to be effective if the irrelevant data can be removed. Dimension reduction methods such as those described in [Chapter 12](#) can be attractive for this purpose.

When you start down the road toward your goal of finding links among different cancer types, you don't know if you will reach your destination. If you don't, before concluding that there are no relationships, it's helpful to rule out some other possibilities. Perhaps the data reduction and data interpretation methods you used are not powerful enough. Another set of methods might be better. Or perhaps there isn't enough data to be able to detect the patterns you are looking for.

Simulations can help here. To illustrate, consider a rather simple data reduction technique for the `NCI60` microarray data. If the expression of a probe is the same or very similar across all the different cancers, there's nothing that that probe can tell us about the links among cancers. One way to quantify the variation in a probe from cell line to cell line is the standard deviation of microarray readings for that probe.

It is a straightforward exercise in data wrangling to calculate this for each probe. The `NCI60` data come in a wide form: a matrix that's 60 columns wide (one for each cell line) and

41,078 rows long (one row for each probe). This pipeline will find the standard deviation across cell lines for each probe.

```
library(tidyverse)
library(mdsr)
NCI60 <- etl_NCI60()
spreads <- NCI60 %>%
  pivot_longer(
    -Probe, values_to = "expression",
    names_to = "cellLine"
  ) %>%
  group_by(Probe) %>%
  summarize(N = n(), spread = sd(expression)) %>%
  arrange(desc(spread)) %>%
  mutate(order = row_number())
```

NCI60 has been rearranged into narrow format in spreads, with columns Probe and spread for each of 32,344 probes. (A large number of the probes appear several times in the microarray, in one case as many as 14 times.) We arrange this dataset in descending order by the size of the standard deviation, so we can collect the probes that exhibit the most variation across cell lines by taking the topmost ones in spreads. For ease in plotting, we add the variable order to mark the order of each probe in the list.

How many of the probes with top standard deviations should we include in further data reduction and interpretation? 1? 10? 1000? 10,000? How should we go about answering this question? We'll use a simulation to help determine the number of probes that we select.

```
sim_spreads <- NCI60 %>%
  pivot_longer(
    -Probe, values_to = "expression",
    names_to = "cellLine"
  ) %>%
  mutate(Probe = mosaic::shuffle(Probe)) %>%
  group_by(Probe) %>%
  summarize(N = n(), spread = sd(expression)) %>%
  arrange(desc(spread)) %>%
  mutate(order = row_number())
```

What makes this a simulation is the `mutate()` command where we call `shuffle()`. In that line, we replace each of the probe labels with a randomly selected label. The result is that the expression has been statistically disconnected from any other variable, particularly cellLine. The simulation creates the kind of data that would result from a system in which the probe expression data is meaningless. In other words, the simulation mechanism matches the null hypothesis that the probe labels are irrelevant. By comparing the real NCI60 data to the simulated data, we can see which probes give evidence that the null hypothesis is false. Let's compare the top-500 spread values in spreads and sim_spreads.

```
spreads %>%
  filter(order <= 500) %>%
  ggplot(aes(x = order, y = spread)) +
  geom_line(color = "blue", size = 2) +
  geom_line(
    data = filter(sim_spreads, order <= 500),
```

```

    color = "red",
    size = 2
) +
geom_text(
  label = "simulated", x = 275, y = 4.4,
  size = 3, color = "red"
) +
geom_text(
  label = "observed", x = 75, y = 5.5,
  size = 3, color = "blue"
)
)

```

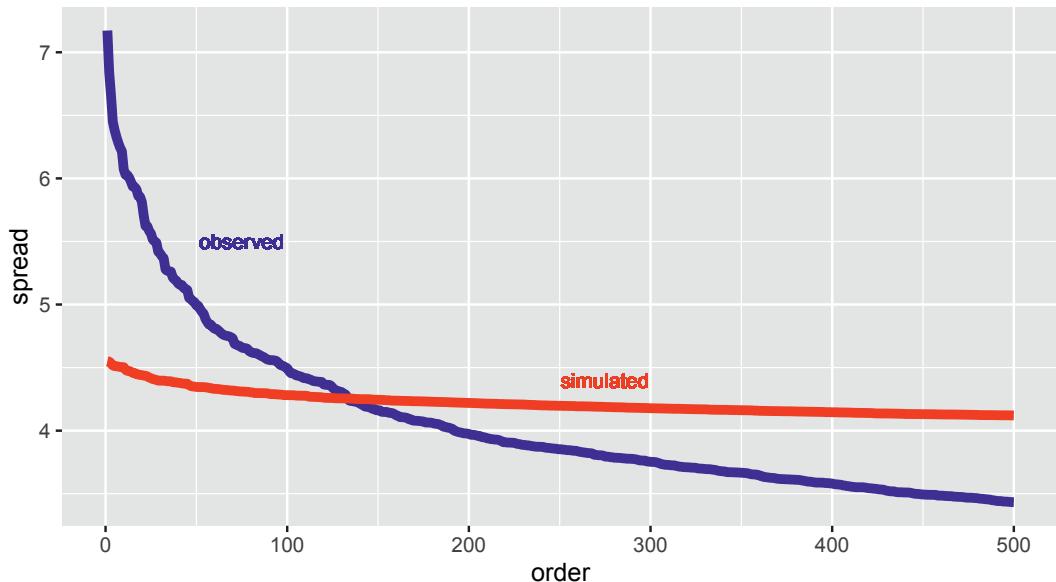


Figure 13.1: Comparing the variation in expression for individual probes across cell lines data (blue) and a simulation of a null hypothesis (red).

We can tell a lot from the results of the simulation shown in Figure 13.1. If we decided to use the top-500 probes, we would risk including many that were no more variable than random noise (i.e., that which could have been generated under the null hypothesis).

But if we set the threshold much lower, including, say, only those probes with a spread greater than 5.0, we would be unlikely to include any that were generated by a mechanism consistent with the null hypothesis. The simulation is telling us that it would be good to look at roughly the top-50 probes, since that is about how many in NCI60 were out of the range of the simulated results for the null hypothesis. Methods of this sort are often identified as *false discovery rate* methods.

13.3 Randomizing functions

There are as many possible simulations as there are possible hypotheses—that is, an unlimited number. Different hypotheses call for different techniques for building simulations. But there are some techniques that appear in a wide range of simulations. It's worth knowing about these.

The previous example about false discovery rates in gene expression uses an everyday method of randomization: *shuffling*. Shuffling is, of course, a way of destroying any genuine order in a sequence, leaving only those appearances of order that are due to chance. Closely-related methods, *sampling* and *resampling*, were introduced in [Chapter 9](#) when we used simulation to assess the statistical significance of patterns observed in data.

Counter-intuitively, the use of random numbers is an important component of many simulations. In simulation, we want to induce variation. For instance, the simulated probes for the cancer example do not all have the same spread. But in creating that variation, we do not want to introduce any structure other than what we specify explicitly in the simulation. Using random numbers ensures that any structure that we find in the simulation is either due to the mechanism we've built for the simulation or is purely accidental.

The workhorse of simulation is the generation of random numbers in the range from zero to one, with each possibility being equally likely. In **R**, the most widely-used such *uniform random number generator* is `runif()`. For instance, here we ask for five uniform random numbers:

```
runif(5)
```

```
[1] 0.7614 0.1023 0.2699 0.6333 0.0527
```

Other randomization devices can be built out of uniform random number generators. To illustrate, here is a device for selecting one value at random from a vector:

```
select_one <- function(vec) {  
  n <- length(vec)  
  ind <- which.max(runif(n))  
  vec[ind]  
}  
select_one(letters) # letters are a, b, c, ..., z
```

```
[1] "c"
```

```
select_one(letters)
```

```
[1] "e"
```

The `select_one()` function is functionally equivalent to `slice_sample()` with the `size` argument set to 1. Random numbers are so important that you should try to use generators that have been written by experts and vetted by the community.

There is a lot of sophisticated theory behind programs that generate uniform random numbers. After all, you generally don't want sequences of random numbers to repeat themselves. (An exception is described in [Section 13.6](#).) The theory has to do with techniques for making repeated sub-sequences as rare as possible.

Perhaps the widest use of simulation in data analysis involves the randomness introduced

by sampling, resampling, and shuffling. These operations are provided by the functions `sample()`, `resample()`, and `shuffle()` from the **mosaic** package. These functions sample uniformly at random from a data frame (or vector) with or without replacement, or permute the rows of a data frame. `resample()` is equivalent to `sample()` with the `replace` argument set to `TRUE`, while `shuffle()` is equivalent to `sample()` with `size` equal to the number of rows in the data frame and `replace` equal to `FALSE`. Non-uniform sampling can be achieved using the `prob` option.

Other important functions for building simulations are those that generate random numbers with certain important properties. We've already seen `runif()` for creating uniform random numbers. Very widely used are `rnorm()`, `rexp()`, and `rpois()` for generating numbers that are distributed normally (that is, in the bell-shaped, *Gaussian distribution*, exponentially, and with a Poisson (count) pattern, respectively. These different distributions correspond to idealized descriptions of mechanisms in the real world. For instance, events that are equally likely to happen at any time (e.g., earthquakes) will tend to have a time spacing between events that is exponential. Events that have a rate that remains the same over time (e.g., the number of cars passing a point on a low-traffic road in one minute) are often modeled using a *Poisson distribution*. There are many other forms of distributions that are considered good models of particular random processes. Functions analogous to `runif()` and `rnorm()` are available for other common probability distributions (see the Probability Distributions CRAN Task View).

13.4 Simulating variability

13.4.1 The partially-planned rendezvous

Imagine a situation where Sally and Joan plan to meet to study in their college campus center (Mosteller, 1987). They are both impatient people who will wait only 10 minutes for the other before leaving.

But their planning was incomplete. Sally said, “Meet me between 7 and 8 tonight at the center.” When should Joan plan to arrive at the campus center? And what is the probability that they actually meet?

A simulation can help answer these questions. Joan might reasonably assume that it doesn't really matter when she arrives, and that Sally is equally likely to arrive any time between 7:00 and 8:00 pm.

So to Joan, Sally's arrival time is random and uniformly distributed between 7:00 and 8:00 pm. The same is true for Sally. Such a simulation is easy to write: generate uniform random numbers between 0 and 60 minutes after 7:00 pm. For each pair of such numbers, check whether or not the time difference between them is 10 minutes or less. If so, they successfully met. Otherwise, they missed each other.

Here's an implementation in **R**, with 100,000 trials of the simulation being run to make sure that the possibilities are well covered.

```
n <- 100000
sim_meet <- tibble(
  sally = runif(n, min = 0, max = 60),
  joan = runif(n, min = 0, max = 60),
```

```

result = ifelse(
  abs(sally - joan) <= 10, "They meet", "They do not"
)
)
mosaic::tally(~ result, format = "percent", data = sim_meet)

result
They do not    They meet
       69.4        30.6
mosaic::binom.test(~result, n, success = "They meet", data = sim_meet)

```

```

data: sim_meet$result [with success = They meet]
number of successes = 30601, number of trials = 1e+05, p-value
<2e-16
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.303 0.309
sample estimates:
probability of success
 0.306

```

There's about a 30% chance that they meet (the true probability is $11/36 \approx 0.3055556$). The confidence interval, the width of which is determined in part by the number of simulations, is relatively narrow. For any reasonable decision that Joan might consider ("Oh, it seems unlikely we'll meet. I'll just skip it.") would be the same regardless of which end of the confidence interval is considered. So the simulation is good enough for Joan's purposes. (If the interval was not narrow enough for this, you would want to add more trials. The $1/\sqrt{n}$ rule for the width of a confidence interval described in Chapter 9 can guide your choice.)

```

ggplot(data = sim_meet, aes(x = joan, y = sally, color = result)) +
  geom_point(alpha = 0.3) +
  geom_abline(intercept = 10, slope = 1) +
  geom_abline(intercept = -10, slope = 1) +
  scale_color_brewer(palette = "Set2")

```

Often, it's valuable to visualize the possibilities generated in the simulation as in Figure 13.2. The arrival times uniformly cover the rectangle of possibilities, but only those that fall into the stripe in the center of the plot are successful. Looking at the plot, Joan notices a pattern. For any arrival time she plans, the probability of success is the fraction of a vertical band of the plot that is covered in blue. For instance, if Joan chose to arrive at 7:20, the probability of success is the proportion of blue in the vertical band with boundaries of 20 minutes and 30 minutes on the horizontal axis. Joan observes that near 0 and 60 minutes, the probability goes down, since the diagonal band tapers. This observation guides an important decision: Joan will plan to arrive somewhere from 7:10 to 7:50. Following this strategy, what is the probability of success? (Hint: Repeat the simulation but set Joan's `min()` to 10 and her `max()` to 50.) If Joan had additional information about Sally ("She wouldn't arrange to meet at 7:21—most likely at 7:00, 7:15, 7:30, or 7:45.") the simulation can be easily modified, e.g., `sally = resample(c(0, 15, 30, 45), n)` to incorporate that hypothesis.

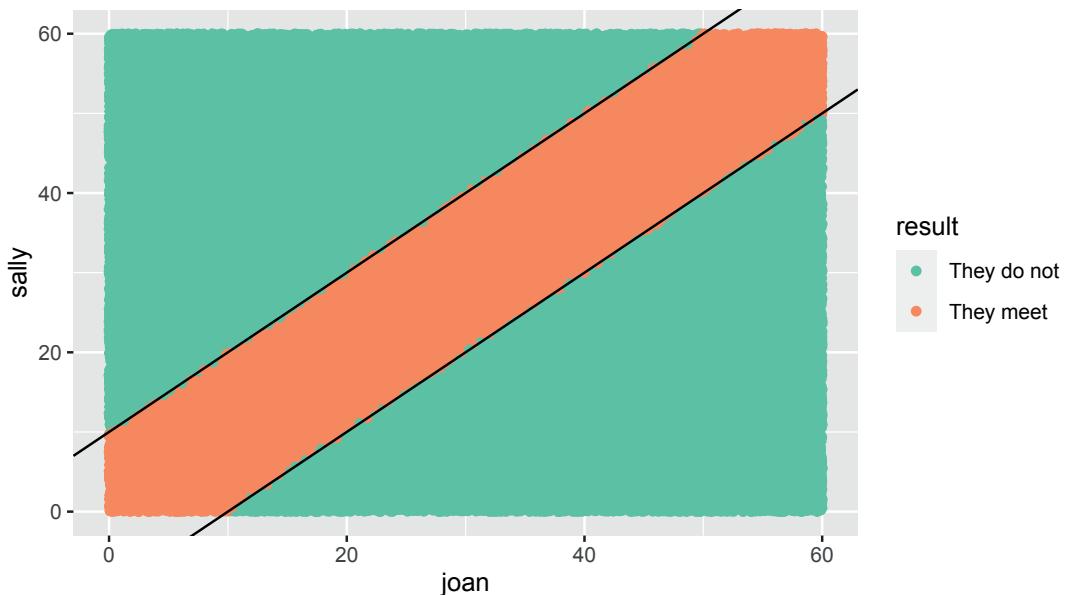


Figure 13.2: Distribution of Sally and Joan arrival times (shaded area indicates where they meet).

13.4.2 The jobs report

One hour before the opening of the stock market on the first Friday of each month, the *U.S. Bureau of Labor Statistics* releases the employment report. This widely anticipated estimate of the monthly change in non-farm payroll is an economic indicator that often leads to stock market shifts.

If you read the financial blogs, you'll hear lots of speculation before the report is released, and lots to account for the change in the stock market in the minutes *after* the report comes out. And you'll hear a lot of anticipation of the consequences of that month's job report on the prospects for the economy as a whole. It happens that many financiers read a lot into the ups and downs of the jobs report. (And other people, who don't take the report so seriously, see opportunities in responding to the actions of the believers.)

You are a skeptic. You know that in the months after the jobs report, an updated number is reported that is able to take into account late-arriving data that couldn't be included in the original report. One analysis, the article "How not to be misled by the jobs report" from the May 1, 2014 *New York Times* modeled the monthly report as a random number from a Gaussian distribution with a mean of 150,000 jobs and a standard deviation of 65,000 jobs.

You are preparing a briefing for your bosses to convince them not to take the jobs report itself seriously as an economic indicator. For many bosses, the phrases "Gaussian distribution," "standard deviation," and "confidence interval" will trigger a primitive "I'm not listening!" response, so your message won't get through in that form.

It turns out that many such people will have a better understanding of a simulation than of theoretical concepts. You decide on a strategy: Use a simulation to generate a year's worth of job reports. Ask the bosses what patterns they see and what they would look for in the next month's report. Then inform them that there are no actual patterns in the graphs you showed them.

```

jobs_true <- 150
jobs_se <- 65 # in thousands of jobs
gen_samp <- function(true_mean, true_sd,
                      num_months = 12, delta = 0, id = 1) {
  samp_year <- rep(true_mean, num_months) +
    rnorm(num_months, mean = delta * (1:num_months), sd = true_sd)
  return(
    tibble(
      jobs_number = samp_year,
      month = as.factor(1:num_months),
      id = id
    )
  )
}

```

We begin by defining some constants that will be needed, along with a function to calculate a year's worth of monthly samples from this known truth. Since the default value of `delta` is equal to zero, the “true” value remains constant over time. When the function argument `true_sd` is set to `0`, no random noise is added to the system.

Next, we prepare a data frame that contains the function argument values over which we want to simulate. In this case, we want our first simulation to have no random noise—thus the `true_sd` argument will be set to `0` and the `id` argument will be set to `Truth`. Following that, we will generate three random simulations with `true_sd` set to the assumed value of `jobs_se`. The data frame `params` contains complete information about the simulations we want to run.

```

n_sims <- 3
params <- tibble(
  sd = c(0, rep(jobs_se, n_sims)),
  id = c("Truth", paste("Sample", 1:n_sims))
)
params

# A tibble: 4 x 2
  sd   id
  <dbl> <chr>
1     0 Truth
2    65 Sample 1
3    65 Sample 2
4    65 Sample 3

```

Finally, we will actually perform the simulation using the `pmap_dfr()` function from the `purrr` package (see [Chapter 7](#)). This will iterate over the `params` data frame and apply the appropriate values to each simulation.

```

df <- params %>%
  pmap_dfr(~gen_samp(true_mean = jobs_true, true_sd = ..1, id = ..2))

```

Note how the two arguments are given in a compact and flexible form `(..1` and `..2)`.

```

ggplot(data = df, aes(x = month, y = jobs_number)) +
  geom_hline(yintercept = jobs_true, linetype = 2) +
  geom_col()

```

```
facet_wrap(~ id) +
ylab("Number of new jobs (in thousands)")
```

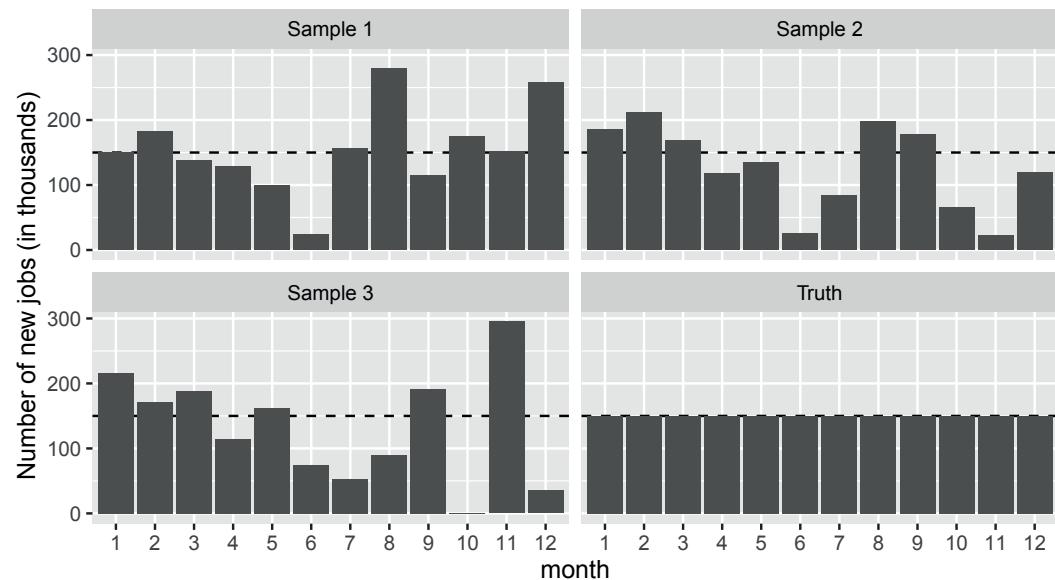


Figure 13.3: True number of new jobs from simulation as well as three realizations from a simulation.

Figure 13.3 displays the “true” number as well as three realizations from the simulation. While all of the three samples are taken from a “true” universe where the jobs number is constant, each could easily be misinterpreted to conclude that the numbers of new jobs was decreasing at some point during the series. The moral is clear: It is important to be able to understand the underlying variability of a system before making inferential conclusions.

13.4.3 Restaurant health and sanitation grades

We take our next simulation from the data set of restaurant health violations in *New York City*. To help ensure the safety of patrons, health inspectors make unannounced inspections at least once per year to each restaurant. Establishments are graded based on a range of criteria including food handling, personal hygiene, and vermin control. Those with a score between 0 and 13 points receive a coveted A grade, those with 14 to 27 points receive the less desirable B, and those of 28 or above receive a C. We’ll display values in a subset of this range to focus on the threshold between an A and B grade, after grouping by `dba` (Doing Business As) and `score`. We focus our analysis on the year 2015.

```
minval <- 7
maxval <- 19
violation_scores <- Violations %>%
  filter(lubridate::year(inspection_date) == 2015) %>%
  filter(score >= minval & score <= maxval) %>%
  select(dba, score)

ggplot(data = violation_scores, aes(x = score)) +
  geom_histogram(binwidth = 0.5) +
```

```
geom_vline(xintercept = 13, linetype = 2) +
scale_x_continuous(breaks = minval:maxval) +
annotate(
  "text", x = 10, y = 15000,
  label = "'A' grade: score of 13 or less"
)
```

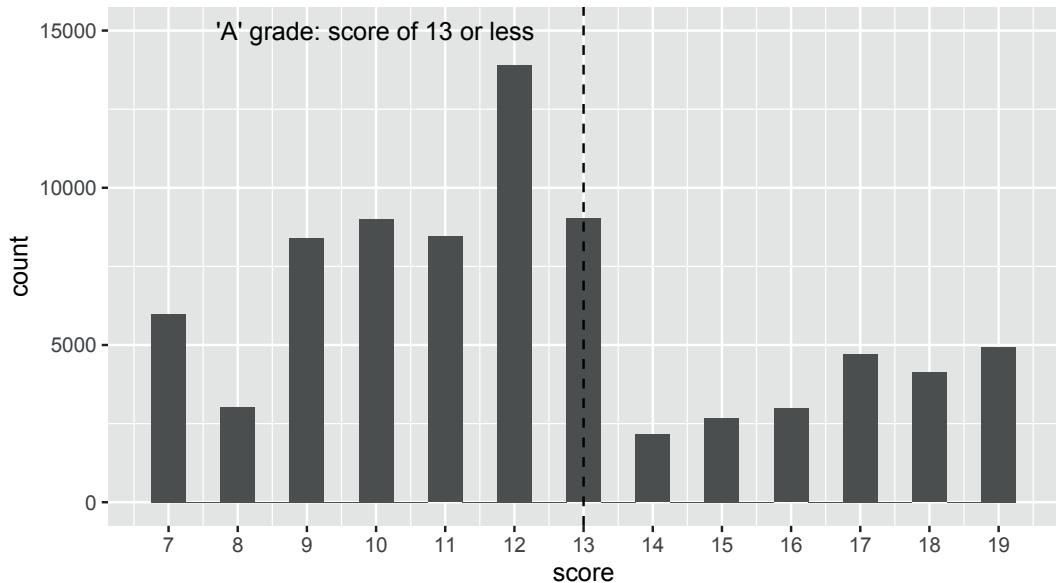


Figure 13.4: Distribution of NYC restaurant health violation scores.

Figure 13.4 displays the distribution of restaurant violation scores. Is something unusual happening at the threshold of 13 points (the highest value to still receive an A)? Or could sampling variability be the cause of the dramatic decrease in the frequency of restaurants graded between 13 and 14 points? Let's carry out a simple simulation in which a grade of 13 or 14 is equally likely. The `nflip()` function allows us to flip a fair coin that determines whether a grade is a 14 (heads) or 13 (tails).

```
scores <- mosaic::tally(~score, data = violation_scores)
scores
```

```
score
 7     8     9    10    11    12    13    14    15    16    17    18
5985  3026  8401  9007  8443 13907  9021  2155  2679  2973  4720  4119
 19
4939
```

```
mean(scores[c("13", "14")])
```

```
[1] 5588
```

```
random_flip <- 1:1000 %>%
  map_dbl(~mosaic::nflip(scores["13"] + scores["14"])) %>%
  enframe(name = "sim", value = "heads")
head(random_flip, 3)
```

```
# A tibble: 3 x 2
  sim heads
  <int> <dbl>
1     1  5648
2     2  5614
3     3  5642

ggplot(data = random_flip, aes(x = heads)) +
  geom_histogram(binwidth = 10) +
  geom_vline(xintercept = scores["14"], col = "red") +
  annotate(
    "text", x = 2200, y = 75,
    label = "observed", hjust = "left"
  ) +
  xlab("Number of restaurants with scores of 14 (if equal probability)")
```

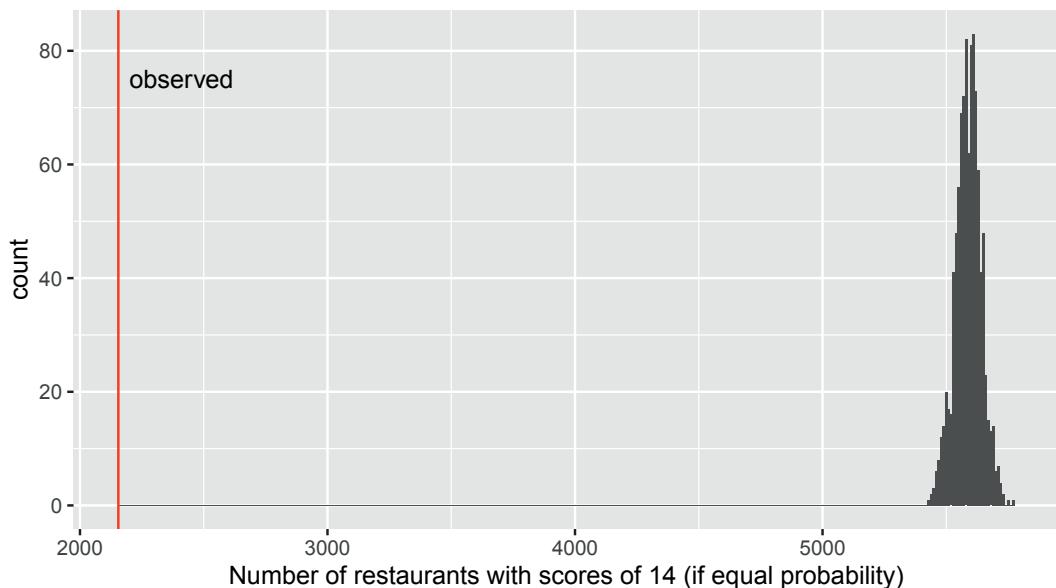


Figure 13.5: Distribution of health violation scores under a randomization procedure (permutation test).

Figure 13.5 demonstrates that the observed number of restaurants with a 14 are nowhere near what we would expect if there was an equal chance of receiving a score of 13 or 14. While the number of restaurants receiving a 13 might exceed the number receiving a 14 by 100 or so due to chance alone, there is essentially no chance of observing 5,000 more 13s than 14s if the two scores are truly equally likely. (It is not surprising given the large number of restaurants inspected in New York City that we wouldn't observe much sampling variability in terms of the proportion that are 14.) It appears as if the inspectors tend to give restaurants near the threshold the benefit of the doubt, and not drop their grade from A to B if the restaurant is on the margin between a 13 and 14 grade.

This is another situation where simulation can provide a more intuitive solution starting from first principles than an investigation using more formal statistical methods. (A more nuanced test of the “edge effect” might be considered given the drop in the numbers of restaurants with violation scores between 14 and 19.)

13.5 Random networks

As noted in [Chapter 2](#), a network (or graph) is a collection of nodes, along with edges that connect certain pairs of those nodes. Networks are often used to model real-world systems that contain these pairwise relationships. Although these networks are often simple to describe, many of the interesting problems in the mathematical discipline of graph theory are very hard to solve analytically, and intractable computationally (Garey and Johnson, 1979). For this reason, simulation has become a useful technique for exploring questions in *network science*. We illustrate how simulation can be used to verify properties of random graphs in [Chapter 20](#).

13.6 Key principles of simulation

Many of the key principles needed to develop the capacity to simulate come straight from computer science, including aspects of design, modularity, and reproducibility. In this section, we will briefly propose guidelines for simulations.

13.6.1 Design

It is important to consider design issues relative to simulation. As the analyst, you control all aspects and decide what assumptions and scenarios to explore. You have the ability (and responsibility) to determine which scenarios are relevant and what assumptions are appropriate.

The choice of scenarios depends on the underlying model: they should reflect plausible situations that are relevant to the problem at hand. It is often useful to start with a simple setting, then gradually add complexity as needed.

13.6.2 Modularity

It is very helpful to write a function to implement the simulation, which can be called repeatedly with different options and parameters (see [Appendix C](#)). Spending time planning what features the simulation might have, and how these can be split off into different functions (that might be reused in other simulations) will pay off handsomely.

13.6.3 Reproducibility and random number seeds

It is important that simulations are both reproducible and representative. Sampling variability is inherent in simulations: Our results will be sensitive to the number of computations that we are willing to carry out. We need to find a balance to avoid unneeded calculations while ensuring that our results aren't subject to random fluctuation. What is a reasonable number of simulations to consider? Let's revisit Sally and Joan, who will meet only if they both arrive within 10 minutes of each other. How variable are our estimates if we carry out only `num_sims = 100` simulations? We'll assess this by carrying out 5,000 replications, saving the results from each simulation of 100 possible meetings. Then we'll repeat the process, but with `num_sims = 400` and `num_sims = 1600`. Note that we can do this efficiently

using `map_dfr()` twice (once to iterate over the changing number of simulations, and once to repeat the procedure 5,000 times.

```

campus_sim <- function(sims = 1000, wait = 10) {
  sally <- runif(sims, min = 0, max = 60)
  joan <- runif(sims, min = 0, max = 60)
  return(
    tibble(
      num_sims = sims,
      meet = sum(abs(sally - joan) <= wait),
      meet_pct = meet / num_sims,
    )
  )
}

reps <- 5000
sim_results <- 1:reps %>%
  map_dfr(~map_dfr(c(100, 400, 1600), campus_sim))

sim_results %>%
  group_by(num_sims) %>%
  skim(meet_pct)

-- Variable type: numeric -----
  var      num_sims     n     na   mean     sd    p0    p25    p50    p75    p100
1 meet_pct      100  5000     0 0.305 0.0460 0.12  0.28   0.3    0.33   0.49
2 meet_pct      400  5000     0 0.306 0.0231 0.23  0.290  0.305  0.322  0.39
3 meet_pct     1600  5000     0 0.306 0.0116 0.263 0.298  0.306  0.314  0.352

```

Note that each of the simulations yields an unbiased estimate of the true probability that they meet, but there is variability within each individual simulation (of size 100, 400, or 1600). The standard deviation is halved each time we increase the number of simulations by a factor of 4. We can display the results graphically (see [Figure 13.6](#)).

```

sim_results %>%
  ggplot(aes(x = meet_pct, color = factor(num_sims))) +
  geom_density(size = 2) +
  geom_vline(aes(xintercept = 11/36), linetype = 3) +
  scale_x_continuous("Proportion of times that Sally and Joan meet") +
  scale_color_brewer("Number\nof sims", palette = "Set2")

```

What would be a reasonable value for `num_sims` in this setting? The answer depends on how accurate we want to be. (And we can also simulate to see how variable our results are!) Carrying out 20,000 simulations yields relatively little variability and would likely be sufficient for a first pass. We could state that these results have *converged* sufficiently close to the true value since the sampling variability due to the simulation is negligible.

```

1:reps %>%
  map_dfr(~campus_sim(20000)) %>%
  group_by(num_sims) %>%
  skim(meet_pct)

```

Given the inherent nature of variability due to sampling, it can be very useful to set (and save) a *seed* for the pseudo-random number generator (using the `set.seed()` function). This

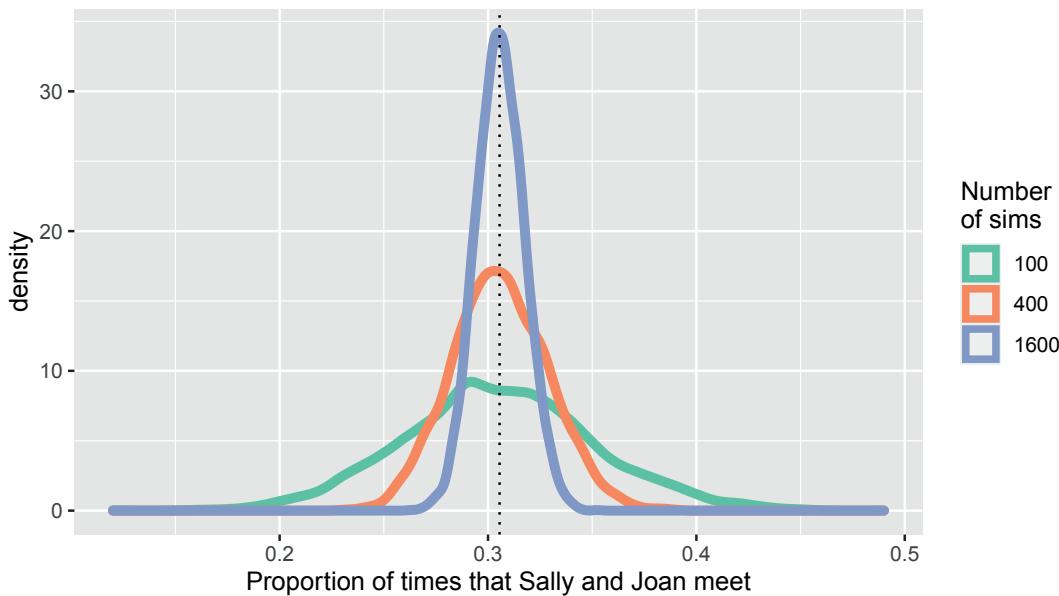


Figure 13.6: Convergence of the estimate of the proportion of times that Sally and Joan meet.

ensures that the results are the same each time the simulation is run since the simulation will use the same list of random numbers. The seed itself is arbitrary, but each seed defines a different sequence of random numbers.

```
set.seed(1974)
campus_sim()

# A tibble: 1 x 3
  num_sims  meet meet_pct
  <dbl>    <int>    <dbl>
1     1000    308    0.308

campus_sim()

# A tibble: 1 x 3
  num_sims  meet meet_pct
  <dbl>    <int>    <dbl>
1     1000    331    0.331

set.seed(1974)
campus_sim()

# A tibble: 1 x 3
  num_sims  meet meet_pct
  <dbl>    <int>    <dbl>
1     1000    308    0.308
```

13.7 Further resources

This chapter has been a basic introduction to simulation. Over the last 30 years, the ability to use simulation to match observed data has become an essential component of *Bayesian statistics*. A central technique is called *Markov Chain Monte Carlo* (MCMC). For an accessible introduction to Bayesian methods, see Albert and Hu (2019).

Rizzo (2019) provides a comprehensive introduction to statistical computing in **R**, while Horton et al. (2004) and Horton (2013) describe the use of **R** for simulation studies. The importance of simulation as part of an analyst's toolbox is enunciated in American Statistical Association Undergraduate Guidelines Workgroup (2014), Horton (2015), and National Academies of Science, Engineering, and Medicine (2018). The **simstudy** package can be used to simplify data generation or exploration using simulation.

13.8 Exercises

Problem 1 (Medium): The time a manager takes to interview a job applicant has an exponential distribution with mean of half an hour, and these times are independent of each other. The applicants are scheduled at quarter-hour intervals beginning at 8:00 am, and all of the applicants arrive exactly on time (this is an excellent thing to do, by the way). When the applicant with an 8:15 am appointment arrives at the manager's office office, what is the probability that she will have to wait before seeing the manager? What is the expected time that her interview will finish?

Problem 2 (Medium): Consider an example where a recording device that measures remote activity is placed in a remote location. The time, T , to failure of the remote device has an exponential distribution with mean of 3 years. Since the location is so remote, the device will not be monitored during its first 2 years of service. As a result, the time to discovery of its failure is $X = \max(T, 2)$. The problem here is to determine the average of the time to discovery of the truncated variable (in probability parlance, the expected value of the observed variable X , $E[X]$).

The analytic solution is fairly straightforward but requires calculus. We need to evaluate:

$$E[X] = \int_0^2 2f(u)du + \int_2^\infty uf(u)du,$$

where $f(u) = 1/3 \exp(-1/3u)$ for $u > 0$.

We can use the calculus functions in the **mosaicCalc** package to find the answer.

Is calculus strictly necessary here? Conduct a simulation to estimate (or check) the value for the average time to discovery.

Problem 3 (Medium): Two people toss a fair coin 4 times each. Find the probability that they throw equal numbers of heads. Also estimate the probability that they throw equal numbers of heads using a simulation in R (with an associated 95% confidence interval for your estimate).

Problem 4 (Medium): In this chapter, we considered a simulation where the true jobs

number remained constant over time. Modify the call to the function provided in that example so that the true situation is that there are 15,000 new jobs created every month. Set your random number seed to the value 1976. Summarize what you might conclude from these results as if you were a journalist without a background in data science.

Problem 5 (Medium): The `violations` dataset in the `mdsr` package contains information about health violations across different restaurants in New York City. Is there evidence that restaurant health inspectors in New York City give the benefit of the doubt to those at the threshold between a B grade (14 to 27) or C grade (28 or above)?

Problem 6 (Medium): Sally and Joan plan to meet to study in their college campus center. They are both impatient people who will only wait 10 minutes for the other before leaving. Rather than pick a specific time to meet, they agree to head over to the campus center sometime between 7:00 and 8:00 pm. Let both arrival times be normally distributed with mean 30 minutes past and a standard deviation of 10 minutes. Assume that they are independent of each other. What is the probability that they actually meet? Estimate the answer using simulation techniques introduced in this chapter, with at least 10,000 simulations.

Problem 7 (Medium): What is the impact if the residuals from a linear regression model are skewed (and not from a normal distribution)?

Repeatedly generate data from a “true” model given by:

```
n <- 250
rmse <- 1
x1 <- rep(c(0, 1), each = n / 2) # x1 resembles 0 0 0 ... 1 1 1
x2 <- runif(n, min = 0, max = 5)
beta0 <- -1
beta1 <- 0.5
beta2 <- 1.5
y <- beta0 + beta1 * x1 + beta2 * x2 + rexp(n, rate = 1 / 2)
```

For each simulation, fit the linear regression model and display the distribution of 1,000 estimates of the β_1 parameter (note that you need to generate the vector of outcomes each time).

Problem 8 (Medium): What is the impact of the violation of the equal variance assumption for linear regression models? Repeatedly generate data from a “true” model given by the following code.

```
n <- 250
rmse <- 1
x1 <- rep(c(0, 1), each = n / 2) # x1 resembles 0 0 0 ... 1 1 1
x2 <- runif(n, min = 0, max = 5)
beta0 <- -1
beta1 <- 0.5
beta2 <- 1.5
y <- beta0 + beta1 * x1 + beta2 * x2 + rnorm(n, mean = 0, sd = rmse + x2)
```

For each simulation, fit the linear regression model and display the distribution of 1,000 estimates of the β_1 parameter (note that you need to generate the vector of outcomes each time). Does the distribution of the parameter follow a normal distribution?

Problem 9 (Medium): Generate $n = 5,000$ observations from a logistic regression model with parameters intercept $\beta_0 = -1$, slope $\beta_1 = 0.5$, and distribution of the predictor

being normal with mean 1 and standard deviation 1. Calculate and interpret the resulting parameter estimates and confidence intervals.

13.9 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/simulation.html#simulation-online-exercises>

Part III

Part III: Topics in Data Science



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

14

Dynamic and customized data graphics

As we discussed in [Chapter 1](#), the practice of data science involves many different elements. In [Part I](#), we laid a foundation for data science by developing a basic understanding of data wrangling, data visualization, and ethics. In [Part II](#), we focused on building statistical models and using those models to learn from data. However, to this point we have focused mainly on traditional two-dimensional data (e.g., rows and columns) and data graphics. In this part, we tackle the heterogeneity found in many modern data: spatial, text, network, and relational data. We explore interactive data graphics that leap out of the printed page. Finally, we address the volume of data—concluding with a discussion of “big data” and the tools that you are likely to see when working with it.

In [Chapter 2](#), we laid out a systematic framework for composing data graphics. A similar grammar of graphics employed by the **ggplot2** package provided a mechanism for creating static data graphics in [Chapter 3](#). In this chapter, we explore a few alternatives for making more sophisticated—and in particular, dynamic—data graphics.

14.1 Rich Web content using **d3.js** and **htmlwidgets**

As Web browsers became more complex, the desire to have interactive data visualizations in the browser grew. Thus far, all of the data visualization techniques that we have discussed are based on static images. However, newer tools have made it considerably easier to create interactive data graphics.

JavaScript is a programming language that allows Web developers to create client-side *web applications*. This means that computations are happening *in the client’s browser*, as opposed to taking place on the host’s Web servers. *JavaScript* applications can be more responsive to client interaction than dynamically-served Web pages that rely on a server-side scripting language, like *PHP* or *Ruby*.

The current state of the art for client-side dynamic data graphics on the Web is a JavaScript library called **d3.js**, or just **d3**, which stands for “data-driven documents.” One of the lead developers of **d3** is Mike Bostock, formerly of *The New York Times* and *Stanford University*.

More recently, Ramnath Vaidyanathan and the developers at **RStudio** have created the **htmlwidgets** package, which provides a bridge between **R** and **d3**. Specifically, the **htmlwidgets** framework allows **R** developers to create packages that render data graphics in HTML using **d3**. Thus, **R** programmers can now make use of **d3** without having to learn JavaScript. Furthermore, since R Markdown documents also render as HTML, **R** users can easily create interactive data graphics embedded in annotated Web documents. This is a

highly active area of development. In what follows, we illustrate a few of the more obviously useful **htmlwidgets** packages.

14.1.1 Leaflet

Perhaps the **htmlwidgets** that is getting the greatest attention is **leaflet**, which enables dynamic geospatial maps to be drawn using the Leaflet JavaScript library and the *OpenStreetMaps* API. The use of this package requires knowledge of spatial data, and thus we postpone our illustration of its use until [Chapter 17](#).

14.1.2 Plot.ly

Plot.ly specializes in online dynamic data visualizations and, in particular, the ability to translate code to generate data graphics between **R**, Python, and other data software tools. This project is based on the **plotly.js** JavaScript library, which is available under an open-source license. The functionality of Plot.ly can be accessed in **R** through the **plotly** package.

What makes **plotly** especially attractive is that it can convert any **ggplot2** object into a **plotly** object using the **ggplotly()** function. This enables immediate interactivity for existing data graphics. Features like *brushing* (where selected points are marked) and *mouse-over* annotations (where points display additional information when the mouse hovers over them) are automatic. For example, in [Figure 14.1](#) we display a static plot of the frequency of the names of births in the United States of the four members of the *Beatles* over time (using data from the **babynames** package).

```
library(tidyverse)
library(mdsr)
library(babynames)
Beatles <- babynames %>%
  filter(name %in% c("John", "Paul", "George", "Ringo") & sex == "M") %>%
  mutate(name = factor(name, levels = c("John", "George", "Paul", "Ringo")))
beatles_plot <- ggplot(data = Beatles, aes(x = year, y = n)) +
  geom_line(aes(color = name), size = 2)
beatles_plot
```

After running the **ggplotly()** function on that object, a plot is displayed in **RStudio** or in a Web browser. The exact values can be displayed by mousing-over the lines. In addition, brushing, panning, and zooming are supported. In [Figure 14.2](#), we show that image.

```
library(plotly)
ggplotly(beatles_plot)
```

14.1.3 DataTables

The **DataTables** (**DT**) package provides a quick way to make data tables interactive. Simply put, it enables tables to be searchable, sortable, and pageable automatically. [Figure 14.3](#) displays the first 10 rows of the **Beatles** table as rendered by **DT**. Note the search box and clickable sorting arrows.

```
datatable(Beatles, options = list(pageLength = 10))
```

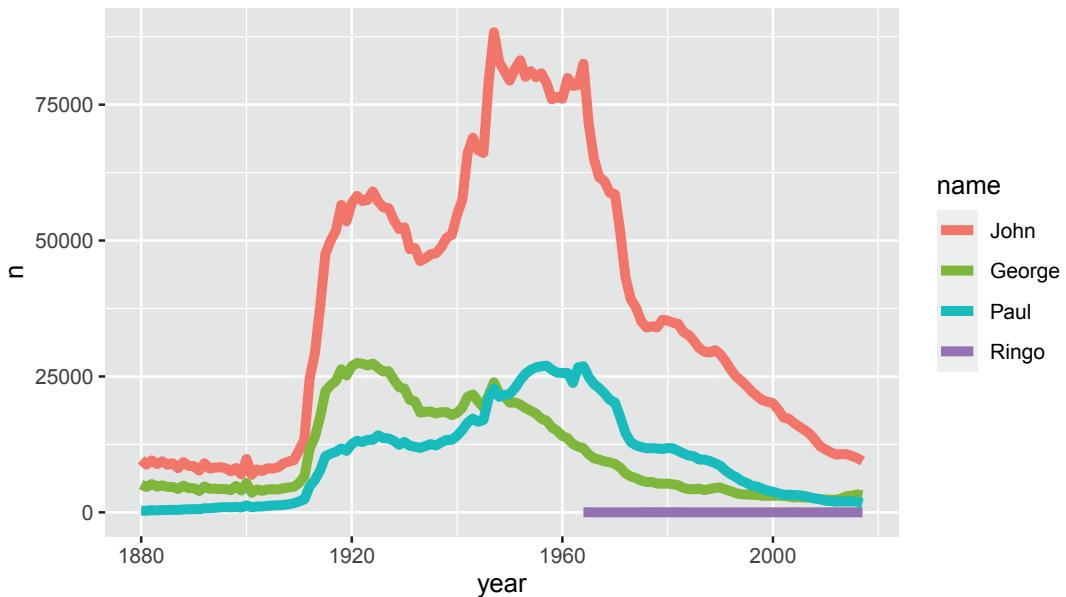


Figure 14.1: ggplot2 depiction of the frequency of Beatles names over time.

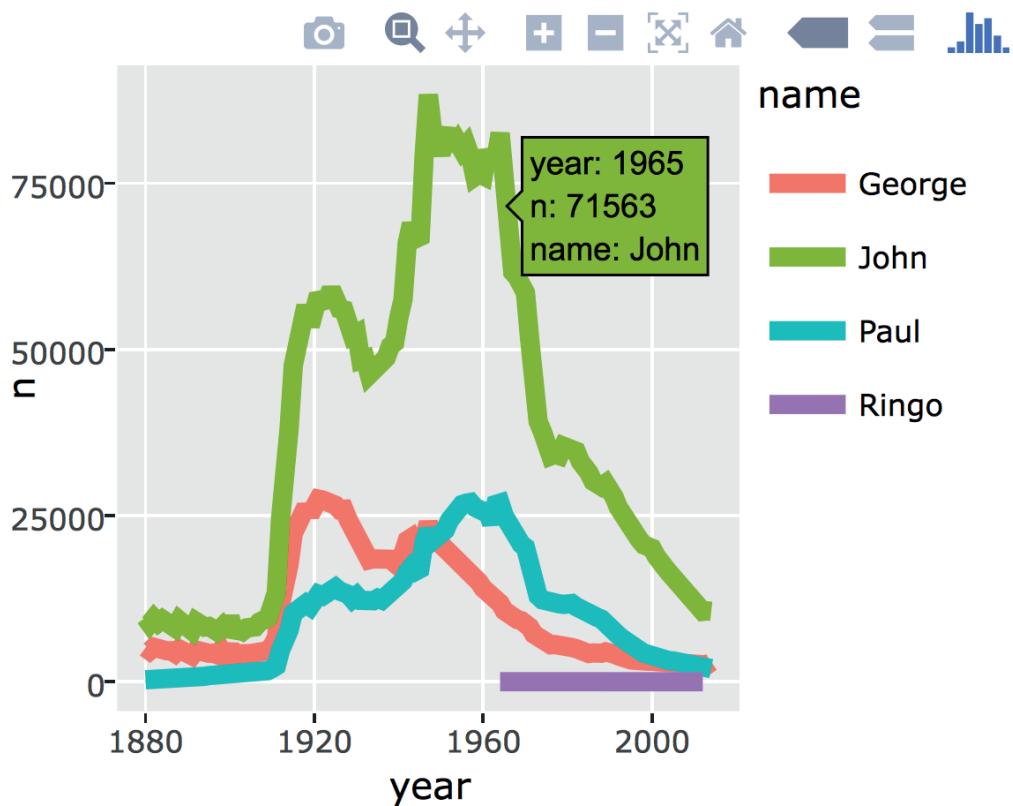


Figure 14.2: An interactive plot of the frequency of Beatles names over time.

Show 10 entries Search:

	year	sex	name	n
1	1880	M	John	9655
2	1880	M	George	5126
3	1880	M	Paul	301
4	1881	M	John	8769
5	1881	M	George	4664
6	1881	M	Paul	291
7	1882	M	John	9557
8	1882	M	George	5193
9	1882	M	Paul	397
10	1883	M	John	8894

Showing 1 to 10 of 442 entries

Previous 1 2 3 4 F

Figure 14.3: Output of the **DT** package applied to the Beatles names.

14.1.4 Dygraphs

The **dygraphs** package generates interactive time series plots with the ability to brush over time intervals and zoom in and out. For example, the popularity of Beatles names could be made dynamic with just a little bit of extra code. Here, the dynamic range selector allows for the easy selection of specific time periods on which to focus. In [Figure 14.4](#), one can zoom in on the uptick in the popularity of the names **John** and **Paul** during the first half of the 1960s.

```
library(dygraphs)
Beatles %>%
  filter(sex == "M") %>%
  select(year, name, prop) %>%
  pivot_wider(names_from = name, values_from = prop) %>%
  dygraph(main = "Popularity of Beatles names over time") %>%
  dyRangeSelector(dateWindow = c("1940", "1980"))
```

14.1.5 Streamgraphs

A *streamgraph* is a particular type of time series plot that uses area as a visual cue for quantity. Streamgraphs allow you to compare the values of several time series at once. The **streamgraph** `htmlwidget` provides access to the `streamgraphs.js` D3 library. [Figure 14.5](#) displays our Beatles names time series as a streamgraph.

```
# remotes::install_github("hrbrmstr/streamgraph")
library(streamgraph)
Beatles %>%
```

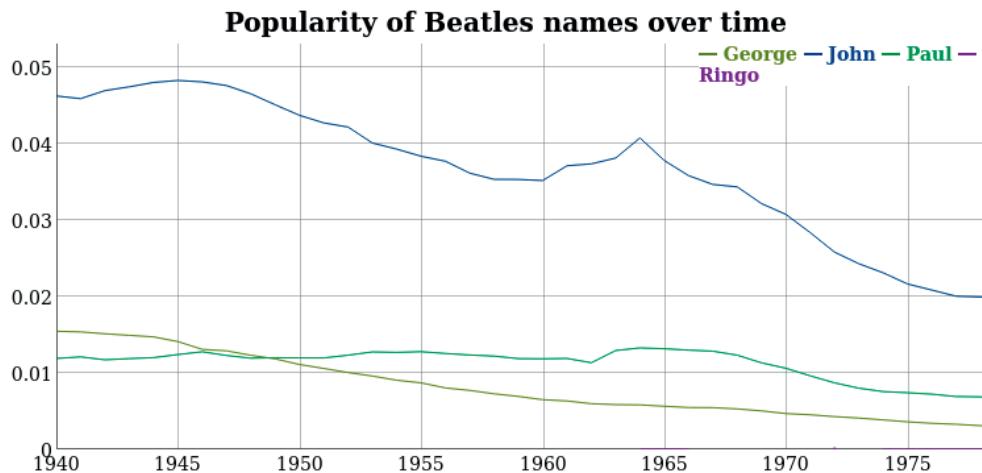


Figure 14.4: The `dygraphs` display of the popularity of Beatles names over time. In this screenshot, the years range from 1940 to 1980, and one can expand or contract that timespan.

```
streamgraph(key = "name", value = "n", date = "year") %>%
  sg_fill_brewer("Accent")
```

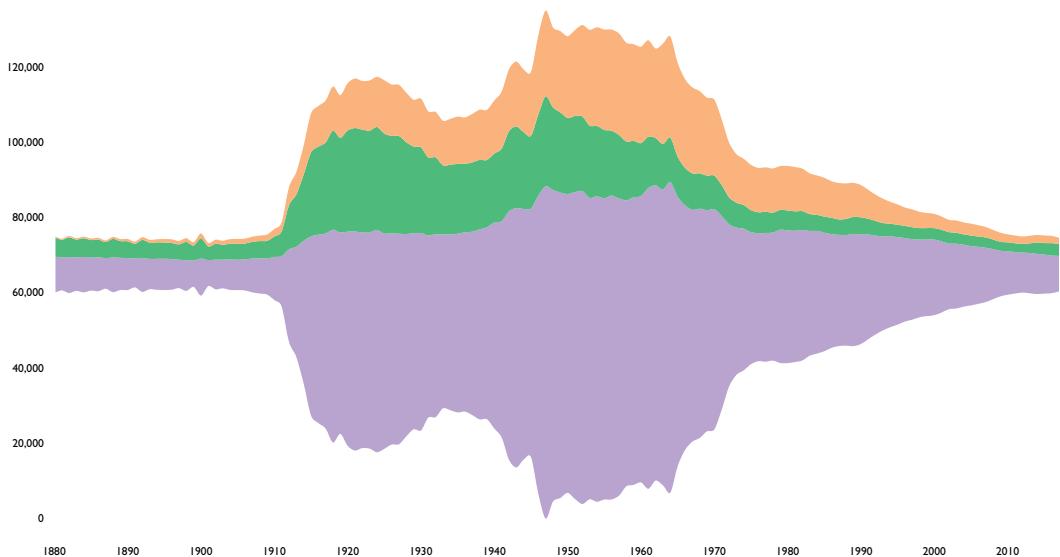


Figure 14.5: A screenshot of the `streamgraph` display of Beatles names over time.

14.2 Animation

The **gganimate** package provides a simple way to create animations (i.e., *GIFs*) from **ggplot2** data graphics. In [Figure 14.6](#), we illustrate a simple transition, wherein the lines indicating the popularity of each band member's name over time grows and shrinks.

```
library(gganimate)
library(transformr)
beatles_animation <- beatles_plot +
  transition_states(
    name,
    transition_length = 2,
    state_length = 1
  ) +
  enter_grow() +
  exit_shrink()

animate(beatles_animation, height = 400, width = 800)
```

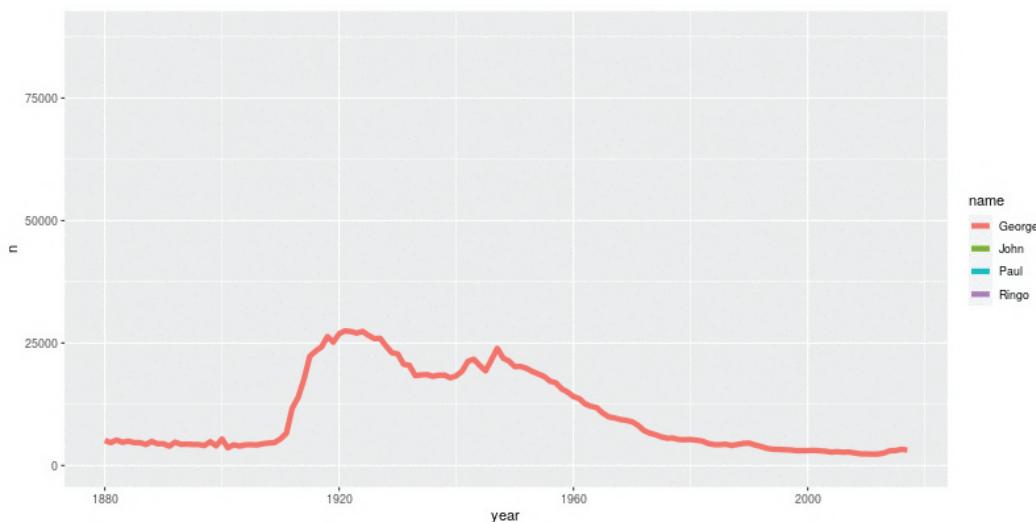


Figure 14.6: Evolving Beatles plot created by **gganimate**.

14.3 Flexdashboard

The **flexdashboard** package provides a straightforward way to create and publish data visualizations as a *dashboard*. Dashboards are a common way that data scientists make data available to managers and others to make decisions. They will often include a mix of graphical and textual displays that can be targeted to their needs.

Here we provide an example of an R Markdown file that creates a static dashboard of

information from the **palmerpenguins** package. **flexdashboard** divides up the page into rows and columns. In this case, we create two columns of nearly equal width. The second column (which appears on the right in [Figure 14.8](#)) is further subdivided into two rows, each marked by a third-level section header.

```
---
```

```
title: "Flexdashboard example (Palmer Penguins)"
output:
  flexdashboard::flex_dashboard:
    orientation: columns
    vertical_layout: fill
---
```

```
```{r setup, include=FALSE}
library(flexdashboard)
library(palmerpenguins)
library(tidyverse)
```

Column {data-width=400}
-----
```

```
### Chart A
```

```
```{r}
ggplot(
 penguins,
 aes(x = bill_length_mm, y = bill_depth_mm, color = species)
) +
 geom_point()
```

Column {data-width=300}
-----
```

```
### Chart B
```

```
```{r}
DT::datatable(penguins)
```

### Chart C
```

```
```{r}
roundval <- 2
cleanmean <- function(x, roundval = 2, na.rm = TRUE) {
 return(round(mean(x, na.rm = na.rm), digits = roundval))
}
summarystat <- penguins %>%
 group_by(species) %>%
 summarize(
```

```

`Average bill length (mm)` = cleanmean(bill_length_mm),
`Average bill depth (mm)` = cleanmean(bill_depth_mm)
)
knitr:::kable(summarystat)
```

```

Figure 14.7: Sample flexdashboard input file.

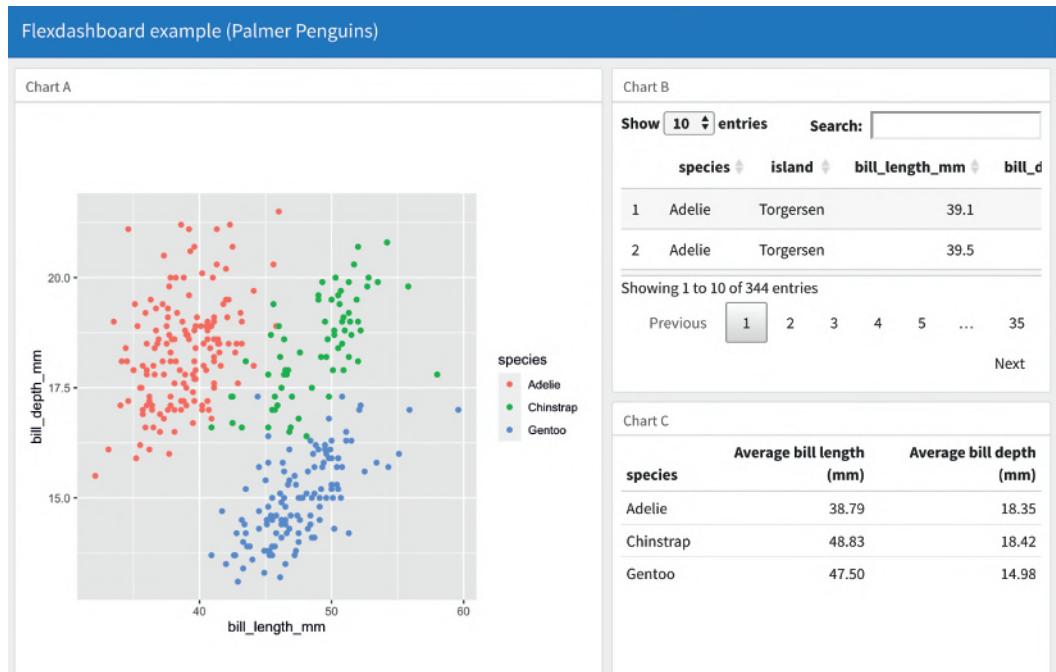


Figure 14.8: Sample flexdashboard output.

The upper-right panel of this dashboard employs **DT** to provide a data table that the user can interact with. However, the dashboard itself is not interactive, in the sense that the user can only change the display through this HTML widget. Changing the display in that upper-right panel has no effect on the other panels. To create a fully interactive *web application*, we need a more powerful tool, which we introduce in the next section.

14.4 Interactive web apps with Shiny

Shiny is a framework for **R** that can be used to create interactive *web applications* and dynamic dashboards. It is particularly attractive because it provides a high-level structure to easily prototype and deploy apps. While a full discussion of Shiny is outside the scope of this book, we will demonstrate how one might create a dynamic web app that allows the user to explore the data set of babies with the same names as the Beatles.

One way to write a Shiny app involves creating a `ui.R` file that controls the user interface,

and a `server.R` file to display the results. (Alternatively, the two files can be combined into a single `app.R` file that includes both components.) These files communicate with each other using *reactive objects* `input` and `output`. Reactive expressions are special constructions that use `input` from *widgets* to return a value. These allow the application to automatically update when the user clicks on a button, changes a slider, or provides other input.

14.4.1 Example: interactive display of the Beatles

For this example, we'd like to let the user pick the start and end years along with a set of checkboxes to include their favorite Beatles.

The `ui.R` file shown in [Figure 14.9](#) sets up a title, creates inputs for the start and end years (with default values), creates a set of check boxes for each of the Beatles' names, then plots the result.

```
# ui.R
beatles_names <- c("John", "Paul", "George", "Ringo")

shinyUI(
  bootstrapPage(
    h3("Frequency of Beatles names over time"),
    numericInput(
      "startyear", "Enter starting year",
      value = 1960, min = 1880, max = 2014, step = 1
    ),
    numericInput(
      "endyear", "Enter ending year",
      value = 1970, min = 1881, max = 2014, step = 1
    ),
    checkboxGroupInput(
      'names', 'Names to display:',
      sort(unique(beatles_names)),
      selected = c("George", "Paul")
    ),
    plotOutput("plot")
  )
)
```

Figure 14.9: User interface code for a simple Shiny app.

The `server.R` file shown in [Figure 14.10](#) loads needed packages, performs some data wrangling, extracts the reactive objects using the `input` object, then generates the desired plot. The `renderPlot()` function returns a reactive object called `plot` that is referenced in `ui.R`. Within this function, the values for the years and Beatles are used within a call to `filter()` to identify what to plot.

```
# server.R
library(tidyverse)
library(babynames)
library(shiny)
```

```
Beatles <- babynames %>%
  filter(name %in% c("John", "Paul", "George", "Ringo") & sex == "M")

shinyServer(
  function(input, output) {
    output$plot <- renderPlot({
      ds <- Beatles %>%
        filter(
          year >= input$startyear, year <= input$endyear,
          name %in% input$names
        )
      ggplot(data = ds, aes(x = year, y = prop, color = name)) +
        geom_line(size = 2)
    })
  }
)
```

Figure 14.10: Server processing code for a simple Shiny app.

Shiny Apps can be run locally within **RStudio**, or deployed on a Shiny App server (such as <http://shinyapps.io>). Please see the book website at <https://mdsr-book.github.io> for access to the code files. Figure 14.11 displays the results when only Paul and George are checked when run locally.

```
library(shiny)
runApp('')
```

14.4.2 More on reactive programming

Shiny is an extremely powerful and complicated system to master. Repeated and gradual exposure to reactive programming and widgets will pay off in terms of flexible and attractive displays. For this example, we demonstrate some additional features that show off some of the possibilities: more general reactive objects, dynamic user interfaces, and progress indicators.

Here we display information about health violations from *New York City* restaurants. The user has the option to specify a *borough* (district) within New York City and a cuisine. Since not every cuisine is available within every borough, we need to dynamically filter the list. We do this by calling `uiOutput()`. This references a reactive object created within the `server()` function. The output is displayed in a `dataTableOutput()` widget from the **DT** package.

```
library(tidyverse)
library(shiny)
library(shinybusy)
library(mdsr)

mergedViolations <- Violations %>%
  left_join(Cuisines)

ui <- fluidPage(
  titlePanel("Restaurant Explorer"),
  ...)
```

Frequency of Beatles names over time

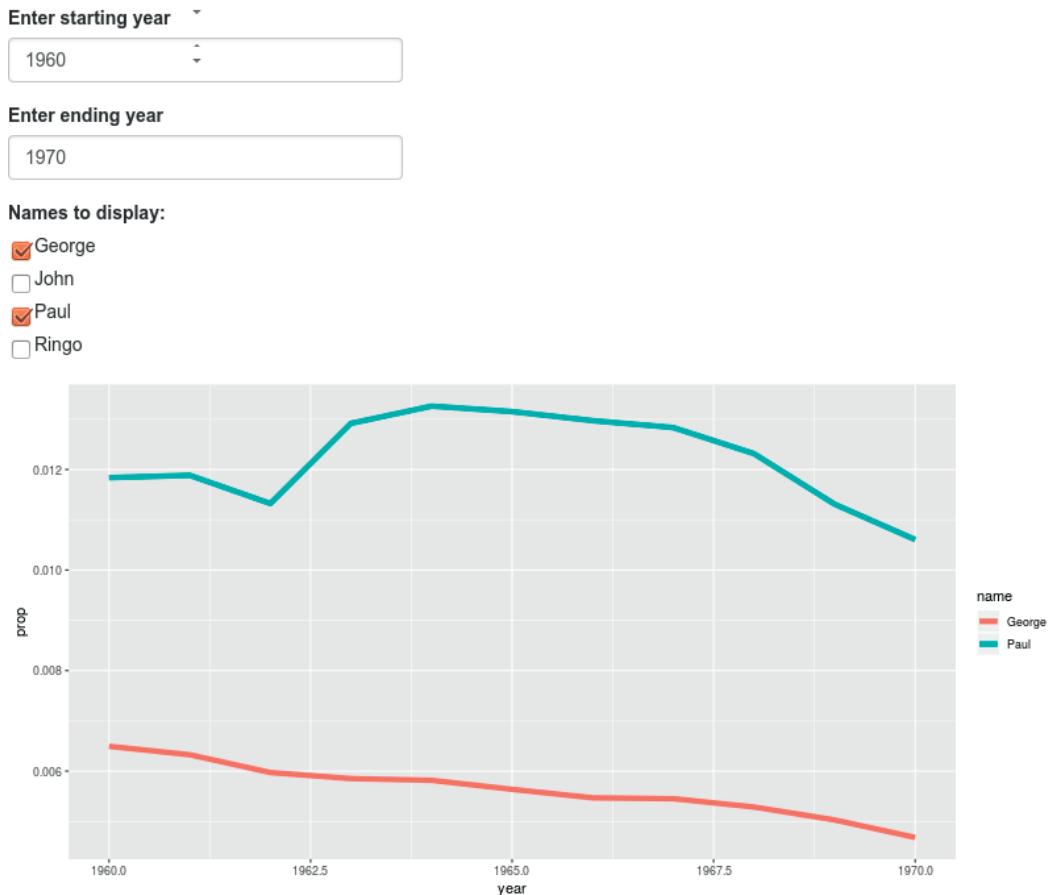


Figure 14.11: A screenshot of the Shiny app displaying babies with Beatles names.

```
fluidRow(
  # some things take time: this lets users know
  add_busy_spinner(spin = "fading-circle"),
  column(
    4,
    selectInput(inputId = "boro",
      label = "Borough:",
      choices = c(
        "ALL",
        unique(as.character(mergedViolations$boro))
      )
    ),
    # display dynamic list of cuisines
    column(4, uiOutput("cuisinecontrols"))
  ),
)
```

```
# Create a new row for the table.
fluidRow(
  DT::dataTableOutput("table")
)
)
```

Figure 14.12: User interface processing code for a more sophisticated Shiny app.

The code shown in Figure 14.12 also includes a call to the `add_busy_spinner()` function from the `shinybusy` package. It takes time to render the various reactive objects, and the spinner shows up to alert the user that there will be a slight delay.

```
server <- function(input, output) {
  datasetboro <- reactive({ # Filter data based on selections
    data <- mergedViolations %>%
      select(
        dba, cuisine_code, cuisine_description, street,
        boro, zipcode, score, violation_code, grade_date
      ) %>%
      distinct()
    req(input$boro) # wait until there's a selection
    if (input$boro != "ALL") {
      data <- data %>%
        filter(boro == input$boro)
    }
    data
  })

  datasetcuisine <- reactive({ # dynamic list of cuisines
    req(input$cuisine) # wait until list is available
    data <- datasetboro() %>%
      unique()
    if (input$cuisine != "ALL") {
      data <- data %>%
        filter(cuisine_description == input$cuisine)
    }
    data
  })

  output$table <- DT::renderDataTable(DT::datatable(datasetcuisine()))

  output$cuisinecontrols <- renderUI({
    availablelevels <-
      unique(sort(as.character(datasetboro()$cuisine_description)))
    selectInput(
      inputId = "cuisine",
      label = "Cuisine:",
      choices = c("ALL", availablelevels)
    )
  })
}
```

```

}

shinyApp(ui = ui, server = server)

```

Figure 14.13: Server processing code for a more sophisticated Shiny app.

The code shown in [Figure 14.13](#) makes up the rest of the Shiny app. We create a reactive object that is dynamically filtered based on which borough and cuisine are selected. Calls made to the `req()` function wait until the reactive inputs are available (at startup these will take time to populate with the default values). The two functions are linked with a call to the `shinyApp()` function. [Figure 14.14](#) displays the Shiny app when it is running.

| | dba | cuisine_code | cuisine_description | street | boro | zipcode |
|------|-----------------------|--------------|---------------------|----------------|------------------|---------|
| 5985 | THE
BAKE
SHOPPE | 8 | Bakery | PAGE
AVENUE | STATEN
ISLAND | 10309 |
| 5986 | THE
BAKE
SHOPPE | 8 | Bakery | PAGE
AVENUE | STATEN
ISLAND | 10309 |

Figure 14.14: A screenshot of the Shiny app displaying New York City restaurants.

14.5 Customization of *ggplot2* graphics

There are endless possibilities for customizing plots in **R** and **ggplot2**. One important concept is the notion of *themes*. In the next section, we will illustrate how to customize a **ggplot2** theme by defining one we include in the **mdsr** package.

ggplot2 provides many different ways to change the appearance of a plot. A comprehensive system of customizations is called a *theme*. In **ggplot2**, a theme is a list of 93 different attributes that define how axis labels, titles, grid lines, etc. are drawn. The default theme is `theme_grey()`.

```
length(theme_grey())
```

```
[1] 93
```

For example, notable features of `theme_grey()` are the distinctive grey background and white grid lines. The `panel.background` and `panel.grid` properties control these aspects of the theme.

```
theme_grey() %>%
  pluck("panel.background")
```

```
List of 5
```

```
$ fill      : chr "grey92"
$ colour    : logi NA
$ size      : NULL
$ linetype   : NULL
$ inherit.blank: logi TRUE
- attr(*, "class")= chr [1:2] "element_rect" "element"
```

```
theme_grey() %>%
  pluck("panel.grid")
```

```
List of 6
```

```
$ colour    : chr "white"
$ size      : NULL
$ linetype   : NULL
$ lineend    : NULL
$ arrow      : logi FALSE
$ inherit.blank: logi TRUE
- attr(*, "class")= chr [1:2] "element_line" "element"
```

A number of useful themes are built into **ggplot2**, including `theme_bw()` for a more traditional white background, `theme_minimal()`, and `theme_classic()`. These can be invoked using the eponymous functions. We compare `theme_grey()` with `theme_bw()` in [Figure 14.15](#).

```
beatles_plot
beatles_plot + theme_bw()
```

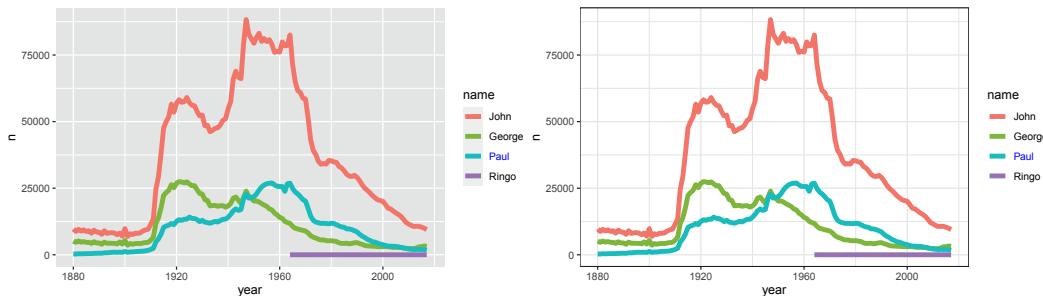


Figure 14.15: Comparison of two **ggplot2** themes. At left, the default grey theme. At right, the black-and-white theme.

We can modify a theme on-the-fly using the `theme()` function. In [Figure 14.16](#) we illustrate how to change the background color and major grid lines color.

```
beatles_plot +
  theme(
    panel.background = element_rect(fill = "cornsilk"),
    panel.grid.major = element_line(color = "dodgerblue")
  )
```

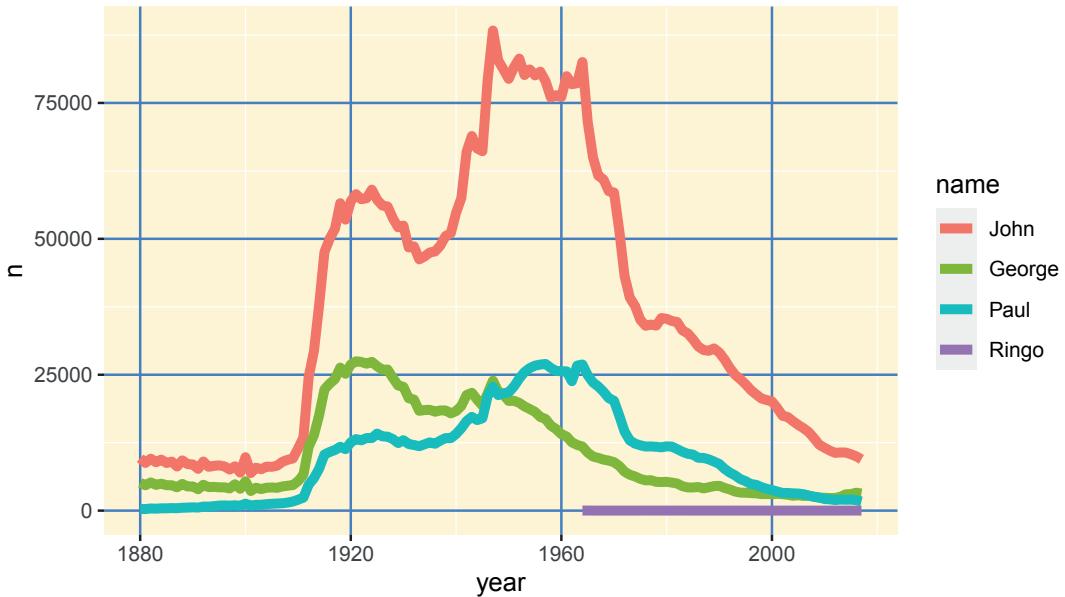


Figure 14.16: Beatles plot with custom **ggplot2** theme.

How did we know the names of those colors? You can display R’s built-in colors using the `colors()` function. There are more intuitive color maps on the Web.

```
head(colors())
```

```
[1] "white"        "aliceblue"     "antiquewhite"  "antiquewhite1"
[5] "antiquewhite2" "antiquewhite3"
```

To create a new theme, write a function that will return a complete **ggplot2** theme. One could write this function by completely specifying all 93 items. However, in this case we illustrate how the `%+replace%` operator can be used to modify an existing theme. We start with `theme_grey()` and change the background color, major and minor grid lines colors, and the default font.

```
theme_mdsr <- function(base_size = 12, base_family = "Helvetica") {
  theme_grey(base_size = base_size, base_family = base_family) %+replace%
  theme(
    axis.text      = element_text(size = rel(0.8)),
    axis.ticks     = element_line(color = "black"),
    legend.key     = element_rect(color = "grey80"),
    panel.background = element_rect(fill = "whitesmoke", color = NA),
    panel.border   = element_rect(fill = NA, color = "grey50"),
    panel.grid.major = element_line(color = "grey80", size = 0.2),
    panel.grid.minor = element_line(color = "grey92", size = 0.5),
    strip.background = element_rect(fill = "grey80", color = "grey50"),
```

```
    size = 0.2)
)
}
```

With our new theme defined, we can apply it in the same way as any of the built-in themes—namely, by calling the `theme_mdsr()` function. Figure 14.17 shows how this stylizes the faceted Beatles time series plot.

```
beatles_plot + facet_wrap(~name) + theme_mdsr()
```

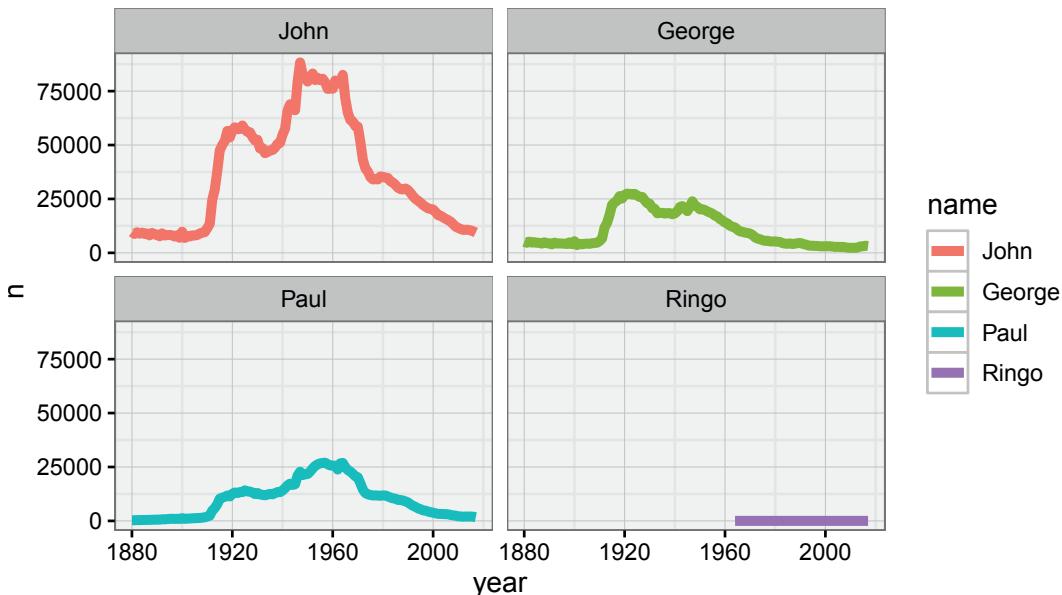


Figure 14.17: Beatles plot with customized `mdsr` theme.

Many people have taken to creating their own themes for `ggplot2`. In particular, the `ggthemes` package features useful (`theme_solarized()`), humorous (`theme_tufte()`), whimsical (`theme_fivethirtyeight()`), and even derisive (`theme_excel()`) themes. Another humorous theme is `theme_xkcd()`, which attempts to mimic the popular Web comic's distinctive hand-drawn styling. This functionality is provided by the `xkcd` package.

```
library(xkcd)
```

To set `xkcd` up, we need to download the pseudo-handwritten font, import it, and then `loadfonts()`. Note that the destination for the fonts is system dependent: On Mac OS X this should be `~/Library/Fonts` while for Ubuntu it is `~/.fonts`.

```
download.file(
  "http://simonsoftware.se/other/xkcd.ttf",
  # ~/Library/Fonts/ for Mac OS X
  dest = "~/.fonts/xkcd.ttf", mode = "wb"
)

font_import(pattern = "[X/x]kcd", prompt = FALSE)
loadfonts()
```

In Figure 14.18, we show the xkcd-styled plot of the popularity of the Beatles names.

```
beatles_plot + theme_xkcd()
```

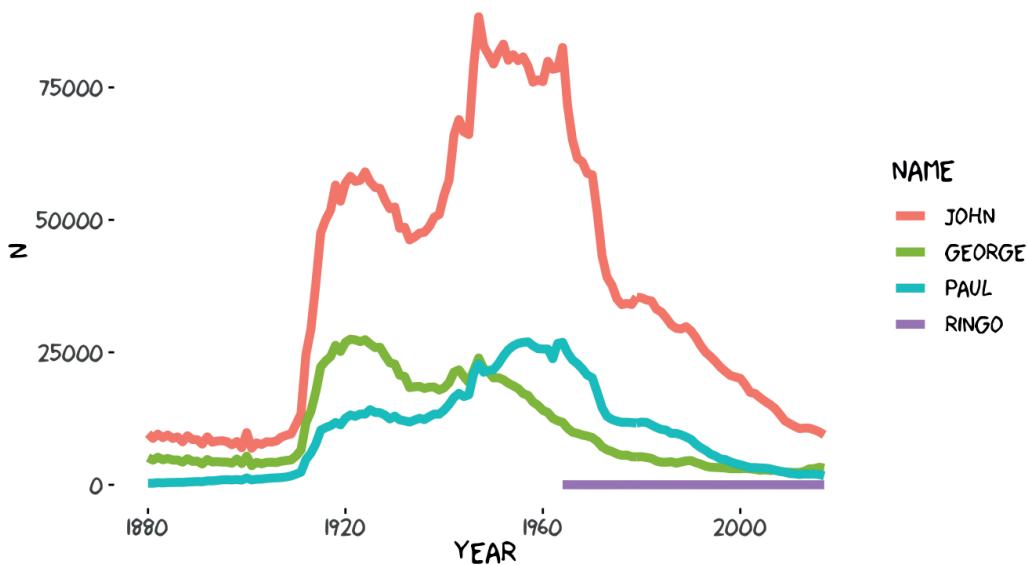


Figure 14.18: Prevalence of Beatles names drawn in the style of an **xkcd** Web comic.

14.6 Extended example: Hot dog eating

Writing in 2011, former *New York Times* data graphic intern Nathan Yau noted that “*Adobe Illustrator* is the industry standard. Every graphic that goes to print at *The New York Times* either was created or edited in Illustrator” (Yau, 2011). To underscore his point, Yau presents the data graphic shown in Figure 14.19, created in **R** but modified in Illustrator.

Ten years later, *The New York Times* data graphic department now produces much of their content using **d3.js**, an interactive JavaScript library that we discussed in Section 14.1. What follows is our best attempt to recreate a static version of Figure 14.19 entirely within **R** using **ggplot2** graphics. After saving the plot as a PDF, we can open it in Illustrator or *Inkscape* for further customization if necessary.

Pro Tip 33. Undertaking such “Copy the Master” exercises (Nolan and Perrett, 2016) is a good way to deepen your skills.

```
library(tidyverse)
library(mdsr)
hd <- read_csv(
  "http://datasets.flowingdata.com/hot-dog-contest-winners.csv"
) %>%
  janitor::clean_names()
glimpse(hd)
```

Winners from Nathan's Hot Dog Eating Contest

Since 1916, the annual eating competition has grown substantially attracting competitors from around the world. This year's competition will be televised on July 4, 2008 at 12pm EDT live on ESPN.

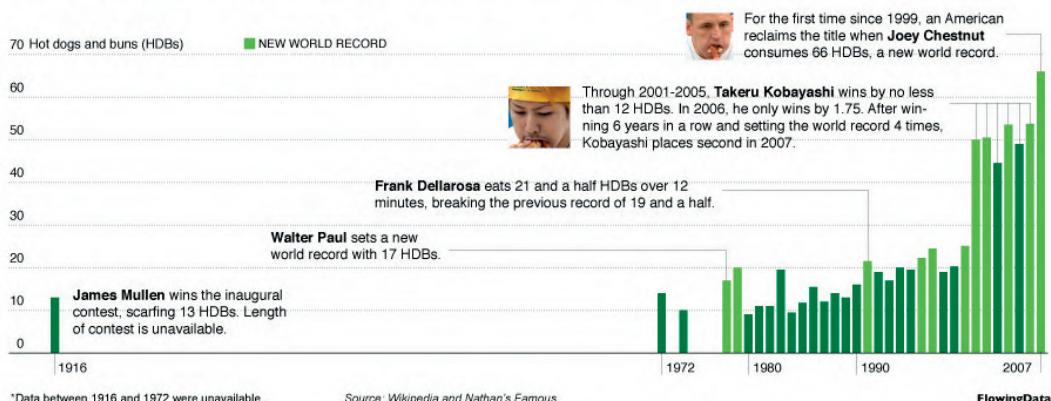


Figure 14.19: Nathan Yau's Hot Dog Eating data graphic that was created in R but modified using Adobe Illustrator (reprinted with permission from flowingdata.com).

Rows: 31

Columns: 5

```
$ year      <dbl> 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988...
$ winner    <chr> "Paul Siederman & Joe Baldini", "Thomas DeBerry", "S...
$ dogs_eaten <dbl> 9.1, 11.0, 11.0, 19.5, 9.5, 11.8, 15.5, 12.0, 14.0, ...
$ country   <chr> "United States", "United States", "United States", "...
$ new_record <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1...
```

The hd data table doesn't provide any data from before 1980, so we need to estimate them from Figure 14.19 and manually add these rows to our data frame.

```
new_data <- tibble(
  year = c(1979, 1978, 1974, 1972, 1916),
  winner = c(NA, "Walter Paul", NA, NA, "James Mullen"),
  dogs_eaten = c(19.5, 17, 10, 14, 13),
  country = rep(NA, 5), new_record = c(1,1,0,0,0)
)
hd <- hd %>%
  bind_rows(new_data)
glimpse(hd)
```

Rows: 36

Columns: 5

```
$ year      <dbl> 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988...
$ winner    <chr> "Paul Siederman & Joe Baldini", "Thomas DeBerry", "S...
$ dogs_eaten <dbl> 9.1, 11.0, 11.0, 19.5, 9.5, 11.8, 15.5, 12.0, 14.0, ...
$ country   <chr> "United States", "United States", "United States", "...
$ new_record <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1...
```

Note that we only want to draw some of the years on the horizontal axis and only every 10th value on the vertical axis.

```
xlabs <- c(1916, 1972, 1980, 1990, 2007)
ylabs <- seq(from = 0, to = 70, by = 10)
```

Finally, the plot only shows the data up until 2008, even though the file contains more recent information than that. Let's define a subset that we'll use for plotting.

```
hd_plot <- hd %>%
  filter(year < 2008)
```

Our most basic plot is shown in [Figure 14.20](#).

```
p <- ggplot(data = hd_plot, aes(x = year, y = dogs_eaten)) +
  geom_col()
p
```

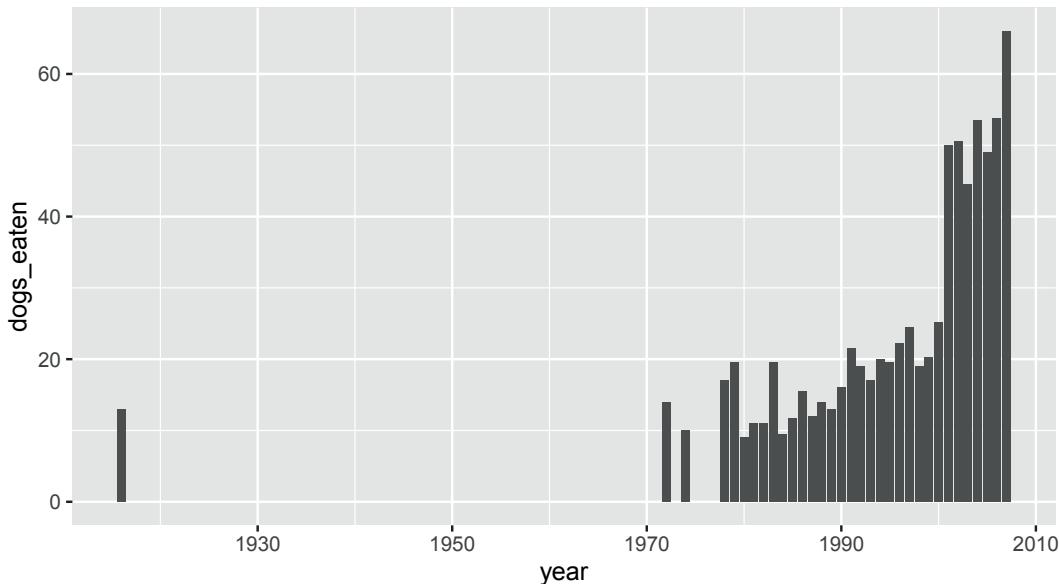


Figure 14.20: A simple bar graph of hot dog eating.

This doesn't provide the context of [Figure 14.19](#), nor the pizzazz. Although most of the important data are already there, we still have a great deal of work to do to make this data graphic as engaging as [Figure 14.19](#). Our recreation is shown in [Figure 14.21](#).

We aren't actually going to draw the y -axis—instead we are going to places the labels for the y values on the plot. We'll put the locations for those values in a data frame.

```
ticks_y <- tibble(x = 1912, y = ylabs)
```

There are many text annotations, and we will collect those into a single data frame. Here, we use the `tribble()` function to create a data frame row-by-row. The format of the input is similar to a CSV (see [Section 6.4.1.1](#)).

```
text <- tribble(
  ~x, ~y, ~label, ~adj,
  # Frank Dellarosa
  1953, 37, paste(
    "Frank Dellarosa eats 21 and a half HDBs over 12",
```

```

"\nminutes, breaking the previous record of 19 and a half."), 0,
# Joey Chestnut
1985, 69, paste(
  "For the first time since 1999, an American",
  "\nreclaims the title when Joey Chestnut",
  "\nconsumes 66 HDBs, a new world record."), 0,
# Kobayashi
1972, 55, paste(
  "Through 2001-2005, Takeru Kobayashi wins by no less",
  "\nthan 12 HDBs. In 2006, he only wins by 1.75. After win-",
  "\nnning 6 years in a row and setting the world record 4 times",
  "\nKobayashi places second in 2007."), 0,
# Walter Paul
1942, 26, paste(
  "Walter Paul sets a new",
  "\nworld record with 17 HDBs."), 0,
# James Mullen
1917, 10.5, paste(
  "James Mullen wins the inaugural",
  "\ncontest, scarfing 13 HDBs. Length",
  "\nof contest unavailable."), 0,
1935, 72, "NEW WORLD RECORD", 0,
1914, 72, "Hot dogs and buns (HDBs)", 0,
1940, 2, "*Data between 1916 and 1972 were unavailable", 0,
1922, 2, "Source: FlowingData", 0,
)

```

The grey segments that connect the text labels to the bars in the plot must be manually specified in another data frame. Here, we use `tribble()` to construct a data frame in which each row corresponds to a single segment. Next, we use the `unnest()` function to expand the data frame so that each row corresponds to a single point. This will allow us to pass it to the `geom_segment()` function.

```

segments <- tribble(
  ~x, ~y,
  c(1978, 1991, 1991, NA), c(37, 37, 21, NA),
  c(2004, 2007, 2007, NA), c(69, 69, 66, NA),
  c(1998, 2006, 2006, NA), c(58, 58, 53.75, NA),
  c(2005, 2005, NA), c(58, 49, NA),
  c(2004, 2004, NA), c(58, 53.5, NA),
  c(2003, 2003, NA), c(58, 44.5, NA),
  c(2002, 2002, NA), c(58, 50.5, NA),
  c(2001, 2001, NA), c(58, 50, NA),
  c(1955, 1978, 1978), c(26, 26, 17)
) %>%
  unnest(cols = c(x, y))

```

Finally, we draw the plot, layering on each of the elements that we defined above.

```

p +
  geom_col(aes(fill = factor(new_record))) +
  geom_hline(yintercept = 0, color = "darkgray") +

```

```
scale_fill_manual(name = NULL,
  values = c("0" = "#006f3c", "1" = "#81c450"))
) +
scale_x_continuous(
  name = NULL, breaks = xlabs, minor_breaks = NULL,
  limits = c(1912, 2008), expand = c(0, 1)
) +
scale_y_continuous(
  name = NULL, breaks = ylabs, labels = NULL,
  minor_breaks = NULL, expand = c(0.01, 1)
) +
geom_text(
  data = ticks_y, aes(x = x, y = y + 2, label = y),
  size = 3
) +
labs(
  title = "Winners from Nathan's Hot Dog Eating Contest",
  subtitle = paste(
    "Since 1916, the annual eating competition has grown substantially",
    "attracting competitors from around\nthe world.",
    "This year's competition will be televised on July 4, 2008",
    "at 12pm EDT live on ESPN.\n\n"
  )
) +
geom_text(
  data = text, aes(x = x, y = y, label = label),
  hjust = "left", size = 3
) +
geom_path(
  data = segments, aes(x = x, y = y), col = "darkgray"
) +
# Key
geom_rect(
  xmin = 1933, ymin = 70.75, xmax = 1934.3, ymax = 73.25,
  fill = "#81c450", color = "white"
) +
guides(fill = FALSE) +
theme(
  panel.background = element_rect(fill = "white"),
  panel.grid.major.y =
    element_line(color = "gray", linetype = "dotted"),
  plot.title = element_text(face = "bold", size = 16),
  plot.subtitle = element_text(size = 10),
  axis.ticks.length = unit(0, "cm")
)
```

Winners from Nathan's Hot Dog Eating Contest

Since 1916, the annual eating competition has grown substantially attracting competitors from around the world. This year's competition will be televised on July 4, 2008 at 12pm EDT live on ESPN.

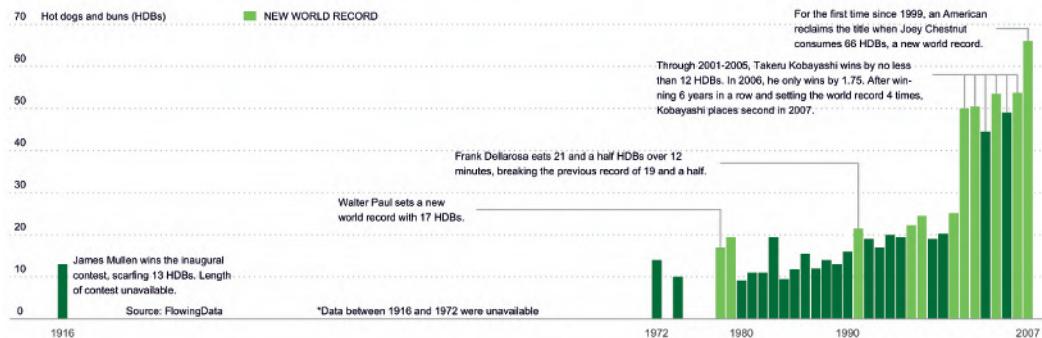


Figure 14.21: Recreation of the hot dog graphic.

14.7 Further resources

The `htmlwidgets` website includes a gallery of showcase applications of JavaScript in **R**. Details and examples of use of the `flexdashboard` package can be found at <https://rmarkdown.rstudio.com/flexdashboard>.

The Shiny gallery (<http://shiny.rstudio.com/gallery>) includes a number of interactive visualizations (and associated code), many of which feature JavaScript libraries. Nearly 200 examples of widgets and idioms in Shiny are available at <https://github.com/rstudio/shiny-examples>. The **RStudio** Shiny cheat sheet is a useful reference. Wickham (2020b) provides a comprehensive guide to many aspects of Shiny development.

The `extrafont` package makes use of the full suite of fonts that are installed on your computer, rather than the relatively small sets of fonts that **R** knows about. (These are often device and operating system dependent, but three fonts—`sans`, `serif`, and `mono`—are always available.) For a more extensive tutorial on how to use the `extrafont` package, see <http://tinyurl.com/fonts-rcharts>.

14.8 Exercises

Problem 1 (Easy): Modify the Shiny app that displays the frequency of Beatles names over time so that it has a `checkboxInput()` widget that uses the `theme_tufte()` theme from the `ggthemes` package.

Problem 2 (Medium): Create a Shiny app that demonstrates the use of at least five widgets.

Problem 3 (Medium): The `macleish` package contains weather data collected every 10 minutes in 2015 from two weather stations in Whately, Massachusetts.

Using the `ggplot2` package, create a data graphic that displays the average temperature over

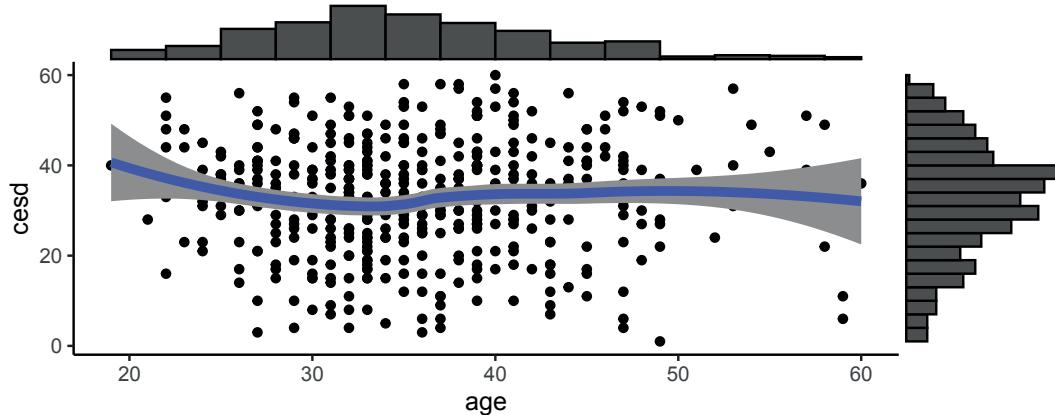
each 10-minute interal (`temperature`) as a function of time (`when`) from the `whately_2015` dataframe. Create annotations to include context about the four seasons: the date of the vernal and autumnal equinoxes, and the summer and winter solstices.

Problem 4 (Medium): Modify the restaurant violations Shiny app so that it displays a table of the number of restaurants within a given type of cuisine along with a count of restaurants (as specified by the `dba` variable). (Hint: Be sure not to double count. The dataset should include 842 unique pizza restaurants in all boroughs and 281 Caribbean restaurants in Brooklyn.)

Problem 5 (Medium): Create your own `ggplot2` theme. Describe the choices you made and justify why you made them using the principles introduced earlier.

Problem 6 (Medium): The following code generates a scatterplot with marginal histograms.

```
p <- ggplot(HELPrc, aes(x = age, y = cesd)) +
  geom_point() +
  theme_classic() +
  stat_smooth(method = "loess", formula = y ~ x, size = 2)
ggExtra::ggMarginal(p, type = "histogram", binwidth = 3)
```



Find an example where such a display might be useful. Be sure to interpret your graphical display

Problem 7 (Medium): Using data from the `palmerpenguins` package, create a Shiny app that displays measurements from the `penguins` dataframe. Allow the user to select a species or a gender, and to choose between various attributes on a scatterplot. (Hint: examples of similar apps can be found at the Shiny gallery).

Problem 8 (Medium): Create a Shiny app to display an interactive time series plot of the `macleish` weather data. Include a selection box to alternate between data from the `whately_2015` and `orchard_2015` weather stations. Add a selector of dates to include in the display. Do you notice any irregularities?

Problem 9 (Hard): Repeat the earlier question using the weather data from the MacLeish field station, but include context on major storms listed on the Wikipedia pages: 2014–2015 North American Winter and 2015–2016 North American Winter.

Problem 10 (Hard): Using data from the `Lahman` package, create a Shiny app that displays career leaderboards similar to the one at http://www.baseball-reference.com/leaders/HR_season.shtml. Allow the user to select a statistic of their choice, and to choose between

Career, Active, Progressive, and Yearly League leaderboards. (Hint: examples of similar apps can be found at the Shiny gallery.)

14.9 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/dataviz-III.html#dataviz-III-online-exercises>

Database querying using SQL

Thus far, most of the data that we have encountered in this book (such as the **Lahman** baseball data in [Chapter 4](#)) has been small—meaning that it will fit easily in a personal computer’s memory. In this chapter, we will explore approaches for working with data sets that are larger—let’s call them *medium* data. These data will fit on a personal computer’s hard disk, but not necessarily in its memory. Thankfully, a venerable solution for retrieving medium data from a database has been around since the 1970s: *SQL* (structured query language). Database management systems implementing SQL provide a ubiquitous architecture for storing and querying data that is relational in nature. While the death of SQL has been presaged many times, it continues to provide an effective solution for medium data. Its wide deployment makes it a “must-know” tool for data scientists. For those of you with bigger appetites, we will consider some extensions that move us closer to a true big data setting in [Chapter 21](#).

15.1 From dplyr to SQL

Recall the **airlines** data that we encountered in [Chapter 9](#). Using the **dplyr** verbs that we developed in [Chapters 4](#) and [5](#), consider retrieving the top on-time carriers with at least 100 flights arriving at JFK in September 2016. If the data are stored in data frames called **flights** and **carriers**, then we might write a **dplyr** pipeline like this:

```
q <- flights %>%
  filter(
    year == 2016 & month == 9,
    dest == "JFK"
  ) %>%
  inner_join(carriers, by = c("carrier" = "carrier")) %>%
  group_by(name) %>%
  summarize(
    N = n(),
    pct_ontime = sum(arr_delay <= 15) / n()
  ) %>%
  filter(N >= 100) %>%
  arrange(desc(pct_ontime))
head(q, 4)

# Source:     lazy query [?? x 3]
# Database:   mysql 5.6.40-log
#   [mdsr_public@mdsr.cdc7tgkkqd0n.us-east-1.rds.amazonaws.com:/airlines]
# Ordered by: desc(pct_ontime)
```

| | | N | pct_ontime |
|---|------------------------|-------|------------|
| | <chr> | <dbl> | <dbl> |
| 1 | Delta Air Lines Inc. | 2396 | 0.869 |
| 2 | Virgin America | 347 | 0.833 |
| 3 | JetBlue Airways | 3463 | 0.817 |
| 4 | American Airlines Inc. | 1397 | 0.782 |

However, the `flights` data frame can become very large. Going back to 1987, there are more than 169 million individual flights—each comprising a different row in this table. These data occupy nearly 20 gigabytes as CSVs, and thus are problematic to store in a personal computer’s memory. Instead, we write these data to disk, and use a querying language to access only those rows that interest us. In this case, we configured `dplyr` to access the `flights` data on a MySQL server. The `dbConnect_scidb()` function from the `mdsr` package provides a connection to the `airlines` database that lives on a remote MySQL server and stores it as the object `db`. The `tbl()` function from `dplyr` maps the `flights` table in that `airlines` database to an object in `R`, in this case also called `flights`. The same is done for the `carriers` table.

```
library(tidyverse)
library(mdsr)
db <- dbConnect_scidb("airlines")
flights <- tbl(db, "flights")
carriers <- tbl(db, "carriers")
```

Note that while we can use the `flights` and `carriers` objects *as if* they were data frames, they are not, in fact, `data.frames`. Rather, they have class `tbl_MySQLConnection`, and more generally, `tbl_sql`. A `tbl` is a special kind of object created by `dplyr` that behaves similarly to a `data.frame`.

```
class(flights)
```

```
[1] "tbl_MySQLConnection" "tbl_dbi"                  "tbl_sql"
[4] "tbl_lazy"           "tbl"
```

Note also that in the output of our pipeline above, there is an explicit mention of a MySQL database. We set up this database ahead of time (see [Chapter 16](#) for instructions on doing this), but `dplyr` allows us to interact with these `tbls` as if they were `data.frames` in our `R` session. This is a powerful and convenient illusion!

What is actually happening is that `dplyr` translates our pipeline into SQL. We can see the translation by passing the pipeline through the `show_query()` function using our previously created query.

```
show_query(q)
```

```
<SQL>
SELECT *
FROM (SELECT `name`, COUNT(*) AS `N`, SUM(`arr_delay` <= 15.0) /
      COUNT(*) AS `pct_ontime`
      FROM (SELECT `year`, `month`, `day`, `dep_time`, `sched_dep_time`, `dep_delay`,
                  `arr_time`, `sched_arr_time`, `arr_delay`, `LHS`.`carrier` AS `carrier`,
                  `tailnum`, `flight`, `origin`, `dest`, `air_time`, `distance`, `cancelled`,
                  `diverted`, `hour`, `minute`, `time_hour`, `name`
                  FROM (SELECT *
                        FROM `flights`
```

```

WHERE ((`year` = 2016.0 AND `month` = 9.0) AND (`dest` = 'JFK'))) `LHS`
INNER JOIN `carriers` AS `RHS`
ON (`LHS`.`carrier` = `RHS`.`carrier`)
) `q01`
GROUP BY `name` `q02`
WHERE (`N` >= 100.0)
ORDER BY `pct_ontime` DESC

```

Understanding this output is not important—the translator here is creating temporary tables with unintelligible names—but it should convince you that even though we wrote our pipeline in **R**, it was translated to SQL. **dplyr** will do this automatically any time you are working with objects of class `tbl_sql`. If we were to write an SQL query equivalent to our pipeline, we would write it in a more readable format:

```

SELECT
  c.name,
  SUM(1) AS N,
  SUM(arr_delay <= 15) / SUM(1) AS pct_ontime
FROM flights AS f
JOIN carriers AS c ON f.carrier = c.carrier
WHERE year = 2016 AND month = 9
  AND dest = 'JFK'
GROUP BY name
HAVING N >= 100
ORDER BY pct_ontime DESC
LIMIT 0,4;

```

How did **dplyr** perform this translation?¹ As we learn SQL, the parallels will become clear (e.g., the **dplyr** verb `filter()` corresponds to the SQL `WHERE` clause). But what about the formulas we put in our `summarize()` command? Notice that the **R** command `n()` was converted into ("COUNT(*) in SQL. This is not magic either: the `translate_sql()` function provides translation between **R** commands and SQL commands. For example, it will translate basic mathematical expressions.

```

library(dbplyr)
translate_sql(mean(arr_delay, na.rm = TRUE))

```

```
<SQL> AVG(`arr_delay`) OVER ()
```

However, it only recognizes a small set of the most common operations—it cannot magically translate any **R** function into SQL. It can be easily tricked. For example, if we make a copy of the very common **R** function `paste0()` (which concatenates strings) called `my_paste()`, that function is not translated.

```

my_paste <- paste0

translate_sql(my_paste("this", "is", "a", "string"))

<SQL> my_paste('this', 'is', 'a', 'string')

```

This is a good thing—since it allows you to pass arbitrary SQL code through. But you have

¹The difference between the SQL query that we wrote and the translated SQL query that **dplyr** generated from our pipeline is a consequence of the syntactic logic of **dplyr** and needn't concern us.

to know what you are doing. Since there is no SQL function called `my_paste()`, this will throw an error, even though it is a perfectly valid **R** expression.

```
carriers %>%
  mutate(name_code = my_paste(name, "(", carrier, ")"))

Error in .local(conn, statement, ...): could not run statement:
execute command denied to user 'mdsr_public'@'%' for 'airlines.my_paste'
class(carriers)

[1] "tbl_MySQLConnection" "tbl_db"           "tbl_sql"
[4] "tbl_lazy"             "tbl"
```

Because `carriers` is a `tbl_sql` and not a `data.frame`, the MySQL server is actually doing the computations here. The `dplyr` pipeline is simply translated into SQL and submitted to the server. To make this work, we need to replace `my_paste()` with its MySQL equivalent command, which is `CONCAT()`.

```
carriers %>%
  mutate(name_code = CONCAT(name, "(", carrier, ")"))

# Source:  lazy query [?? x 3]
# Database: mysql 5.6.40-log
#   [mdsr_public@mdsr.cdc7tgkkqd0n.us-east-1.rds.amazonaws.com:/airlines]
  carrier name          name_code
  <chr>   <chr>
1 02Q    Titan Airways  Titan Airways(02Q)
2 04Q    Tradewind Aviation Tradewind Aviation(04Q)
3 05Q    Comlux Aviation, AG Comlux Aviation, AG(05Q)
4 06Q    Master Top Linhas Aereas Ltd. Master Top Linhas Aereas Ltd.(06Q)
5 07Q    Flair Airlines Ltd.  Flair Airlines Ltd.(07Q)
6 09Q    Swift Air, LLC     Swift Air, LLC(09Q)
7 0BQ    DCA                DCA(0BQ)
8 0CQ    ACM AIR CHARTER GmbH ACM AIR CHARTER GmbH(0CQ)
9 0GQ    Inter Island Airways, d/b/a I~ Inter Island Airways, d/b/a Inter-
10 0HQ   Polar Airlines de Mexico d/b/~ Polar Airlines de Mexico d/b/a No~
# ... with more rows
```

The syntax of this looks a bit strange, since `CONCAT()` is not a valid **R** expression—but it works.

Another alternative is to pull the `carriers` data into **R** using the `collect()` function first, and then use `my_paste()` as before.² The `collect()` function breaks the connection to the MySQL server and returns a `data.frame` (which is also a `tbl_df`).

```
carriers %>%
  collect() %>%
  mutate(name_code = my_paste(name, "(", carrier, ")"))

# A tibble: 1,610 x 3
  carrier name          name_code
  <chr>   <chr>
```

²Of course, this will work well when the `carriers` table is not too large but could become problematic if it is.

```

1 02Q    Titan Airways          Titan Airways(02Q)
2 04Q    Tradewind Aviation    Tradewind Aviation(04Q)
3 05Q    Comlux Aviation, AG   Comlux Aviation, AG(05Q)
4 06Q    Master Top Linhas Aereas Ltd. Master Top Linhas Aereas Ltd.(06Q)
5 07Q    Flair Airlines Ltd.    Flair Airlines Ltd.(07Q)
6 09Q    Swift Air, LLC        Swift Air, LLC(09Q)
7 0BQ    DCA                   DCA(0BQ)
8 0CQ    ACM AIR CHARTER GmbH  ACM AIR CHARTER GmbH(0CQ)
9 0GQ    Inter Island Airways, d/b/a I~ Inter Island Airways, d/b/a Inter~
10 0HQ   Polar Airlines de Mexico d/b/~ Polar Airlines de Mexico d/b/a No~
# ... with 1,600 more rows

```

This example illustrates that when using **dplyr** with a `tbl_sql` backend, one must be careful to use expressions that SQL can understand. This is just one more reason why it is important to know SQL on its own and not rely entirely on the **dplyr** front-end (as wonderful as it is).

For querying a database, the choice of whether to use **dplyr** or SQL is largely a question of convenience. If you want to work with the result of your query in **R**, then use **dplyr**. If, on the other hand, you are pulling data into a *web application*, you likely have no alternative other than writing the SQL query yourself. **dplyr** is just one SQL client that only works in **R**, but there are SQL servers all over the world, in countless environments. Furthermore, as we will see in [Chapter 21](#), even the big data tools that supersede SQL assume prior knowledge of SQL. Thus, in this chapter, we will learn how to write SQL queries.

15.2 Flat-file databases

It may be the case that all of the data that you have encountered thus far has been in a application-specific format (e.g., **R**, *Minitab*, *SPSS*, *Stata* or has taken the form of a single CSV (comma-separated value) file. This file consists of nothing more than rows and columns of data, usually with a header row providing names for each of the columns. Such a file is known as known as a *flat file*, since it consists of just one flat (e.g., two-dimensional) file. A *spreadsheet* application—like *Excel* or *Google Sheets*—allows a user to open a flat file, edit it, and also provides a slew of features for generating additional columns, formatting cells, etc. In **R**, the `read_csv()` command from the `readr` package converts a flat file database into a `data.frame`.

These flat-file databases are both extremely common and extremely useful, so why do we need anything else? One set of limitations comes from computer hardware. A personal computer has two main options for storing data:

- Memory (RAM): the amount of data that a computer can work on at once. Modern computers typically have a few gigabytes of memory. A computer can access data in memory extremely quickly (tens of GBs per second).
- Hard Disk: the amount of data that a computer can store permanently. Modern computers typically have hundreds or even thousands of gigabytes (terabytes) of storage space. However, accessing data on disk is orders of magnitude slower than accessing data in memory (hundreds of MBs per second).

Thus, there is a trade-off between storage space (disks have more room) and speed (memory

is much faster to access). It is important to recognize that these are *physical* limitations—if you only have 4 Gb of RAM on your computer, you simply can't read more than 4 Gb of data into memory.³

In general, all objects in your **R** workspace are stored in memory. Note that the `carriers` object that we created earlier occupies very little memory (since the data still lives on the SQL server), whereas `collect(carriers)` pulls the data into **R** and occupies much more memory.

Pro Tip 34. You can find out how much memory an object occupies in **R** using the `object.size()` function and its `print` method.

```
carriers %>%
  object.size() %>%
  print(units = "Kb")
```

3.6 Kb

```
carriers %>%
  collect() %>%
  object.size() %>%
  print(units = "Kb")
```

234.8 Kb

For a typical **R** user, this means that it can be difficult or impossible to work with a data set stored as a `data.frame` that is larger than a few Gb. The following bit of code will illustrate that a data set of random numbers with 100 columns and 1 million rows occupies more than three-quarters of a Gb of memory on this computer.

```
n <- 100 * 1e6
x <- matrix(runif(n), ncol = 100)
dim(x)

[1] 1000000      100
print(object.size(x), units = "Mb")
```

762.9 Mb

Thus, by the time that `data.frame` reached 10 million rows, it would be problematic for most personal computers—probably making your machine sluggish and unresponsive—and it could never reach 100 million rows. But Google processes over 3.5 *billion* search queries per day! We know that they get stored somewhere—where do they all go?

To work effectively with larger data, we need a system that stores *all* of the data on disk, but allows us to access a portion of the data in memory easily. A *relational database*—which stores data in a collection of linkable tables—provides a powerful solution to this problem. While more sophisticated approaches are available to address big data challenges, databases are a venerable solution for medium data.

³In practice, the limit is much lower than that, since the operating system occupies a fair amount of memory. Virtual memory, which uses the hard disk to allocate extra memory, can be another workaround, but cannot sidestep the throughput issue given the inherent limitations of hard drives or solid state devices.

15.3 The SQL universe

SQL (Structured Query Language) is a programming language for *relational database management systems*. Originally developed in the 1970s, it is a mature, powerful, and widely used storage and retrieval solution for data of many sizes. *Google, Facebook, Twitter, Reddit, LinkedIn, Instagram*, and countless other companies all access large datastores using SQL.

Relational database management systems (RDBMS) are very efficient for data that is naturally broken into a series of *tables* that are linked together by *keys*. A table is a two-dimensional array of data that has *records* (rows) and *fields* (columns). It is very much like a `data.frame` in **R**, but there are some important differences that make SQL more efficient under certain conditions.

The theoretical foundation for SQL is based on *relational algebra* and *tuple relational calculus*. These ideas were developed by mathematicians and computer scientists, and while they are not required knowledge for our purposes, they help to solidify SQL's standing as a data storage and retrieval system.

SQL has been an American National Standards Institute (ANSI) standard since 1986, but that standard is only loosely followed by its implementing developers. Unfortunately, this means that there are many different dialects of SQL, and translating between them is not always trivial. However, the broad strokes of the SQL language are common to all, and by learning one dialect, you will be able to easily understand any other (Kline et al., 2008).

Major implementations of SQL include:

- *Oracle*: corporation that claims #1 market share by revenue—now owns MySQL.
- *Microsoft SQL Server*: another widespread corporate SQL product.
- *SQLite*: a lightweight, open-source version of SQL that has recently become the most widely used implementation of SQL, in part due to its being embedded in *Android*, the world's most popular mobile operating system. SQLite is an excellent choice for relatively simple applications—like storing data associated with a particular mobile app—but has neither the features nor the scalability for persistent, multi-user, multi-purpose applications.
- *MySQL*: the most popular client-server RDBMS. It is open source, but is now owned by Oracle Corporation, and that has caused some tension in the open-source community. One of the original developers of MySQL, Monty Widenius, now maintains *MariaDB* as a community fork. MySQL is used by Facebook, Google, LinkedIn, and Twitter.
- *PostgreSQL*: a feature-rich, standards-compliant, open-source implementation growing in popularity. PostgreSQL hews closer to the ANSI standard than MySQL, supports more functions and data types, and provides powerful procedural languages that can extend its base functionality. It is used by Reddit and Instagram, among others.
- *MonetDB* and *MonetDBLite*: open-source implementations that are column-based, rather than the traditional row-based systems. Column-based RDBMSs scale better for big data. **MonetDBLite** is an **R** package that provides a local experience similar to SQLite.
- *Vertica*: a commercial column-based implementation founded by Postgres originator Michael Stonebraker and now owned by *Hewlett-Packard*.

We will focus on MySQL, but most aspects are similar in PostgreSQL or SQLite (see [Appendix F](#) for setup instructions).

15.4 The SQL data manipulation language

MySQL is based on a client-server model. This means that there is a *database server* that stores the data and executes queries. It can be located on the user's local computer or on a remote server. We will be connecting to a server hosted by *Amazon Web Services*. To retrieve data from the server, one can connect to it via any number of client programs. One can of course use the command-line `mysql` program, or the official *GUI* application: *MySQL Workbench*. While we encourage the reader to explore both options—we most often use the Workbench for MySQL development—the output you will see in this presentation comes directly from the MySQL command line client.

Pro Tip 35. Even though `dplyr` enables one to execute most queries using **R** syntax, and without even worrying so much where the data are stored, learning *SQL* is valuable in its own right due to its ubiquity.

Pro Tip 36. If you are just learning *SQL* for the first time, use the command-line client and/or one of the *GUI* applications. The former provides the most direct feedback, and the latter will provide lots of helpful information.

Information about setting up a MySQL database can be found in [Appendix F](#): we assume that this has been done on a local or remote machine. In what follows, you will see SQL commands and their results in tables. To run these on your computer, please see [Section F.4](#) for information about connecting to a MySQL server.

As noted in [Chapter 1](#), the `airlines` package streamlines construction an SQL database containing over 169 million flights. These data come directly from the *United States Bureau of Transportation Statistics*. We access a remote SQL database that we have already set up and populated using the `airlines` package. Note that this database is relational and consists of multiple tables.

```
SHOW TABLES;
```

Tables_in_airlines

airports
carriers
flights
planes

Note that every SQL statement must end with a semicolon. To see what columns are present in the `airports` table, we ask for a description.

```
DESCRIBE airports;
```

Field	Type	Null	Key	Default	Extra
faa	varchar(3)	NO	PRI		
name	varchar(255)	YES		NA	
lat	decimal(10,7)	YES		NA	
lon	decimal(10,7)	YES		NA	
alt	int(11)	YES		NA	
tz	smallint(4)	YES		NA	
dst	char(1)	YES		NA	
city	varchar(255)	YES		NA	
country	varchar(255)	YES		NA	

This command tells us the names of the field (or variables) in the table, as well as their data type, and what kind of keys might be present (we will learn more about keys in [Chapter 16](#)).

Next, we want to build a *query*. Queries in SQL start with the `SELECT` keyword and consist of several clauses, which have to be written in this order:

- `SELECT` allows you to list the columns, or functions operating on columns, that you want to retrieve. This is an analogous operation to the `select()` verb in `dplyr`, potentially combined with `mutate()`.
- `FROM` specifies the table where the data are.
- `JOIN` allows you to stitch together two or more tables using a key. This is analogous to the `inner_join()` and `left_join()` commands in `dplyr`.
- `WHERE` allows you to filter the records according to some criteria. This is an analogous operation to the `filter()` verb in `dplyr`.
- `GROUP BY` allows you to aggregate the records according to some shared value. This is an analogous operation to the `group_by()` verb in `dplyr`.
- `HAVING` is like a `WHERE` clause that operates on the result set—not the records themselves. This is analogous to applying a second `filter()` command in `dplyr`, after the rows have already been aggregated.
- `ORDER BY` is exactly what it sounds like—it specifies a condition for ordering the rows of the result set. This is analogous to the `arrange()` verb in `dplyr`.
- `LIMIT` restricts the number of rows in the output. This is similar to the `R` commands `head()` and `slice()`.

Only the `SELECT` and `FROM` clauses are required. Thus, the simplest query one can write is:

```
SELECT * FROM flights;
```

DO NOT EXECUTE THIS QUERY! This will cause all 169 million records to be dumped! This will not only crash your machine, but also tie up the server for everyone else!

A safe query is:

```
SELECT * FROM flights LIMIT 0,10;
```

We can specify a subset of variables to be displayed. [Table 15.1](#) displays the results, limited to the specified fields and the first 10 records.

```
SELECT year, month, day, dep_time, sched_dep_time, dep_delay, origin
FROM flights
LIMIT 0, 10;
```

Table 15.1: Specifying a subset of variables.

year	month	day	dep_time	sched_dep_time	dep_delay	origin
2010	10	1	1	2100	181	EWR
2010	10	1	1	1920	281	FLL
2010	10	1	3	2355	8	JFK
2010	10	1	5	2200	125	IAD
2010	10	1	7	2245	82	LAX
2010	10	1	7	10	-3	LAX
2010	10	1	7	2150	137	ATL
2010	10	1	8	15	-7	SMF
2010	10	1	8	10	-2	LAS
2010	10	1	10	2225	105	SJC

Table 15.2: Equivalent commands in SQL and R, where *a* and *b* are SQL tables and R dataframes.

Concept	SQL	R
Filter by rows & columns	<code>SELECT col1, col2 FROM a WHERE col3 = 'x'</code>	<code>a %>% filter(col3 == 'x')</code>
Aggregate by rows	<code>SELECT id, SUM(col1) FROM a GROUP BY id</code>	<code>a %>% group_by(id) %>% summarize(SUM(col1))</code>
Combine two tables	<code>SELECT * FROM a JOIN b ON a.id = b.id</code>	<code>a %>% inner_join(b, by = c('id' = 'id'))</code>

The astute reader will recognize the similarities between the five idioms for single table analysis and the join operations discussed in [Chapters 4](#) and [5](#) and the SQL syntax. This is not a coincidence! In the contrary, **dplyr** represents a concerted effort to bring the almost natural language SQL syntax to **R**. For this book, we have presented the **R** syntax first, since much of our content is predicated on the basic data wrangling skills developed previously. But historically, SQL predated the **dplyr** by decades. In [Table 15.2](#), we illustrate the functional equivalence of SQL and **dplyr** commands.

15.4.1 **SELECT...FROM**

As noted above, every SQL **SELECT** query must contain **SELECT** and **FROM**. The analyst may specify columns to be retrieved. We saw above that the **airports** table contains seven columns. If we only wanted to retrieve the FAA code and name of each airport, we could write the following query.

```
SELECT faa, name FROM airports;
```

faa	name
04G	Lansdowne Airport
06A	Moton Field Municipal Airport
06C	Schaumburg Regional

In addition to columns that are present in the database, one can retrieve columns that are

functions of other columns. For example, if we wanted to return the geographic coordinates of each airport as an (x, y) pair, we could combine those fields.

```
SELECT
    name,
    CONCAT('(', lat, ', ', lon, ')')
FROM airports
LIMIT 0, 6;
```

name	CONCAT('(', lat, ', ', lon, ')')
Lansdowne Airport	(41.1304722, -80.6195833)
Moton Field Municipal Airport	(32.4605722, -85.6800278)
Schaumburg Regional	(41.9893408, -88.1012428)
Randall Airport	(41.4319120, -74.3915611)
Jekyll Island Airport	(31.0744722, -81.4277778)
Elizabethton Municipal Airport	(36.3712222, -82.1734167)

Note that the column header for the derived column above is ungainly, since it consists of the entire formula that we used to construct it! This is difficult to read, and would be cumbersome to work with. An easy fix is to give this derived column an *alias*. We can do this using the keyword **AS**.

```
SELECT
    name,
    CONCAT('(', lat, ', ', lon, ')') AS coords
FROM airports
LIMIT 0, 6;
```

name	coords
Lansdowne Airport	(41.1304722, -80.6195833)
Moton Field Municipal Airport	(32.4605722, -85.6800278)
Schaumburg Regional	(41.9893408, -88.1012428)
Randall Airport	(41.4319120, -74.3915611)
Jekyll Island Airport	(31.0744722, -81.4277778)
Elizabethton Municipal Airport	(36.3712222, -82.1734167)

We can also use **AS** to refer to a column in the table by a different name in the result set.

```
SELECT
    name AS airport_name,
    CONCAT('(', lat, ', ', lon, ')') AS coords
FROM airports
LIMIT 0, 6;
```

airport_name	coords
Lansdowne Airport	(41.1304722, -80.6195833)
Moton Field Municipal Airport	(32.4605722, -85.6800278)
Schaumburg Regional	(41.9893408, -88.1012428)
Randall Airport	(41.4319120, -74.3915611)
Jekyll Island Airport	(31.0744722, -81.4277778)
Elizabethton Municipal Airport	(36.3712222, -82.1734167)

This brings an important distinction to the fore: In SQL, it is crucial to distinguish between

clauses that operate *on the rows of the original table* versus those that operate *on the rows of the result set*. Here, `name`, `lat`, and `lon` are columns in the original table—they are written to the disk on the SQL server. On the other hand, `airport_name` and `coords` exist only in the result set—which is passed from the server to the client and is not written to the disk.

The preceding examples show the SQL equivalents of the `dplyr` commands `select()`, `mutate()`, and `rename()`.

15.4.2 WHERE

The `WHERE` clause is analogous to the `filter()` command in `dplyr`—it allows you to restrict the set of rows that are retrieved to only those rows that match a certain condition. Thus, while there are several million rows in the `flights` table in each year—each corresponding to a single flight—there were only a few dozen flights that left *Bradley International Airport* on June 26th, 2013.

SELECT

```
year, month, day, origin, dest,
flight, carrier
FROM flights
WHERE year = 2013 AND month = 6 AND day = 26
AND origin = 'BDL'
LIMIT 0, 6;
```

year	month	day	origin	dest	flight	carrier
2013	6	26	BDL	EWR	4714	EV
2013	6	26	BDL	MIA	2015	AA
2013	6	26	BDL	DTW	1644	DL
2013	6	26	BDL	BWI	2584	WN
2013	6	26	BDL	ATL	1065	DL
2013	6	26	BDL	DCA	1077	US

It would be convenient to search for flights in a date range. Unfortunately, there is no date field in this table—but rather separate columns for the `year`, `month`, and `day`. Nevertheless, we can tell SQL to interpret these columns as a date, using the `STR_TO_DATE()` function.⁴ Unlike in `R` code, function names in SQL code are customarily capitalized.

Pro Tip 37. *Dates and times can be challenging to wrangle. To learn more about these date tokens, see the MySQL documentation for `STR_TO_DATE()`.*

SELECT

```
STR_TO_DATE(CONCAT(year, '-', month, '-', day), '%Y-%m-%d') AS theDate,
origin,
flight, carrier
FROM flights
WHERE year = 2013 AND month = 6 AND day = 26
AND origin = 'BDL'
LIMIT 0, 6;
```

⁴The analogous function in PostgreSQL is called `TO_DATE()`. To do this, we first need to collect these columns as a string, and then tell SQL how to parse that string into a date.

theDate	origin	flight	carrier
2013-06-26	BDL	4714	EV
2013-06-26	BDL	2015	AA
2013-06-26	BDL	1644	DL
2013-06-26	BDL	2584	WN
2013-06-26	BDL	1065	DL
2013-06-26	BDL	1077	US

Note that here we have used a `WHERE` clause on columns that are not present in the result set. We can do this because `WHERE` operates only on the rows of the original table. Conversely, if we were to try and use a `WHERE` clause on `theDate`, it would not work, because (as the error suggests), `theDate` is not the name of a column in the `flights` table.

```
SELECT
  STR_TO_DATE(CONCAT(year, ' - ', month, ' - ', day), '%Y-%m-%d') AS theDate,
  origin, flight, carrier
FROM flights
WHERE theDate = '2013-06-26'
  AND origin = 'BDL'
LIMIT 0, 6;
```

ERROR 1054 (42S22) at line 1: Unknown column 'theDate' in 'where clause'

A workaround is to copy and paste the definition of `theDate` into the `WHERE` clause, since `WHERE` can operate on functions of columns in the original table (results not shown).

```
SELECT
  STR_TO_DATE(CONCAT(year, ' - ', month, ' - ', day), '%Y-%m-%d') AS theDate,
  origin, flight, carrier
FROM flights
WHERE STR_TO_DATE(CONCAT(year, ' - ', month, ' - ', day), '%Y-%m-%d') =
  '2013-06-26'
  AND origin = 'BDL'
LIMIT 0, 6;
```

This query will work, but here we have stumbled onto another wrinkle that exposes subtleties in how SQL executes queries. The previous query was able to make use of indices defined on the `year`, `month`, and `day` columns. However, the latter query is not able to make use of these indices because it is trying to filter on functions of a combination of those columns. This makes the latter query very slow. We will return to a fuller discussion of indices in Section 16.1.

Finally, we can use the `BETWEEN` syntax to filter through a range of dates. The `DISTINCT` keyword limits the result set to one row per unique value of `theDate`.

```
SELECT
  DISTINCT STR_TO_DATE(CONCAT(year, ' - ', month, ' - ', day), '%Y-%m-%d')
    AS theDate
FROM flights
WHERE year = 2013 AND month = 6 AND day BETWEEN 26 and 30
  AND origin = 'BDL'
LIMIT 0, 6;
```

theDate

2013-06-26
2013-06-27
2013-06-28
2013-06-29
2013-06-30

Similarly, we can use the `IN` syntax to search for items in a specified list. Note that flights on the 27th, 28th, and 29th of June are retrieved in the query using `BETWEEN` but not in the query using `IN`.

```
SELECT
  DISTINCT STR_TO_DATE(CONCAT(year, '-', month, '-', day), '%Y-%m-%d')
    AS theDate
FROM flights
WHERE year = 2013 AND month = 6 AND day IN (26, 30)
    AND origin = 'BDL'
LIMIT 0, 6;
```

theDate

2013-06-26
2013-06-30

SQL also supports `OR` clauses in addition to `AND` clauses, but one must always be careful with parentheses when using `OR`. Note the difference in the numbers of rows returned by the following two queries (557,874 vs. 2,542). The `COUNT` function simply counts the number of rows. The criteria in the `WHERE` clause are not evaluated left to right, but rather the `AND`s are evaluated first. This means that in the first query below, all flights on the 26th day of any month, regardless of year or month, would be returned.

```
/* returns 557,874 records */
SELECT
  COUNT(*) AS N
FROM flights
WHERE year = 2013 AND month = 6 OR day = 26
    AND origin = 'BDL';
```

```
/* returns 2,542 records */
SELECT
  COUNT(*) AS N
FROM flights
WHERE year = 2013 AND (month = 6 OR day = 26)
    AND origin = 'BDL';
```

15.4.3 GROUP BY

The `GROUP BY` clause allows one to *aggregate* multiple rows according to some criteria. The challenge when using `GROUP BY` is specifying *how* multiple rows of data should be reduced into a single value. Aggregate functions (e.g., `COUNT()`, `SUM()`, `MAX()`, and `AVG()`) are necessary.

We know that there were 65 flights that left Bradley Airport on June 26th, 2013, but how

many belonged to each airline carrier? To get this information we need to aggregate the individual flights, based on who the carrier was.

```
SELECT
    carrier,
    COUNT(*) AS numFlights,
    SUM(1) AS numFlightsAlso
FROM flights
WHERE year = 2013 AND month = 6 AND day = 26
    AND origin = 'BDL'
GROUP BY carrier;
```

carrier	numFlights	numFlightsAlso
9E	5	5
AA	4	4
B6	5	5
DL	11	11
EV	5	5
MQ	5	5
UA	1	1
US	7	7
WN	19	19
YV	3	3

For each of these airlines, which flight left the earliest in the morning?

```
SELECT
    carrier,
    COUNT(*) AS numFlights,
    MIN(dep_time)
FROM flights
WHERE year = 2013 AND month = 6 AND day = 26
    AND origin = 'BDL'
GROUP BY carrier;
```

carrier	numFlights	MIN(dep_time)
9E	5	0
AA	4	559
B6	5	719
DL	11	559
EV	5	555
MQ	5	0
UA	1	0
US	7	618
WN	19	601
YV	3	0

This is a bit tricky to figure out because the `dep_time` variable is stored as an integer, but would be better represented as a `time` data type. If it is a three-digit integer, then the first digit is the hour, but if it is a four-digit integer, then the first two digits are the hour. In either case, the last two digits are the minutes, and there are no seconds recorded. The

`MAKETIME()` function combined with the `IF(condition, value if true, value if false)` statement can help us with this.

```
SELECT
    carrier,
    COUNT(*) AS numFlights,
    MAKETIME(
        IF(LENGTH(MIN(dep_time)) = 3,
            LEFT(MIN(dep_time), 1),
            LEFT(MIN(dep_time), 2)
        ),
        RIGHT(MIN(dep_time), 2),
        0
    ) AS firstDepartureTime
FROM flights
WHERE year = 2013 AND month = 6 AND day = 26
    AND origin = 'BDL'
GROUP BY carrier
LIMIT 0, 6;
```

carrier	numFlights	firstDepartureTime
9E	5	00:00:00
AA	4	05:59:00
B6	5	07:19:00
DL	11	05:59:00
EV	5	05:55:00
MQ	5	00:00:00

We can also group by more than one column, but need to be careful to specify that we apply an aggregate function to each column that we are *not* grouping by. In this case, every time we access `dep_time`, we apply the `MIN()` function, since there may be many different values of `dep_time` associated with each unique combination of `carrier` and `dest`. Applying the `MIN()` function returns the smallest such value unambiguously.

```
SELECT
    carrier, dest,
    COUNT(*) AS numFlights,
    MAKETIME(
        IF(LENGTH(MIN(dep_time)) = 3,
            LEFT(MIN(dep_time), 1),
            LEFT(MIN(dep_time), 2)
        ),
        RIGHT(MIN(dep_time), 2),
        0
    ) AS firstDepartureTime
FROM flights
WHERE year = 2013 AND month = 6 AND day = 26
    AND origin = 'BDL'
GROUP BY carrier, dest
LIMIT 0, 6;
```

carrier	dest	numFlights	firstDepartureTime
9E	CVG	2	00:00:00
9E	DTW	1	18:20:00
9E	MSP	1	11:25:00
9E	RDU	1	09:38:00
AA	DFW	3	07:04:00
AA	MIA	1	05:59:00

15.4.4 ORDER BY

The use of aggregate function allows us to answer some very basic exploratory questions. Combining this with an `ORDER BY` clause will bring the most interesting results to the top. For example, which destinations are most common from Bradley in 2013?

```
SELECT
    dest, SUM(1) AS numFlights
FROM flights
WHERE year = 2013
    AND origin = 'BDL'
GROUP BY dest
ORDER BY numFlights DESC
LIMIT 0, 6;
```

dest	numFlights
ORD	2657
BWI	2613
ATL	2277
CLT	1842
MCO	1789
DTW	1523

Pro Tip 38. Note that since the `ORDER BY` clause cannot be executed until all of the data are retrieved, it operates on the result set, and not the rows of the original data. Thus, derived columns can be referenced in the `ORDER BY` clause.

Which of those destinations had the lowest average arrival delay time?

```
SELECT
    dest, SUM(1) AS numFlights,
    AVG(arr_delay) AS avg_arr_delay
FROM flights
WHERE year = 2013
    AND origin = 'BDL'
GROUP BY dest
ORDER BY avg_arr_delay ASC
LIMIT 0, 6;
```

dest	numFlights	avg_arr_delay
CLE	57	-13.07
LAX	127	-10.31
CVG	708	-7.37
MSP	981	-3.66
MIA	404	-3.27
DCA	204	-2.90

Cleveland Hopkins International Airport (CLE) has the smallest average arrival delay time.

15.4.5 HAVING

Although flights to Cleveland had the lowest average arrival delay—more than 13 minutes ahead of schedule—there were only 57 flights that went to from Bradley to Cleveland in all of 2013. It probably makes more sense to consider only those destinations that had, say, at least two flights per day. We can filter our result set using a `HAVING` clause.

```
SELECT
  dest, SUM(1) AS numFlights,
  AVG(arr_delay) AS avg_arr_delay
FROM flights
WHERE year = 2013
  AND origin = 'BDL'
GROUP BY dest
HAVING numFlights > 365 * 2
ORDER BY avg_arr_delay ASC
LIMIT 0, 6;
```

dest	numFlights	avg_arr_delay
MSP	981	-3.664
DTW	1523	-2.148
CLT	1842	-0.120
FLL	1011	0.277
DFW	1062	0.750
ATL	2277	4.470

We can see now that among the airports that are common destinations from Bradley, Minneapolis-St. Paul has the lowest average arrival delay time, at nearly 4 minutes ahead of schedule, on average.

Note that MySQL and SQLite support the use of derived column aliases in `HAVING` clauses, but PostgreSQL does not.

It is important to understand that the `HAVING` clause operates on the result set. While `WHERE` and `HAVING` are similar in spirit and syntax (and indeed, in `dplyr` they are both masked by the `filter()` function), they are different, because `WHERE` operates on the original data in the table and `HAVING` operates on the result set. Moving the `HAVING` condition to the `WHERE` clause will not work.

```
SELECT
  dest, SUM(1) AS numFlights,
  AVG(arr_delay) AS avg_arr_delay
FROM flights
```

```
WHERE year = 2013
  AND origin = 'BDL'
  AND numFlights > 365 * 2
GROUP BY dest
ORDER BY avg_arr_delay ASC
LIMIT 0, 6;
```

ERROR 1054 (42S22) at line 1: Unknown column 'numFlights' in 'where clause'

On the other hand, moving the **WHERE** conditions to the **HAVING** clause will work, but could result in a major loss of efficiency. The following query will return the same result as the one we considered previously.

```
SELECT
  origin, dest, SUM(1) AS numFlights,
  AVG(arr_delay) AS avg_arr_delay
FROM flights
WHERE year = 2013
GROUP BY origin, dest
HAVING numFlights > 365 * 2
  AND origin = 'BDL'
ORDER BY avg_arr_delay ASC
LIMIT 0, 6;
```

Moving the `origin = 'BDL'` condition to the **HAVING** clause means that *all* airport destinations had to be considered. With this condition in the **WHERE** clause, the server can quickly identify only those flights that left Bradley, perform the aggregation, and then filter this relatively small result set for those entries with a sufficient number of flights. Conversely, with this condition in the **HAVING** clause, the server is forced to consider *all* 3 million flights from 2013, perform the aggregation for all pairs of airports, and then filter this much larger result set for those entries with a sufficient number of flights from Bradley. The filtering of the result set is not importantly slower, but the aggregation over 3 million rows as opposed to a few thousand is.

Pro Tip 39. To maximize query efficiency, put conditions in a **WHERE** clause as opposed to a **HAVING** clause whenever possible.

15.4.6 LIMIT

A **LIMIT** clause simply allows you to truncate the output to a specified number of rows. This achieves an effect analogous to the **R** commands `head()` or `slice()`.

```
SELECT
  dest, SUM(1) AS numFlights,
  AVG(arr_delay) AS avg_arr_delay
FROM flights
WHERE year = 2013
  AND origin = 'BDL'
GROUP BY dest
HAVING numFlights > 365*2
ORDER BY avg_arr_delay ASC
LIMIT 0, 6;
```

dest	numFlights	avg_arr_delay
MSP	981	-3.664
DTW	1523	-2.148
CLT	1842	-0.120
FLL	1011	0.277
DFW	1062	0.750
ATL	2277	4.470

Note, however, that it is also possible to retrieve rows not at the beginning. The first number in the `LIMIT` clause indicates the number of rows to skip, and the latter indicates the number of rows to retrieve. Thus, this query will return the 4th–7th airports in the previous list.

```
SELECT
  dest, SUM(1) AS numFlights,
  AVG(arr_delay) AS avg_arr_delay
FROM flights
WHERE year = 2013
  AND origin = 'BDL'
GROUP BY dest
HAVING numFlights > 365*2
ORDER BY avg_arr_delay ASC
LIMIT 3,4;
```

dest	numFlights	avg_arr_delay
FLL	1011	0.277
DFW	1062	0.750
ATL	2277	4.470
BWI	2613	5.032

15.4.7 JOIN

In [Chapter 5](#), we presented several `dplyr` join operators: `inner_join()` and `left_join()`. Other functions (e.g., `semi_join()`) are also available. As you might expect, these operations are fundamental to SQL—and moreover, the success of the RDBMS paradigm is predicated on the ability to efficiently join tables together. Recall that SQL is a *relational* database management system—the relations between the tables allow you to write queries that efficiently tie together information from multiple sources. The syntax for performing these operations in SQL requires the `JOIN` keyword.

In general, there are four pieces of information that you need to specify in order to join two tables:

- The name of the first table that you want to join
- (optional) The *type* of join that you want to use
- The name of the second table that you want to join
- The *condition(s)* under which you want the records in the first table to match the records in the second table

There are many possible permutations of how two tables can be joined, and in many cases, a single query may involve several or even dozens of tables. In practice, the `JOIN` syntax varies among SQL implementations. In MySQL, `OUTER JOINs` are not available, but the following join types are:

- **JOIN**: includes all of the rows that are present in *both* tables and match.
- **LEFT JOIN**: includes all of the rows that are present in the first table. Rows in the first table that have no match in the second are filled with **NULLS**.
- **RIGHT JOIN**: include all of the rows that are present in the second table. This is the opposite of a **LEFT JOIN**.
- **CROSS JOIN**: the Cartesian product of the two tables. Thus, all possible combinations of rows matching the joining condition are returned.

Recall that in the `flights` table, the `origin` and `destination` of each flight are recorded.

```
SELECT
    origin, dest,
    flight, carrier
FROM flights
WHERE year = 2013 AND month = 6 AND day = 26
    AND origin = 'BDL'
LIMIT 0, 6;
```

origin	dest	flight	carrier
BDL	EWR	4714	EV
BDL	MIA	2015	AA
BDL	DTW	1644	DL
BDL	BWI	2584	WN
BDL	ATL	1065	DL
BDL	DCA	1077	US

Note that the `flights` table contains only the three-character FAA airport codes for both airports—not the full name of the airport. These cryptic abbreviations are not easily understood by humans. Which airport is `EWR`? Wouldn't it be more convenient to have the airport name in the table? It would be more convenient, but it would also be significantly less efficient from a storage and retrieval point of view, as well as more problematic from a *database integrity* point of view. The solution is to store information *about airports* in the `airports` table, along with these cryptic codes—which we will now call *keys*—and to only store these keys in the `flights` table—which is about *flights*, not airports. However, we can use these keys to join the two tables together in our query. In this manner, we can *have our cake and eat it too*: The data are stored in separate tables for efficiency, but we can still have the full names in the result set if we choose. Note how once again, the distinction between the rows of the original table and the result set is critical.

To write our query, we simply have to specify the table we want to join onto `flights` (e.g., `airports`) and the condition by which we want to match rows in `flights` with rows in `airports`. In this case, we want the airport code listed in `flights.dest` to be matched to the airport code in `airports.faa`. We need to specify that we want to see the `name` column from the `airports` table in the result set (see Table 15.3).

```
SELECT
    origin, dest,
    airports.name AS dest_name,
    flight, carrier
FROM flights
JOIN airports ON flights.dest = airports.faa
WHERE year = 2013 AND month = 6 AND day = 26
```

Table 15.3: Using JOIN to retrieve airport names.

origin	dest	dest_name	flight	carrier
BDL	EWR	Newark Liberty Intl	4714	EV
BDL	MIA	Miami Intl	2015	AA
BDL	DTW	Detroit Metro Wayne Co	1644	DL
BDL	BWI	Baltimore Washington Intl	2584	WN
BDL	ATL	Hartsfield Jackson Atlanta Intl	1065	DL
BDL	DCA	Ronald Reagan Washington Natl	1077	US

Table 15.4: Using JOIN with table aliases.

origin	dest	dest_name	flight	carrier
BDL	EWR	Newark Liberty Intl	4714	EV
BDL	MIA	Miami Intl	2015	AA
BDL	DTW	Detroit Metro Wayne Co	1644	DL
BDL	BWI	Baltimore Washington Intl	2584	WN
BDL	ATL	Hartsfield Jackson Atlanta Intl	1065	DL
BDL	DCA	Ronald Reagan Washington Natl	1077	US

```
AND origin = 'BDL'
LIMIT 0, 6;
```

This is much easier to read for humans. One quick improvement to the readability of this query is to use *table aliases*. This will save us some typing now, but a considerable amount later on. A table alias is often just a single letter after the reserved word `AS` in the specification of each table in the `FROM` and `JOIN` clauses. Note that these aliases can be referenced anywhere else in the query (see [Table 15.4](#)).

```
SELECT
    origin, dest,
    a.name AS dest_name,
    flight, carrier
FROM flights AS o
JOIN airports AS a ON o.dest = a.faa
WHERE year = 2013 AND month = 6 AND day = 26
    AND origin = 'BDL'
LIMIT 0, 6;
```

In the same manner, there are cryptic codes in `flights` for the airline carriers. The full name of each carrier is stored in the `carriers` table, since that is the place where information about carriers are stored. We can join this table to our result set to retrieve the name of each carrier (see [Table 15.5](#)).

```
SELECT
    dest, a.name AS dest_name,
    o.carrier, c.name AS carrier_name
FROM flights AS o
JOIN airports AS a ON o.dest = a.faa
JOIN carriers AS c ON o.carrier = c.carrier
```

Table 15.5: Using JOIN with multiple tables.

dest	dest_name	carrier	carrier_name
EWR	Newark Liberty Intl	EV	ExpressJet Airlines Inc.
MIA	Miami Intl	AA	American Airlines Inc.
DTW	Detroit Metro Wayne Co	DL	Delta Air Lines Inc.
BWI	Baltimore Washington Intl	WN	Southwest Airlines Co.
ATL	Hartsfield Jackson Atlanta Intl	DL	Delta Air Lines Inc.
DCA	Ronald Reagan Washington Natl	US	US Airways Inc.

Table 15.6: Using JOIN on the same table more than once.

flight	orig_name	dest_name	carrier_name
4714	Bradley Intl	Newark Liberty Intl	ExpressJet Airlines Inc.
2015	Bradley Intl	Miami Intl	American Airlines Inc.
1644	Bradley Intl	Detroit Metro Wayne Co	Delta Air Lines Inc.
2584	Bradley Intl	Baltimore Washington Intl	Southwest Airlines Co.
1065	Bradley Intl	Hartsfield Jackson Atlanta Intl	Delta Air Lines Inc.
1077	Bradley Intl	Ronald Reagan Washington Natl	US Airways Inc.

```
WHERE year = 2013 AND month = 6 AND day = 26
      AND origin = 'BDL'
LIMIT 0, 6;
```

Finally, to retrieve the name of the originating airport, we can join onto the same table more than once. Here the table aliases are necessary.

```
SELECT
    flight,
    a2.name AS orig_name,
    a1.name AS dest_name,
    c.name AS carrier_name
FROM flights AS o
JOIN airports AS a1 ON o.dest = a1.faa
JOIN airports AS a2 ON o.origin = a2.faa
JOIN carriers AS c ON o.carrier = c.carrier
WHERE year = 2013 AND month = 6 AND day = 26
      AND origin = 'BDL'
LIMIT 0, 6;
```

Table 15.6 displays the results. Now it is perfectly clear that *ExpressJet* flight 4714 flew from Bradley International airport to *Newark Liberty International airport* on June 26th, 2013. However, in order to put this together, we had to join four tables. Wouldn't it be easier to store these data in a single table that looks like the result set? For a variety of reasons, the answer is no.

First, there are very literal storage considerations. The `airports.name` field has room for 255 characters.

```
DESCRIBE airports;
```

Field	Type	Null	Key	Default	Extra
faa	varchar(3)	NO	PRI		
name	varchar(255)	YES		NA	
lat	decimal(10,7)	YES		NA	
lon	decimal(10,7)	YES		NA	
alt	int(11)	YES		NA	
tz	smallint(4)	YES		NA	
dst	char(1)	YES		NA	
city	varchar(255)	YES		NA	
country	varchar(255)	YES		NA	

This takes up considerably more space on disk than the four-character abbreviation stored in `airports.faa`. For small data sets, this overhead might not matter, but the `flights` table contains 169 million rows, so replacing the four-character `origin` field with a 255-character field would result in a noticeable difference in space on disk. (Plus, we'd have to do this twice, since the same would apply to `dest`.) We'd suffer a similar penalty for including the full name of each carrier in the `flights` table. Other things being equal, tables that take up less room on disk are faster to search.

Second, it would be logically inefficient to store the full name of each airport in the `flights` table. The name of the airport doesn't change for each flight. It doesn't make sense to store the full name of the airport any more than it would make sense to store the full name of the month, instead of just the integer corresponding to each month.

Third, what if the name of the airport *did* change? For example, in 1998 the airport with code DCA was renamed from Washington National to *Ronald Reagan Washington National*. It is still the same airport in the same location, and it still has code DCA—only the full name has changed. With separate tables, we only need to update a single field: the `name` column in the `airports` table for the DCA row. Had we stored the full name in the `flights` table, we would have to make millions of substitutions, and would risk ending up in a situation in which both "Washington National" and "Reagan National" were present in the table.

When designing a database, how do you know whether to create a separate table for pieces of information? The short answer is that if you are designing a persistent, scalable database for speed and efficiency, then every *entity* should have its own table. In practice, very often it is not worth the time and effort to set this up if we are simply doing some quick analysis. But for permanent systems—like a database backend to a website—proper curation is necessary. The notions of normal forms, and specifically *third normal form* (3NF), provide guidance for how to properly design a database. A full discussion of this is beyond the scope of this book, but the basic idea is to "keep like with like."

Pro Tip 40. *If you are designing a database that will be used for a long time or by a lot of people, take the extra time to design it well.*

15.4.7.1 LEFT JOIN

Recall that in a `JOIN`—also known as an *inner* or *natural* or *regular* `JOIN`—all possible matching pairs of rows from the two tables are included. Thus, if the first table has n rows and the second table has m , as many as nm rows could be returned. However, in

the `airports` table each row has a unique airport code, and thus every row in `flights` will match the destination field to *at most* one row in the `airports` table. What happens if no such entry is present in `airports`? That is, what happens if there is a destination airport in `flights` that has no corresponding entry in `airports`? If you are using a `JOIN`, then the offending row in `flights` is simply not returned. On the other hand, if you are using a `LEFT JOIN`, then every row in the first table is returned, and the corresponding entries from the second table are left blank. In this example, no airport names were found for several airports.

SELECT

```
year, month, day, origin, dest,
a.name AS dest_name,
flight, carrier
FROM flights AS o
LEFT JOIN airports AS a ON o.dest = a.faa
WHERE year = 2013 AND month = 6 AND day = 26
    AND a.name IS NULL
LIMIT 0, 6;
```

year	month	day	origin	dest	dest_name	flight	carrier
2013	6	26	BOS	SJU	NA	261	B6
2013	6	26	JFK	SJU	NA	1203	B6
2013	6	26	JFK	PSE	NA	745	B6
2013	6	26	JFK	SJU	NA	1503	B6
2013	6	26	JFK	BQN	NA	839	B6
2013	6	26	JFK	BQN	NA	939	B6

The output indicates that the airports are all in *Puerto Rico*: SJU is in *San Juan*, BQN is in *Aguadilla*, and PSE is in *Ponce*.

The result set from a `LEFT JOIN` is always a superset of the result set from the same query with a regular `JOIN`. A `RIGHT JOIN` is simply the opposite of a `LEFT JOIN`—that is, the tables have simply been specified in the opposite order. This can be useful in certain cases, especially when you are joining more than two tables.

15.4.8 UNION

Two separate queries can be combined using a `UNION` clause.

```
(SELECT
    year, month, day, origin, dest,
    flight, carrier
FROM flights
WHERE year = 2013 AND month = 6 AND day = 26
    AND origin = 'BDL' AND dest = 'MSP')
UNION
(SELECT
    year, month, day, origin, dest,
    flight, carrier
FROM flights
WHERE year = 2013 AND month = 6 AND day = 26)
```

Table 15.7: First set of six airports outside the lower 48 states.

faa	name	tz	city
369	Atmautluak Airport	-9	Atmautluak
6K8	Tok Junction Airport	-9	Tok
ABL	Ambler Airport	-9	Ambler
ADK	Adak Airport	-9	Adak Island
ADQ	Kodiak	-9	Kodiak
AET	Allakaket Airport	-9	Allakaket

```
AND origin = 'JFK' AND dest = 'ORD')
LIMIT 0,10;
```

year	month	day	origin	dest	flight	carrier
2013	6	26	BDL	MSP	797	DL
2013	6	26	BDL	MSP	3338	9E
2013	6	26	BDL	MSP	1226	DL
2013	6	26	JFK	ORD	905	B6
2013	6	26	JFK	ORD	1105	B6
2013	6	26	JFK	ORD	3523	9E
2013	6	26	JFK	ORD	1711	AA
2013	6	26	JFK	ORD	105	B6
2013	6	26	JFK	ORD	3521	9E
2013	6	26	JFK	ORD	3525	9E

This is analogous to the `dplyr` operation `bind_rows()`.

15.4.9 Subqueries

It is also possible to use a result set as if it were a table. That is, you can write one query to generate a result set, and then use that result set in a larger query as if it were a table, or even just a list of values. The initial query is called a *subquery*.

For example, Bradley is listed as an “international” airport, but with the exception of trips to *Montreal* and *Toronto* and occasional flights to *Mexico* and *Europe*, it is more of a regional airport. Does it have any flights coming from or going to *Alaska* and *Hawaii*? We can retrieve the list of airports outside the lower 48 states by filtering the airports table using the time zone `tz` column (see Table 15.7 for the first six).

```
SELECT faa, name, tz, city
FROM airports AS a
WHERE tz < -8
LIMIT 0, 6;
```

Now, let’s use the airport codes generated by that query as a list to filter the flights leaving from Bradley in 2013. Note the subquery in parentheses in the query below.

```
SELECT
  dest, a.name AS dest_name,
  SUM(1) AS N, COUNT(distinct carrier) AS numCarriers
FROM flights AS o
  WHERE dest IN (SELECT faa
    FROM airports AS a
    WHERE tz < -8)
```

```
LEFT JOIN airports AS a ON o.dest = a.faa
WHERE year = 2013
  AND origin = 'BDL'
  AND dest IN
    (SELECT faa
     FROM airports
      WHERE tz < -8)
GROUP BY dest;
```

No results are returned. As it turns out, Bradley did not have any outgoing flights to Alaska or Hawaii. However, it did have some flights to and from airports in the Pacific Time Zone.

```
SELECT
  dest, a.name AS dest_name,
  SUM(1) AS N, COUNT(distinct carrier) AS numCarriers
FROM flights AS o
LEFT JOIN airports AS a ON o.origin = a.faa
WHERE year = 2013
  AND dest = 'BDL'
  AND origin IN
    (SELECT faa
     FROM airports
      WHERE tz < -7)
GROUP BY origin;
```

dest	dest_name	N	numCarriers
BDL	Mc Carran Intl	262	1
BDL	Los Angeles Intl	127	1

We could also employ a similar subquery to create an ephemeral table (results not shown).

```
SELECT
  dest, a.name AS dest_name,
  SUM(1) AS N, COUNT(distinct carrier) AS numCarriers
FROM flights AS o
JOIN (SELECT *
         FROM airports
          WHERE tz < -7) AS a
  ON o.origin = a.faa
WHERE year = 2013 AND dest = 'BDL'
GROUP BY origin;
```

Of course, we could have achieved the same result with a **JOIN** and **WHERE** (results not shown).

```
SELECT
  dest, a.name AS dest_name,
  SUM(1) AS N, COUNT(distinct carrier) AS numCarriers
FROM flights AS o
LEFT JOIN airports AS a ON o.origin = a.faa
WHERE year = 2013
  AND dest = 'BDL'
  AND tz < -7
GROUP BY origin;
```

It is important to note that while subqueries are often convenient, they cannot make use of indices. In most cases it is preferable to write the query using joins as opposed to subqueries.

15.5 Extended example: FiveThirtyEight flights

Over at FiveThirtyEight, Nate Silver wrote an article about airline delays using the same Bureau of Transportation Statistics data that we have in our database (see the link in the footnote⁵). We can use this article as an exercise in querying our airlines database.

The article makes a number of claims. We'll walk through some of these. First, the article states:

In 2014, the 6 million domestic flights the U.S. government tracked required an extra 80 million minutes to reach their destinations.

The majority of flights (54%) arrived ahead of schedule in 2014. (The 80 million minutes figure cited earlier is a net number. It consists of about 115 million minutes of delays minus 35 million minutes saved from early arrivals.)

Although there are a number of claims here, we can verify them with a single query. Here, we compute the total number of flights, the percentage of those that were on time and ahead of schedule, and the total number of minutes of delays.

```
SELECT
    SUM(1) AS numFlights,
    SUM(IF(arr_delay < 15, 1, 0)) / SUM(1) AS ontimePct,
    SUM(IF(arr_delay < 0, 1, 0)) / SUM(1) AS earlyPct,
    SUM(arr_delay) / 1e6 AS netMinLate,
    SUM(IF(arr_delay > 0, arr_delay, 0)) / 1e6 AS minLate,
    SUM(IF(arr_delay < 0, arr_delay, 0)) / 1e6 AS minEarly
FROM flights AS o
WHERE year = 2014
LIMIT 0, 6;
```

numFlights	ontimePct	earlyPct	netMinLate	minLate	minEarly
5819811	0.787	0.542	41.6	77.6	-36

We see the right number of flights (about 6 million), and the percentage of flights that

⁵<https://fivethirtyeight.com/features/fastest-airlines-fastest-airports/>

were early (about 54%) is also about right. The total number of minutes early (about 36 million) is also about right. However, the total number of minutes late is way off (about 78 million vs. 115 million), and as a consequence, so is the net number of minutes late (about 42 million vs. 80 million). In this case, you have to read the fine print. A description of the methodology used in this analysis contains some information about the *estimates*⁶ of the arrival delay for cancelled flights. The problem is that cancelled flights have an `arr_delay` value of 0, yet in the real-world experience of travelers, the practical delay is much longer. The FiveThirtyEight data scientists concocted an estimate of the actual delay experienced by travelers due to cancelled flights.

A quick-and-dirty answer is that cancelled flights are associated with a delay of four or five hours, on average. However, the calculation varies based on the particular circumstances of each flight.

Unfortunately, reproducing the estimates made by FiveThirtyEight is likely impossible, and certainly beyond the scope of what we can accomplish here. Since we only care about the aggregate number of minutes, we can amend our computation to add, say, 270 minutes of delay time for each cancelled flight.

```
SELECT
    SUM(1) AS numFlights,
    SUM(IF(arr_delay < 15, 1, 0)) / SUM(1) AS ontimePct,
    SUM(IF(arr_delay < 0, 1, 0)) / SUM(1) AS earlyPct,
    SUM(IF(cancelled = 1, 270, arr_delay)) / 1e6 AS netMinLate,
    SUM(
        IF(cancelled = 1, 270, IF(arr_delay > 0, arr_delay, 0))
    ) / 1e6 AS minLate,
    SUM(IF(arr_delay < 0, arr_delay, 0)) / 1e6 AS minEarly
FROM flights AS o
WHERE year = 2014
LIMIT 0, 6;
```

numFlights	ontimePct	earlyPct	netMinLate	minLate	minEarly
5819811	0.787	0.542	75.9	112	-36

This again puts us in the neighborhood of the estimates from the article. One has to read the fine print to properly vet these estimates. The problem is not that the estimates reported by Silver are inaccurate—on the contrary, they seem plausible and are certainly better than not correcting for cancelled flights at all. However, it is not immediately clear from reading the article (you have to read the separate methodology article) that these estimates—which account for roughly 25% of the total minutes late reported—are in fact estimates and not hard data.

Later in the article, Silver presents a figure that breaks down the percentage of flights that were on time, had a delay of 15 to 119 minutes, or were delayed longer than 2 hours. We can

⁶Somehow, the word “estimate” is not used to describe what is being calculated.

pull the data for this figure with the following query. Here, in order to plot these results, we need to actually bring them back into **R**. To do this, we will use the functionality provided by the **knitr** package (see [Section F.4.3](#) for more information about connecting to a MySQL server from within **R**). The results of this query will be saved to an **R** data frame called `res`.

```
SELECT o.carrier, c.name,
  SUM(1) AS numFlights,
  SUM(IF(arr_delay > 15 AND arr_delay <= 119, 1, 0)) AS shortDelay,
  SUM(
    IF(arr_delay >= 120 OR cancelled = 1 OR diverted = 1, 1, 0)
  ) AS longDelay
FROM
  flights AS o
LEFT JOIN
  carriers c ON o.carrier = c.carrier
WHERE year = 2014
GROUP BY carrier
ORDER BY shortDelay DESC
```

Reproducing the figure requires a little bit of work. We begin by pruning less informative labels from the carriers.

```
res <- res %>%
  as_tibble() %>%
  mutate(
    name = str_remove_all(name, "Air(lines|ways| Lines)"),
    name = str_remove_all(name, "(Inc\\.|Co\\.|Corporation)"),
    name = str_remove_all(name, "\\(.*\\)"),
    name = str_remove_all(name, " *$")
  )
res %>%
  pull(name)

[1] "Southwest"      "ExpressJet"       "SkyWest"        "Delta"
[5] "American"        "United"          "Envoy Air"      "US"
[9] "JetBlue"         "Frontier"        "Alaska"         "AirTran"
[13] "Virgin America" "Hawaiian"
```

Next, it is now clear that FiveThirtyEight has considered airline mergers and regional carriers that are not captured in our data. Specifically: “We classify all remaining *AirTran* flights as *Southwest* flights.” *Envoy Air* serves *American Airlines*. However, there is a bewildering network of alliances among the other regional carriers. Greatly complicating matters, *ExpressJet* and *SkyWest* serve multiple national carriers (primarily United, American, and Delta) under different flight numbers. FiveThirtyEight provides a footnote detailing how they have assigned flights carried by these regional carriers, but we have chosen to ignore that here and include ExpressJet and SkyWest as independent carriers. Thus, the data that we show in [Figure 15.1](#) does not match the figure from FiveThirtyEight.

```
carriers_2014 <- res %>%
  mutate(
    groupName = case_when(
      name %in% c("Envoy Air", "American Eagle") ~ "American",
      name == "AirTran" ~ "Southwest",
```

```

TRUE ~ name
)
) %>%
group_by(groupName) %>%
summarize(
  numFlights = sum(numFlights),
  wShortDelay = sum(shortDelay),
  wLongDelay = sum(longDelay)
) %>%
mutate(
  wShortDelayPct = wShortDelay / numFlights,
  wLongDelayPct = wLongDelay / numFlights,
  delayed = wShortDelayPct + wLongDelayPct,
  ontime = 1 - delayed
)
carriers_2014

# A tibble: 12 x 8
# ... with 2 more variables: delayed <dbl>, ontime <dbl>
  groupName numFlights wShortDelay wLongDelay wShortDelayPct wLongDelayPct
  <chr>      <dbl>       <dbl>       <dbl>        <dbl>        <dbl>
1 Alaska     160257      18366      2613        0.115       0.0163
2 American   930398      191071     53641       0.205       0.0577
3 Delta      800375      105194     19818       0.131       0.0248
4 ExpressJ~  686021      136207     59663       0.199       0.0870
5 Frontier   85474       18410      2959        0.215       0.0346
6 Hawaiian   74732       5098       514         0.0682      0.00688
7 JetBlue    249693      46618      12789       0.187       0.0512
8 SkyWest    613030      107192     33114       0.175       0.0540
9 Southwest  1254128     275155     44907       0.219       0.0358
10 United    493528      93721      20923       0.190       0.0424
11 US        414665      64505      12328       0.156       0.0297
12 Virgin A~ 57510       8356       1976        0.145       0.0344
# ... with 2 more variables: delayed <dbl>, ontime <dbl>

```

After tidying this data frame using the `pivot_longer()` function (see [Chapter 6](#)), we can draw the figure as a stacked bar chart.

```

carriers_tidy <- carriers_2014 %>%
  select(groupName, wShortDelayPct, wLongDelayPct, delayed) %>%
  pivot_longer(
    -c(groupName, delayed),
    names_to = "delay_type",
    values_to = "pct"
  )
delay_chart <- ggplot(
  data = carriers_tidy,
  aes(x = reorder(groupName, pct, max), y = pct)
) +
  geom_col(aes(fill = delay_type)) +
  scale_fill_manual(
    name = NULL,
    values = c("red", "gold"),

```

```

labels = c(
  "Flights Delayed 120+ Minutes\nCancelled or Diverted",
  "Flights Delayed 15-119 Minutes"
)
) +
scale_y_continuous(limits = c(0, 1)) +
coord_flip() +
labs(
  title = "Southwest's Delays Are Short; United's Are Long",
  subtitle = "As share of scheduled flights, 2014"
) +
ylab(NULL) +
xlab(NULL) +
ggthemes::theme_fivethirtyeight() +
theme(
  plot.title = element_text(hjust = 1),
  plot.subtitle = element_text(hjust = -0.2)
)
)

```

Getting the right text labels in the right places to mimic the display requires additional wrangling. We show our best effort in [Figure 15.1](#). In fact, by comparing the two figures, it becomes clear that many of the long delays suffered by United and American passengers occur on flights operated by ExpressJet and SkyWest.

```

delay_chart +
geom_text(
  data = filter(carriers_tidy, delay_type == "wShortDelayPct"),
  aes(label = paste0(round(pct * 100, 1), "%")),
  hjust = "right",
  size = 2
) +
geom_text(
  data = filter(carriers_tidy, delay_type == "wLongDelayPct"),
  aes(y = delayed - pct, label = paste0(round(pct * 100, 1), "%")),
  hjust = "left",
  nudge_y = 0.01,
  size = 2
)
)

```

The rest of the analysis is predicated on FiveThirtyEight's definition of *target time*, which is different from the scheduled time in the database. To compute it would take us far astray. In another graphic in the article, FiveThirtyEight reports the slowest and fastest airports among the 30 largest airports.

Using arrival delay time instead of the FiveThirtyEight-defined target time, we can produce a similar table by joining the results of two queries together.

```

SELECT
  dest,
  SUM(1) AS numFlights,
  AVG(arr_delay) AS avgArrivalDelay
FROM
  flights AS o

```

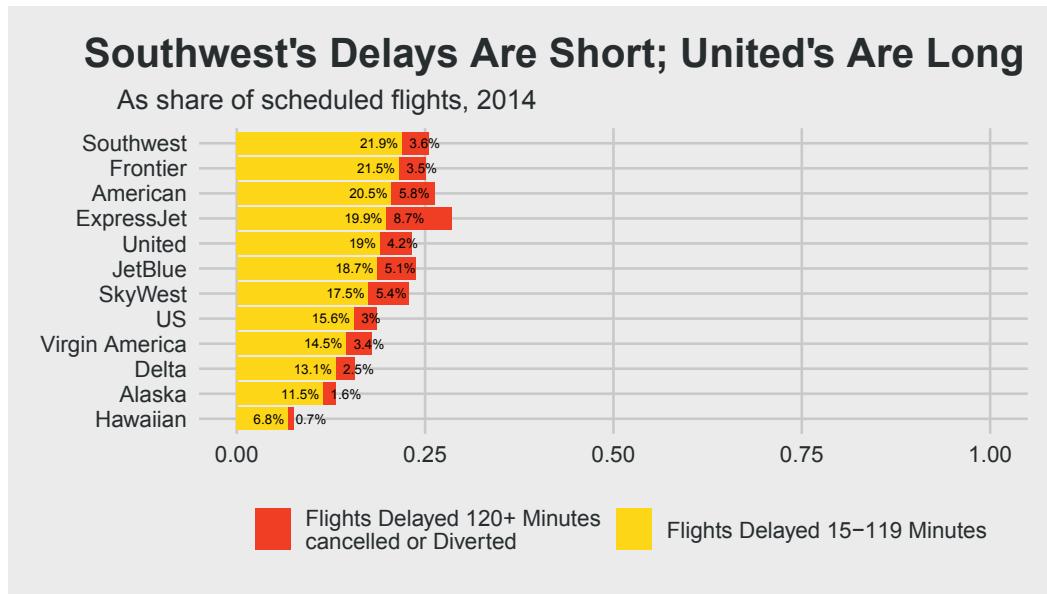


Figure 15.1: Recreation of the FiveThirtyEight plot on flight delays.

```
WHERE year = 2014
```

```
GROUP BY dest
```

```
ORDER BY numFlights DESC
```

```
LIMIT 0, 30
```

```
SELECT
```

```
origin,
```

```
SUM(1) AS numFlights,
```

```
AVG(arr_delay) AS avgDepartDelay
```

```
FROM
```

```
flights AS o
```

```
WHERE year = 2014
```

```
GROUP BY origin
```

```
ORDER BY numFlights DESC
```

```
LIMIT 0, 30
```

```
dests %>%
```

```
left_join(origins, by = c("dest" = "origin")) %>%
```

```
select(dest, avgDepartDelay, avgArrivalDelay) %>%
```

```
arrange(desc(avgDepartDelay)) %>%
```

```
as_tibble()
```

```
# A tibble: 30 x 3
```

	dest	avgDepartDelay	avgArrivalDelay
1	ORD	14.3	13.1
2	MDW	12.8	7.40
3	DEN	11.3	7.60
4	IAD	11.3	7.45

```

5 HOU          11.3        8.07
6 DFW          10.7        9.00
7 BWI          10.2        6.04
8 BNA          9.47       8.94
9 EWR          8.70       9.61
10 IAH         8.41       6.75
# ... with 20 more rows

```

Finally, FiveThirtyEight produces a simple table ranking the airlines by the amount of time added versus *typical*—another of their creations—and target time.

What we can do instead is compute a similar table for the average arrival delay time by carrier, *after controlling for the routes*. First, we compute the average arrival delay time for each route.

```

SELECT
  origin, dest,
  SUM(1) AS numFlights,
  AVG(arr_delay) AS avgDelay
FROM
  flights AS o
WHERE year = 2014
GROUP BY origin, dest

```

```
head(routes)
```

	origin	dest	numFlights	avgDelay
1	ABE	ATL	829	5.43
2	ABE	DTW	665	3.23
3	ABE	ORD	144	19.51
4	ABI	DFW	2832	10.70
5	ABQ	ATL	893	1.92
6	ABQ	BWI	559	6.60

Next, we perform the same calculation, but this time, we add `carrier` to the `GROUP BY` clause.

```

SELECT
  origin, dest,
  o.carrier, c.name,
  SUM(1) AS numFlights,
  AVG(arr_delay) AS avgDelay
FROM
  flights AS o
LEFT JOIN
  carriers c ON o.carrier = c.carrier
WHERE year = 2014
GROUP BY origin, dest, o.carrier

```

Next, we merge these two data sets, matching the routes traveled by each carrier with the route averages across all carriers.

```

routes_aug <- routes_carriers %>%
  left_join(routes, by = c("origin" = "origin", "dest" = "dest")) %>%

```

```
as_tibble()
head(routes_aug)
```

```
# A tibble: 6 x 8
  origin dest carrier name numFlights.x avgDelay.x numFlights.y avgDelay.y
  <chr>  <chr> <chr>   <chr>      <dbl>       <dbl>       <dbl>       <dbl>
1 ABE    ATL    DL     Delt~      186        1.67       829        5.43
2 ABE    ATL    EV     Expr~      643        6.52       829        5.43
3 ABE    DTW    EV     Expr~      665        3.23       665        3.23
4 ABE    ORD    EV     Expr~      144       19.5       144       19.5
5 ABI    DFW    EV     Expr~      219         7       2832       10.7
6 ABI    DFW    MQ     Envo~     2613       11.0       2832       10.7
```

Note that `routes_aug` contains both the average arrival delay time for each carrier on each route that it flies (`avgDelay.x`) as well as the average arrival delay time for each route across all carriers (`avgDelay.y`). We can then compute the difference between these times, and aggregate the weighted average for each carrier.

```
routes_aug %>%
  group_by(carrier) %>%
  # use str_remove_all() to remove parentheses
  summarize(
    carrier_name = str_remove_all(first(name), "\\\(.*\\\\)"),
    numRoutes = n(),
    numFlights = sum(numFlights.x),
    wAvgDelay = sum(
      numFlights.x * (avgDelay.x - avgDelay.y),
      na.rm = TRUE
    ) / sum(numFlights.x)
  ) %>%
  arrange(wAvgDelay)
```

```
# A tibble: 14 x 5
  carrier carrier_name          numRoutes numFlights wAvgDelay
  <chr>   <chr>              <int>      <dbl>       <dbl>
1 VX      Virgin America       72        57510      -2.69
2 FL      AirTran Airways Corporation 170        79495      -1.55
3 AS      Alaska Airlines Inc.  242       160257      -1.44
4 US      US Airways Inc.     378       414665      -1.31
5 DL      Delta Air Lines Inc. 900       800375      -1.01
6 UA      United Air Lines Inc. 621       493528      -0.982
7 MQ      Envoy Air           442       392701      -0.455
8 AA      American Airlines Inc. 390       537697      -0.0340
9 HA      Hawaiian Airlines Inc. 56        74732       0.272
10 OO     SkyWest Airlines Inc. 1250      613030      0.358
11 B6     JetBlue Airways      316       249693      0.767
12 EV     ExpressJet Airlines Inc. 1534      686021      0.845
13 WN     Southwest Airlines Co. 1284      1174633     1.13
14 F9     Frontier Airlines Inc. 326       85474       2.29
```

15.6 SQL vs. R

This chapter contains an introduction to the database querying language SQL. However, along the way we have highlighted the similarities and differences between the way certain things are done in **R** versus how they are done in SQL. While the rapid development of **dplyr** has brought fusion to the most common data management operations shared by both **R** and SQL, while at the same time shielding the user from concerns about where certain operations are being performed, it is important for a practicing data scientist to understand the relative strengths and weaknesses of each of their tools.

While the process of slicing and dicing data can generally be performed in either **R** or SQL, we have already seen tasks for which one is more appropriate (e.g., faster, simpler, or more logically structured) than the other. **R** is a statistical computing environment that is developed for the purpose of data analysis. If the data are small enough to be read into memory, then **R** puts a vast array of data analysis functions at your fingertips. However, if the data are large enough to be problematic in memory, then SQL provides a robust, parallelizable, and scalable solution for data storage and retrieval. The SQL query language, or the **dplyr** interface, enable one to efficiently perform basic data management operations on smaller pieces of the data. However, there is an upfront cost to creating a well-designed SQL database. Moreover, the analytic capabilities of SQL are very limited, offering only a few simple statistical functions (e.g., `AVG()`, `SD()`, etc.—although user-defined extensions are possible). Thus, while SQL is usually a more robust solution for *data management*, it is a poor substitute for **R** when it comes to *data analysis*.

15.7 Further resources

The documentation for MySQL, PostgreSQL, and SQLite are the authoritative sources for complete information on their respective syntaxes. We have also found Kline et al. (2008) to be a useful reference.

15.8 Exercises

Problem 1 (Easy): How many rows are available in the `Measurements` table of the Smith College Wideband Auditory Immittance database?

```
library(tidyverse)
library(mdsr)
library(RMySQL)
con <- dbConnect(
  MySQL(), host = "scidb.smith.edu",
  user = "waiuser", password = "smith_waiDB",
  dbname = "wai"
)
Measurements <- tbl(con, "Measurements")
```

Problem 2 (Easy): Identify what years of data are available in the `flights` table of the `airlines` database.

```
library(tidyverse)
library(mdsr)
library(RMySQL)
con <- dbConnect_scidb("airlines")
```

Problem 3 (Easy): Use the `dbConnect_scidb` function to connect to the `airlines` database to answer the following problem. How many domestic flights flew into Dallas-Fort Worth (DFW) on May 14, 2010?

Problem 4 (Easy): Wideband acoustic immittance (WAI) is an area of biomedical research that strives to develop WAI measurements as noninvasive auditory diagnostic tools. WAI measurements are reported in many related formats, including absorbance, admittance, impedance, power reflectance, and pressure reflectance. More information can be found about this public facing WAI database at <http://www.science.smith.edu/wai-database/home/about>.

```
library(RMySQL)
db <- dbConnect(
  MySQL(),
  user = "waiuser",
  password = "smith_waiDB",
  host = "scidb.smith.edu",
  dbname = "wai"
)
```

- a. How many female subjects are there in total across all studies?
- b. Find the average absorbance for participants for each study, ordered by highest to lowest value.
- c. Write a query to count all the measurements with a calculated absorbance of less than 0.

Problem 5 (Medium): Use the `dbConnect_scidb` function to connect to the `airlines` database to answer the following problem. Of all the destinations from Chicago O'Hare (ORD), which were the most common in 2010?

Problem 6 (Medium): Use the `dbConnect_scidb` function to connect to the `airlines` database to answer the following problem. Which airport had the highest average arrival delay time in 2010?

Problem 7 (Medium): Use the `dbConnect_scidb` function to connect to the `airlines` database to answer the following problem. How many domestic flights came into or flew out of Bradley Airport (BDL) in 2012?

Problem 8 (Medium): Use the `dbConnect_scidb` function to connect to the `airlines` database to answer the following problem. List the airline and flight number for all flights between LAX and JFK on September 26th, 1990.

Problem 9 (Medium): The following questions require use of the `Lahman` package and reference basic baseball terminology. (See https://en.wikipedia.org/wiki/Baseball_statistics for explanations of any acronyms.)

- a. List the names of all batters who have at least 300 home runs (HR) and 300 stolen bases (SB) in their careers and rank them by career batting average (H/AB).
- b. List the names of all pitchers who have at least 300 wins (W) and 3,000 strikeouts (SO) in their careers and rank them by career winning percentage ($W/(W + L)$).
- c. The attainment of either 500 home runs (HR) or 3,000 hits (H) in a career is considered to be among the greatest achievements to which a batter can aspire. These milestones are thought to guarantee induction into the Baseball Hall of Fame, and yet several players who have attained either milestone have not been inducted into the Hall of Fame. Identify them.

Problem 10 (Medium): Use the `dbConnect_scidb` function to connect to the `airlines` database to answer the following problem. Find all flights between `JFK` and `SFO` in 1994. How many were canceled? What percentage of the total number of flights were canceled?

Problem 11 (Hard): The following open-ended question may require more than one query and a thoughtful response. Based on data from 2012 only, and assuming that transportation to the airport is not an issue, would you rather fly out of `JFK`, `LaGuardia (LGA)`, or `Newark (EWR)`? Why or why not? Use the `dbConnect_scidb` function to connect to the `airlines` database.

15.9 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/sql-I.html#sqlI-online-exercises>

16

Database administration

In [Chapter 15](#), we learned how to write `SELECT` queries to retrieve data from an existing SQL server. Of course, these queries depend on that server being configured, and the proper data loaded into it. In this chapter, we provide the tools necessary to set up a new database and populate it. Furthermore, we present concepts that will help you construct efficient databases that enable faster query performance. While the treatment herein is not sufficient to make you a seasoned database administrator, it should be enough to allow you to start experimenting with SQL databases on your own.

As in [Chapter 15](#), the code that you see in this chapter illustrates exchanges between a MySQL server and a client. In places where **R** is involved, we will make that explicit. We assume that you are able to log in to a MySQL server. (See [Appendix F](#) for instructions on how to install, configure, and log in to an SQL server.)

16.1 Constructing efficient SQL databases

While it is often helpful to think about SQL tables as being analogous to `data.frames` in **R**, there are some important differences. In **R**, a `data.frame` is a `list` of vectors that have the same length. Each of those vectors has a specific data type (e.g., integers, character strings, etc.), but those data types can vary across the columns. The same is true of tables in SQL, but there are additional constraints that we can impose on SQL tables that can improve both the logical integrity of our data, as well as the performance we can achieve when searching it.

16.1.1 Creating new databases

Once you have logged into MySQL, you can see what databases are available to you by running the `SHOW DATABASES` command at the `mysql>` prompt.

```
SHOW DATABASES;
```

Database

```
information_schema  
airlines  
fec  
imdb  
lahman  
nyctaxi
```

In this case, the output indicates that the `airlines` database already exists. If it didn't, we could create it using the `CREATE DATABASE` command.

```
CREATE DATABASE airlines;
```

Since we will continue to work with the `airlines` database, we can save ourselves some typing by utilizing the `USE` command to make that connection explicit.

```
USE airlines;
```

Now that we are confined to the `airlines` database, there is no ambiguity in asking what tables are present.

```
SHOW TABLES;
```

Tables_in_airlines

airports
carriers
flights
planes

16.1.2 CREATE TABLE

Recall that in [Chapter 15](#) we used the `DESCRIBE` statement to display the definition of each table. This lists each field, its data type, whether there are keys or indices defined on it, and whether `NUL` values are allowed. For example, the `airports` table has the following definition.

```
DESCRIBE airports;
```

Field	Type	Null	Key	Default	Extra
faa	varchar(3)	NO	PRI		
name	varchar(255)	YES		NA	
lat	decimal(10,7)	YES		NA	
lon	decimal(10,7)	YES		NA	
alt	int(11)	YES		NA	
tz	smallint(4)	YES		NA	
dst	char(1)	YES		NA	
city	varchar(255)	YES		NA	
country	varchar(255)	YES		NA	

We can see from the output that the `faa`, `name`, `city`, and `country` fields are defined as `varchar` (or variable character) fields. These fields contain character strings, but the length of the strings allowed varies. We know that the `faa` code is restricted to three characters, and so we have codified that in the table definition. The `dst` field contains only a single character, indicating whether daylight saving time is observed at each airport. The `lat` and `lon` fields contain geographic coordinates, which can be three-digit numbers (i.e., the maximum value is 180) with up to seven decimal places. The `tz` field can be up to a four-digit integer, while the `alt` field is allowed eleven digits. In this case, `NUL` values are allowed—and are the default—in all of the fields except for `faa`, which is the primary key. **R** is translating the null character in SQL (`NULL`) to the null character in **R** (`NA`).

These definitions did not come out of thin air, nor were they automatically generated. In this case, we wrote them by hand, in the following `CREATE TABLE` statement:

```
SHOW CREATE TABLE airports;

CREATE TABLE `airports` (
  `faa` varchar(3) NOT NULL DEFAULT '',
  `name` varchar(255) DEFAULT NULL,
  `lat` decimal(10,7) DEFAULT NULL,
  `lon` decimal(10,7) DEFAULT NULL,
  `alt` int(11) DEFAULT NULL,
  `tz` smallint(4) DEFAULT NULL,
  `dst` char(1) DEFAULT NULL,
  `city` varchar(255) DEFAULT NULL,
  `country` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`faa`)
```

The `CREATE TABLE` command starts by defining the name of the table, and then proceeds to list the field definitions in a comma-separated list. If you want to build a database from scratch—as we do in [Section 16.3](#)—you will have to write these definitions for each table.¹ Tables that are already created can be modified using the `ALTER TABLE` command. For example, the following will change the `tz` field to two digits and change the default value to zero.

```
ALTER TABLE airports CHANGE tz tz smallint(2) DEFAULT 0;
```

16.1.3 Keys

Two related but different concepts are *keys* and *indices*. The former offers some performance advantages but is primarily useful for imposing constraints on possible entries in the database, while the latter is purely about improving the speed of retrieval.

Different relational database management systems (RDBMS) may implement a variety of different kinds of keys, but three types are most common. In each case, suppose that we have a table with n rows and p columns.

- **PRIMARY KEY:** a column or set of columns in a table that uniquely identifies each row. By convention, this column is often called `id`. A table can have at most one *primary key*, and in general it is considered good practice to define a primary key on every table (although there are exceptions to this rule). If the index spans $k < p$ columns, then even though the primary key must by definition have n rows itself, it only requires nk pieces of data, rather than the np that the full table occupies. Thus, the primary key is always smaller than the table itself, and is thus faster to search. A second critically important role of the primary key is enforcement of non-duplication. If you try to insert a row into a table that would result in a duplicate entry for the primary key, you will get an error.
- **UNIQUE KEY:** a column or set of columns in a table that uniquely identifies each row, except for rows that contain `NULL` in some of those attributes. Unlike primary keys, a single table may have many unique keys. A typical use for these are in a lookup table. For example, Ted Turocy maintains a register of player `ids` for professional baseball players across multiple data providers. Each row in this table is a different player, and the primary key is a randomly-generated hash—each player gets exactly one value. However, each row also contains that same player’s `id` in systems designed by *MLBAM*, *Baseball-Reference*, *Baseball Prospectus*, *Fangraphs*, etc. This is tremendously useful for researchers working

¹There are ways of automatically generating table schemas, but in many cases some manual tweaking is recommended.

with multiple data providers, since they can easily link a player's statistics in one system to his information in another. However, this ability is predicated on the *uniqueness* of each player's id in *each* system. Moreover, many players may not have an `id` in every system, since some data providers track minor league baseball, or even the Japanese and Korean professional leagues. Thus, the imposition of a unique key—which allows `NULLS`—is necessary to maintain the integrity of these data.

- **FOREIGN KEY:** a column or set of columns that reference a primary key in another table. For example, the primary key in the `carriers` table is `carrier`. The `carrier` column in the `flights` table, which consists of carrier IDs, is a *foreign key* that references `carriers.carrier`. Foreign keys don't offer any performance enhancements, but they are important for maintaining *referential integrity*, especially in transactional databases that have many insertions and deletions.

You can use the `SHOW KEYS` command to identify the keys in a table. Note that the `carriers` table has only one key defined: a primary key on `carrier`.

```
SHOW KEYS FROM carriers;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Cardinality
carriers	0	PRIMARY	1	carrier	1610

16.1.4 Indices

While keys help maintain the integrity of the data, indices impose no constraints—they simply enable faster retrieval. An index is a lookup table that helps SQL keep track of which records contain certain values. Judicious use of indices can dramatically speed up retrieval times. The technical implementation of efficient indices is an active area of research among computer scientists, and fast indices are one of the primary advantages that differentiate SQL tables from large **R** data frames.

Indices have to be built by the database in advance, and they are then written to the disk. Thus, indices take up space on the disk (this is one of the reasons that they aren't implemented in **R**). For some tables with many indices, the size of the indices can even exceed the size of the raw data. Thus, when building indices, there is a trade-off to consider: You want just enough indices but not too many.

Consider the task of locating all of the rows in the `flights` table that contain the `origin` value `BDL`. These rows are strewn about the table in no particular order. How would you find them? A simple approach would be to start with the first row, examine the `origin` field, grab it if it contains `BDL`, and otherwise move to the second row. In order to ensure that all of the matching rows are returned, this algorithm must check every single one of the $n = 48$ million rows² in this table! So its speed is $O(n)$. However, we have built an index on the `origin` column, and this index contains only 2,266 rows (see [Table 16.1](#)). Each row in the index corresponds to exactly one value of `origin`, and contains a lookup for the exact rows in the table that are specific to that value. Thus, when we ask for the rows for which `origin` is equal to `BDL`, the database will use the index to deliver those rows very quickly. In practice, the retrieval speed for indexed columns can be $O(\ln n)$ (or better)—which is a tremendous advantage when n is large.

The speed-up that indices can provide is often especially apparent when joining two large tables. To see why, consider the following toy example. Suppose we want to merge two tables

²The `db` instance referenced here contains flight data from 2010–2017 only.

Table 16.1: Indices in the flights table.

Table	Non_unique	Key_name	Seq_in_index	Column_name	Cardinality
flights	1	Year	1	year	7
flights	1	Date	1	year	7
flights	1	Date	2	month	89
flights	1	Date	3	day	2712
flights	1	Origin	1	origin	2266
flights	1	Dest	1	dest	2266
flights	1	Carrier	1	carrier	133
flights	1	tailNum	1	tailnum	37861

on the columns whose values are listed below. To merge these records correctly, we have to do a lot of work to find the appropriate value in the second list that matches each value in the first list.

```
[1] 5 18 2 3 4 2 1
[1] 5 6 3 18 4 7 1 2
```

On the other hand, consider performing the same task on the same set of values, but having the values sorted ahead of time. Now, the merging task is very fast, because we can quickly locate the matching records. In effect, by keeping the records sorted, we have off-loaded the sorting task when we do a merge, resulting in much faster merging performance. However, this requires that we sort the records in the first place and then keep them sorted. This may slow down other operations—such as inserting new records—which now have to be done more carefully.

```
[1] 1 2 2 3 4 5 18
[1] 1 2 3 4 5 6 7 18
SHOW INDEXES FROM flights;
```

In MySQL the `SHOW INDEXES` command is equivalent to `SHOW KEYS`. Note that the `flights` table has several keys defined, but no primary key (see [Table 16.1](#)). The key `Date` spans the three columns `year`, `month`, and `day`.

16.1.5 Query plans

It is important to have the right indices built for your specific data and the queries that are likely to be run on it. Unfortunately, there is not always a straightforward answer to the question of which indices to build. For the `flights` table, it seems likely that many queries will involve searching for flights from a particular origin, or to a particular destination, or during a particular year (or range of years), or on a specific carrier, and so we have built indices on each of these columns. We have also built the `Date` index, since it seems likely that people would want to search for flights on a certain date. However, it does not seem so likely that people would search for flights in a specific month across all years, and thus we have not built an index on `month` alone. The `Date` index contains the `month` column, but this index can only be used if `year` is also part of the query.

You can ask MySQL for information about how it is going to perform a query using the `EXPLAIN` syntax. This will help you understand how onerous your query is, without actually

running it—saving you the time of having to wait for it to execute. This output reflects the query plan returned by the MySQL server.

If we were to run a query for long flights using the `distance` column the server will have to inspect each of the 48 million rows, since this column is not indexed. This is the slowest possible search, and is often called a *table scan*. The 48 million number that you see in the `rows` column is an estimate of the number of rows that MySQL will have to consult in order to process your query. In general, more rows mean a slower query.

```
EXPLAIN SELECT * FROM flights WHERE distance > 3000;
```

table	type	possible_key	key_len	ref	rows
flights	ALL	NA	NA	NA	47932811

On the other hand, if we search for recent flights using the `year` column, which has an index built on it, then we only need to consider a fraction of those rows (about 6.3 million).

```
EXPLAIN SELECT * FROM flights WHERE year = 2013;
```

table	type	possible_key	key_len	ref	rows
flights	ALL	Year,Date	NA	NA	6369482

Note that for the second example the server could have used either the index `Year` or the index `Date` (which contains the column `year`). Because of the index, only the 6.3 million flights from 2013 were consulted. Similarly, if we search by year and month, we can use the `Date` index.

```
EXPLAIN SELECT * FROM flights WHERE year = 2013 AND month = 6;
```

table	type	possible_key	key_len	ref	rows
flights	ref	Year,Date	Date	6	const,const

But if we search for months across all years, we can't! The query plan results in a table scan again.

```
EXPLAIN SELECT * FROM flights WHERE month = 6;
```

table	type	possible_key	key_len	ref	rows
flights	ALL	NA	NA	NA	47932811

This is because although `month` is part of the `Date` index, it is the *second* column in the index, and thus it doesn't help us when we aren't filtering on `year`. Thus, if it were common for our users to search on `month` without `year`, it would probably be worth building an index on `month`. Were we to actually run these queries, there would be a significant difference in computational time.

Using indices is especially important when performing `JOIN` operations on large tables. In this example, both queries use indices. However, because the cardinality of the index on `tailnum` is smaller than the cardinality of the index on `year` (see Table 16.1), the number of rows in `flights` associated with each unique value of `tailnum` is smaller than for each unique value of `year`. Thus, the first query runs faster.

```
EXPLAIN
```

```
SELECT * FROM planes p
```

```
LEFT JOIN flights o ON p.tailnum = o.TailNum
WHERE manufacturer = 'BOEING';
```

table	type	possible_key	key	key_len	ref	rows
p	ALL	NA	NA	NA	NA	3322
o	ref	tailNum	tailNum	9	airlines.p.tailnum	1266

EXPLAIN

```
SELECT * FROM planes p
LEFT JOIN flights o ON p.Year = o.Year
WHERE manufacturer = 'BOEING';
```

table	type	possible_key	key	key_len	ref	rows
p	ALL	NA	NA	NA	NA	3322
o	ref	Year,Date	Year	3	airlines.p.year	6450117

16.1.6 Partitioning

Another approach to speeding up queries on large tables (like `flights`) is *partitioning*. Here, we could create partitions based on the year. For `flights` this would instruct the server to physically write the `flights` table as a series of smaller tables, each one specific to a single value of `year`. At the same time, the server would create a logical supertable, so that to the user, the appearance of `flights` would be unchanged. This acts like a preemptive index on the `year` column.

If most of the queries to the `flights` table were for a specific year or range of years, then partitioning could significantly improve performance, since most of the rows would never be consulted. For example, if most of the queries to the `flights` database were for the past three years, then partitioning would reduce the search space of most queries on the full data set to the roughly 20 million flights in the last three years instead of the 169 million rows in the last 20 years. But here again, if most of the queries to the `flights` table were about carriers across years, then this type of partitioning would not help at all. It is the job of the database designer to tailor the database structure to the pattern of queries coming from the users. As a data scientist, this may mean that you have to tailor the database structure to the queries that you are running.

16.2 Changing SQL data

In [Chapter 15](#), we described how to query an SQL database using the `SELECT` command. Thus far in this chapter, we have discussed how to set up an SQL database, and how to optimize it for speed. None of these operations actually change data in an existing database. In this section, we will briefly touch upon the `UPDATE` and `INSERT` commands, which allow you to do exactly that.

16.2.1 Changing data

The `UPDATE` command allows you to reset values in a table across all rows that match a certain criteria. For example, in [Chapter 15](#) we discussed the possibility that airports could

change names over time. The airport in *Washington, D.C.* with code `DCA` is now called *Ronald Reagan Washington National*.

```
SELECT faa, name FROM airports WHERE faa = 'DCA';
```

faa	name
DCA	Ronald Reagan Washington Natl

However, the “Ronald Reagan” prefix was added in 1998. If—for whatever reason—we wanted to go back to the old name, we could use an `UPDATE` command to change that information in the `airports` table.

```
UPDATE airports
  SET name = 'Washington National'
  WHERE faa = 'DCA';
```

An `UPDATE` operation can be very useful when you have to apply wholesale changes over a large number of rows. However, extreme caution is necessary, since an imprecise `UPDATE` query can wipe out large quantities of data, and there is no “undo” operation!

Pro Tip 41. *Exercise extraordinary caution when performing `UPDATES`.*

16.2.2 Adding data

New data can be appended to an existing table with the `INSERT` commands. There are actually three things that can happen, depending on what you want to do when you have a primary key conflict. This occurs when one of the new rows that you are trying to insert has the same primary key value as one of the existing rows in the table.

- `INSERT`: Try to insert the new rows. If there is a primary key conflict, quit and throw an error.
- `INSERT IGNORE`: Try to insert the new rows. If there is a primary key conflict, skip inserting the conflicting rows and leave the existing rows untouched. Continue inserting data that does not conflict.
- `REPLACE`: Try to insert the new rows. If there is a primary key conflict, overwrite the existing rows with the new ones. Continue inserting data that does not conflict.

Recall that in [Chapter 15](#) we found that the airports in *Puerto Rico* were not present in the `airports` table. If we wanted to add these manually, we could use `INSERT`.

```
INSERT INTO airports (faa, name)
  VALUES ('SJU', 'Luis Munoz Marin International Airport');
```

Since `faa` is the primary key on this table, we can insert this row without contributing values for all of the other fields. In this case, the new row corresponding to `SJU` would have the `faa` and `name` fields as noted above, and the default values for all of the other fields. If we were to run this operation a second time, we would get an error, because of the primary key collision on `SJU`. We could avoid the error by choosing to `INSERT IGNORE` or `REPLACE` instead of `INSERT`.

16.2.3 Importing data from a file

In practice, we rarely add new data manually in this manner. Instead, new data are most often added using the `LOAD DATA` command. This allows a file containing new data—usually

a CSV—to be inserted in bulk. This is very common, when, for example, your data comes to you daily in a CSV file and you want to keep your database up to date. The primary key collision concepts described above also apply to the `LOAD DATA` syntax, and are important to understand for proper database maintenance. We illustrate the use of `LOAD DATA` in Section 16.3.

16.3 Extended example: Building a database

The *extract-transform-load* (ETL) paradigm is common among data professionals. The idea is that many data sources need to be extracted from some external source, transformed into a different format, and finally loaded into a database system. Often, this is an iterative process that needs to be done every day, or even every hour. In such cases, developing the infrastructure to automate these steps can result in dramatically increased productivity.

In this example, we will illustrate how to set up a MySQL database for the `babynames` data using the command line and SQL, but not **R**. As noted previously, while the `dplyr` package has made **R** a viable interface for querying and populating SQL databases, it is occasionally necessary to get “under the hood” with SQL. The files that correspond to this example can be found on the book website at <http://mdsr-book.github.io/>.

16.3.1 Extract

In this case, our data already lives in an **R** package, but in most cases, your data will live on a website, or be available in a different format. Our goal is to take that data from wherever it is and download it. For the `babynames` data, there isn’t much to do, since we already have the data in an **R** package. We will simply load it.

```
library(babynames)
```

16.3.2 Transform

Since SQL tables conform to a row-and-column paradigm, our goal during the transform phase is to create CSV files (see Chapter 6) for each of the tables. In this example, we will create tables for the `babynames` and `births` tables. You can try to add the `applicants` and `lifetables` tables on your own. We will simply write these data to CSV files using the `write_csv()` command. Since the `babynames` table is very long (nearly 1.8 million rows), we will just use the more recent data.

```
babynames %>%
  filter(year > 1975) %>%
  write_csv("babynames.csv")
births %>%
  write_csv("births.csv")
list.files(".", pattern = ".csv")
```

```
[1] "babynames.csv" "births.csv"
```

This raises an important question: what should we call these objects? The `babynames` package includes a data frame called `babynames` with one row per sex per year per name.

Having both the database and a table with the same name may be confusing. To clarify which is which we will call the database `babynamedb` and the table `babynames`.

Pro Tip 42. *Spending time thinking about the naming of databases, tables, and fields before you create them can help avoid confusion later on.*

16.3.3 Load into MySQL database

Next, we need to write a script that will define the table structure for these two tables in a MySQL database (instructions for creation of a database in SQLite can be found in [Section F.4.4](#)). This script will have four parts:

1. a `USE` statement that ensures we are in the right schema/database
2. a series of `DROP TABLE` statements that drop any old tables with the same names as the ones we are going to create
3. a series of `CREATE TABLE` statements that specify the table structures
4. a series of `LOAD DATA` statements that read the data from the CSVs into the appropriate tables

The first part is easy:

```
USE babynamedb;
```

This assumes that we have a local database called `babynamedb`—we will create this later. The second part is easy in this case, since we only have two tables. These ensure that we can run this script as many times as we want.

```
DROP TABLE IF EXISTS babynames;
DROP TABLE IF EXISTS births;
```

Pro Tip 43. *Be careful with the `DROP TABLE` statement. It destroys data.*

The third step is the trickiest part—we need to define the columns precisely. The use of `str()`, `summary()`, and `glimpse()` are particularly useful for matching up **R** data types with MySQL data types. Please see the MySQL documentation for more information about what data types are supported.

```
glimpse(babynames)
```

```
Rows: 1,924,665
Columns: 5
$ year <dbl> 1880, 1880, 1880, 1880, 1880, 1880, 1880, 1880, 1880...
$ sex <chr> "F", "F", "F", "F", "F", "F", "F", "F", "F", "F"...
$ name <chr> "Mary", "Anna", "Emma", "Elizabeth", "Minnie", "Margaret",...
$ n <int> 7065, 2604, 2003, 1939, 1746, 1578, 1472, 1414, 1320, 1288...
$ prop <dbl> 0.07238, 0.02668, 0.02052, 0.01987, 0.01789, 0.01617, 0.01...
```

In this case, we know that the `year` variable will only contain four-digit integers, so we can specify that this column take up only that much room in SQL. Similarly, the `sex` variable is just a single character, so we can restrict the width of that column as well. These savings probably won't matter much in this example, but for large tables they can make a noticeable difference.

```
CREATE TABLE `babynames` (
  `year` smallint(4) NOT NULL DEFAULT 0,
  `sex` char(1) NOT NULL DEFAULT 'F',
  `name` varchar(255) NOT NULL DEFAULT '',
  `n` mediumint(7) NOT NULL DEFAULT 0,
  `prop` decimal(21,20) NOT NULL DEFAULT 0,
  PRIMARY KEY (`year`, `sex`, `name`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

In this table, each row contains the information about one name for one sex in one year. Thus, each row contains a unique combination of those three variables, and we can therefore define a primary key across those three fields. Note the use of backquotes (to denote tables and variables) and the use of regular quotes (for default values).

```
glimpse(births)
```

```
Rows: 109
Columns: 2
$ year <int> 1909, 1910, 1911, 1912, 1913, 1914, 1915, 1916, 1917, 19...
$ births <int> 2718000, 2777000, 2809000, 2840000, 2869000, 2966000, 29...
```

```
CREATE TABLE `births` (
  `year` smallint(4) NOT NULL DEFAULT 0,
  `births` mediumint(8) NOT NULL DEFAULT 0,
  PRIMARY KEY (`year`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

Finally, we have to tell MySQL where to find the CSV files and where to put the data it finds in them. This is accomplished using the `LOAD DATA` command. You may also need to add a `LINES TERMINATED BY \r\n` clause, but we have omitted that for clarity. Please be aware that lines terminate using different characters in different operating systems, so Windows, Mac, and Linux users may have to tweak these commands to suit their needs. The `SHOW WARNINGS` commands are not necessary, but they will help with debugging.

```
LOAD DATA LOCAL INFILE './babynames.csv' INTO TABLE `babynames`
  FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '\"' IGNORE 1 LINES;
SHOW WARNINGS;
LOAD DATA LOCAL INFILE './births.csv' INTO TABLE `births`
  FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '\"' IGNORE 1 LINES;
SHOW WARNINGS;
```

Putting this all together, we have the following script:

```
USE babynamedb;

DROP TABLE IF EXISTS babynames;
DROP TABLE IF EXISTS births;

CREATE TABLE `babynames` (
  `year` smallint(4) NOT NULL DEFAULT 0,
  `sex` char(1) NOT NULL DEFAULT 'F',
  `name` varchar(255) NOT NULL DEFAULT '',
  `n` mediumint(7) NOT NULL DEFAULT 0,
  `prop` decimal(21,20) NOT NULL DEFAULT 0,
```

```

PRIMARY KEY (`year`, `sex`, `name`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE `births` (
  `year` smallint(4) NOT NULL DEFAULT 0,
  `births` mediumint(8) NOT NULL DEFAULT 0,
  PRIMARY KEY (`year`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

LOAD DATA LOCAL INFILE './babynames.csv' INTO TABLE `babynames`
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '\"' IGNORE 1 LINES;
LOAD DATA LOCAL INFILE './births.csv' INTO TABLE `births`
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '\"' IGNORE 1 LINES;

SELECT year
  , COUNT(DISTINCT name) AS numNames
  , SUM(n) AS numBirths
 FROM babynames
GROUP BY year
ORDER BY numBirths DESC
LIMIT 0,10;

```

Note that we have added a `SELECT` query just to verify that our table is populated. To load this into MySQL, we must first make sure that the `babynamedb` database exists, and if not, we must create it.

First, we check to see if `babynamedb` exists. We can do this from the command line using shell commands:

```
mysql -e "SHOW DATABASES;"
```

If it doesn't exist, then we must create it:

```
mysql -e "CREATE DATABASE babynamedb;"
```

Finally, we run our script. The `--show-warnings` and `-v` flags are optional, but will help with debugging.

```
mysql --local-infile --show-warnings -v babynamedb
< babynamedb.mysql
```

In practice, if your SQL script is not perfect, you will see errors or warnings the first time you try this. But by iterating this process, you will eventually refine your script such that it works as desired. If you get an 1148 error, make sure that you are using the `--local-infile` flag.

```
ERROR 1148 (42000): The used command is not allowed with this MySQL version
```

If you get a 29 error, make sure that the file exists in this location and that the `mysql` user has permission to read and execute it.

```
ERROR 29 (HY000): File './babynames.csv' not found (Errcode: 13)
```

Once the MySQL database has been created, the following commands can be used to access it from **R** using `dplyr`:

```
db <- dbConnect(RMySQL::MySQL(), dbname = "babynamedb")
babynames <- tbl(db, "babynames")
babynames %>%
  filter(name == "Benjamin")
```

16.4 Scalability

With the exception of SQLite, RDBMSs scale very well on a single computer to databases that take up dozens of gigabytes. For a dedicated server, even terabytes are workable on a single machine. Beyond this, many companies employ distributed solutions called *clusters*. A cluster is simply more than one machine (i.e., a node) linked together running the same RDBMS. One machine is designated as the head node, and this machine controls all of the other nodes. The actual data are distributed across the various nodes, and the head node manages queries—parcelling them to the appropriate cluster nodes.

A full discussion of clusters and other distributed architectures (including replication) are beyond the scope of this book. In [Chapter 21](#), we discuss alternatives to SQL that may provide higher-end solutions for bigger data.

16.5 Further resources

The *SQL in a Nutshell* book (Kline et al., 2008) is a useful reference for all things SQL.

16.6 Exercises

Problem 1 (Easy): Alice is searching for cancelled flights in the `flights` table, and her query is running very slowly. She decides to build an index on `cancelled` in the hopes of speeding things up. Discuss the relative merits of her plan. What are the trade-offs? Will her query be any faster?

Problem 2 (Medium): The `Master` table of the `Lahman` database contains biographical information about baseball players. The primary key is the `playerID` variable. There are also variables for `retroID` and `bbrefID`, which correspond to the player's identifier in other baseball databases. Discuss the ramifications of placing a primary, unique, or foreign key on `retroID`.

Problem 3 (Medium): Bob wants to analyze the on-time performance of United Airlines flights across the decade of the 1990s. Discuss how the partitioning scheme of the `flights` table based on `year` will affect the performance of Bob's queries, relative to an unpartitioned table.

Problem 4 (Hard): Use the `macleish` package to download the weather data at the

MacLeish Field Station. Write your own table schema from scratch and import these data into the database server of your choice.

Problem 5 (Hard): Write a full table schema for the two tables in the `fuелеconomy` package and import them into the database server of your choice.

Problem 6 (Hard): Write a full table schema for the `mtcars` data set and import it into the database server of your choice.

Problem 7 (Hard): Write a full table schema for the five tables in the `nasaweather` package and import them into the database server of your choice.

Problem 8 (Hard): Write a full table schema for two of the ten tables in the `usdanutrients` package and import them into the database server of your choice.

```
# remotes::install_github("hadley/usdanutrients")
library(usdanutrients)
# data(package="usdanutrients")
```

16.7 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/sql-II.html#sqlII-online-exercises>

Working with geospatial data

When data contain geographic coordinates, they can be considered a type of *spatial data*. Like the “text as data” that we explore in [Chapter 19](#), spatial data are fundamentally different from the numerical data with which we most often work. While spatial coordinates are often encoded as numbers, these numbers have special meaning, and our ability to understand them will suffer if we do not recognize their spatial nature.

The field of *spatial statistics* concerns building and interpreting models that include spatial coordinates. For example, consider a model for airport traffic using the `airlines` data. These data contain the geographic coordinates of each airport, so they are spatially-aware. But simply including the coordinates for latitude and longitude as covariates in a multiple regression model does not take advantage of the special meaning that these coordinates encode. In such a model we might be led towards the meaningless conclusion that airports at higher latitudes are associated with greater airplane traffic—simply due to the limited nature of the model and our careless and inappropriate use of these spatial data.

A full treatment of spatial statistics is beyond the scope of this book. While we won’t be building spatial models in this chapter, we will learn how to manage and visualize geospatial data in **R**. We will learn about how to work with *shapefiles*, which are a *de facto* open specification data structure for encoding spatial information. We will learn about *projections* (from three-dimensional space into two-dimensional space), colors (again), and how to create informative, and not misleading, spatially-aware visualizations. Our goal—as always—is to provide the reader with the technical ability and intellectual know-how to derive meaning from geospatial data.

17.1 Motivation: What’s so great about geospatial data?

The most famous early analysis of geospatial data was done by physician John Snow in 1854. In a certain London neighborhood, an outbreak of *cholera* killed 127 people in three days, resulting in a mass exodus of the local residents. At the time it was thought that cholera was an airborne disease caused by breathing foul air. Snow was critical of this theory, and set about discovering the true transmission mechanism.

Consider how you might use data to approach this problem. At the hospital, they might have a list of all of the patients that died of cholera. Those data might look like what is presented in [Table 17.1](#).

Snow’s genius was in focusing his analysis on the `Address` column. In a literal sense, the `Address` variable is a character vector—it stores text. This text has no obvious medical significance with respect to cholera. But we as human beings recognize that these strings of

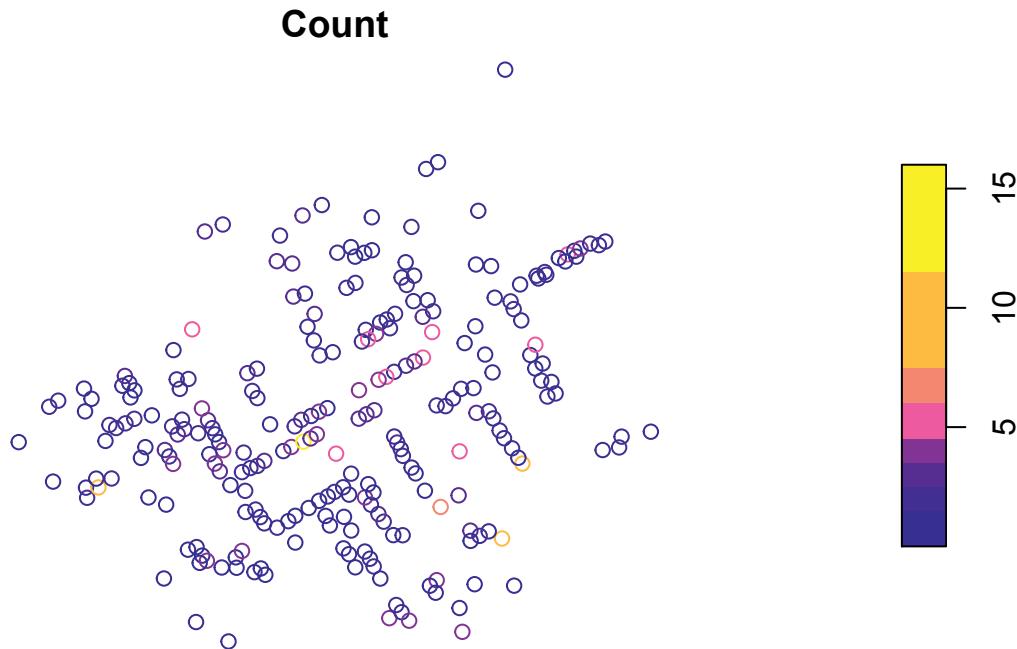
Table 17.1: Hypothetical data from 1854 cholera outbreak.

Date	Last_Name	First_Name	Address	Age	Cause_death
Aug 31, 1854	Jones	Thomas	26 Broad St.	37	cholera
Aug 31, 1854	Jones	Mary	26 Broad St.	11	cholera
Oct 1, 1854	Warwick	Martin	14 Broad St.	23	cholera

text encode *geographic locations*—they are *geospatial* data. Snow’s insight into this outbreak involved simply plotting these data in a geographically relevant way (see [Figure 17.2](#)).

The `CholeraDeaths` data are included in the `mdsr` package. When you plot the address of each person who died from cholera, you get something similar to what is shown in [Figure 17.1](#).

```
library(tidyverse)
library(mdsr)
library(sf)
plot(CholeraDeaths[["Count"]])
```

**Figure 17.1:** Context-free plot of 1854 cholera deaths.

While you might see certain patterns in these data, there is no *context* provided. The map that Snow actually drew is presented in [Figure 17.2](#). The underlying map of the London streets provides helpful context that makes the information in [Figure 17.1](#) intelligible.

However, Snow’s insight was driven by another set of data—the locations of the street-side water pumps. It may be difficult to see in the reproduction, but in addition to the lines indicating cholera deaths, there are labeled circles indicating the water pumps. A quick study of the map reveals that nearly all of the cholera cases are clustered around a single



Figure 17.2: John Snow's original map of the 1854 Broad Street cholera outbreak. Source: Wikipedia.

pump on the center of Broad Street. Snow was able to convince local officials that this pump was the probable cause of the epidemic.

While the story presented here is factual, it may be more legend than spatial data analysts would like to believe. Much of the causality is dubious: Snow himself believed that the outbreak petered out more or less on its own, and he did not create his famous map until afterwards. Nevertheless, his map was influential in the realization among doctors that cholera is a waterborne—rather than airborne—disease.

Our idealized conception of Snow's use of spatial analysis typifies a successful episode in data science. First, the key insight was made by combining three sources of data: the cholera deaths, the locations of the water pumps, and the London street map. Second, while we now have the capability to create a spatial model directly from the data that might have led to the same conclusion, constructing such a model is considerably more difficult than simply plotting the data in the proper context. Moreover, the plot itself—properly contextualized—is probably more convincing to most people than a statistical

model anyway. Human beings have a very strong intuitive ability to see spatial patterns in data, but computers have no such sense. Third, the problem was only resolved when the data-based evidence was combined with a plausible model that explained the physical phenomenon. That is, Snow *was a doctor* and his knowledge of disease transmission was sufficient to convince his colleagues that cholera was not transmitted via the air.¹

17.2 Spatial data structures

Spatial data are often stored in special data structures (i.e., not just `data.frames`). The most commonly used format for spatial data is called a *shapefile*. Another common format is *KML*. There are many other formats, and while mastering the details of any of these formats is not realistic in this treatment, there are some important basic notions that one must have in order to work with spatial data.

Shapefiles evolved as the native file format of the ArcView program developed by the *Environmental Systems Research Institute (Esri)* and have since become an open specification. They can be downloaded from many different government websites and other locations that publish spatial data. Spatial data consists not of rows and columns, but of geometric objects like points, lines, and polygons. Shapefiles contain vector-based instructions for drawing the boundaries of countries, counties, and towns, etc. As such, shapefiles are richer—and more complicated—data containers than simple data frames. Working with shapefiles in **R** can be challenging, but the major benefit is that shapefiles allow you to provide your data with a geographic context. The results can be stunning.

First, the term “shapefile” is somewhat of a *misnomer*, as there are several files that you must have in order to read spatial data. These files have extensions like `.shp`, `.shx`, and `.dbf`, and they are typically stored in a common directory.

There are *many* packages for **R** that specialize in working with spatial data, but we will focus on the most recent: `sf`. This package provides a `tidyverse`-friendly set of class definitions and functions for spatial objects in **R**. These will have the class `sf`. (Under the hood, `sf` wraps functionality previously provided by the `rgdal` and `rgeos` packages.²)

To get a sense of how these work, we will make a recreation of Snow’s cholera map. First, download and unzip this file: (http://rtwilson.com/downloads/SnowGIS_SHP.zip). After loading the `sf` package, we explore the directory that contains our shapefiles.

```
library(sf)
# The correct path on your computer may be different
dsn <- fs::path(root, "snow", "SnowGIS_SHP")
list.files(dsn)

[1] "Cholera_Deaths.dbf"           "Cholera_Deaths.prj"
[3] "Cholera_Deaths.sbn"          "Cholera_Deaths.sbx"
[5] "Cholera_Deaths.shp"          "Cholera_Deaths.shx"
[7] "OSMap_Grayscale.tif"         "OSMap_Grayscale.tif"
[9] "OSMap_Grayscale.tif.aux.xml" "OSMap_Grayscale.tif.ovr"
```

¹Unfortunately, the theory of germs and bacteria was still nearly a decade away.

²Note that `rgdal` may require external dependencies. On Ubuntu, it requires the `libgdal-dev` and `libproj-dev` packages. On Mac OS X, it requires GDAL. Also, loading `rgdal` loads `sp`.

```
[11] "OSMap.tfw"           "OSMap.tif"
[13] "Pumps.dbf"           "Pumps.prj"
[15] "Pumps.sbx"           "Pumps.shp"
[17] "Pumps.shx"           "README.txt"
[19] "SnowMap.tfw"          "SnowMap.tif"
[21] "SnowMap.tif.aux.xml" "SnowMap.tif.ovr"
```

Note that there are six files with the name `Cholera_Deaths` and another five with the name `Pumps`. These correspond to two different sets of shapefiles called *layers*, as revealed by the `st_layers()` function.

```
st_layers(dsn)
```

```
Driver: ESRI Shapefile
Available layers:
  layer_name geometry_type features fields
1 Cholera_Deaths      Point       250      2
2         Pumps        Point        8      1
```

We'll begin by loading the `Cholera_Deaths` layer into **R** using the `st_read()` function. Note that `Cholera_Deaths` is a `data.frame` in addition to being an `sf` object. It contains 250 *simple features* – these are the rows of the data frame, each corresponding to a different spatial object. In this case, the geometry type is `POINT` for all 250 rows. We will return to discussion of the mysterious projected CRS in [Section 17.3.2](#), but for now simply note that a specific geographic projection is encoded in these files.

```
CholeraDeaths <- st_read(dsn, layer = "Cholera_Deaths")
```

```
Reading layer `Cholera_Deaths' using driver `ESRI Shapefile'
Simple feature collection with 250 features and 2 fields
geometry type:  POINT
dimension:      XY
bbox:            xmin: 529000 ymin: 181000 xmax: 530000 ymax: 181000
projected CRS:  OSGB 1936 / British National Grid
```

```
class(CholeraDeaths)
```

```
[1] "sf"           "data.frame"
```

```
CholeraDeaths
```

```
Simple feature collection with 250 features and 2 fields
geometry type:  POINT
dimension:      XY
bbox:            xmin: 529000 ymin: 181000 xmax: 530000 ymax: 181000
projected CRS:  OSGB 1936 / British National Grid
```

```
First 10 features:
```

	Id	Count	geometry
1	0	3	POINT (529309 181031)
2	0	2	POINT (529312 181025)
3	0	1	POINT (529314 181020)
4	0	1	POINT (529317 181014)
5	0	4	POINT (529321 181008)
6	0	2	POINT (529337 181006)
7	0	2	POINT (529290 181024)

```
8   0      2 POINT (529301 181021)
9   0      3 POINT (529285 181020)
10  0     2 POINT (529288 181032)
```

There are data associated with each of these points. Every `sf` object is also a `data.frame` that stores values that correspond to each observation. In this case, for each of the points, we have an associated `Id` number and a `Count` of the number of deaths at that location. To plot these data, we can simply use the `plot()` generic function as we did in [Figure 17.1](#). However, in the next section, we will illustrate how `sf` objects can be integrated into a `ggplot2` workflow.

17.3 Making maps

In addition to providing geospatial processing capabilities, `sf` also provides spatial plotting extensions that work seamlessly with `ggplot2`. The `geom_sf()` function extends the grammar of graphics embedded in `ggplot2` that we explored in [Chapter 3](#) to provide native support for plotting spatial objects. Thus, we are only a few steps away from having some powerful mapping functionality.

17.3.1 Static maps

The `geom_sf()` function allows you to plot geospatial objects in any `ggplot2` object. Since the `x` and `y` coordinates are implied by the geometry of the `sf` object, you don't have to explicitly bind the `x` aesthetic (see [Chapter 3](#)) to the longitudinal coordinate and the `y` aesthetic to the latitude. Your map looks like this:

```
ggplot(CholeraDeaths) +
  geom_sf()
```

[Figure 17.3](#) is an improvement over what you would get from `plot()`. It is mostly clear what the coordinates along the axes are telling us (the units are in fact degrees), but we still don't have any context for what we are seeing. What we really want is to overlay these points on the London street map—and this is exactly what `ggspatial` lets us do.

The `annotation_map_tile()` function adds a layer of map tiles pulled from Open Street Map. We can control the `zoom` level, as well as the `type`. Here, we also map the number of deaths at each location to the size of the dot.

```
library(ggspatial)
ggplot(CholeraDeaths) +
  annotation_map_tile(type = "osm", zoomin = 0) +
  geom_sf(aes(size = Count), alpha = 0.7)
```

We note that *John Snow* is now the name of a pub on the corner of Broadwick (formerly Broad) Street and Lexington Street.

But look carefully at [Figure 17.4](#) and [Figure 17.2](#). You will not see the points in the right places. The center of the cluster is not on Broadwick Street, and some of the points are in the middle of the street (where there are no residences). Why? The coordinates in the `CholeraDeaths` object have unfamiliar values, as we can see by accessing the bounding box of the object.

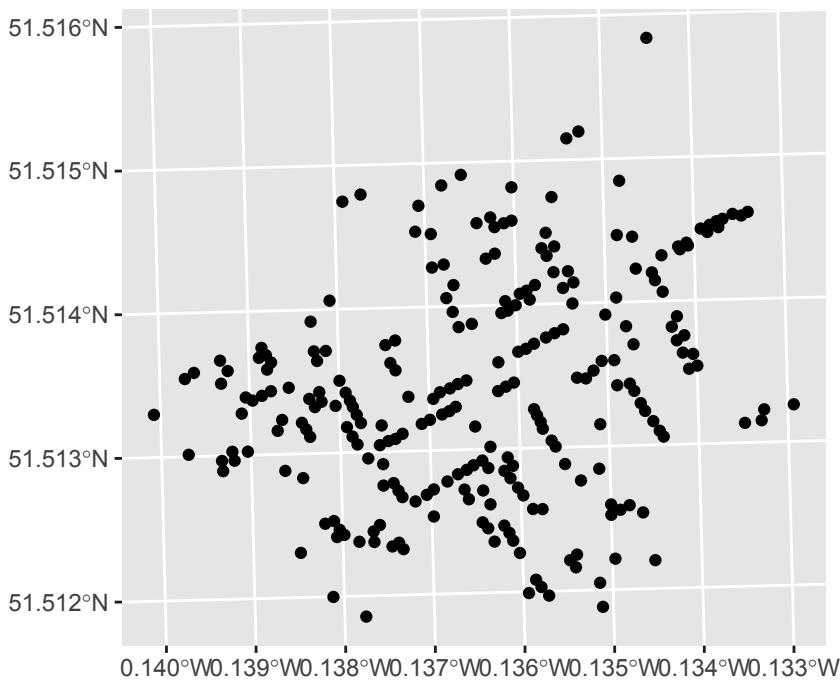


Figure 17.3: A simple `ggplot2` of the cholera deaths, with little context provided.

```
st_bbox(CholeraDeaths)
```

xmin	ymin	xmax	ymax
529160	180858	529656	181306

Both `CholeraDeaths` and the map tiles retrieved by the `ggspatial` package have geospatial coordinates, but those coordinates are not in the same units. While it is true that `annotation_map_tile()` performs some on the fly coordination translation, there remains a discrepancy between our two geospatial data sources. To understand how to get these two spatial data sources to work together properly, we have to understand projections.

17.3.2 Projections

The Earth happens to be an oblate spheroid—a three-dimensional flattened sphere. Yet we would like to create two-dimensional representations of the Earth that fit on pages or computer screens. The process of converting locations in a three-dimensional *geographic coordinate system* to a two-dimensional representation is called *projection*.

Once people figured out that the world was not flat, the question of how to project it followed. Since people have been making nautical maps for centuries, it would be nice if the study of map projection had resulted in a simple, accurate, universally-accepted projection system. Unfortunately, that is not the case. It is simply not possible to faithfully preserve all properties present in a three-dimensional space in a two-dimensional space. Thus there is no one best projection system—each has its own advantages and disadvantages. As noted previously, because the Earth is not a perfect sphere the mathematics behind many of these projections are non-trivial.

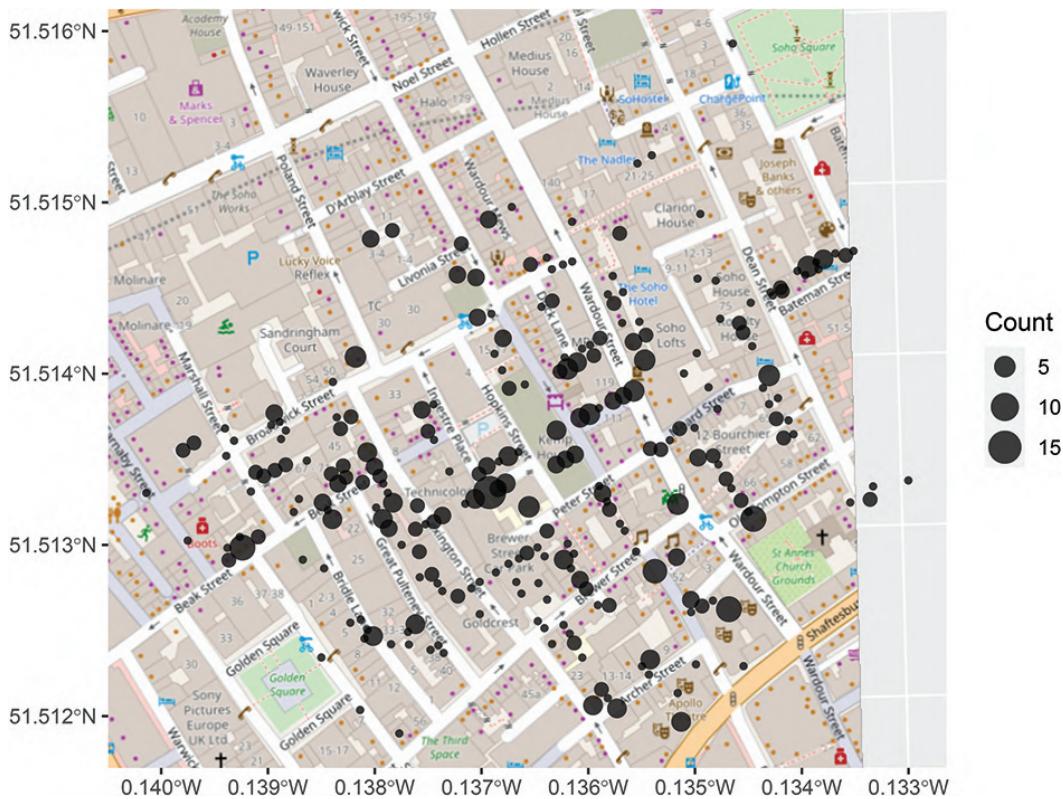


Figure 17.4: Erroneous reproduction of John Snow's original map of the 1854 cholera outbreak. The dots representing the deaths from cholera are off by hundreds of meters.

Two properties that a projection system might preserve—though not simultaneously—are *shape/angle* and *area*. That is, a projection system may be constructed in such a way that it faithfully represents the relative sizes of land masses in two dimensions. The Mercator projection shown at left in Figure 17.5 is a famous example of a projection system that does *not* preserve area. Its popularity is a result of its *angle*-preserving nature, which makes it useful for navigation. Unfortunately, it greatly distorts the size of features near the poles, where land masses become infinitely large.

```
library(mapproj)
library(maps)
map("world", projection = "mercator", wrap = TRUE)
map("world", projection = "cylindricalequalarea", param = 45, wrap = TRUE)
```

The Gall–Peters projection shown at right in Figure 17.5 does preserve area. Note the difference between the two projections when comparing the size of Greenland to Africa. In reality (as shown in the Gall–Peters projection) Africa is 14 times larger than Greenland. However, because Greenland is much closer to the North Pole, its area is greatly distorted in the Mercator projection, making it appear to be larger than Africa.

This particular example—while illustrative—became famous because of the socio-political controversy in which these projections became embroiled. Beginning in the 1960s, a German filmmaker named Arno Peters alleged that the commonly-used Mercator projection was an instrument of *cartographic imperialism*, in that it falsely focused attention on Northern and

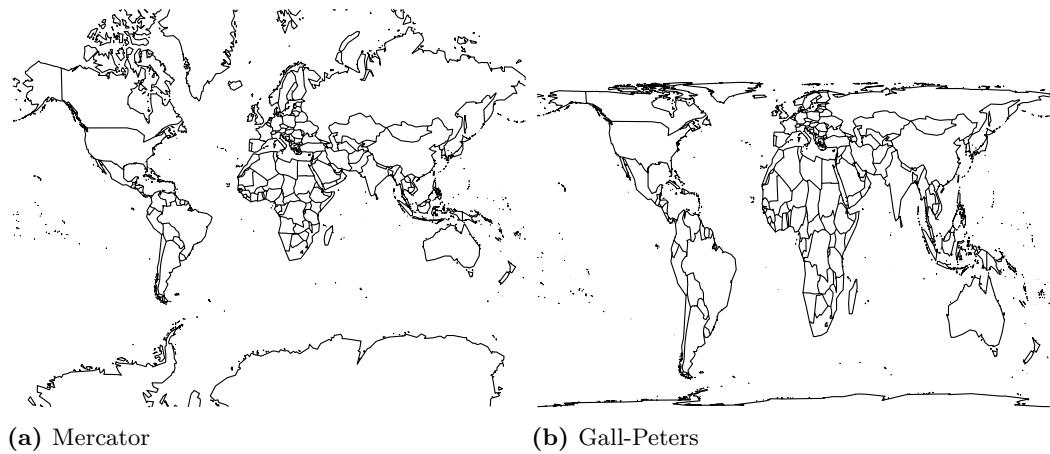


Figure 17.5: The world according to the Mercator (left) and Gall–Peters (right) projections.

Southern countries at the expense of those in Africa and South America closer to the equator. Peters had a point—the Mercator projection has many shortcomings—but unfortunately his claims about the virtues of the Gall–Peters projection (particularly its originality) were mostly false. Peters either ignored or was not aware that cartographers had long campaigned against the Mercator.

Nevertheless, you should be aware that the “default” projection can be very misleading. As a data scientist, your choice of how to project your data can have a direct influence on what viewers will take away from your data maps. Simply ignoring the implications of projections is not an ethically tenable position! While we can’t offer a comprehensive list of map projections here, two common general-purpose map projections are the *Lambert conformal conic* projection and the *Albers equal-area conic* projection (see Figure 17.6). In the former, angles are preserved, while in the latter neither scale nor shape are preserved, but gross distortions of both are minimized.

```
map(
  "state", projection = "lambert",
  parameters = c(lat0 = 20, lat1 = 50), wrap = TRUE
)
map(
  "state", projection = "albers",
  parameters = c(lat0 = 20, lat1 = 50), wrap = TRUE
)
```

Pro Tip 44. Always think about how your data are projected when making a map.

A *coordinate reference system* (CRS) is needed to keep track of geographic locations. Every spatially-aware object in R can have a projection. Three formats that are common for storing information about the projection of a geospatial object are *EPSG*, *PROJ.4*, and *WKT*. The former is simply an integer, while PROJ.4 is a cryptic string of text. The latter can be retrieved (or set) using the `st_crs()` command.



(a) Lambert conformal conic

(b) Albers equal area

Figure 17.6: The contiguous United States according to the Lambert conformal conic (left) and Albers equal area (right) projections. We have specified that the scales are true on the 20th and 50th parallels.

```
st_crs(CholeraDeaths)
```

Coordinate Reference System:

```
User input: OSGB 1936 / British National Grid
wkt:
PROJCRS["OSGB 1936 / British National Grid",
    BASEGEOGCRS["OSGB 1936",
        DATUM["OSGB 1936",
            ELLIPSOID["Airy 1830",6377563.396,299.3249646,
                LENGTHUNIT["metre",1]],
            PRIMEM["Greenwich",0,
                ANGLEUNIT["degree",0.0174532925199433]],
            ID["EPSG",4277]],
        CONVERSION["British National Grid",
            METHOD["Transverse Mercator",
                ID["EPSG",9807]],
            PARAMETER["Latitude of natural origin",49,
                ANGLEUNIT["degree",0.0174532925199433],
                ID["EPSG",8801]],
            PARAMETER["Longitude of natural origin",-2,
                ANGLEUNIT["degree",0.0174532925199433],
                ID["EPSG",8802]],
            PARAMETER["Scale factor at natural origin",0.9996012717,
                SCALEUNIT["unity",1],
                ID["EPSG",8805]],
            PARAMETER["False easting",400000,
                LENGTHUNIT["metre",1],
                ID["EPSG",8806]],
            PARAMETER["False northing",-100000,
                LENGTHUNIT["metre",1],
                ID["EPSG",8807]]],
        CS[Cartesian,2],
        AXIS["(E)",east,
```

```

    ORDER[1],
    LENGTHUNIT["metre",1]],
    AXIS["(N)",north,
        ORDER[2],
        LENGTHUNIT["metre",1]],
    USAGE[
        SCOPE["unknown"],
        AREA["UK - Britain and UKCS 49°46'N to 61°01'N, 7°33'W to 3°33'E"],
        BBOX[49.75,-9.2,61.14,2.88]],
    ID["EPSG",27700]]

```

It should be clear by now that the science of map projection is complicated, and it is likely unclear how to decipher this representation of the projection, which is in a format called *Well-Known Text*. What we can say is that `METHOD["Transverse_Mercator"]` indicates that these data are encoded using a *Transverse Mercator* projection. The *Airy ellipsoid* is being used, and the units are meters. The equivalent EPSG system is 27700. The *datum*—or model of the Earth—is OSGB 1936, which is also known as the *British National Grid*. The rest of the terms in the string are parameters that specify properties of that projection. The unfamiliar coordinates that we saw earlier for the `CholeraDeaths` data set were relative to this CRS.

There are many CRSs, but a few are most common. A set of EPSG (*European Petroleum Survey Group*) codes provides a shorthand for the full descriptions (like the one shown above). The most commonly-used are:

- EPSG:4326 - Also known as WGS84, this is the standard for GPS systems and Google Earth.
- EPSG:3857 - A Mercator projection used in maps tiles³ by Google Maps, Open Street Maps, etc.
- EPSG:27700 - Also known as OSGB 1936, or the British National Grid: United Kingdom Ordnance Survey. It is commonly used in Britain.

The `st_crs()` function will translate from the shorthand EPSG code to the full-text PROJ.4 strings and WKT.

```
st_crs(4326)$epsg
```

```
[1] 4326
```

```
st_crs(3857)$Wkt
```

```
[1] "PROJCS[\\"WGS 84 / Pseudo-Mercator\\",GEOGCS[\\"WGS 84\\",DATUM[\\"WGS_1984\\"],
SPHEROID[\\"WGS 84\\",6378137,298.257223563,AUTHORITY[\\"EPSG\\\",\\\"7030\\"]],
AUTHORITY[\\"EPSG\\\",\\\"6326\\"],PRIMEM[\\"Greenwich\\",0,
AUTHORITY[\\"EPSG\\\",\\\"8901\\"],UNIT[\\"degree\\",0.0174532925199433,
AUTHORITY[\\"EPSG\\\",\\\"9122\\"],AUTHORITY[\\"EPSG\\\",\\\"4326\\"]],
PROJECTION[\\"Mercator_1SP\\"],PARAMETER[\\"central_meridian\\",0],
PARAMETER[\\"scale_factor\\",1],PARAMETER[\\"false_easting\\",0],
PARAMETER[\\"false_northing\\",0],UNIT[\\"metre\\",1,
AUTHORITY[\\"EPSG\\\",\\\"9001\\"],AXIS[\\"Easting\\",EAST],
AXIS[\\"Northing\\",NORTH],EXTENSION[\\"PROJ4\\\",\\\"+proj=merc +a=6378137
+b=6378137 +lat_ts=0 +lon_0=0 +x_0=0 +y_0=0 +k=1 +units=m
```

³Google Maps and other online maps are composed of a series of square static images called *tiles*. These are pre-fetched and loaded as you scroll, creating the appearance of a larger image.

```
+nadgrids=@null +wktext +no_defs\"", AUTHORITY["EPSG","3857"]]]"
st_crs(27700)$proj4string
[1] "+proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=4000000 +y_0=-1000000 +ellps=airy +units=m +no_defs"
```

The `CholeraDeaths` points did not show up on our earlier map because we did not project them into the same coordinate system as the map tiles. Since we can't project the map tiles, we had better project the points in the `CholeraDeaths` data. As noted above, Google Maps tiles (and Open Street Map tiles) are projected in the `espg:3857` system. However, they are confusingly returned with coordinates in the `espg:4326` system. Thus, we use the `st_transform()` function to project our `CholeraDeaths` data to `espg:4326`.

```
cholera_4326 <- CholeraDeaths %>%
  st_transform(4326)
```

Note that the *bounding box* in our new coordinates are in the same familiar units of latitude and longitude.

```
st_bbox(cholera_4326)
```

```
xmin      ymin      xmax      ymax
-0.140  51.512 -0.133  51.516
```

Unfortunately, the code below *still* produces a map with the points in the wrong places.

```
ggplot(cholera_4326) +
  annotation_map_tile(type = "osm", zoomin = 0) +
  geom_sf(aes(size = Count), alpha = 0.7)
```

A careful reading of the help file for `spTransform-methods()` (the underlying machinery) gives some clues to our mistake.

```
help("spTransform-methods", package = "rgdal")
```

Not providing the appropriate `+datum` and `+towgs84` tags may lead to coordinates being out by hundreds of meters. Unfortunately, there is no easy way to provide this information: The user has to know the correct metadata for the data being used, even if this can be hard to discover.

That seems like our problem! The `+datum` and `+towgs84` arguments were missing from our PROJ.4 string.

```
st_crs(CholeraDeaths)$proj4string
```

```
[1] "+proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=4000000 +y_0=-1000000 +ellps=airy +units=m +no_defs"
```

The `CholeraDeaths` object has all of the same specifications as `espg:27700` but without the missing `+datum` and `+towgs84` tags. Furthermore, the documentation for the original data

source suggests using `epsg:27700`. Thus, we first assert that the `CholeraDeaths` data is in `epsg:27700`. Then, projecting to `epsg:4326` works as intended.

```
cholera_latlong <- CholeraDeaths %>%
  st_set_crs(27700) %>%
  st_transform(4326)
snow <- ggplot(cholera_latlong) +
  annotation_map_tile(type = "osm", zoomin = 0) +
  geom_sf(aes(size = Count))
```

All that remains is to add the locations of the pumps.

```
pumps <- st_read(dsn, layer = "Pumps")
```

```
Reading layer 'Pumps' using driver 'ESRI Shapefile'
Simple feature collection with 8 features and 1 field
geometry type:  POINT
dimension:      XY
bbox:           xmin: 529000 ymin: 181000 xmax: 530000 ymax: 181000
projected CRS: OSGB 1936 / British National Grid

pumps_latlong <- pumps %>%
  st_set_crs(27700) %>%
  st_transform(4326)
snow +
  geom_sf(data = pumps_latlong, size = 3, color = "red")
```

In [Figure 17.7](#), we finally see the clarity that judicious uses of spatial data in the proper context can provide. It is not necessary to fit a statistical model to these data to see that nearly all of the cholera deaths occurred in people closest to the Broad Street water pump, which was later found to be drawing fecal bacteria from a nearby cesspit.

17.3.3 Dynamic maps with `leaflet`

`Leaflet` is a powerful open-source JavaScript library for building interactive maps in HTML. The corresponding **R** package `leaflet` brings this functionality to **R** using the `htmlwidgets` platform introduced in [Chapter 14](#).

Although the commands are different, the architecture is very similar to `ggplot2`. However, instead of putting data-based layers on top of a static map, `leaflet` allows you to put data-based layers on top of an interactive map.

Because `leaflet` renders as HTML, you won't be able to take advantage of our plots in the printed book (since they are displayed as screen shots). We encourage you to run this code on your own and explore interactively.

A `leaflet` map widget is created with the `leaflet()` command. We will subsequently add layers to this widget. The first layer that we will add is a `tile` layer containing all of the static map information, which by default comes from OpenStreetMap. The second layer we will add here is a marker, which designates a point location. Note how the `addMarkers()` function can take a `data` argument, just like a `geom_*`() layer in `ggplot2` would.

```
white_house <- tibble(
  address = "The White House, Washington, DC"
) %>%
```

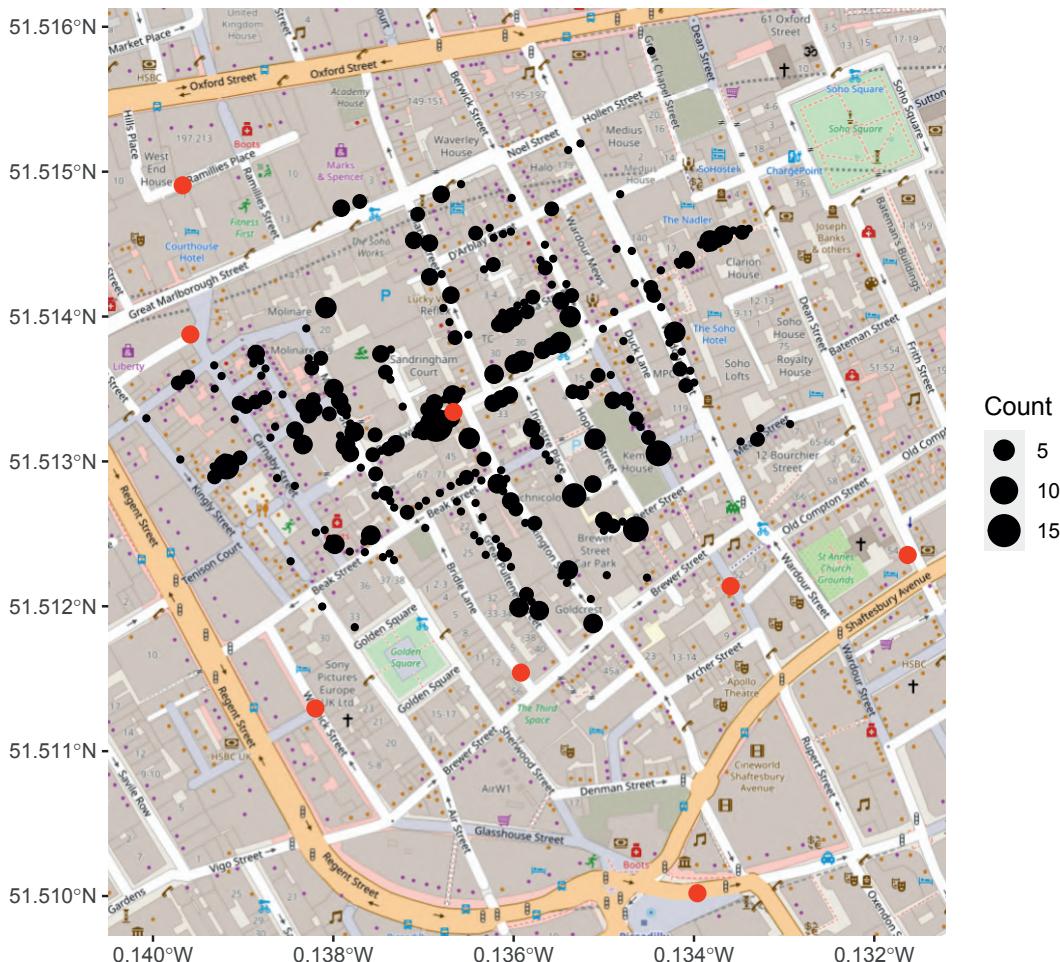


Figure 17.7: Recreation of John Snow's original map of the 1854 cholera outbreak. The size of each black dot is proportional to the number of people who died from cholera at that location. The red dots indicate the location of public water pumps. The strong clustering of deaths around the water pump on Broad(wick) Street suggests that perhaps the cholera was spread through water obtained at that pump.

```
tidygeocoder::geocode(address, method = "osm")

library(leaflet)
white_house_map <- leaflet() %>%
  addTiles() %>%
  addMarkers(data = white_house)
white_house_map
```

When you render this in **RStudio**, or in an **R** Markdown document with HTML output, or in a Web browser using Shiny, you will be able to scroll and zoom on the fly. In [Figure 17.8](#) we display our version.

We can also add a pop-up to provide more information about a particular location, as shown in [Figure 17.9](#).

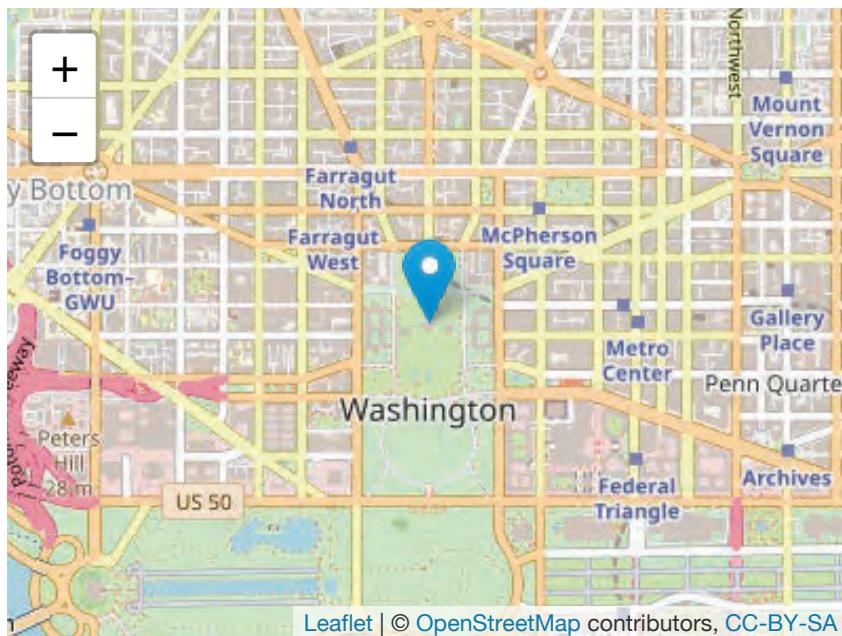


Figure 17.8: A leaflet plot of the White House.

```
white_house <- white_house %>%
  mutate(
    title = "The White House",
    street_address = "1600 Pennsylvania Ave"
  )
white_house_map %>%
  addPopups(
    data = white_house,
    popup = ~paste0("<b>", title, "</b><br>", street_address)
  )
```

Although **leaflet** and **ggplot2** are not syntactically equivalent, they are conceptually similar. Because the map tiles provide geographic context, the dynamic, zoomable, scrollable maps created by **leaflet** can be more informative than the static maps created by **ggplot2**.

17.4 Extended example: Congressional districts

In the 2012 presidential election, the *Republican* challenger Mitt Romney narrowly defeated President Barack Obama in the state of *North Carolina*, winning 50.4% of the popular votes, and thereby earning all 15 electoral votes. Obama had won North Carolina in 2008—becoming the first *Democrat* to do so since 1976. As a swing state, North Carolina has voting patterns that are particularly interesting, and—as we will see—contentious.

The roughly 50/50 split in the popular vote suggests that there are about the same number of Democratic and Republican voters in the state. In the fall of 2020, 10 of North Carolina's



Figure 17.9: A leaflet plot of the White House with a popup.

13 congressional representatives are Republican (with one seat currently vacant). How can this be? In this case, geospatial data can help us understand.

17.4.1 Election results

Our first step is to download the results of the 2012 congressional elections from the Federal Election Commission. These data are available through the **fec12** package.⁴

```
library(fec12)
```

Note that we have slightly more than 435 elections, since these data include U.S. territories like Puerto Rico and the Virgin Islands.

```
results_house %>%
  group_by(state, district_id) %>%
  summarize(N = n()) %>%
  nrow()
```

```
[1] 445
```

According to the *United States Constitution*, congressional districts are apportioned according to population from the 2010 U.S. Census. In practice, we see that this is not quite the case. These are the 10 candidates who earned the most votes in the general election.

```
results_house %>%
  left_join(candidates, by = "cand_id") %>%
  select(state, district_id, cand_name, party, general_votes) %>%
  arrange(desc(general_votes))
```

⁴fec12 is available on GitHub at <https://github.com/baumer-lab/fec12>.

```
# A tibble: 2,343 x 5
  state district_id cand_name      party general_votes
  <chr>    <chr>     <chr>       <chr>        <dbl>
1 PR      00        PIERLUISI, PEDRO R NPP          905066
2 PR      00        ALOMAR, RAFAEL COX PPD          881181
3 PA      02        FATTAH, CHAKA MR. D           318176
4 WA      07        McDERMOTT, JAMES   D           298368
5 MI      14        PETERS, GARY     D           270450
6 MO      01        CLAY, WILLIAM LACY JR D           267927
7 WI      02        POCHAN, MARK    D           265422
8 OR      03        BLUMENAUER, EARL  D           264979
9 MA      08        LYNCH, STEPHEN F  D           263999
10 MN     05        ELLISON, KEITH MAURICE DFL         262102
# ... with 2,333 more rows
```

Note that the representatives from Puerto Rico earned nearly three times as many votes as any other Congressional representative. We are interested in the results from North Carolina. We begin by creating a data frame specific to that state, with the votes aggregated by congressional district. As there are 13 districts, the `nc_results` data frame will have exactly 13 rows.

```
district_elections <- results_house %>%
  mutate(district = parse_number(district_id)) %>%
  group_by(state, district) %>%
  summarize(
    N = n(),
    total_votes = sum(general_votes, na.rm = TRUE),
    d_votes = sum(ifelse(party == "D", general_votes, 0), na.rm = TRUE),
    r_votes = sum(ifelse(party == "R", general_votes, 0), na.rm = TRUE)
  ) %>%
  mutate(
    other_votes = total_votes - d_votes - r_votes,
    r_prop = r_votes / total_votes,
    winner = ifelse(r_votes > d_votes, "Republican", "Democrat")
  )
nc_results <- district_elections %>%
  filter(state == "NC")
nc_results %>%
  select(-state)
```

Adding missing grouping variables: `state`

```
# A tibble: 13 x 9
# Groups: state [1]
  state district     N total_votes  d_votes  r_votes other_votes  r_prop
  <chr>    <dbl> <int>      <dbl>    <dbl>    <dbl>      <dbl>    <dbl>
1 NC        1     4      338066  254644  77288      6134  0.229
2 NC        2     8      311397  128973  174066      8358  0.559
3 NC        3     3      309885  114314  195571      0       0.631
4 NC        4     4      348485  259534  88951      0       0.255
5 NC        5     3      349197  148252  200945      0       0.575
6 NC        6     4      364583  142467  222116      0       0.609
7 NC        7     4      336736  168695  168041      0       0.499
```

```

8 NC      8     8      301824  137139  160695      3990  0.532
9 NC      9    13      375690  171503  194537      9650  0.518
10 NC     10    6      334849  144023  190826       0  0.570
11 NC     11    11      331426  141107  190319       0  0.574
12 NC     12    3      310908  247591  63317       0  0.204
13 NC     13    5      370610  160115  210495       0  0.568
# ... with 1 more variable: winner <chr>

```

We see that the distribution of the number of votes cast across congressional districts in North Carolina is very narrow—all of the districts had between 301,824 and 375,690 votes cast.

```

nc_results %>%
  skim(total_votes) %>%
  select(-na)

-- Variable type: numeric -----
var      state      n      mean      sd      p0      p25      p50      p75      p100
<chr>    <chr>    <int>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
1 total_votes NC      13 337204. 24175. 301824 311397 336736 349197 375690

```

However, as the close presidential election suggests, the votes of North Carolinans were roughly evenly divided among Democratic and Republican congressional candidates. In fact, state Democrats earned a narrow majority—50.6%—of the votes. Yet the Republicans won 9 of the 13 races.⁵

```

nc_results %>%
  summarize(
    N = n(),
    state_votes = sum(total_votes),
    state_d = sum(d_votes),
    state_r = sum(r_votes)
  ) %>%
  mutate(
    d_prop = state_d / state_votes,
    r_prop = state_r / state_votes
  )

# A tibble: 1 x 7
state      N state_votes state_d state_r d_prop r_prop
<chr> <int>    <dbl>    <dbl>    <dbl>    <dbl>
1 NC      13     4383656  2218357  2137167  0.506  0.488

```

One clue is to look more closely at the distribution of the proportion of Republican votes in each district.

```

nc_results %>%
  select(district, r_prop, winner) %>%
  arrange(desc(r_prop))

# A tibble: 13 x 4
# Groups:   state [1]

```

⁵The 7th district was the closest race in the entire country, with Democratic incumbent Mike McIntyre winning by just 655 votes. After McIntyre's retirement, Republican challenger David Rouzer won the seat easily in 2014.

```

state district r_prop winner
<chr>      <dbl>   <dbl> <chr>
1 NC          3  0.631 Republican
2 NC          6  0.609 Republican
3 NC          5  0.575 Republican
4 NC         11  0.574 Republican
5 NC         10  0.570 Republican
6 NC         13  0.568 Republican
7 NC          2  0.559 Republican
8 NC          8  0.532 Republican
9 NC          9  0.518 Republican
10 NC         7  0.499 Democrat
11 NC         4  0.255 Democrat
12 NC         1  0.229 Democrat
13 NC         12 0.204 Democrat

```

In the nine districts that Republicans won, their share of the vote ranged from a narrow (51.8%) to a comfortable (63.1%) majority. With the exception of the essentially even 7th district, the three districts that Democrats won were routs, with the Democratic candidate winning between 75% and 80% of the vote. Thus, although Democrats won more votes across the state, most of their votes were clustered within three overwhelmingly Democratic districts, allowing Republicans to prevail with moderate majorities across the remaining nine districts.

Conventional wisdom states that Democratic voters tend to live in cities, so perhaps they were simply clustered in three cities, while Republican voters were spread out across the state in more rural areas. Let's look more closely at the districts.

17.4.2 Congressional districts

To do this, we first download the congressional district shapefiles for the 113th Congress.

```

src <- "http://cdmaps.polisci.ucla.edu/shp/districts113.zip"
dsn_districts <- usethis::use_zip(src, destdir = fs::path("data_large"))

```

Next, we read these shapefiles into **R** as an **sf** object.

```

library(sf)
st_layers(dsn_districts)

```

```

Driver: ESRI Shapefile
Available layers:
  layer_name geometry_type features fields
1 districts113      Polygon      436      15
districts <- st_read(dsn_districts, layer = "districts113") %>%
  mutate(DISTRICT = parse_number(as.character(DISTRICT)))

```

```

Reading layer `districts113' using driver 'ESRI Shapefile'
Simple feature collection with 436 features and 15 fields
  (with 1 geometry empty)
geometry type:  MULTIPOLYGON
dimension:      XY
bbox:           xmin: -179 ymin: 18.9 xmax: 180 ymax: 71.4

```

```
geographic CRS: NAD83
```

```
glimpse(districts)
```

```
Rows: 436
```

```
Columns: 16
```

```
$ STATENAME <chr> "Louisiana", "Maine", "Maine", "Maryland", "Maryland...
$ ID <chr> "022113114006", "023113114001", "023113114002", "024...
$ DISTRICT <dbl> 6, 1, 2, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7...
$ STARTCONG <chr> "113", "113", "113", "113", "113", "113", "113", ...
$ ENDCONG <chr> "114", "114", "114", "114", "114", "114", "114", "11...
$ DISTRICTSI <chr> NA, ...
$ COUNTY <chr> NA, ...
$ PAGE <chr> NA, ...
$ LAW <chr> NA, ...
$ NOTE <chr> NA, ...
$ BESTDEC <chr> NA, ...
$ FINALNOTE <chr> "{\"From US Census website\"}", "{\"From US Census w...
$ RNOTE <chr> NA, ...
$ LASTCHANGE <chr> "2016-05-29 16:44:10.857626", "2016-05-29 16:44:10.8...
$ FROMCOUNTY <chr> "F", "F", "F", "F", "F", "F", "F", "F", "F", ...
$ geometry <MULTIPOLYGON [°]> MULTIPOLYGON (((-91.8 30.9,..., MULTIPO...
```

We are investigating North Carolina, so we will create a smaller object with only those shapes using the `filter()` function. Note that since every `sf` object is *also* a `data.frame`, we can use all of our usual `dplyr` tools on our geospatial objects.

```
nc_shp <- districts %>%
  filter(STATENAME == "North Carolina")
nc_shp %>%
  st_geometry() %>%
  plot(col = gray.colors(nrow(nc_shp)))
```

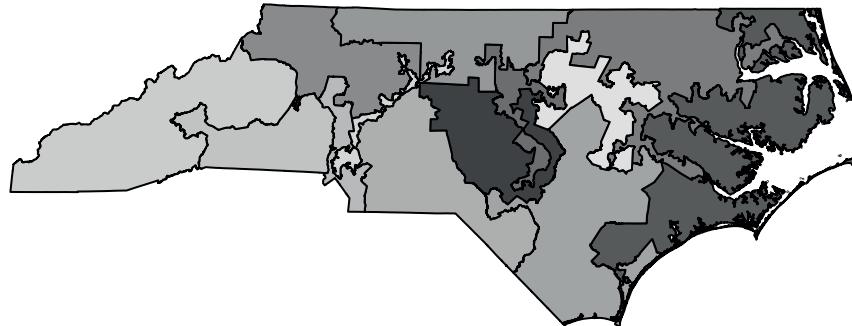


Figure 17.10: A basic map of the North Carolina congressional districts.

It is hard to see exactly what is going on here, but it appears as though there are some traditionally shaped districts, as well as some very strange and narrow districts. Unfortunately the map in [Figure 17.10](#) is devoid of context, so it is not very informative. We need the `nc_results` data to provide that context.

17.4.3 Putting it all together

How to merge these two together? The simplest way is to use the `inner_join()` function from `dplyr` (see Chapter 5). Since both `nc_shp` and `nc_results` are `data.frames`, this will append the election results data to the geospatial data. Here, we merge the `nc_shp` polygons with the `nc_results` election data frame using the district as the key. Note that there are 13 polygons and 13 rows.

```
nc_merged <- nc_shp %>%
  st_transform(4326) %>%
  inner_join(nc_results, by = c("DISTRICT" = "district"))
glimpse(nc_merged)
```

```
Rows: 13
Columns: 24
$ STATENAME <chr> "North Carolina", "North Carolina", "North Carolina...
$ ID <chr> "037113114002", "037113114003", "037113114004", "03...
$ DISTRICT <dbl> 2, 3, 4, 1, 5, 6, 7, 8, 9, 10, 11, 12, 13
$ STARTCONG <chr> "113", "113", "113", "113", "113", "113", "1...
$ ENDCONG <chr> "114", "114", "114", "114", "114", "114", "1...
$ DISTRICTSI <chr> NA, NA
$ COUNTY <chr> NA, NA
$ PAGE <chr> NA, NA
$ LAW <chr> NA, NA
$ NOTE <chr> NA, NA
$ BESTDEC <chr> NA, NA
$ FINALNOTE <chr> "{\"From US Census website\"}", "{\"From US Census ...
$ RNOTE <chr> NA, NA
$ LASTCHANGE <chr> "2016-05-29 16:44:10.857626", "2016-05-29 16:44:10....
$ FROMCOUNTY <chr> "F", "F", "F", "F", "F", "F", "F", "F", "F", ...
$ state <chr> "NC", "NC", "NC", "NC", "NC", "NC", "NC", "NC", "NC...
$ N <int> 8, 3, 4, 4, 3, 4, 4, 8, 13, 6, 11, 3, 5
$ total_votes <dbl> 311397, 309885, 348485, 338066, 349197, 364583, 336...
$ d_votes <dbl> 128973, 114314, 259534, 254644, 148252, 142467, 168...
$ r_votes <dbl> 174066, 195571, 88951, 77288, 200945, 222116, 16804...
$ other_votes <dbl> 8358, 0, 0, 6134, 0, 0, 3990, 9650, 0, 0, 0, 0
$ r_prop <dbl> 0.559, 0.631, 0.255, 0.229, 0.575, 0.609, 0.499, 0....
$ winner <chr> "Republican", "Republican", "Democrat", "Democrat",...
$ geometry <MULTIPOLYGON [°]> MULTIPOLYGON (((-80.1 35.8,..., MULTIP...
```

17.4.4 Using `ggplot2`

We are now ready to plot our map of North Carolina's congressional districts. We start by using a simple red-blue color scheme for the districts.

```
nc <- ggplot(data = nc_merged, aes(fill = winner)) +
  annotation_map_tile(zoom = 6, type = "osm") +
  geom_sf(alpha = 0.5) +
  scale_fill_manual("Winner", values = c("blue", "red")) +
  geom_sf_label(aes(label = DISTRICT), fill = "white") +
  theme_void()

nc
```

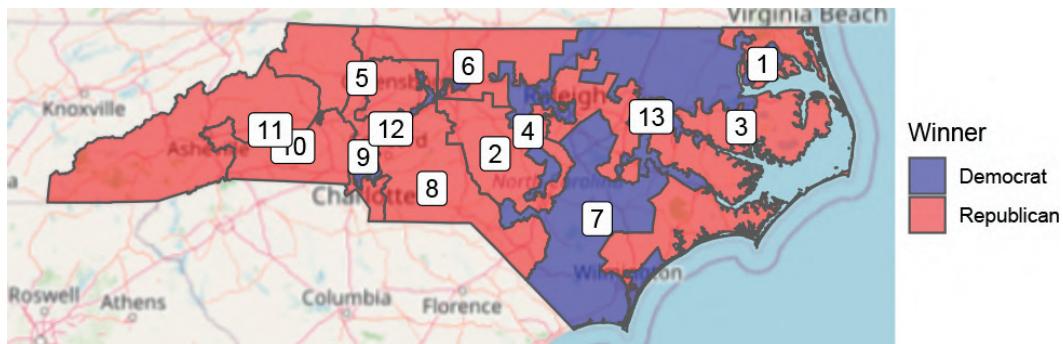


Figure 17.11: Bichromatic choropleth map of the results of the 2012 congressional elections in North Carolina.

Figure 17.11 shows that it was the Democratic districts that tended to be irregularly shaped. Districts 12 and 4 have narrow, tortured shapes—both were heavily Democratic. This plot tells us who won, but it doesn't convey the subtleties we observed about the *margins* of victory. In the next plot, we use a continuous color scale to indicate the proportion of votes in each district. The RdBu diverging color palette comes from **RColorBrewer** (see Chapter 2).

```
nc +
  aes(fill = r_prop) +
  scale_fill_distiller(
    "Proportion\nnRepublican",
    palette = "RdBu",
    limits = c(0.2, 0.8)
  )
```

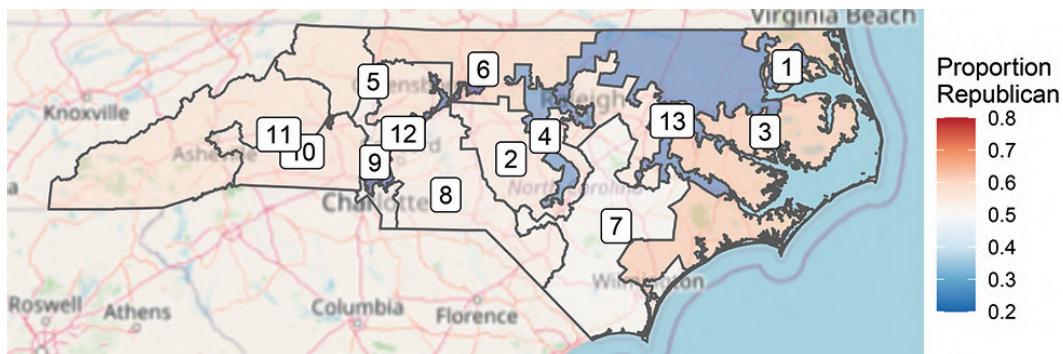


Figure 17.12: Full color choropleth map of the results of the 2012 congressional elections in North Carolina. The clustering of Democratic voters is evident from the deeper blue in Democratic districts, versus the pale red in the more numerous Republican districts.

The `limits` argument to `scale_fill_distiller()` is important. This forces *red* to be the color associated with 80% Republican votes and *blue* to be associated with 80% Democratic votes. Without this argument, red would be associated with the maximum value in that data (about 63%) and blue with the minimum (about 20%). This would result in the neutral

color of white not being at exactly 50%. When choosing color scales, it is critically important to make choices that reflect the data.

Pro Tip 45. Choose colors and scales carefully when making maps.

In [Figure 17.12](#), we can see that the three Democratic districts are “bluer” than the nine Republican counties are “red.” This reflects the clustering that we observed earlier. North Carolina has become one of the more egregious examples of *gerrymandering*, the phenomenon of when legislators of one party use their re-districting power for political gain. This is evident in [Figure 17.12](#), where Democratic votes are concentrated in three curiously-drawn congressional districts. This enables Republican lawmakers to have 69% (9/13) of the voting power in Congress despite earning only 48.8% of the votes.

Since the 1st edition of this book, the North Carolina gerrymandering case went all the way to the *United States Supreme Court*. In a landmark 2018 decision, the Justices ruled 5–4 in *Rucho vs. Common Cause* that while partisan gerrymanders such as those in North Carolina may be problematic for democracy, they are not reviewable by the judicial system.

17.4.5 Using `leaflet`

Was it true that the Democratic districts were weaved together to contain many of the biggest cities in the state? A similar map made in `leaflet` allows us to zoom in and pan out, making it easier to survey the districts.

First, we will define a color palette over the values [0, 1] that ranges from red to blue.

```
library(leaflet)
pal <- colorNumeric(palette = "RdBu", domain = c(0, 1))
```

To make our plot in `leaflet`, we have to add the tiles, and then the polygons defined by the `sf` object `nc_merged`. Since we want red to be associated with the proportion of Republican votes, we will map `1 - r_prop` to color. Note that we also add popups with the actual proportions, so that if you click on the map, it will show the district number and the proportion of Republican votes. The resulting `leaflet` map is shown in [Figure 17.13](#).

```
leaflet_nc <- leaflet(nc_merged) %>%
  addTiles() %>%
  addPolygons(
    weight = 1, fillOpacity = 0.7,
    color = ~pal(1 - r_prop),
    popup = ~paste("District", DISTRICT, "<br>", round(r_prop, 4)))
  ) %>%
  setView(lng = -80, lat = 35, zoom = 7)
```

Indeed, the curiously-drawn districts in blue encompass all seven of the largest cities in the state: *Charlotte, Raleigh, Greensboro, Durham, Winston-Salem, Fayetteville, and Cary*.

17.5 Effective maps: How (not) to lie

The map shown in [Figure 17.12](#) is an example of a *choropleth* map. This is a very common type of map where coloring and/or shading is used to differentiate a region of the map

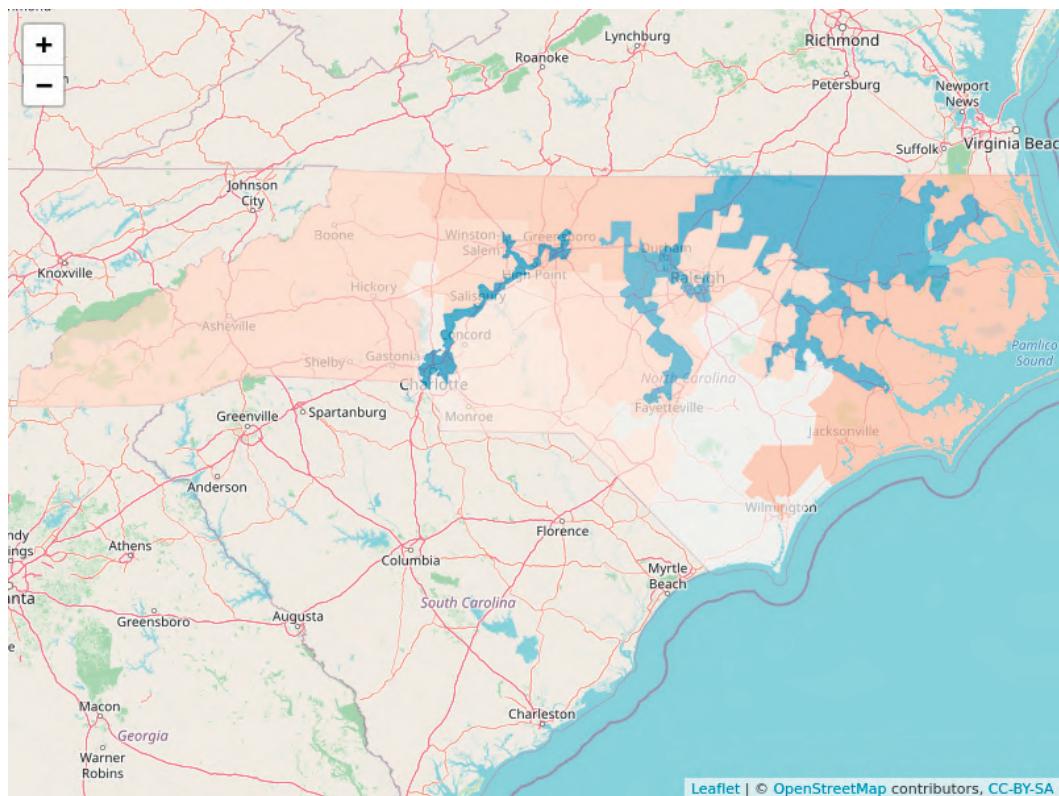


Figure 17.13: A leaflet plot of the North Carolina congressional districts.

based on the value of a variable. These maps are popular, and can be very persuasive, but you should be aware of some challenges when making and interpreting choropleth maps and other data maps. Three common map types include:

- **Choropleth:** color or shade regions based on the value of a variable
- **Proportional symbol:** associate a symbol with each location, but scale its size to reflect the value of a variable
- **Dot density:** place dots for each data point, and view their accumulation

We note that in a proportional symbol map the symbol placed on the map is usually two-dimensional. Its size—*in area*—should be scaled in proportion to the quantity being mapped. Be aware that often the size of the symbol is defined by its *radius*. If the *radius* is in direct proportion to the quantity being mapped, then the area will be disproportionately large.

Pro Tip 46. *Always scale the size of proportional symbols in terms of their area.*

As noted in [Chapter 2](#), the choice of scale is also important and often done poorly. The relationship between various quantities can be altered by scale. In [Chapter 2](#), we showed how the use of logarithmic scale can be used to improve the readability of a scatterplot. In [Figure 17.12](#), we illustrated the importance of properly setting the scale of a proportion so that 0.5 was exactly in the middle. Try making [Figure 17.12](#) without doing this and see if the results are as easily interpretable.

Decisions about colors are also crucial to making an effective map. In [Chapter 2](#), we men-

tioned the color palettes available through **RColorBrewer**. When making maps, categorical variables should be displayed using a *qualitative* palette, while quantitative variables should be displayed using a *sequential* or *diverging* palette. In [Figure 17.12](#) we employed a diverging palette, because Republicans and Democrats are on two opposite ends of the scale, with the neutral white color representing 0.5.

It's important to decide how to deal with missing values. Leaving them a default color (e.g., white) might confuse them with observed values.

Finally, the concept of *normalization* is fundamental. Plotting raw data values on maps can easily distort the truth. This is particularly true in the case of data maps, because area is an implied variable. Thus, on choropleth maps, we almost always want to show some sort of density or ratio rather than raw values (i.e., counts).

17.6 Projecting polygons

It is worth briefly illustrating the hazards of mapping unprojected data. Consider the congressional district map for the entire country. To plot this, we follow the same steps as before, but omit the step of restricting to North Carolina. There is one additional step here for creating a mapping between state names and their abbreviations. Thankfully, these data are built into **R**.

```
districts_full <- districts %>%
  left_join(
    tibble(state.abb, state.name),
    by = c("STATENAME" = "state.name")
  ) %>%
  left_join(
    district_elections,
    by = c("state.abb" = "state", "DISTRICT" = "district")
  )
```

We can make the map by adding white polygons for the generic map data and then adding colored polygons for each congressional district. Some clipping will make this easier to see.

```
box <- st_bbox(districts_full)
world <- map_data("world") %>%
  st_as_sf(coords = c("long", "lat")) %>%
  group_by(group) %>%
  summarize(region = first(region), do_union = FALSE) %>%
  st_cast("POLYGON") %>%
  st_set_crs(4269)
```

We display the Mercator projection of this base map in [Figure 17.14](#). Note how massive Alaska appears to be in relation to the other states. Alaska is big, but it is not that big! This is a distortion of reality due to the projection.

```
map_4269 <- ggplot(data = districts_full) +
  geom_sf(data = world, size = 0.1) +
  geom_sf(aes(fill = r_prop), size = 0.1) +
  scale_fill_distiller(palette = "RdBu", limits = c(0, 1)) +
```

```
theme_void() +
  labs(fill = "Proportion\nRepublican") +
  xlim(-180, -50) + ylim(box[c("ymin", "ymax")])
map_4269
```

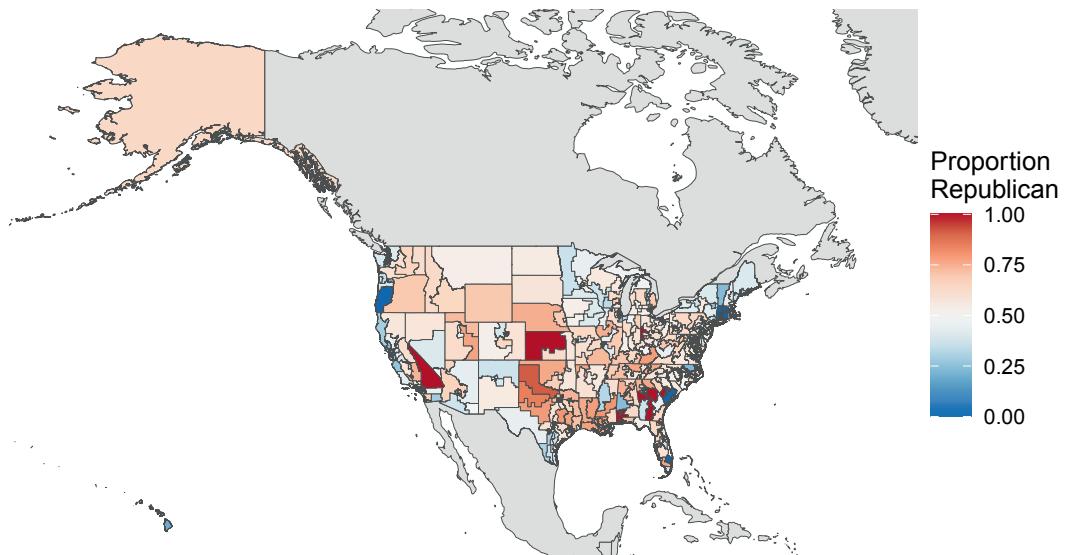


Figure 17.14: U.S. congressional election results, 2012 (Mercator projection).

We can use the Albers equal area projection to make a more representative picture, as shown in [Figure 17.15](#). Note how Alaska is still the biggest state (and district) by area, but it is now much closer in size to Texas.

```
districts_aea <- districts_full %>%
  st_transform(5070)
box <- st_bbox(districts_aea)
map_4269 %+% districts_aea +
  xlim(box[c("xmin", "xmax")]) + ylim(box[c("ymin", "ymax")])
```

17.7 Playing well with others

There are many technologies outside of **R** that allow you to work with spatial data. *ArcGIS* is a proprietary *Geographic Information System* (GIS) software that is considered by many to be the industry state-of-the-art. *QGIS* is its open-source competitor. Both have graphical user interfaces.

Keyhole Markup Language (KML) is an *XML* file format for storing geographic data. KML files can be read by *Google Earth* and other GIS applications. An **sf** object in **R** can be written to KML using the **st_write()** function. These files can then be read by ArcGIS, Google Maps, or Google Earth. Here, we illustrate how to create a KML file for the North Carolina congressional districts data frame that we defined earlier. A screenshot of the resulting output in Google Earth is shown in [Figure 17.16](#).

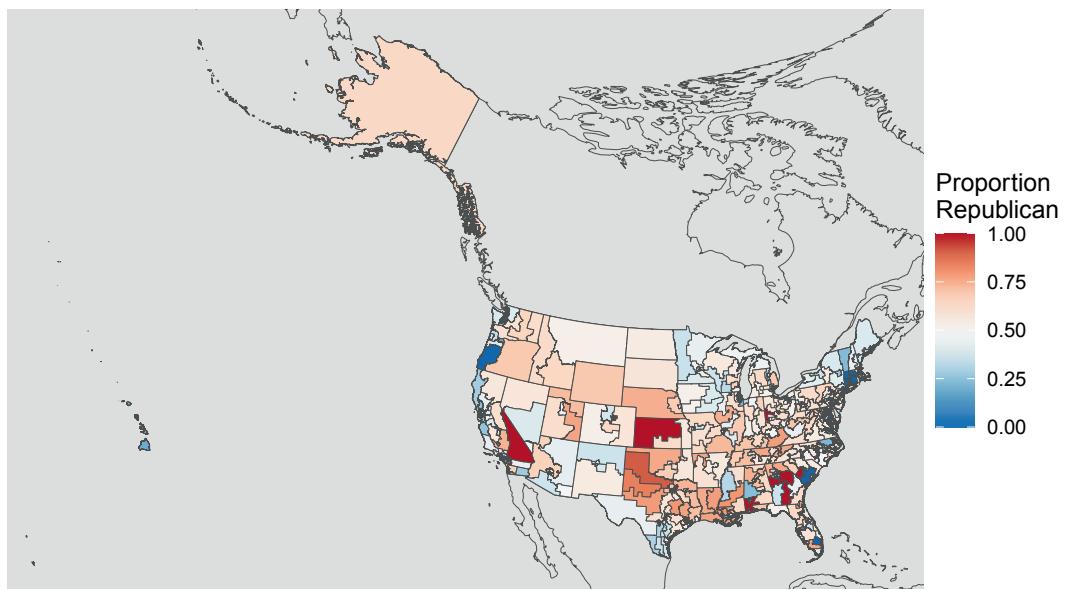


Figure 17.15: U.S. congressional election results, 2012 (Albers equal area projection).

```
nc_merged %>%
  st_transform(4326) %>%
  st_write("/tmp/nc_congress113.kml", driver = "kml")
```

17.8 Further resources

Some excellent resources for spatial methods include Bivand et al. (2013) and Cressie (1993). A helpful pocket guide to CRS systems in **R** contains information about projections, ellipsoids, and datums (reference points). Pebesma (2021) discuss the mechanics of how to work with spatial data in **R** in addition to introducing spatial modeling. The **tigris** package provides access to shapefiles and demographic data from the United States Census Bureau (Walker, 2020b).

The **sf** package has superseded spatial packages **sp**, **rgdal**, and **rgeos** which were used in the first edition of this book. A guide for migrating from **sp** to **sf** is maintained by the **r-spatial** group.

The fascinating story of John Snow and his pursuit of the causes of cholera can be found in Vinten-Johansen et al. (2003).

Quantitative measures of gerrymandering have been a subject of interest to political scientists for some time (Niemi et al., 1990; Engstrom and Wildgen, 1977; Hodge et al., 2010; Mackenzie, 2009).

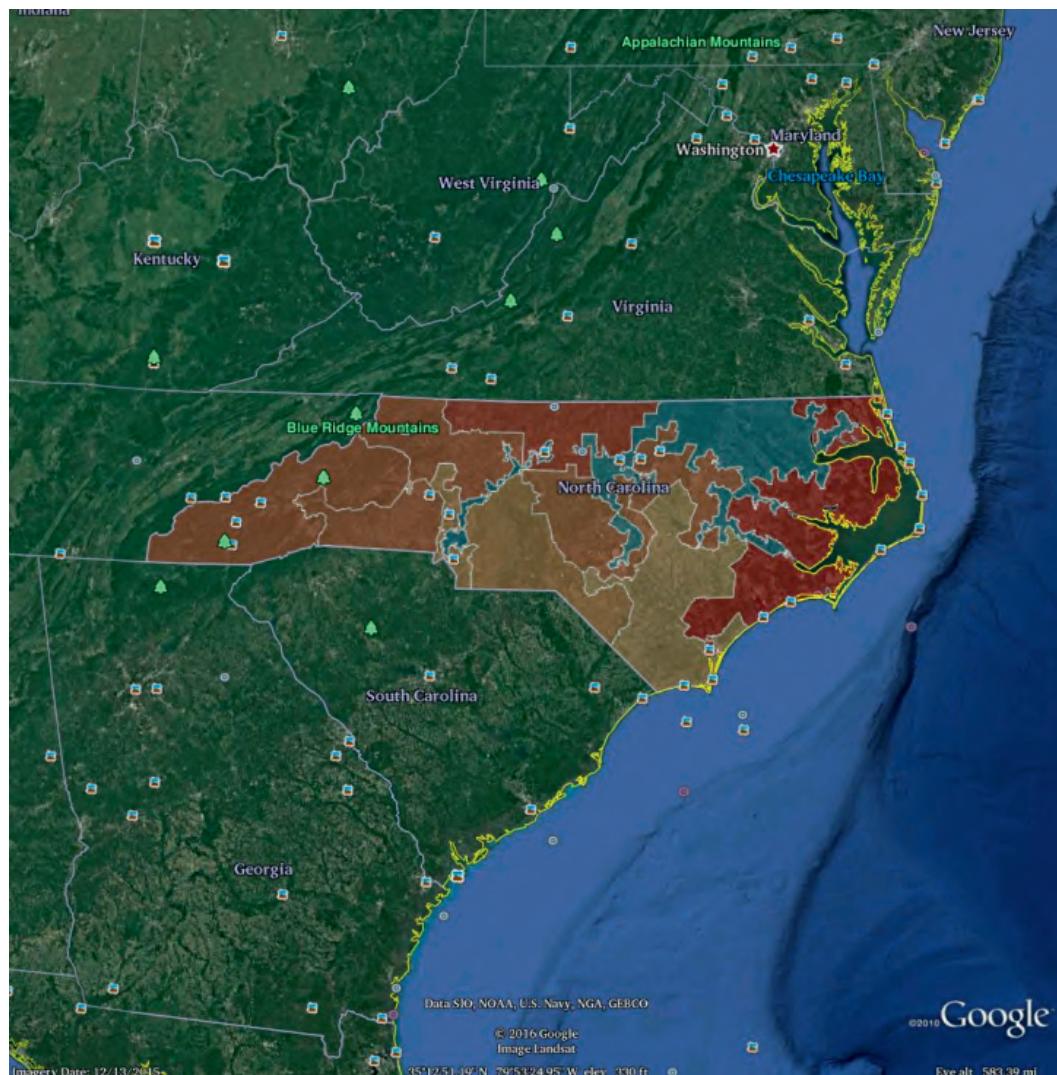


Figure 17.16: Screenshot of the North Carolina congressional districts as rendered in Google Earth, after exporting to KML. Compare with [Figure 17.12](#).

17.9 Exercises

Problem 1 (Easy): Use the `geocode` function from the `tidygeocoder` package to find the latitude and longitude of the Emily Dickinson Museum in Amherst, Massachusetts.

Problem 2 (Medium): The `pdxTrees` package contains a dataset of over 20,000 trees in Portland, Oregon, parks.

- Using `pdxTrees_parks` data, create a informative leaflet map for a tree enthusiast curious about the diversity and types of trees in the Portland area.
- Not all trees were created equal. Create an interactive map that highlights trees in

terms of their overall contribution to sustainability and value to the Portland community using variables such as `carbon_storage_value` and `total_annual_benefits`, etc.

- c. Create an interactive map that helps identify any problematic trees that city officials should take note of.

Problem 3 (Hard): Researchers at UCLA maintain historical congressional district shapefiles (see <http://cdmaps.polisci.ucla.edu>). Use these data to discuss the history of gerrymandering in the United States. Is the problem better or worse today?

Problem 4 (Hard): Use the `tidycensus` package to conduct a spatial analysis of the Census data it contains for your home state. Can you illustrate how the demography of your state varies spatially?

Problem 5 (Hard): Use the `tigris` package to make the congressional election district map for your home state. Do you see evidence of gerrymandering? Why or why not?

17.10 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/geospatial-I.html#geospatialI-online-exercises>



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

Geospatial computations

In [Chapter 17](#), we learned how to work with geospatial data. We learned about *shapefiles*, *map projections*, and how to plot spatial data using both **ggplot2** and **leaflet**. In this chapter, we will learn how to perform computations on geospatial data that will enable us to answer questions about how long or how big spatial features are. We will also learn how to use geometric operations and spatial joins to create new geospatial objects. These capabilities will broaden the spectrum of analytical tasks we can perform, and accordingly expand the range of questions we can answer.

18.1 Geospatial operations

18.1.1 Geocoding, routes, and distances

The process of converting a human-readable address into geographic coordinates is called *geocoding*. While there are numerous APIs available online that will do this for you, the functionality provided in **tidygeocoder** by the `geocode()` function uses Open Street Map and does not require registration to use the API. Here, we build a data frame of the places of business of the three authors, geocode the addresses of the schools, convert the resulting data frame into an **sf** object, and set the projection to `epsg:4326` (see [Chapter 17](#)).

```
library(tidyverse)
library(mdsr)
library(sf)
library(tidygeocoder)
colleges <- tribble(
  ~school, ~address,
  "Smith", "44 College Lane, Northampton, MA 01063",
  "Macalester", "1600 Grand Ave, St Paul, MN 55105",
  "Amherst", "Amherst College, Amherst, MA 01002"
) %>%
  geocode(address, method = "osm") %>%
  st_as_sf(coords = c("long", "lat")) %>%
  st_set_crs(4326)
colleges
```

```
Simple feature collection with 3 features and 2 fields
geometry type:  POINT
dimension:      XY
bbox:           xmin: -93.2 ymin: 42.3 xmax: -72.5 ymax: 44.9
geographic CRS: WGS 84
# A tibble: 3 x 3
```

```

school      address                      geometry
* <chr>      <chr>                      <POINT [°]>
1 Smith     44 College Lane, Northampton, MA 01063 (-72.6 42.3)
2 Macalester 1600 Grand Ave, St Paul, MN 55105      (-93.2 44.9)
3 Amherst    Amherst College, Amherst, MA 01002      (-72.5 42.4)

```

Geodesic distances can be computed using the `st_distance()` function in `sf`. Here, we compute the distance between two of the Five Colleges¹.

```

colleges %>%
  filter(school != "Macalester") %>%
  st_distance()

```

```

Units: [m]
 [,1] [,2]
[1,] 0 11982
[2,] 11982 0

```

The geodesic distance is closer to “*as the crow flies*,” but we might be more interested in the *driving distance* between two locations along the road. To compute this, we need to access a service with a database of roads. Here, we use the openroute service, which requires an API key². The `openrouteservice` package provides this access via the `ors_directions()` function. Note that the value of the variable `mdsr_ors_api_key` is not shown. You will need your own API to use this service.

```

library(openrouteservice)
ors_api_key(mdsr_ors_api_key)

smith_amherst <- colleges %>%
  filter(school != "Macalester") %>%
  st_coordinates() %>%
  as_tibble()

route_driving <- smith_amherst %>%
  ors_directions(profile = "driving-car", output = "sf")

```

Note the difference between the geodesic distance computed above and the driving distance computed below. Of course, the driving distance must be longer than the geodesic distance.

```

route_driving %>%
  st_length()

```

```
13545 [m]
```

If you prefer, you can convert meters to miles using the `set_units()` function from the `units` package.

```

route_driving %>%
  st_length() %>%
  units::set_units("miles")

```

```
8.42 [miles]
```

¹The Five College Consortium consists of Amherst, Hampshire, Mount Holyoke, and Smith Colleges, as well as the University of Massachusetts-Amherst.

²Google Maps requires an API key backed up by either a credit card or credits requested by an instructor.

Given the convenient Norwottuck Rail Trail connecting Northampton and Amherst, we might prefer to bike. Will that be shorter?

```
route_cycling <- smith_amherst %>%
  ors_directions(profile = "cycling-regular", output = "sf")

route_cycling %>%
  st_length()
```

14066 [m]

It turns out that the rail trail path is slightly longer (but far more scenic).

Since the *Calvin Coolidge Bridge* is the only reasonable way to get from Northampton to Amherst when driving, there is only one shortest route between Smith and Amherst, as shown in [Figure 18.1](#). We also show the shortest biking route, which follows the Norwottuck Rail Trail.

```
library(leaflet)
leaflet() %>%
  addTiles() %>%
  addPolylines(data = route_driving, weight = 10) %>%
  addPolylines(data = route_cycling, color = "green", weight = 10)
```



Figure 18.1: The fastest route from Smith College to Amherst College, by both car (blue) and bike (green).

However, shortest paths in a network are not unique (see [Chapter 20](#)). Ben's daily commute to *Citi Field* from his apartment in *Brooklyn* presented three distinct alternatives:

1. One could take the *Brooklyn-Queens Expressway* (I-278 E) to the *Grand Central Parkway* E and pass by *LaGuardia Airport*.
2. One could continue on the *Long Island Expressway* (I-495 E) and then approach Citi Field from the opposite direction on the Grand Central Parkway W.
3. One could avoid highways altogether and take *Roosevelt Avenue* all the way through *Queens*.

The latter route is the shortest but often will take longer due to traffic. The first route is the most convenient approach to the Citi Field employee parking lot. These two routes are overlaid on the map in [Figure 18.2](#).

```
commute <- tribble(
  ~place, ~address,
  "home", "736 Leonard St, Brooklyn, NY",
  "lga", "LaGuardia Airport, Queens, NY",
  "work", "Citi Field, 41 Seaver Way, Queens, NY 11368",
) %>%
  geocode(address, method = "osm") %>%
  st_as_sf(coords = c("long", "lat")) %>%
  st_set_crs(4326)

route_direct <- commute %>%
  filter(place %in% c("home", "work")) %>%
  st_coordinates() %>%
  as_tibble() %>%
  ors_directions(output = "sf", preference = "recommended")

route_gcp <- commute %>%
  st_coordinates() %>%
  as_tibble() %>%
  ors_directions(output = "sf")

leaflet() %>%
  addTiles() %>%
  addMarkers(data = commute, popup = ~place) %>%
  addPolylines(data = route_direct, color = "green", weight = 10) %>%
  addPolylines(data = route_gcp, weight = 10)
```

18.1.2 Geometric operations

Much of the power of working with geospatial data comes from the interactions between various layers of data. The **sf** package provides many features that enable us to compute with geospatial data.

A basic geospatial question is: what part of one series of geospatial objects lies within another set? To illustrate, we use geospatial data from the MacLeish field station in *Whately, MA*. These data are provided by the **macleish** package. [Figure 18.3](#) illustrates that there are several streams that pass through the MacLeish property.

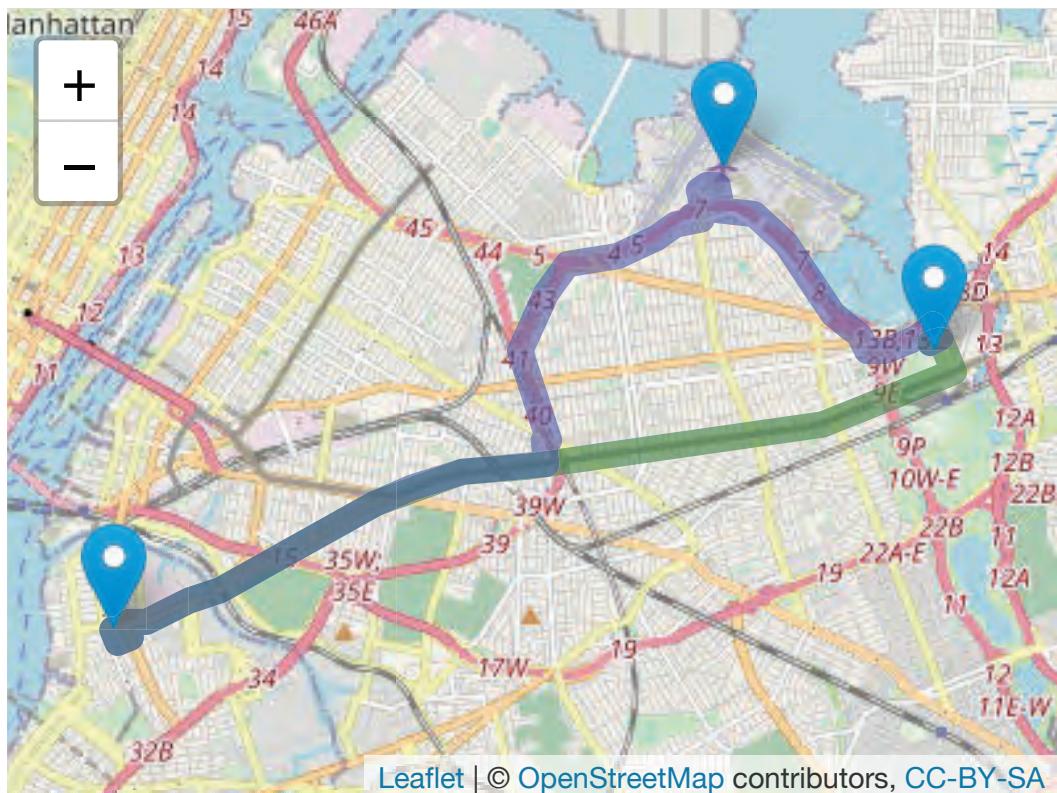


Figure 18.2: Alternative commuting routes from Ben's old apartment in Brooklyn to Citi Field. Note that the routes overlap for most of the way from Brooklyn to the I-278 E onramp on Roosevelt Avenue.

```
library(sf)
library(macleish)

boundary <- macleish_layers %>%
  pluck("boundary")
streams <- macleish_layers %>%
  pluck("streams")

boundary_plot <- ggplot(boundary) +
  geom_sf() +
  scale_x_continuous(breaks = c(-72.677, -72.683))

boundary_plot +
  geom_sf(data = streams, color = "blue", size = 1.5)
```

The data from MacLeish happens to have a variable called `Shape_Area` that contains the precomputed size of the property.

```
boundary %>%
  pull(Shape_Area)
```

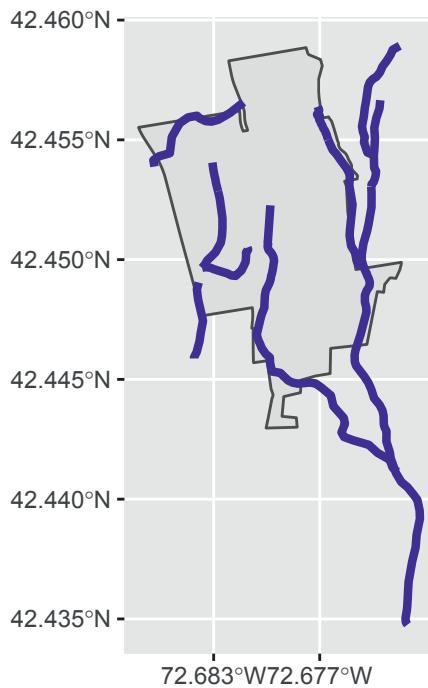


Figure 18.3: Streams cross through the boundary of the MacLeish property.

```
[1] 1033988
```

Is this accurate? We can easily compute basic geometric properties of spatial objects like area and length using the `st_area` function.

```
st_area(boundary)
```

```
1034035 [m^2]
```

The exact computed area is very close to the reported value. We can also convert the area in square meters to *acres* by dividing by a known conversion factor.

```
st_area(boundary) / 4046.8564224
```

```
256 [m^2]
```

Similarly, we can compute the length of each segment of the streams and the location of the *centroid* of the property.

```
streams %>%
  mutate(length = st_length(geometry))
```

```
Simple feature collection with 13 features and 2 fields
geometry type:  LINESTRING
dimension:      XY
bbox:           xmin: -72.7 ymin: 42.4 xmax: -72.7 ymax: 42.5
geographic CRS: WGS 84
First 10 features:
  Id          geometry      length
1  1  LINESTRING (-72.7 42.5, -72...  593.2 [m]
```

```

2  1 LINESTRING (-72.7 42.5, -72... 411.9 [m]
3  1 LINESTRING (-72.7 42.5, -72... 137.8 [m]
4  1 LINESTRING (-72.7 42.5, -72... 40.2 [m]
5  1 LINESTRING (-72.7 42.5, -72... 51.0 [m]
6  1 LINESTRING (-72.7 42.5, -72... 592.5 [m]
7  1 LINESTRING (-72.7 42.5, -72... 2151.1 [m]
8  3 LINESTRING (-72.7 42.5, -72... 1651.7 [m]
9  3 LINESTRING (-72.7 42.5, -72... 316.6 [m]
10 3 LINESTRING (-72.7 42.5, -72... 388.1 [m]
```

```
boundary %>%
  st_centroid()
```

```
Simple feature collection with 1 feature and 3 fields
geometry type: POINT
dimension: XY
bbox: xmin: -72.7 ymin: 42.5 xmax: -72.7 ymax: 42.5
geographic CRS: WGS 84
  OBJECTID Shape_Leng Shape_Area      geometry
1         1      5894  1033988 POINT (-72.7 42.5)
```

As promised, we can also combine two geospatial layers. The functions `st_intersects()` and `st_intersection()` take two geospatial objects and return a `logical` indicating whether they intersect, or another `sf` object representing that intersection, respectively.

```
st_intersects(boundary, streams)
```

```
Sparse geometry binary predicate list of length 1, where the predicate was `intersects'
1: 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...
st_intersection(boundary, streams)
```

```
Simple feature collection with 11 features and 4 fields
geometry type: GEOMETRY
dimension: XY
bbox: xmin: -72.7 ymin: 42.4 xmax: -72.7 ymax: 42.5
geographic CRS: WGS 84
First 10 features:
  OBJECTID Shape_Leng Shape_Area Id      geometry
1         1      5894  1033988 1 LINESTRING (-72.7 42.5, -72...
1.1       1      5894  1033988 1 LINESTRING (-72.7 42.5, -72...
1.2       1      5894  1033988 1 LINESTRING (-72.7 42.5, -72...
1.3       1      5894  1033988 1 MULTILINESTRING ((-72.7 42....
1.4       1      5894  1033988 1 LINESTRING (-72.7 42.4, -72...
1.5       1      5894  1033988 3 LINESTRING (-72.7 42.5, -72...
1.6       1      5894  1033988 3 LINESTRING (-72.7 42.5, -72...
1.7       1      5894  1033988 3 LINESTRING (-72.7 42.5, -72...
1.8       1      5894  1033988 3 LINESTRING (-72.7 42.5, -72...
1.9       1      5894  1033988 3 LINESTRING (-72.7 42.4, -72...
```

`st_intersects()` is called a *predicate* function because it returns a `logical`. It answers the question: “Do these two layers intersect?” On the other hand, `st_intersection()` performs a set operation. It answers the question: “What is the set that represents the intersection of these two layers?” Similar functions compute familiar set operations like unions, differ-

ences, and symmetric differences, while a whole host of additional predicate functions detect containment (`st_contains()`, `st_within()`, etc.), crossings, overlaps, etc.

In [Figure 18.4\(a\)](#) we use the `st_intersection()` function to show only the parts of the streams that are contained within the MacLeish property. In [Figure 18.4\(b\)](#), we show the corresponding set of stream parts that lie *outside* of the MacLeish property.

```
boundary_plot +
  geom_sf(
    data = st_intersection(boundary, streams),
    color = "blue",
    size = 1.5
  )

boundary_plot +
  geom_sf(
    data = st_difference(streams, boundary),
    color = "blue",
    size = 1.5
  )
```

Different spatial geometries intersect in different ways. Above, we saw that the intersection of streams (which are `LINESTRINGS`) and the boundary (which is a `POLYGON`) produced `LINestring` geometries. Below, we compute the intersection of the streams with the trails that exist at MacLeish. The trails are also `LINESTRING` geometries, and the intersection of two `LINESTRING` geometries produces a set of `POINT` geometries.

```
trails <- macleish_layers %>%
  pluck("trails")

st_intersection(trails, streams)

Simple feature collection with 10 features and 3 fields
geometry type:  GEOMETRY
dimension:      XY
bbox:           xmin: -72.7 ymin: 42.4 xmax: -72.7 ymax: 42.5
geographic CRS: WGS 84
#> #>   name  color Id          geometry
#> #>   entry trail  - 3      POINT (-72.7 42.4)
#> #> Eastern Loop Blue 3      POINT (-72.7 42.5)
#> #> Snowmobile Trail <NA> 3 MULTIPOLY ((-72.7 42.5), (...))
#> #> Driveway        <NA> 3      POINT (-72.7 42.4)
#> #> Western Loop   Red 3      POINT (-72.7 42.4)
#> #> Porcupine Trail White 3     POINT (-72.7 42.5)
#> #> Western Loop   Red 3      POINT (-72.7 42.5)
#> #> Vernal Pool Loop Yellow 3    POINT (-72.7 42.4)
#> #> Poplar Hill Road Road 3     POINT (-72.7 42.5)
#> #> Snowmobile Trail <NA> 3      POINT (-72.7 42.5)
```

Note that one of the features is a `MULTIPOINT`. Occasionally, a trail might intersect a stream in more than one place (resulting in a `MULTIPOINT` geometry). This occurs here, where the Snowmobile Trail intersects one of the stream segments in two different places. To clean this up, we first use the `st_cast()` function to convert everything to `MULTIPOINT`, and then

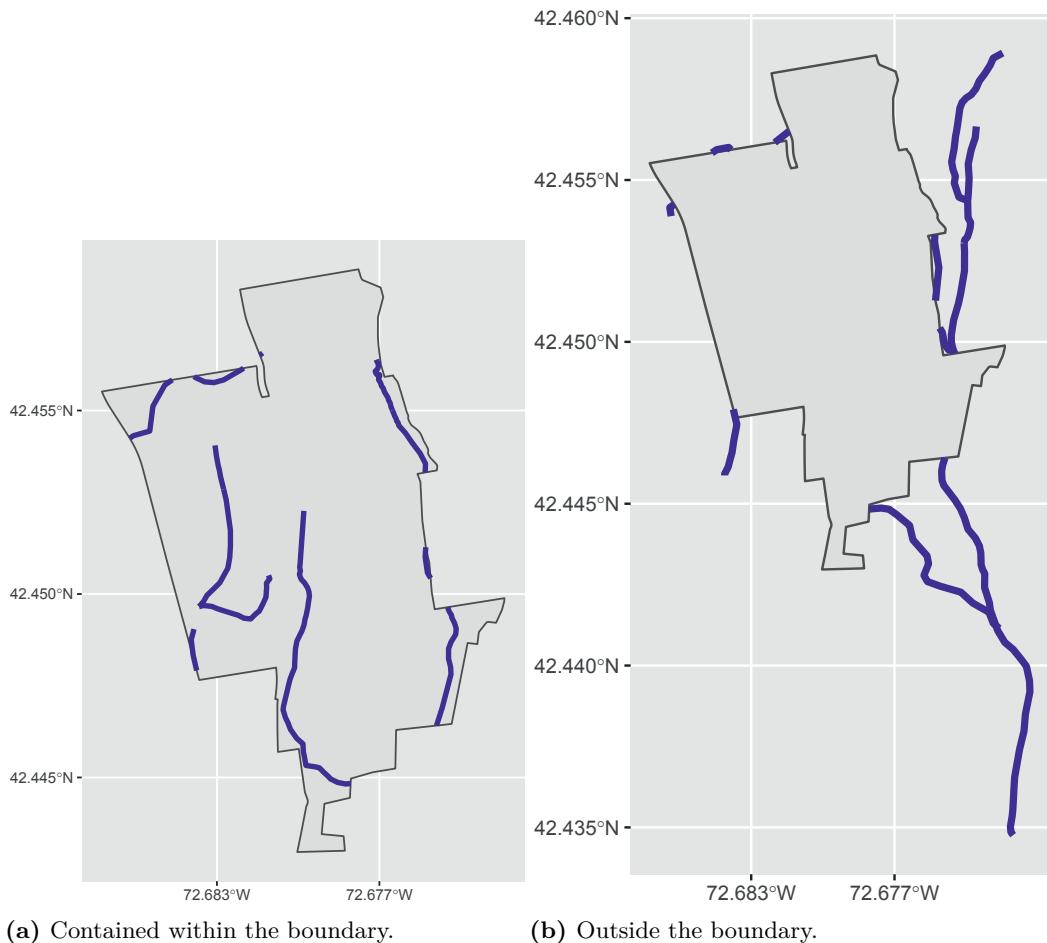


Figure 18.4: Streams on the MacLeish property.

cast everything to POINT. (We can't go straight to POINT because we start with a mixture of POINTs and MULTIPOLYNGs.)

```
bridges <- st_intersection(trails, streams) %>%
  st_cast("MULTIPOINT") %>%
  st_cast("POINT")

nrow(bridges)
```

```
[1] 11
```

Note that we now have 11 features instead of 10. In this case, the intersections of trails and streams has a natural interpretation: these must be bridges of some type! How else could the trail continue through the stream? Figure 18.5 shows the trails, the streams, and these “bridges” (some of the points are hard to see because they partially overlap).

```
boundary_plot +
  geom_sf(data = trails, color = "brown", size = 1.5) +
  geom_sf(data = streams, color = "blue", size = 1.5) +
  geom_sf(data = bridges, pch = 21, fill = "yellow", size = 3)
```

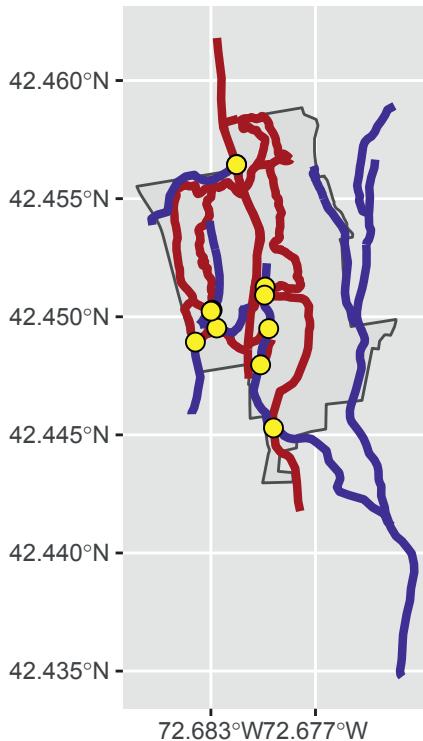


Figure 18.5: Bridges on the MacLeish property where trails and streams intersect.

18.2 Geospatial aggregation

In [Section 18.1.2](#), we saw how we can split `MULTIPOINT` geometries into `POINT` geometries. This was, in a sense, geospatial disaggregation. Here, we consider the perhaps more natural behavior of spatial aggregation.

Just as we saw previously that the intersection of different geometries can produce different resulting geometries, so too will different geometries aggregate in different ways. For example, `POINT` geometries can be aggregated into `MULTIPOINT` geometries.

The `sf` package implements spatial aggregation using the same `group_by()` and `summarize()` function that you learned in [Chapter 4](#). The only difference is that we might have to specify *how* we want the spatial layers to be aggregated. The default aggregation method is `st_union()`, which makes sense for most purposes.

Note that the `trails` layer is broken into segments: the Western Loop is comprised of three different features.

`trails`

```
Simple feature collection with 15 features and 2 fields
geometry type:  LINESTRING
```

```

dimension:      XY
bbox:           xmin: -72.7 ymin: 42.4 xmax: -72.7 ymax: 42.5
geographic CRS: WGS 84
First 10 features:
  name color          geometry
1 Porcupine Trail White LINESTRING (-72.7 42.5, -72...
2 Western Loop Red LINESTRING (-72.7 42.5, -72...
3 Poplar Hill Road Road LINESTRING (-72.7 42.5, -72...
4 Vernal Pool Loop Yellow LINESTRING (-72.7 42.4, -72...
5 Eastern Loop Blue LINESTRING (-72.7 42.5, -72...
6 Western Loop Red LINESTRING (-72.7 42.5, -72...
7 Western Loop Red LINESTRING (-72.7 42.4, -72...
8 entry trail - LINESTRING (-72.7 42.4, -72...
9 Eastern Loop Blue LINESTRING (-72.7 42.5, -72...
10 Easy Out Red LINESTRING (-72.7 42.5, -72...

```

Which trail is the longest? We know we can compute the length of the features with `st_length()`, but then we'd have to add up the lengths of each segment. Instead, we can aggregate the segments and do the length computation on the full trails.

```

trails_full <- trails %>%
  group_by(name) %>%
  summarize(num_segments = n()) %>%
  mutate(trail_length = st_length(geometry)) %>%
  arrange(desc(trail_length))

```

although coordinates are longitude/latitude, `st_union` assumes that they are planar
 although coordinates are longitude/latitude, `st_union` assumes that they are planar
 although coordinates are longitude/latitude, `st_union` assumes that they are planar
 although coordinates are longitude/latitude, `st_union` assumes that they are planar
 although coordinates are longitude/latitude, `st_union` assumes that they are planar
 although coordinates are longitude/latitude, `st_union` assumes that they are planar
 although coordinates are longitude/latitude, `st_union` assumes that they are planar
 although coordinates are longitude/latitude, `st_union` assumes that they are planar
 although coordinates are longitude/latitude, `st_union` assumes that they are planar
 although coordinates are longitude/latitude, `st_union` assumes that they are planar
`trails_full`

```

Simple feature collection with 9 features and 3 fields
geometry type:  GEOMETRY
dimension:      XY
bbox:           xmin: -72.7 ymin: 42.4 xmax: -72.7 ymax: 42.5
geographic CRS: WGS 84
# A tibble: 9 x 4
  name    num_segments          geometry  trail_length
  <fct>     <int>        <GEOMETRY [°]>    [m]
1 Snowmob~      2 MULTILINESTRING (((-72.7 42.5, -72.7 4~   2575
2 Eastern ~     2 MULTILINESTRING (((-72.7 42.5, -72.7 4~   1940
3 Western ~     3 MULTILINESTRING (((-72.7 42.5, -72.7 4~   1351
4 Poplar H~     2 MULTILINESTRING (((-72.7 42.5, -72.7 4~   1040
5 Porcupin~     1 LINESTRING (-72.7 42.5, -72.7 42.5, --    700
6 Vernal P~     1 LINESTRING (-72.7 42.4, -72.7 42.4, --    361
7 entry tr~     1 LINESTRING (-72.7 42.4, -72.7 42.4, --    208

```

8 Driveway	1 LINESTRING (-72.7 42.4, -72.7 42.4, --)	173
9 Easy Out	2 MULTILINESTRING ((-72.7 42.5, -72.7 4~	136

18.3 Geospatial joins

In [Section 17.4.3](#), we show how the `inner_join()` function can be used to merge geospatial data with additional data. This works because the geospatial data was stored in an `sf` object, which is also a data frame. In that case, since the second data frame was not spatial, by necessity the key on which the join was performed was a non-spatial attribute.

A geospatial join is a fundamentally different type of operation, in which *both* data frames are geospatial, and the joining key is a geospatial attribute. This operation is implemented by the `st_join()` function, which behaves similarly to the `inner_join()` function, but with some additional complexities due to the different nature of its task.

To illustrate this, we consider the question of in which type of forest the two campsites at MacLeish lie (see [Figure 18.6](#)).

```
forests <- macleish_layers %>%
  pluck("forests")

camp_sites <- macleish_layers %>%
  pluck("camp_sites")

boundary_plot +
  geom_sf(data = forests, fill = "green", alpha = 0.1) +
  geom_sf(data = camp_sites, size = 4) +
  geom_sf_label(
    data = camp_sites, aes(label = name),
    nudge_y = 0.001
  )
```

It is important to note that this question is inherently spatial. There simply is no variable between the `forests` layer and the `camp_sites` layer that would allow you to connect them other than their geospatial location.

Like `inner_join()`, the `st_join()` function takes two data frames as its first two arguments. There is no `st_left_join()` function, but instead `st_join()` takes a `left` argument, that is set to `TRUE` by default. Finally, the `join` argument takes a predicate function that determines the criteria for whether the spatial features match. The default is `st_intersects()`, but here we employ `st_within()`, since we want the `POINT` geometries of the camp sites to lie *within* the `POLYGON` geometries of the forests.

```
st_join(camp_sites, forests, left = FALSE, join = st_within) %>%
  select(name, type)
```

```
Simple feature collection with 2 features and 2 fields
geometry type:  POINT
dimension:      XY
bbox:           xmin: -72.7 ymin: 42.5 xmax: -72.7 ymax: 42.5
geographic CRS: WGS 84
```

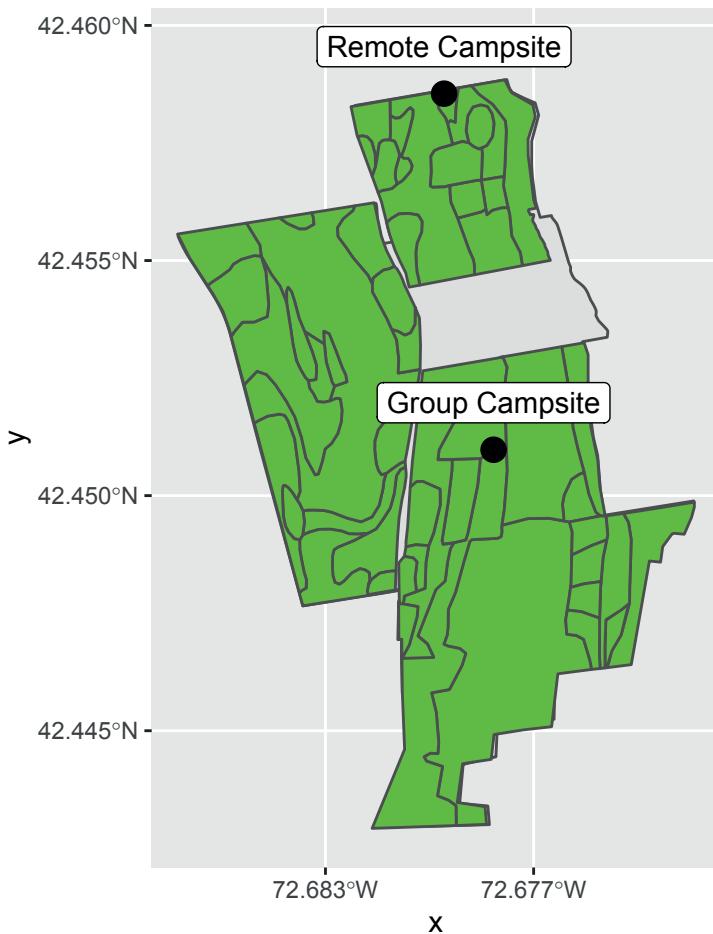


Figure 18.6: The MacLeish property has two campsites and many different types of forest.

```
# A tibble: 2 x 3
  name      type            geometry
  <chr>     <fct>           <POINT [°]>
1 Group Campsite Old Field White Pine Forest (-72.7 42.5)
2 Remote Campsite Sugar Maple Forest       (-72.7 42.5)
```

We see that the Group Campsite is in a *Eastern White Pine* forest, while the Remote Campsite is in a *Sugar Maple* forest.

18.4 Extended example: Trail elevations at MacLeish

Many hiking trails provide trail elevation maps (or elevation profiles) that depict changes in elevation along the trail. These maps can help hikers understand the interplay between the uphill and downhill segments of the trail and where they occur along their hike.

More formally, various trail railing systems exist to numerically score the difficulty of a hike. *Shenandoah National Park* uses this simple trail rating system:

$$\text{rating} = \sqrt{\text{gain} \cdot 2 \cdot \text{distance}}$$

A rating below 50 corresponds to the easiest class of hike.

In this example, we will construct an elevation profile and compute the trail rating for the longest trail at MacLeish. The **macleish** package contains elevation contours that are 30 feet apart. These are relatively sparse, but they will suffice for our purposes.

```
elevations <- macleish_layers %>%
  pluck("elevation")
```

First, we leverage the spatial aggregation work that we did previously to isolate the longest trail.

```
longest_trail <- trails_full %>%
  head(1)
longest_trail
```

```
Simple feature collection with 1 feature and 3 fields
geometry type:  MULTILINESTRING
dimension:      XY
bbox:           xmin: -72.7 ymin: 42.4 xmax: -72.7 ymax: 42.5
geographic CRS: WGS 84
# A tibble: 1 x 4
  name    num_segments          geometry  trail_length
  <fct>     <int>   <MULTILINESTRING [°]>      [m]
1 Snowmob~       2 ((-72.7 42.5, -72.7 42.5, -72.7 42.5,~      2575
```

Next, we compute the geospatial intersection between the trails and the elevation contours, both of which are **LINESTRING** geometries. This results in **POINTS**, but just as we saw above, there are several places where the trail crosses the same contour line more than once. If you think about walking up a hill and then back down the other side, this should make sense (see [Figure 18.7](#)). These multiple intersections result in **MULTIPOINT** geometries. We unravel these just as we did before: by casting everything to **MULTIPOINT** and then casting everything back to **POINT**.

```
trail_elevations <- longest_trail %>%
  st_intersection(elevations) %>%
  st_cast("MULTIPOINT") %>%
  st_cast("POINT")
```

[Figure 18.7](#) reveals that the Snowmobile trail starts near the southernmost edge of the property at about 750 feet above sea level, snakes along a ridge at 780 feet, before climbing to the local peak at 870 feet, and finally descending the back side of the hill near the northern border of the property.

```
boundary_plot +
  geom_sf(data = elevations, color = "dark gray") +
  geom_sf(data = longest_trail, color = "brown", size = 1.5) +
  geom_sf(data = trail_elevations, fill = "yellow", pch = 21, size = 3) +
  geom_sf_label(
    data = trail_elevations,
```

```
aes(label = CONTOUR_FT),
  hjust = "right",
  size = 2.5,
  nudge_x = -0.0005
)
```

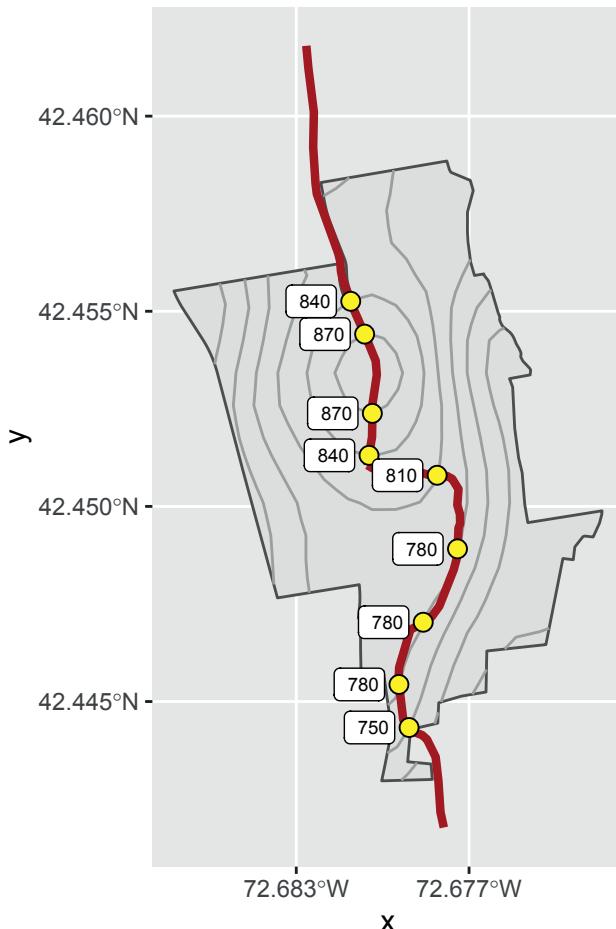


Figure 18.7: The Snowmobile Trail at MacLeish, with contour lines depicted.

Finally, we need to put the features in order, so that we can compute the distance from the start of the trail. Luckily, in this case the trail goes directly south to north, so that we can use the latitude coordinate as an ordering variable.

In this case, we use `st_distance()` to compute the geodesic distances between the elevation contours. This function returns a $n \times n$ matrix with all of the pairwise distances, but since we only want the distances from the southernmost point (which is the first element), we only select the first column of the resulting matrix.

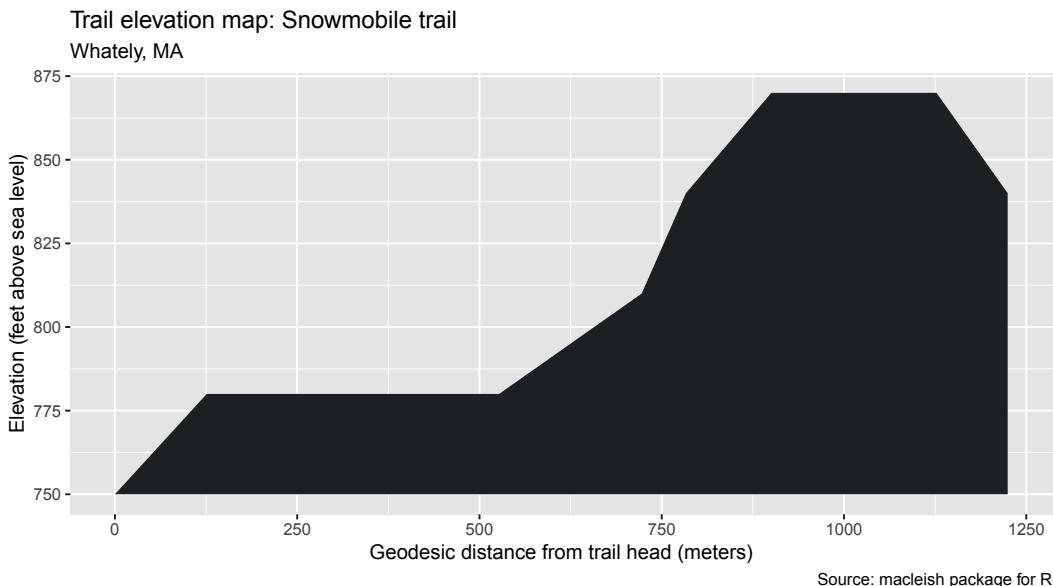
To compute the actual distances (i.e., along the trail) we would have to split the trail into pieces. We leave this as an exercise.

```
trail_elevations <- trail_elevations %>%
  mutate(lat = st_coordinates(geometry)[, 2]) %>%
```

```
arrange(lat) %>%
  mutate(distance_from_start = as.numeric(st_distance(geometry)[, 1]))
```

Figure 18.8 shows our elevation profile for the Snowmobile trail.

```
ggplot(trail_elevations, aes(x = distance_from_start)) +
  geom_ribbon(aes(ymax = CONTOUR_FT, ymin = 750)) +
  scale_y_continuous("Elevation (feet above sea level)") +
  scale_x_continuous("Geodesic distance from trail head (meters)") +
  labs(
    title = "Trail elevation map: Snowmobile trail",
    subtitle = "Whately, MA",
    caption = "Source: macleish package for R"
  )
```



Source: macleish package for R

Figure 18.8: Trail elevation map for the Snowmobile trail at MacLeish.

With a rating under 20, this trail rates as one of the easiest according to the Shenandoah system.

```
trail_elevations %>%
  summarize(
    gain = max(CONTOUR_FT) - min(CONTOUR_FT),
    trail_length = max(units::set_units(trail_length, "miles")),
    rating = sqrt(gain * 2 * as.numeric(trail_length))
  )
```

although coordinates are longitude/latitude, st_union assumes that they are planar

```
Simple feature collection with 1 feature and 3 fields
geometry type:  MULTIPOINT
dimension:      XY
bbox:           xmin: -72.7 ymin: 42.4 xmax: -72.7 ymax: 42.5
geographic CRS: WGS 84
```

```
# A tibble: 1 x 4
  gain trail_length rating      geometry
  <dbl>      <dbl>   <dbl> <MULTIPOINT [°]>
1    120       1.6     19.6 ((-72.7 42.5), (-72.7 42.5), (-72.7 42.5), (-72.7 42.5))
```

18.5 Further resources

Lovelace et al. (2019) and Engel (2019) were both helpful in updating this material to take advantage of `sf`.

18.6 Exercises

Problem 1 (Medium): The `Violations` data frame in the `mdsr` package contains information on violations noted in Board of Health inspections of New York City restaurants. These data contain spatial information in the form of addresses and zip codes.

- Use the `geocode` function in `tidygeocoder` to obtain spatial coordinates for these restaurants.
- Using the spatial coordinates you obtained in the previous exercise, create an informative static map using `ggspatial` that illustrates the nature and extent of restaurant violations in New York City.
- Using the spatial coordinates you obtained in the previous exercises, create an informative interactive map using `leaflet` that illustrates the nature and extent of restaurant violations in New York City.

Problem 2 (Medium):

- Use the spatial data in the `macleish` package and `ggspatial` to make an informative static map of the MacLeish Field Station property.
- Use the spatial data in the `macleish` package and `leaflet` to make an informative interactive map of the MacLeish Field Station property.

Problem 3 (Hard): GIS data in the form of shapefiles is all over the Web. Government agencies are particularly good sources for these. The following code downloads bike trail data in Massachusetts from MassGIS. Use `bike_trails` to answer the following questions:

```
if (!file.exists("./biketrails_arc.zip")) {
  part1 <- "http://download.massgis.digital.mass.gov/"
  part2 <- "shapefiles/state/biketrails_arc.zip"
  url <- paste(part1, part2, sep = "")
  local_file <- basename(url)
  download.file(url, destfile = local_file)
  unzip(local_file, exdir = "./biketrails/")
}
```

```
library(sf)
dsn <- path.expand("./biketrails/biketrails_arc")
st_layers(dsn)

Driver: ESRI Shapefile
Available layers:
  layer_name geometry_type features fields
1 biketrails_arc   Line String      272       13
bike_trails <- read_sf(dsn)
```

- a. How many distinct bike trail segments are there?
- b. What is the longest individual bike trail segment?
- c. How many segments are associated with the Norwottuck Rail Trail?
- d. Among all of the named trails (which may have multiple features), which one has the longest total length?
- e. The bike trails are in a Lambert conformal conic projection. Note that the units of the coordinates are very different from lat/long. In order to get these data onto our leaflet map, we need to re-project them. Convert the bike trails to EPSG:4326, and create a leaflet map.
- f. Color-code the bike trails based on their length, and add an informative legend to the plot.

Problem 4 (Hard): The MacLeish snowmobile trail map generated in the book is quite rudimentary. Generate your own map that improves upon the aesthetics and information content.

18.7 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/geospatial-II.html#geospatialII-online-exercises>

Text as data

So far, we have focused primarily on numerical data, but there is a whole field of research that focuses on unstructured textual data. Fields such as *natural language processing* and *computational linguistics* work directly with text documents to extract meaning algorithmically. Not surprisingly, the fact that computers are really good at storing text, but not very good at understanding it, whereas humans are really good at understanding text, but not very good at storing it, is a fundamental challenge.

Processing text data requires an additional set of wrangling skills. In this chapter, we will introduce how text can be ingested, how *corpora* (collections of text documents) can be created, sentiments extracted, patterns described, and how *regular expressions* can be used to automate searches that would otherwise be excruciatingly labor-intensive.

19.1 Regular expressions using *Macbeth*

As noted previously, working with textual data requires new tools. In this section, we introduce the powerful grammar of regular expressions.

19.1.1 Parsing the text of the *Scottish play*

Project Gutenberg contains the full-text for all of William Shakespeare's plays. In this example, we will use text mining techniques to explore *The Tragedy of Macbeth*. The text can be downloaded directly from Project Gutenberg. Alternatively, the `Macbeth_raw` object is also included in the `mdsr` package.

```
library(tidyverse)
library(mdsr)
macbeth_url <- "http://www.gutenberg.org/cache/epub/1129/pg1129.txt"
Macbeth_raw <- RCurl::getURL(macbeth_url)

data(Macbeth_raw)
```

Note that `Macbeth_raw` is a *single* string of text (i.e., a character vector of length 1) that contains the entire play. In order to work with this, we want to split this single string into a vector of strings using the `str_split()` function from the `stringr`. To do this, we just have to specify the end-of-line character(s), which in this case are: `\r\n`.

```
# str_split returns a list: we only want the first element
macbeth <- Macbeth_raw %>%
  str_split("\r\n") %>%
```

```
pluck(1)
length(macbeth)
```

```
[1] 3194
```

Now let's examine the text. Note that each speaking line begins with two spaces, followed by the speaker's name in capital letters.

```
macbeth[300:310]
```

```
[1] "meeting a bleeding Sergeant."
[2] ""
[3] " DUNCAN. What bloody man is that? He can report,"
[4] " As seemeth by his plight, of the revolt"
[5] " The newest state."
[6] " MALCOLM. This is the sergeant"
[7] " Who like a good and hardy soldier fought"
[8] " 'Gainst my captivity. Hail, brave friend!"
[9] " Say to the King the knowledge of the broil"
[10] " As thou didst leave it."
[11] " SERGEANT. Doubtful it stood,"
```

The power of text mining comes from quantifying ideas embedded in the text. For example, how many times does the character Macbeth speak in the play? Think about this question for a moment. If you were holding a physical copy of the play, how would you compute this number? Would you flip through the book and mark down each speaking line on a separate piece of paper? Is your algorithm scalable? What if you had to do it for *all* characters in the play, and not just Macbeth? What if you had to do it for *all 37* of Shakespeare's plays? What if you had to do it for all plays written in English?

Naturally, a computer cannot read the play and figure this out, but we can find all instances of Macbeth's speaking lines by cleverly counting patterns in the text.

```
macbeth_lines <- macbeth %>%
  str_subset(" MACBETH")
length(macbeth_lines)
```

```
[1] 147
```

```
head(macbeth_lines)
```

```
[1] " MACBETH, Thane of Glamis and Cawdor, a general in the King's"
[2] " MACBETH. So foul and fair a day I have not seen."
[3] " MACBETH. Speak, if you can. What are you?"
[4] " MACBETH. Stay, you imperfect speakers, tell me more."
[5] " MACBETH. Into the air, and what seem'd corporal melted"
[6] " MACBETH. Your children shall be kings."
```

The `str_subset()` function works using a *needle* in a *haystack* paradigm, wherein the first argument is the character vector in which you want to find patterns (i.e., the haystack) and the second argument is the *regular expression* (or pattern) you want to find (i.e., the needle). Alternatively, `str_which()` returns the *indices* of the haystack in which the needles were found. By changing the needle, we find different results:

```
macbeth %>%
```

```
str_subset(" MACDUFF") %>%
length()
```

```
[1] 60
```

The `str_detect()` function—which we use in the example in the next section—uses the same syntax but returns a logical vector as long as the haystack. Thus, while the length of the vector returned by `str_subset()` is the number of matches, the length of the vector returned by `str_detect()` is always the same as the length of the haystack vector.¹

```
macbeth %>%
str_subset(" MACBETH") %>%
length()
```

```
[1] 147
```

```
macbeth %>%
str_detect(" MACBETH") %>%
length()
```

```
[1] 3194
```

To extract the piece of each matching line that actually matched, use the `str_extract()` function from the `stringr` package.

```
pattern <- " MACBETH"
macbeth %>%
str_subset(pattern) %>%
str_extract(pattern) %>%
head()
```

```
[1] " MACBETH" " MACBETH" " MACBETH" " MACBETH" " MACBETH" " MACBETH"
```

Above, we use a literal string (e.g., “MACBETH”) as our needle to find exact matches in our haystack. This is the simplest type of pattern for which we could have searched, but the needle that `str_extract()` searches for can be any regular expression.

Regular expression syntax is very powerful and as a result, can become very complicated. Still, regular expressions are a grammar, so that learning a few basic concepts will allow you to build more efficient searches.

- **Metacharacters:** `.` is a *metacharacter* that matches any character. Note that if you want to search for the literal value of a metacharacter (e.g., a period), you have to escape it with a backslash. To use the pattern in R, two backslashes are needed. Note the difference in the results below.

```
macbeth %>%
str_subset("MAC.") %>%
head()
```

```
[1] "MACHINE READABLE COPIES MAY BE DISTRIBUTED SO LONG AS SUCH COPIES"
[2] "MACHINE READABLE COPIES OF THIS ETEXT, SO LONG AS SUCH COPIES"
[3] "WITH PERMISSION. ELECTRONIC AND MACHINE READABLE COPIES MAY BE"
[4] "THE TRAGEDY OF MACBETH"
```

¹`str_subset()`, `str_which()`, and `str_detect()` replicate the functionality of the base R functions `grep()` and `grepl()`, but with a more consistent and pipeable syntax.

```
[5] " MACBETH, Thane of Glamis and Cawdor, a general in the King's"
[6] " LADY MACBETH, his wife"

macbeth %>%
  str_subset("MACBETH\\.") %>%
  head()
```

```
[1] " MACBETH. So foul and fair a day I have not seen."
[2] " MACBETH. Speak, if you can. What are you?"
[3] " MACBETH. Stay, you imperfect speakers, tell me more."
[4] " MACBETH. Into the air, and what seem'd corporal melted"
[5] " MACBETH. Your children shall be kings."
[6] " MACBETH. And Thane of Cawdor too. Went it not so?"
```

- **Character sets:** Use brackets to define sets of characters to match. This pattern will match any lines that contain MAC followed by any capital letter other than A. It will match MACBETH but not MACALESTER.

```
macbeth %>%
  str_subset("MAC[B-Z]") %>%
  head()
```

```
[1] "MACHINE READABLE COPIES MAY BE DISTRIBUTED SO LONG AS SUCH COPIES"
[2] "MACHINE READABLE COPIES OF THIS ETEXT, SO LONG AS SUCH COPIES"
[3] "WITH PERMISSION. ELECTRONIC AND MACHINE READABLE COPIES MAY BE"
[4] "THE TRAGEDY OF MACBETH"
[5] " MACBETH, Thane of Glamis and Cawdor, a general in the King's"
[6] " LADY MACBETH, his wife"
```

- **Alternation:** To search for a few specific alternatives, use the | wrapped in parentheses. This pattern will match any lines that contain either MACB or MACD.

```
macbeth %>%
  str_subset("MAC(B|D)") %>%
  head()
```

```
[1] "THE TRAGEDY OF MACBETH"
[2] " MACBETH, Thane of Glamis and Cawdor, a general in the King's"
[3] " LADY MACBETH, his wife"
[4] " MACDUFF, Thane of Fife, a nobleman of Scotland"
[5] " LADY MACDUFF, his wife"
[6] " MACBETH. So foul and fair a day I have not seen."
```

- **Anchors:** Use ^ to anchor a pattern to the beginning of a piece of text, and \$ to anchor it to the end.

```
macbeth %>%
  str_subset("^ MAC[B-Z]") %>%
  head()
```

```
[1] " MACBETH, Thane of Glamis and Cawdor, a general in the King's"
[2] " MACDUFF, Thane of Fife, a nobleman of Scotland"
[3] " MACBETH. So foul and fair a day I have not seen."
[4] " MACBETH. Speak, if you can. What are you?"
[5] " MACBETH. Stay, you imperfect speakers, tell me more."
[6] " MACBETH. Into the air, and what seem'd corporal melted"
```

- **Repetitions:** We can also specify the number of times that we want certain patterns to occur: ? indicates zero or one time, * indicates zero or more times, and + indicates one or more times. This quantification is applied to the previous element in the pattern—in this case, a space.

```
macbeth %>%
  str_subset("^\s?MAC[B-Z]") %>%
  head()

[1] "MACHINE READABLE COPIES MAY BE DISTRIBUTED SO LONG AS SUCH COPIES"
[2] "MACHINE READABLE COPIES OF THIS ETEXT, SO LONG AS SUCH COPIES"

macbeth %>%
  str_subset("^\s*MAC[B-Z]") %>%
  head()

[1] "MACHINE READABLE COPIES MAY BE DISTRIBUTED SO LONG AS SUCH COPIES"
[2] "MACHINE READABLE COPIES OF THIS ETEXT, SO LONG AS SUCH COPIES"
[3] " MACBETH, Thane of Glamis and Cawdor, a general in the King's"
[4] " MACDUFF, Thane of Fife, a nobleman of Scotland"
[5] " MACBETH. So foul and fair a day I have not seen."
[6] " MACBETH. Speak, if you can. What are you?"

macbeth %>%
  str_subset("^\s+MAC[B-Z]") %>%
  head()

[1] " MACBETH, Thane of Glamis and Cawdor, a general in the King's"
[2] " MACDUFF, Thane of Fife, a nobleman of Scotland"
[3] " MACBETH. So foul and fair a day I have not seen."
[4] " MACBETH. Speak, if you can. What are you?"
[5] " MACBETH. Stay, you imperfect speakers, tell me more."
[6] " MACBETH. Into the air, and what seem'd corporal melted"
```

Combining these basic rules can automate incredibly powerful and sophisticated searches and are an increasingly necessary tool in every data scientist's toolbox.

Pro Tip 47. *Regular expressions are a powerful and commonly-used tool. They are implemented in many programming languages. Developing a working understanding of regular expressions will pay off in text wrangling.*

19.1.2 Life and death in *Macbeth*

Can we use these techniques to analyze the speaking patterns in Macbeth? Are there things we can learn about the play simply by noting who speaks when? Four of the major characters in *Macbeth* are the titular character, his wife Lady Macbeth, his friend Banquo, and Duncan, the King of Scotland.

We might learn something about the play by knowing when each character speaks as a function of the line number in the play. We can retrieve this information using `str_detect()`.

```
macbeth_chars <- tribble(
  ~name, ~regexp,
  "Macbeth", " MACBETH\\\\.",
  "Lady Macbeth", " LADY MACBETH\\\\.,"
```

```

  "Banquo", "  BANQUO\\.",  

  "Duncan", "  DUNCAN\\.",  

) %>%  

  mutate(speaks = map(regexp, str_detect, string = macbeth))

```

However, for plotting purposes we will want to convert these logical vectors into numeric vectors, and tidy up the data. Since there is unwanted text at the beginning and the end of the play text, we will also restrict our analysis to the actual contents of the play (which occurs from line 218 to line 3172).

```

speaker_freq <- macbeth_chars %>%
  unnest(cols = speaks) %>%
  mutate(
    line = rep(1:length(macbeth), 4),
    speaks = as.numeric(speaks)
  ) %>%
  filter(line > 218 & line < 3172)
glimpse(speaker_freq)

```

```

Rows: 11,812
Columns: 4
$ name   <chr> "Macbeth", "Macbeth", "Macbeth", "Macbeth", "Macbeth", "...  

$ regexp <chr> " MACBETH\\.", " MACBETH\\.", " MACBETH\\.", " MACBE...  

$ speaks <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...  

$ line   <int> 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 2...

```

Before we create the plot, we will gather some helpful contextual information about when each Act begins.

```

acts <- tibble(  

  line = str(which(macbeth, "ACT [I|V]+")),
  line_text = str_subset(macbeth, "ACT [I|V]+"),
  labels = str_extract(line_text, "ACT [I|V]+")
)

```

Finally, [Figure 19.1](#) illustrates how King Duncan of Scotland is killed early in Act II (never to speak again), with Banquo to follow in Act III. Soon afterwards in Act IV, Lady Macbeth—overcome by guilt over the role she played in Duncan's murder—kills herself. The play and Act V conclude with a battle in which Macbeth is killed.

```

ggplot(data = speaker_freq, aes(x = line, y = speaks)) +  

  geom_smooth(  

    aes(color = name), method = "loess",
    se = FALSE, span = 0.4
  ) +
  geom_vline(  

    data = acts,
    aes(xintercept = line),
    color = "darkgray", lty = 3
  ) +
  geom_text(  

    data = acts,
    aes(y = 0.085, label = labels),

```

```

    hjust = "left", color = "darkgray"
) +
ylim(c(0, NA)) +
xlab("Line Number") +
ylab("Proportion of Speeches") +
scale_color_brewer(palette = "Set2")

```

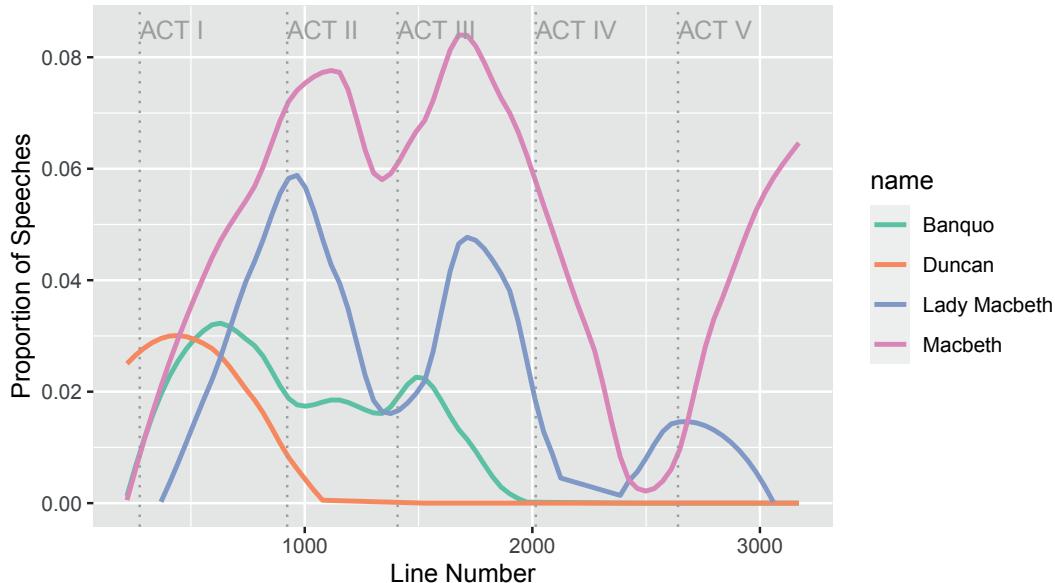


Figure 19.1: Speaking parts for four major characters. Duncan is killed early in the play and never speaks again.

19.2 Extended example: Analyzing textual data from arXiv.org

The *arXiv* (pronounced “archive”) is a fast-growing electronic repository of preprints of scientific papers from many disciplines. The **aRxiv** package provides an application programming interface (API) to the files and metadata available on the arXiv. We will use the 1,089 papers that matched the search term “data science” in the repository as of August, 2020 to try to better understand the discipline. The following code was used to generate this file.

```

library(aRxiv)
DataSciencePapers <- arxiv_search(
  query = '"Data Science"',
  limit = 20000,
  batchsize = 100
)

```

We have also included the resulting data frame **DataSciencePapers** in the **mdsr** package, so to use this selection of papers downloaded from the archive, you can simply load it (this will avoid unduly straining the arXiv server).

```
data(DataSciencePapers)
```

Note that there are two columns in this data set (`submitted` and `updated`) that are clearly storing dates, but they are stored as character vectors.

```
glimpse(DataSciencePapers)
```

```
Rows: 1,089
Columns: 15
$ id              <chr> "astro-ph/0701361v1", "0901.2805v1", "0901.311...
$ submitted       <chr> "2007-01-12 03:28:11", "2009-01-19 10:38:33", ...
$ updated         <chr> "2007-01-12 03:28:11", "2009-01-19 10:38:33", ...
$ title           <chr> "How to Make the Dream Come True: The Astronom...
$ abstract        <chr> " Astronomy is one of the most data-intensive...
$ authors          <chr> "Ray P Norris", "Heinz Andernach", "O. V. Verk...
$ affiliations    <chr> "", "", "Special Astrophysical Observatory, Ni...
$ link_abstract   <chr> "http://arxiv.org/abs/astro-ph/0701361v1", "ht...
$ link_pdf         <chr> "http://arxiv.org/pdf/astro-ph/0701361v1", "ht...
$ link_doi         <chr> "", "http://dx.doi.org/10.2481/dsj.8.41", "htt...
$ comment          <chr> "Submitted to Data Science Journal Presented a...
$ journal_ref     <chr> "", "", "", "", "EPJ Data Science, 1:9, 2012",...
$ doi              <chr> "", "10.2481/dsj.8.41", "10.2481/dsj.8.34", "...
$ primary_category <chr> "astro-ph", "astro-ph.IM", "astro-ph.IM", "ast...
$ categories       <chr> "astro-ph", "astro-ph.IM|astro-ph.CO", "astro-...
```

To make sure that **R** understands those variables as dates, we will once again use the **lubridate** package (see [Chapter 6](#)). After this conversion, **R** can deal with these two columns as measurements of time.

```
library(lubridate)
DataSciencePapers <- DataSciencePapers %>%
  mutate(
    submitted = lubridate::ymd_hms(submitted),
    updated = lubridate::ymd_hms(updated)
  )
glimpse(DataSciencePapers)
```

```
Rows: 1,089
Columns: 15
$ id              <chr> "astro-ph/0701361v1", "0901.2805v1", "0901.311...
$ submitted       <dttm> 2007-01-12 03:28:11, 2009-01-19 10:38:33, 200...
$ updated         <dttm> 2007-01-12 03:28:11, 2009-01-19 10:38:33, 200...
$ title           <chr> "How to Make the Dream Come True: The Astronom...
$ abstract        <chr> " Astronomy is one of the most data-intensive...
$ authors          <chr> "Ray P Norris", "Heinz Andernach", "O. V. Verk...
$ affiliations    <chr> "", "", "Special Astrophysical Observatory, Ni...
$ link_abstract   <chr> "http://arxiv.org/abs/astro-ph/0701361v1", "ht...
$ link_pdf         <chr> "http://arxiv.org/pdf/astro-ph/0701361v1", "ht...
$ link_doi         <chr> "", "http://dx.doi.org/10.2481/dsj.8.41", "htt...
$ comment          <chr> "Submitted to Data Science Journal Presented a...
$ journal_ref     <chr> "", "", "", "", "EPJ Data Science, 1:9, 2012",...
$ doi              <chr> "", "10.2481/dsj.8.41", "10.2481/dsj.8.34", "...
$ primary_category <chr> "astro-ph", "astro-ph.IM", "astro-ph.IM", "ast...
```

```
$ categories      <chr> "astro-ph", "astro-ph.IM|astro-ph.CO", "astro-...
```

We begin by examining the distribution of submission years. How has interest grown in data science?

```
mosaic::tally(~ year(submitted), data = DataSciencePapers)
```

```
year(submitted)
2007 2009 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020
 1     3     3     7    15    25    52    94   151   187   313   238
```

We see that the first paper was submitted in 2007, but that submissions have increased considerably since then.

Let's take a closer look at one of the papers, in this case one that focuses on causal inference.

```
DataSciencePapers %>%
  filter(id == "1809.02408v2") %>%
  glimpse()
```

```
Rows: 1
Columns: 15
$ id              <chr> "1809.02408v2"
$ submitted       <dttm> 2018-09-07 11:26:51
$ updated         <dttm> 2019-03-05 04:38:35
$ title           <chr> "A Primer on Causality in Data Science"
$ abstract        <chr> " Many questions in Data Science are fundamen...
$ authors          <chr> "Hachem Saddiki|Laura B. Balzer"
$ affiliations    <chr> ""
$ link_abstract   <chr> "http://arxiv.org/abs/1809.02408v2"
$ link_pdf         <chr> "http://arxiv.org/pdf/1809.02408v2"
$ link_doi         <chr> ""
$ comment          <chr> "26 pages (with references); 4 figures"
$ journal_ref     <chr> ""
$ doi              <chr> ""
$ primary_category <chr> "stat.AP"
$ categories       <chr> "stat.AP|stat.ME|stat.ML"
```

We see that this is a primer on causality in data science that was submitted in 2018 and updated in 2019 with a primary category of `stat.AP`.

What fields are generating the most papers in our dataset? A quick glance at the `primary_category` variable reveals a cryptic list of fields and sub-fields starting alphabetically with astronomy.

```
DataSciencePapers %>%
  group_by(primary_category) %>%
  count() %>%
  head()
```

```
# A tibble: 6 x 2
# Groups:   primary_category [6]
  primary_category     n
  <chr>             <int>
1 astro-ph            1
2 astro-ph.CO         3
```

3 astro-ph.EP	1
4 astro-ph.GA	7
5 astro-ph.IM	20
6 astro-ph.SR	6

It may be more helpful to focus simply on the primary field (the part before the period). We can use a regular expression to extract only the primary field, which may contain a dash (-), but otherwise is all lowercase characters. Once we have this information extracted, we can `tally()` those primary fields.

```
DataSciencePapers <- DataSciencePapers %>%
  mutate(
    field = str_extract(primary_category, "^[a-z,-]+"),
  )
mosaic::tally(x = ~field, margins = TRUE, data = DataSciencePapers) %>%
  sort()
```

field	gr-qc	hep-ph	nucl-th	hep-th	econ	quant-ph	cond-mat	q-fin
	1	1	1	3	5	7	12	15
q-bio		eess	astro-ph	physics	math	stat	cs	Total
	16	29	38	62	103	150	646	1089

It appears that more than half ($646/1089 = 59\%$) of these papers come from computer science, while roughly one quarter come from mathematics and statistics.

19.2.1 Corpora

Text mining is often performed not just on one text document, but on a collection of many text documents, called a *corpus*. Can we use the arXiv.org papers to learn more about papers in data science?

The **tidytext** package provides a consistent and elegant approach to analyzing text data. The `unnest_tokens()` function helps prepare data for text analysis. It uses a *tokenizer* to split the text lines. By default the function maps characters to lowercase.

Here we use this function to count word frequencies for each of the papers (other options include N-grams, lines, or sentences).

```
library(tidytext)
DataSciencePapers %>%
  unnest_tokens(word, abstract) %>%
  count(id, word, sort = TRUE)
```

# A tibble: 120,330 x 3	id	word	n
	<chr>	<chr>	<int>
1	2003.11213v1	the	31
2	1508.02387v1	the	30
3	1711.10558v1	the	30
4	1805.09320v2	the	30
5	2004.04813v2	the	27
6	2007.08242v1	the	27
7	1711.09726v3	the	26

```

8 1805.11012v1 the      26
9 1909.10578v1 the      26
10 1404.5971v2 the     25
# ... with 120,320 more rows

```

We see that the word `the` is the most common word in many abstracts. This is not a particularly helpful insight. It's a common practice to exclude *stop words* such as `a`, `the`, and `you`. The `get_stopwords()` function from the `tidytext` package uses the `stopwords` package to facilitate this task. Let's try again.

```

arxiv_words <- DataSciencePapers %>%
  unnest_tokens(word, abstract) %>%
  anti_join(get_stopwords(), by = "word")

arxiv_words %>%
  count(id, word, sort = TRUE)

# A tibble: 93,559 x 3
  id       word     n
  <chr>    <chr>   <int>
1 2007.03606v1 data     20
2 1708.04664v1 data     19
3 1606.06769v1 traffic   17
4 1705.03451v2 data     17
5 1601.06035v1 models    16
6 1807.09127v2 job      16
7 2003.10534v1 data     16
8 1611.09874v1 ii       15
9 1808.04849v1 data     15
10 1906.03418v1 data    15
# ... with 93,549 more rows

```

We now see that the word `data` is, not surprisingly, the most common non-stop word in many of the abstracts.

It is convenient to save a variable (`abstract_clean`) with the abstract after removing stop-words and mapping all characters to lowercase.

```

arxiv_abstracts <- arxiv_words %>%
  group_by(id) %>%
  summarise(abstract_clean = paste(word, collapse = " "))

arxiv_papers <- DataSciencePapers %>%
  left_join(arxiv_abstracts, by = "id")

```

We can now see the before and after for the first part of the abstract of our previously selected paper.

```

single_paper <- arxiv_papers %>%
  filter(id == "1809.02408v2")
single_paper %>%
  pull(abstract) %>%
  strwrap() %>%
  head()

```

```
[1] "Many questions in Data Science are fundamentally causal in that our"
[2] "objective is to learn the effect of some exposure, randomized or"
[3] "not, on an outcome interest. Even studies that are seemingly"
[4] "non-causal, such as those with the goal of prediction or prevalence"
[5] "estimation, have causal elements, including differential censoring"
[6] "or measurement. As a result, we, as Data Scientists, need to"
```

```
single_paper %>%
  pull(abstract_clean) %>%
  strwrap() %>%
  head(4)
```

```
[1] "many questions data science fundamentally causal objective learn"
[2] "effect exposure randomized outcome interest even studies seemingly"
[3] "non causal goal prediction prevalence estimation causal elements"
[4] "including differential censoring measurement result data scientists"
```

19.2.2 Word clouds

At this stage, we have taken what was a coherent English abstract and reduced it to a collection of individual, non-trivial English words. We have transformed something that was easy for humans to read into *data*. Unfortunately, it is not obvious how we can learn from these data.

One rudimentary approach is to construct a *word cloud*—a kind of multivariate histogram for words. The **wordcloud** package can generate these graphical depictions of word frequencies.

```
library(wordcloud)
set.seed(1966)
arxiv_papers %>%
  pull(abstract_clean) %>%
  wordcloud(
    max.words = 40,
    scale = c(8, 1),
    colors = topo.colors(n = 30),
    random.color = TRUE
  )
```

Although word clouds such as the one shown in Figure 19.2 have limited abilities to convey meaning, they can be useful for quickly visualizing the prevalence of words in large corpora.

19.2.3 Sentiment analysis

Can we start to automate a process to discern some meaning from the text? The use of *sentiment analysis* is a simplistic but straightforward way to begin. A *lexicon* is a word list with associated sentiments (e.g., positivity, negativity) that have been labeled. A number of such lexicons have been created with such tags. Here is a sample of sentiment scores for one lexicon.

```
afinn <- get_sentiments("afinn")
afinn %>%
  slice_sample(n = 15) %>%
  arrange(desc(value))
```



Figure 19.2: A word cloud of terms that appear in the abstracts of arXiv papers on data science.

```
# A tibble: 15 x 2
  word      value
  <chr>     <dbl>
1 impress      3
2 joyfully     3
3 advantage     2
4 faith        1
5 grant        1
6 laugh        1
7 apologise   -1
8 lurk         -1
9 ghost        -1
10 deriding    -2
11 detention    -2
12 dirtiest     -2
13 embarrassment -2
14 mocks        -2
15 mournful    -2
```

For the AFINN (Nielsen, 2011) lexicon, each word is associated with an integer value, ranging from -5 to 5 .

We can join this lexicon with our data to calculate a sentiment score.

```
arxiv_words %>%
  inner_join(afinn, by = "word") %>%
  select(word, id, value)
```

```
# A tibble: 7,393 x 3
  word      id          value
  <chr>     <chr>        <dbl>
1 ambitious astro-ph/0701361v1    2
2 powerful  astro-ph/0701361v1    2
3 impotent   astro-ph/0701361v1   -2
4 like       astro-ph/0701361v1    2
5 agree      astro-ph/0701361v1    1
6 better     0901.2805v1         2
7 better     0901.2805v1         2
8 better     0901.2805v1         2
9 improve    0901.2805v1         2
10 support   0901.3118v2        2
# ... with 7,383 more rows

arxiv_sentiments <- arxiv_words %>%
  left_join(afinn, by = "word") %>%
  group_by(id) %>%
  summarize(
    num_words = n(),
    sentiment = sum(value, na.rm = TRUE),
    .groups = "drop"
  ) %>%
  mutate(sentiment_per_word = sentiment / num_words) %>%
  arrange(desc(sentiment))
```

Here we used `left_join()` to ensure that if no words in the abstract matched words in the lexicon, we will still have something to sum (in this case a number of NA's, which sum to 0). We can now add this new variable to our dataset of papers.

```
arxiv_papers <- arxiv_papers %>%
  left_join(arxiv_sentiments, by = "id")
arxiv_papers %>%
  skim(sentiment, sentiment_per_word)

-- Variable type: numeric -----
var           n    na   mean     sd     p0    p25    p50    p75
1 sentiment    1089    0 4.02    7.00   -26     0 4      8
2 sentiment_per_word 1089    0 0.0360 0.0633  -0.227    0 0.0347 0.0714
  p100
1 39
2 0.333
```

The average sentiment score of these papers is 4, but they range from -26 to 39. Surely, abstracts with more words might accrue a higher sentiment score. We can control for abstract length by dividing by the number of words. The paper with the highest sentiment score per word had a score of 0.333. Let's take a closer look at the most positive abstract.

```
most_positive <- arxiv_papers %>%
  filter(sentiment_per_word == max(sentiment_per_word)) %>%
  pull(abstract)
strwrap(most_positive)
```

```
[1] "Data science is creating very exciting trends as well as"
```

- [2] "significant controversy. A critical matter for the healthy"
- [3] "development of data science in its early stages is to deeply"
- [4] "understand the nature of data and data science, and to discuss the"
- [5] "various pitfalls. These important issues motivate the discussions"
- [6] "in this article."

We see a number of positive words (e.g., “exciting”, “significant”, “important”) included in this upbeat abstract.

We can also explore if there are time trends or differences between different disciplines (see [Figure 19.3](#)).

```
ggplot(
  arxiv_papers,
  aes(
    x = submitted, y = sentiment_per_word,
    color = field == "cs"
  )
) +
  geom_smooth(se = TRUE) +
  scale_color_brewer("Computer Science?", palette = "Set2") +
  labs(x = "Date submitted", y = "Sentiment score per word")
```

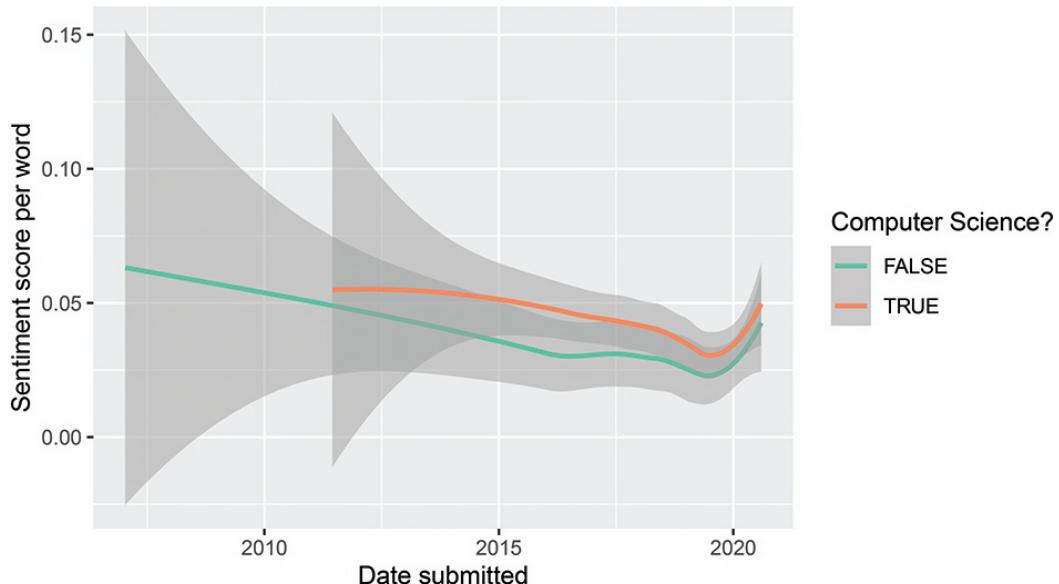


Figure 19.3: Average sum sentiment scores over time by field.

There's mild evidence for a downward trend over time. Computer science papers have slightly higher sentiment, but the difference is modest.

19.2.4 Bigrams and N-grams

We can also start to explore more sophisticated patterns within our corpus. An *N-gram* is a contiguous sequence of *n* “words.” Thus, a 1-gram is a single word (e.g., “text”), while a

2-gram (*bigram*) is a pair of words (e.g. “text mining”). We can use the same techniques to identify the most common pairs of words.

```
arxiv_bigrams <- arxiv_papers %>%
  unnest_tokens(
    arxiv_bigram,
    abstract_clean,
    token = "ngrams",
    n = 2
  ) %>%
  select(arxiv_bigram, id)
arxiv_bigrams

# A tibble: 121,454 x 2
  arxiv_bigram      id
  <chr>            <chr>
1 astronomy one   astro-ph/0701361v1
2 one data        astro-ph/0701361v1
3 data intensive  astro-ph/0701361v1
4 intensive sciences  astro-ph/0701361v1
5 sciences data   astro-ph/0701361v1
6 data technology  astro-ph/0701361v1
7 technology accelerating astro-ph/0701361v1
8 accelerating quality  astro-ph/0701361v1
9 quality effectiveness  astro-ph/0701361v1
10 effectiveness research  astro-ph/0701361v1
# ... with 121,444 more rows

arxiv_bigrams %>%
  count(arxiv_bigram, sort = TRUE)

# A tibble: 96,822 x 2
  arxiv_bigram     n
  <chr>           <int>
1 data science    953
2 machine learning 403
3 big data       139
4 state art       121
5 data analysis   111
6 deep learning   108
7 neural networks 100
8 real world      97
9 large scale     83
10 data driven     80
# ... with 96,812 more rows
```

Not surprisingly, *data science* is the most common bigram.

19.2.5 Document term matrices

Another important technique in text mining involves the calculation of a *term frequency-inverse document frequency* (*tf-idf*), or *document term matrix*. The term frequency of a term t in a document d is denoted $tf(t, d)$ and is simply equal to the number of times that the

term t appears in document d divided by the number of words in the document. On the other hand, the inverse document frequency measures the prevalence of a term across a set of documents D . In particular,

$$idf(t, D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}.$$

Finally, $tf_idf(t, d, D) = tf(t, d) \cdot idf(t, D)$. The tf_idf is commonly used in search engines, when the relevance of a particular word is needed across a body of documents.

Note that unless they are excluded (as we have done above) commonly-used words like `the` will appear in every document. Thus, their inverse document frequency score will be zero, and thus their tf_idf will also be zero regardless of the term frequency. This is a desired result, since words like `the` are never important in full-text searches. Rather, documents with high tf_idf scores for a particular term will contain that particular term many times relative to its appearance across many documents. Such documents are likely to be more relevant to the search term being used.

The most commonly-used words in our corpora are listed below. Not surprisingly “data” and “science” are at the top of the list.

```
arxiv_words %>%
  count(word) %>%
  arrange(desc(n)) %>%
  head()

# A tibble: 6 x 2
  word      n
  <chr>    <int>
1 data     3222
2 science   1122
3 learning  804
4 can       731
5 model     540
6 analysis  488
```

However, the term frequency metric is calculated on a per word, per document basis. It answers the question of which abstracts use a word most often.

```
tidy_DTM <- arxiv_words %>%
  count(id, word) %>%
  bind_tf_idf(word, id, n)
tidy_DTM %>%
  arrange(desc(tf)) %>%
  head()

# A tibble: 6 x 6
  id        word          n      tf     idf tf_idf
  <chr>    <chr>    <int> <dbl> <dbl>  <dbl>
1 2007.03606v1 data        20  0.169  0.128  0.0217
2 1707.07029v1 concept    1  0.167  3.30   0.551 
3 1707.07029v1 data        1  0.167  0.128  0.0214
4 1707.07029v1 implications 1  0.167  3.77   0.629 
5 1707.07029v1 reflections 1  0.167  6.30   1.05  
6 1707.07029v1 science     1  0.167  0.408  0.0680
```

We see that among all terms in all papers, “data” has the highest term frequency for paper 2007.03606v1 (0.169). Nearly 17% of the non-stopwords in this papers abstract were “data.” However, as we saw above, since “data” is the most common word in the entire corpus, it has the *lowest* inverse document frequency (0.128). The `tf_idf` score for “data” in paper 2007.03606v1 is thus $0.169 \cdot 0.128 = 0.022$. This is not a particularly large value, so a search for “data” would not bring this paper to the top of the list.

```
tidy_DTM %>%
  arrange(desc(idf), desc(n)) %>%
  head()
```

```
# A tibble: 6 x 6
  id      word     n      tf     idf tf_idf
  <chr>    <chr> <int>  <dbl>  <dbl>  <dbl>
1 1507.00333v3 mf      14 0.107   6.99   0.747
2 1611.09874v1 fe      13 0.0549  6.99   0.384
3 1611.09874v1 mg      11 0.0464  6.99   0.325
4 2003.00646v1 wildfire 10 0.0518  6.99   0.362
5 1506.08903v7 ph       9 0.0703  6.99   0.492
6 1710.06905v1 homeless  9 0.0559  6.99   0.391
```

On the other hand, “wildfire” has a high `idf` score since it is included in only one abstract (though it is used 10 times).

```
arxiv_papers %>%
  pull(abstract) %>%
  str_subset("wildfire") %>%
  strwrap() %>%
  head()
```

```
[1] "Artificial intelligence has been applied in wildfire science and"
[2] "management since the 1990s, with early applications including"
[3] "neural networks and expert systems. Since then the field has"
[4] "rapidly progressed congruently with the wide adoption of machine"
[5] "learning (ML) in the environmental sciences. Here, we present a"
[6] "scoping review of ML in wildfire science and management. Our"
```

In contrast, “implications” appears in 25 abstracts.

```
tidy_DTM %>%
  filter(word == "implications")
```

```
# A tibble: 25 x 6
  id      word     n      tf     idf tf_idf
  <chr>    <chr> <int>  <dbl>  <dbl>  <dbl>
1 1310.4461v2 implications  1 0.00840  3.77  0.0317
2 1410.6646v1 implications  1 0.00719  3.77  0.0272
3 1511.07643v1 implications  1 0.00621  3.77  0.0234
4 1601.04890v2 implications  1 0.00680  3.77  0.0257
5 1608.05127v1 implications  1 0.00595  3.77  0.0225
6 1706.03102v1 implications  1 0.00862  3.77  0.0325
7 1707.07029v1 implications  1 0.167    3.77  0.629
8 1711.04712v1 implications  1 0.00901  3.77  0.0340
9 1803.05991v1 implications  1 0.00595  3.77  0.0225
```

```
10 1804.10846v6 implications      1 0.00909  3.77 0.0343
# ... with 15 more rows
```

The `tf_idf` field can be used to help identify keywords for an article. For our previously selected paper, “causal,” “exposure,” or “question” would be good choices.

```
tidy_DTM %>%
  filter(id == "1809.02408v2") %>%
  arrange(desc(tf_idf)) %>%
  head()
```

```
# A tibble: 6 x 6
  id          word      n    tf    idf tf_idf
  <chr>      <chr> <int> <dbl> <dbl>   <dbl>
1 1809.02408v2 causal     10 0.0775  4.10 0.318
2 1809.02408v2 exposure    2 0.0155  5.38 0.0835
3 1809.02408v2 question    3 0.0233  3.23 0.0752
4 1809.02408v2 roadmap     2 0.0155  4.80 0.0744
5 1809.02408v2 parametric   2 0.0155  4.16 0.0645
6 1809.02408v2 effect      2 0.0155  3.95 0.0612
```

A search for “covid” yields several papers that address the pandemic directly.

```
tidy_DTM %>%
  filter(word == "covid") %>%
  arrange(desc(tf_idf)) %>%
  head() %>%
  left_join(select(arxiv_papers, id, abstract), by = "id")
```

```
# A tibble: 6 x 7
  id          word      n    tf    idf tf_idf abstract
  <chr>      <chr> <int> <dbl> <dbl>   <dbl> <chr>
1 2006.00~ covid     10 0.0637  4.80 0.305   " Context: The dire consequence~
2 2004.09~ covid      5 0.0391  4.80 0.187   " The Covid-19 outbreak, beyond~
3 2003.08~ covid      3 0.0246  4.80 0.118   " The relative case fatality ra~
4 2006.01~ covid      3 0.0222  4.80 0.107   " This document analyzes the ro~
5 2003.12~ covid      3 0.0217  4.80 0.104   " The COVID-19 pandemic demands~
6 2006.05~ covid      3 0.0170  4.80 0.0817  " This paper aims at providing ~
```

The (document, term) pair with the highest overall `tf_idf` is “reflections” (a rarely-used word having a high `idf` score), in a paper that includes only six non-stopwords in its abstract. Note that “implications” and “society” also garner high `tf_idf` scores for that same paper.

```
tidy_DTM %>%
  arrange(desc(tf_idf)) %>%
  head() %>%
  left_join(select(arxiv_papers, id, abstract), by = "id")
```

```
# A tibble: 6 x 7
  id          word      n    tf    idf tf_idf abstract
  <chr>      <chr> <int> <dbl> <dbl>   <dbl> <chr>
1 1707.07~ reflec~     1 0.167  6.30  1.05   " Reflections on the Concept ~
2 2007.12~ fintech     8 0.123  6.99  0.861  " Smart FinTech has emerged a~
3 1507.00~ mf          14 0.107  6.99  0.747  " Low-rank matrix factorizati~
4 1707.07~ implic~     1 0.167  3.77  0.629  " Reflections on the Concept ~
```

```
5 1707.07~ society      1 0.167   3.70  0.616 " Reflections on the Concept ~
6 1906.04~ utv         8 0.0860  6.99  0.602 " In this work, a novel rank--
```

The `cast_dtm()` function can be used to create a document term matrix.

```
tm_DTM <- arxiv_words %>%
  count(id, word) %>%
  cast_dtm(id, word, n, weighting = tm::weightTfIdf)
tm_DTM
```

```
<<DocumentTermMatrix (documents: 1089, terms: 12317)>>
Non-/sparse entries: 93559/13319654
Sparsity           : 99%
Maximal term length: 37
Weighting          : tf-idf (normalized)
```

By default, each entry in that matrix records the *term frequency* (i.e., the number of times that each word appeared in each document). However, in this case we will specify that the entries record the normalized *tf_idf* as defined above. Note that the `DTM` matrix is very sparse—99% of the entries are 0. This makes sense, since most words do not appear in most documents (abstracts, for our example).

We can now use tools from other packages (e.g., `tm`) to explore associations. We can now use the `findFreqTerms()` function with the `DTM` object to find the words with the highest *tf_idf* scores. Note how these results differ from the word cloud in [Figure 19.2](#). By term frequency, the word `data` is by far the most common, but this gives it a low *idf* score that brings down its *tf_idf*.

```
tm::findFreqTerms(tm_DTM, lowfreq = 7)
```

```
[1] "analysis"    "information" "research"     "learning"    "time"
[6] "network"     "problem"     "can"        "algorithm"  "algorithms"
[11] "based"       "methods"     "model"       "models"     "machine"
```

Since `tm_DTM` contains all of the *tf_idf* scores for each word, we can extract those values and calculate the score of each word across all of the abstracts.

```
tm_DTM %>%
  as.matrix() %>%
  as_tibble() %>%
  map_dbl(sum) %>%
  sort(decreasing = TRUE) %>%
  head()
```

	learning	model	models	machine	analysis	algorithms
	10.10	9.30	8.81	8.04	7.84	7.72

Moreover, we can identify which terms tend to show up in the same documents as the word `causal` using the `findAssocs()` function. In this case, we explore the words that have a correlation of at least 0.35 with the terms `causal`.

```
tm::findAssocs(tm_DTM, terms = "causal", corlimit = 0.35)
```

\$causal	estimand	laan	petersen	stating	tmle	exposure	der
	0.57	0.57	0.57	0.57	0.57	0.39	0.38
censoring		gave					

0.35 0.35

19.3 Ingesting text

In [Chapter 6](#) (see [Section 6.4.1.2](#)) we illustrated how the `rvest` package can be used to convert tabular data presented on the Web in HTML format into a proper **R** data table. Here, we present another example of how this process can bring text data into **R**.

19.3.1 Example: Scraping the songs of the Beatles

In [Chapter 14](#), we explored the popularity of the names for the four members of the *Beatles*. During their heyday from 1962–1970, the Beatles were prolific—recording hundreds of songs. In this example, we explore some of who sang and what words were included in song titles. We begin by downloading the contents of the Wikipedia page that lists the Beatles’ songs.

```
library(rvest)
url <- "http://en.wikipedia.org/wiki/List_of_songs_recorded_by_the_Beatles"
tables <- url %>%
  read_html() %>%
  html_nodes("table")
Beatles_songs <- tables %>%
  purrr::pluck(3) %>%
  html_table(fill = TRUE) %>%
  janitor::clean_names() %>%
  select(song, lead_vocal_s_d)
glimpse(Beatles_songs)
```

Rows: 213
Columns: 2
\$ song <chr> "\"Across the Universe\"[e]", "\"Act Naturally\"..."
\$ lead_vocal_s_d <chr> "John Lennon", "Ringo Starr", "Lennon", "Paul Mc...

We need to clean these data a bit. Note that the `song` variable contains quotation marks. The `lead_vocal_s_d` variable would benefit from being renamed.

```
Beatles_songs <- Beatles_songs %>%
  mutate(song = str_remove_all(song, pattern = '\\"')) %>%
  rename(vocals = lead_vocal_s_d)
```

Most of the Beatles’ songs were sung by some combination of John Lennon and Paul McCartney. While their productive but occasionally contentious working relationship is well-documented, we might be interested in determining how many songs each person is credited with singing.

```
Beatles_songs %>%
  group_by(vocals) %>%
  count() %>%
  arrange(desc(n))
```

A tibble: 18 × 2

# Groups: vocals [18]	
vocals	n
<chr>	<int>
1 Lennon	66
2 McCartney	60
3 Harrison	28
4 LennonMcCartney	15
5 Lennon(with McCartney)	12
6 Starr	10
7 McCartney(with Lennon)	9
8 Lennon(with McCartneyand Harrison)	3
9 Instrumental	1
10 John Lennon	1
11 Lennon(with Yoko Ono)	1
12 LennonHarrison	1
13 LennonMcCartneyHarrison	1
14 McCartney(with Lennon,Harrison, and Starr)	1
15 McCartneyLennonHarrison	1
16 Paul McCartney	1
17 Ringo Starr	1
18 Sound Collage	1

Lennon and McCartney sang separately and together. Other band members (notably Ringo Starr and George Harrison) also sang, along with many rarer combinations.

Regular expressions can help us parse these data. We already saw the number of songs sung by each person individually, and it isn't hard to figure out the number of songs that each person contributed to in some form in terms of vocals.

```
Beatles_songs %>%
  pull(vocals) %>%
  str_subset("McCartney") %>%
  length()
```

```
[1] 103
```

```
Beatles_songs %>%
  pull(vocals) %>%
  str_subset("Lennon") %>%
  length()
```

```
[1] 111
```

John was credited with singing on more songs than Paul.

How many of these songs were the product of some type of Lennon-McCartney collaboration? Given the inconsistency in how the vocals are attributed, it requires some ingenuity to extract these data. We can search the `vocals` variable for either `McCartney` or `Lennon` (or both), and count these instances.

```
Beatles_songs %>%
  pull(vocals) %>%
  str_subset("(McCartney|Lennon)") %>%
  length()
```

```
[1] 172
```

At this point, we need another regular expression to figure out how many songs they both sang on. The following will find the pattern consisting of either McCartney or Lennon, followed by a possibly empty string of characters, followed by another instance of either McCartney or Lennon.

```
pj_regex <- "(McCartney|Lennon).*(McCartney|Lennon)"
Beatles_songs %>%
  pull(vocals) %>%
  str_subset(pj_regex) %>%
  length()
```

```
[1] 42
```

Note also that we can use `str_detect()` in a `filter()` command to retrieve the list of songs upon which Lennon and McCartney both sang.

```
Beatles_songs %>%
  filter(str_detect(vocals, pj_regex)) %>%
  select(song, vocals) %>%
  head()
```

song	vocals
1 All Together Now	McCartney(with Lennon)
2 Any Time at All	Lennon(with McCartney)
3 Baby's in Black	LennonMcCartney
4 Because	LennonMcCartneyHarrison
5 Birthday	McCartney(with Lennon)
6 Carry That Weight	McCartney(with Lennon,Harrison, and Starr)

The Beatles have had such a profound influence upon musicians of all stripes that it might be worth investigating the titles of their songs. What were they singing about?

```
Beatles_songs %>%
  unnest_tokens(word, song) %>%
  anti_join(get_stopwords(), by = "word") %>%
  count(word, sort = TRUE) %>%
  arrange(desc(n)) %>%
  head()
```

word n
1 love 9
2 want 7
3 got 6
4 hey 6
5 long 6
6 baby 4

Fittingly, “Love” is the most common word in the title of Beatles songs.

19.4 Further resources

Silge and Robinson's *Tidy Text Mining in R* book has an extensive set of examples of text mining and sentiment analysis (Silge and Robinson, 2017, 2016). Emil Hvitfeldt and Julia Silge have announced a tidy approach to supervised machine learning for text analysis.

Text analytics has a rich history of being used to infer authorship of the Federalist papers (Mosteller and Wallace, 1963) and Beatles songs (Glickman et al., 2019).

Google has collected n -grams for a huge number of books and provides an interface to these data.

Wikipedia provides a clear overview of syntax for sophisticated pattern-matching within strings using regular expressions.

There are many sources to find text data online. Project Gutenberg is a massive free online library. Project Gutenberg collects the full-text of more than 50,000 books whose copyrights have expired. It is great for older, classic books. You won't find anything by Stephen King (but there is one by Stephen King-Hall). Direct access to Project Gutenberg is available in **R** through the **gutenbergr** package.

The **tidytext** and **textdata** packages support other lexicons for sentiment analysis, including "bing," "nrc," and "loughran."

19.5 Exercises

Problem 1 (Easy): Use the `Macbeth_raw` data from the `mdsr` package to answer the following questions:

- Speaking lines in Shakespeare's plays are identified by a line that starts with two spaces, then a string of capital letters and spaces (the character's name) followed by a period. Use `grep` to find all of the speaking lines in *Macbeth*. How many are there?
- Find all the hyphenated words in *Macbeth*.

Problem 2 (Easy):

- Find all of the adjectives in *Macbeth* that end in *more* or *less* using `Macbeth_raw` in `mdsr`.
- Find all of the lines containing the stage direction *Exit* or *Exeunt* in *Macbeth*.

Problem 3 (Easy): Given the vector of words below, determine the output of the following regular expressions without running the R code.

```
x <- c(
  "popular", "popularity", "popularize", "popularise",
  "Popular", "Population", "repopulate", "reproduce",
  "happy family", "happier\tfamily", " happy family", "P6dn")
```

```
)  
x
```

```
[1] "popular"           "popularity"        "popularize"       "popularise"  
[5] "Popular"          "Population"         "repopulate"       "reproduce"  
[9] "happy family"     "happier\tfamily"  " happy family"   "P6dn"  
  
str_subset(x, pattern = "pop")                      #1  
str_detect(x, pattern = "^pop")                     #2  
str_detect(x, pattern = "populari[sz]e")            #3  
str_detect(x, pattern = "pop.*e")                   #4  
str_detect(x, pattern = "p[a-z]*e")                 #5  
str_detect(x, pattern = "^[Pp][a-z]+.*n")           #6  
str_subset(x, pattern = "^[^Pp]")                   #7  
str_detect(x, pattern = "^[A-Za-p]")                #8  
str_detect(x, pattern = "[ ]")                      #9  
str_subset(x, pattern = "[\t]")                     #10  
str_detect(x, pattern = "[ \t]")                    #11  
str_subset(x, pattern = "^[ ]")                     #12
```

Problem 4 (Easy): Use the `babynames` data table from the `babynames` package to find the 10 most popular:

- a. Boys' names ending in a vowel.
- b. Names ending with `joe`, `jo`, `Joe`, or `Jo` (e.g., *Billyjoe*).

Problem 5 (Easy): Wikipedia defines a hashtag as “a type of metadata tag used on social networks such as Twitter and other microblogging services, allowing users to apply dynamic, user-generated tagging which makes it possible for others to easily find messages with a specific theme or content.” A hashtag must begin with a hash character followed by other characters, and is terminated by a space or end of message. It is always safe to precede the `#` with a space, and to include letters without diacritics (e.g., accents), digits, and underscores.” Provide a regular expression that matches whether a string contains a valid hashtag.

```
strings <- c(  
  "This string has no hashtags",  
  "#hashtag city!",  
  "This string has a #hashtag",  
  "This string has #two #hashtags"  
)
```

Problem 6 (Easy): A ZIP (zone improvement program) code is a code used by the United States Postal Service to route mail. The Zip + 4 code include the five digits of the ZIP Code, followed by a hyphen and four digits that designate a more specific location. Provide a regular expression that matches strings that consist of a Zip + 4 code.

Problem 7 (Medium): Create a DTM (document term matrix) for the collection of Emily Dickinson's poems in the `DickinsonPoems` package. Find the terms with the highest `tf.idf` scores. Choose one of these terms and find any of its strongly correlated terms.

```
# remotes::install_github("Amherst-Statistics/DickinsonPoems")
```

Problem 8 (Medium): A text analytics project is using scanned data to create a corpus. Many of the lines have been hyphenated in the original text.

```
text_lines <- tibble(
  lines = c("This is the first line.",
            "This line is hyphen- ",
            "ated. It's very diff-",
            "icult to use at present.")
)
```

Write a function that can be used to remove the hyphens and concatenate the parts of the words that are split on the line where they first appeared.

Problem 9 (Medium): Find all titles of Emily Dickinson's poems (not including the Roman numerals) in the first 10 poems of the `DickinsonPoems` package. (Hint: the titles are all caps.)

Problem 10 (Medium):

Classify Emily Dickinson's poem *The Lonely House* as either positive or negative using the AFINN lexicon. Does this match with your own interpretation of the poem? Use the `DickinsonPoems` package.

```
library(DickinsonPoems)
poem <- get_poem("gutenberg1.txt014")
```

Problem 11 (Medium): Generate a regular expression to return the second word in a vector.

```
x <- c("one two three", "four five six", "SEVEN EIGHT")
```

When applied to vector x, the result should be:

```
[1] "two"   "five"   "EIGHT"
```

Problem 12 (Hard): The `pdxTrees_parks` dataset from the `pdxTrees` package contains information on thousands of trees in the Portland, Oregon area. Using the `species_factoid` variable, investigate any interesting trends within the facts.

19.6 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/text.html#text-online-exercises>

20

Network science

Network science is an emerging interdisciplinary field that studies the properties of large and complex networks. Network scientists are interested in both theoretical properties of networks (e.g., mathematical models for degree distribution) and data-based discoveries in real networks (e.g., the distribution of the number of friends on Facebook).

20.1 Introduction to network science

20.1.1 Definitions

The roots of network science are in the mathematical discipline of *graph theory*. There are a few basic definitions that we need before we can proceed.

- A *graph* $G = (V, E)$ is simply a set of *vertices* (or nodes) V , and a set of *edges* (or links, or even ties) E between those nodes. It may be more convenient to think about a graph as being a *network*. For example, in a network model of Facebook, each user is a vertex and each friend relation is an edge connecting two users. Thus, one can think of Facebook as a *social network*, but the underlying mathematical structure is just a graph. Discrete mathematicians have been studying graphs since Leonhard Euler posed the *Seven Bridges of Königsberg* problem in 1736 (Euler, 1953).
- Edges in graphs can be *directed* or *undirected*. The difference is whether the relationship is mutual or one-sided. For example, edges in the Facebook social network are undirected, because friendship is a mutual relationship. Conversely, edges in Twitter are directed, since you may follow someone who does not necessarily follow you.
- Edges (or less commonly, vertices) may be *weighted*. The value of the weight represents some quantitative measure. For example, an airline may envision its flight network as a graph, in which each airport is a node, and edges are weighted according to the distance (in miles) from one airport to another. (If edges are unweighted, this is equivalent to setting all weights to 1.)
- A *path* is a non-self-intersecting sequence of edges that connect two vertices. More formally, a path is a special case of a *walk*, which does allow self-intersections (i.e., a vertex may appear in the walk more than once). There may be many paths, or no paths, between two vertices in a graph, but if there are any paths, then there is at least one *shortest path* (or *geodesic*). The notion of a shortest path is dependent upon a distance measure in the graph (usually, just the number of edges, or the sum of the edge weights). A graph is *connected* if there is a path between all pairs of vertices.
- The *diameter* of a graph is the length of the longest geodesic (i.e., the longest shortest [sic] path) between any pairs of vertices. The *eccentricity* of a vertex v in a graph is the

length of the longest geodesic starting at that vertex. Thus, in some sense a vertex with a low eccentricity is more central to the graph.

- In general, graphs do not have coordinates. Thus, there is no right way to draw a graph. Visualizing a graph is more art than science, but several graph layout algorithms are popular.
- Centrality: Since graphs don't have coordinates, there is no obvious measure of *centrality*. That is, it is frequently of interest to determine which nodes are most "central" to the network, but there are many different notions of centrality. We will discuss three:
 - Degree centrality: The *degree* of a vertex within a graph is the number of edges incident to it. Thus, the degree of a node is a simple measure of centrality in which more highly connected nodes rank higher. President Obama has almost 100 million followers on Twitter, whereas the vast majority of users have fewer than a thousand. Thus, the degree of the vertex representing President Obama in the Twitter network is in the millions, and he is more central to the network in terms of degree centrality.

- Betweenness centrality: If a vertex v is more central to a graph, then you would suspect that more shortest paths between vertices would pass through v . This is the notion of *betweenness centrality*. Specifically, let $\sigma(s, t)$ be the number of geodesics between vertices s and t in a graph. Let $\sigma_v(s, t)$ be the number of shortest paths between s and t that pass through v . Then the betweenness centrality for v is the sum of the fractions $\sigma_v(s, t)/\sigma(s, t)$ over all possible pairs (s, t) . This figure ($C_B(v)$) is often normalized by dividing by the number of pairs of vertices that do not include v in the graph.

$$C_B(v) = \frac{2}{(n-1)(n-2)} \sum_{s, t \in V \setminus \{v\}} \frac{\sigma_v(s, t)}{\sigma(s, t)},$$

where n is the number of vertices in the graph. Note that President Obama's high degree centrality would not necessarily translate into a high betweenness centrality.

- *Eigenvector centrality*: This is the essence of Google's *PageRank* algorithm, which we will discuss in [Section 20.3](#). Note that there are also notions of edge centrality that we will not discuss further.
- In a social network, it is usually believed that if Alice and Bob are friends, and Alice and Carol are friends, then it is more likely than it otherwise would be that Bob and Carol are friends. This is the notion of *triadic closure*, and it leads to measurements of *clusters* in real-world networks.

20.1.2 A brief history of network science

As noted above, the study of graph theory began in the 1700s, but the inception of the field of network science was a paper published in 1959 by the legendary Paul Erdős and Alfréd Rényi (Erdős and Rényi, 1959). Erdős and Rényi proposed a model for a *random graph*, where the number of vertices n is fixed, but the probability of an edge connecting any two vertices is p . What do such graphs look like? What properties do they have? It is obvious that if p is very close to 0, then the graph will be almost empty, while conversely, if p is very close to 1, then the graph will be almost complete. Erdős and Rényi unexpectedly proved that for many graph properties c (e.g., connectedness, the existence of a cycle of a certain size, etc.), there is a threshold function $p_c(n)$ around which the structure of the graph seems to change rapidly. That is, for values of p slightly less than $p_c(n)$, the probability that a

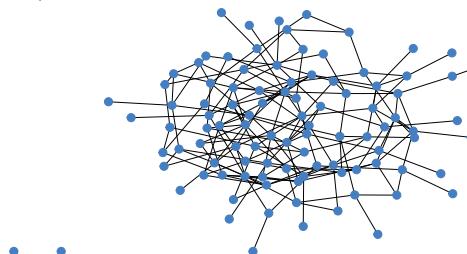
random graph is connected is close to zero, while for values of p just a bit larger than $p_c(n)$, the probability that a random graph is connected is close to one (see [Figure 20.1](#)).

This somewhat bizarre behavior has been called the *phase transition* in allusion to physics, because it evokes at a molecular level how solids turn to liquids and liquids turn to gasses. When temperatures are just above 32 degrees Fahrenheit, water is a liquid, but at just below 32 degrees, it becomes a solid.

```
library(tidyverse)
library(mdsr)
library(tidygraph)
library(ggraph)
set.seed(21)
n <- 100
p_star <- log(n)/n

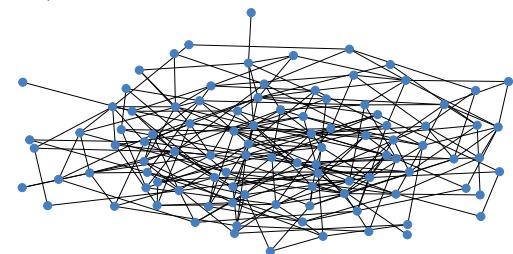
plot_er <- function(n, p) {
  g <- play_erdos_renyi(n, p, directed = FALSE)
  ggraph(g) +
    geom_edge_fan(width = 0.1) +
    geom_node_point(size = 3, color = "dodgerblue") +
    labs(
      title = "Erdős--Rényi random graph",
      subtitle = paste0("n = ", n, ", p = ", round(p, 4))
    ) +
    theme_void()
}
plot_er(n, p = 0.8 * p_star)
plot_er(n, p = 1.2 * p_star)
```

Erd.s--Rényi random graph
n = 100, p = 0.0368



(a) A graph that is not connected.

Erd.s--Rényi random graph
n = 100, p = 0.0553



(b) A connected graph.

Figure 20.1: Two Erdős–Rényi random graphs on 100 vertices with different values of p . The graph at left is not connected, but the graph at right is. The value of p hasn't changed by much.

While many properties of the phase transition have been proven mathematically, they can also be illustrated using simulation (see [Chapter 13](#)). Throughout this chapter, we use the **tidygraph** package for constructing and manipulating graphs¹, and the **ggraph** package for plotting graphs as **ggplot2** objects. The **tidygraph** package provides the `play_erdos_renyi()` function for simulating Erdős–Rényi random graphs. In [Figure 20.2](#),

¹**tidygraph** wraps much of the functionality of the **igraph** package.

we show how the phase transition for connectedness appears around the threshold value of $p(n) = \log n / n$. With $n = 1000$, we have $p(n) = 0.007$. Note how quickly the probability of being connected increases near the value of the threshold function.

```

n <- 1000
p_star <- log(n)/n
p <- rep(seq(from = 0, to = 2 * p_star, by = 0.001), each = 100)

sims <- tibble(n, p) %>%
  mutate(
    g = map2(n, p, play_erdos_renyi, directed = FALSE),
    is_connected = map_int(g, ~with_graph(., graph_is_connected()))
  )

ggplot(data = sims, aes(x = p, y = is_connected)) +
  geom_vline(xintercept = p_star, color = "darkgray") +
  geom_text(
    x = p_star, y = 0.9, label = "Threshold value", hjust = "right"
  ) +
  labs(
    x = "Probability of edge existing",
    y = "Probability that random graph is connected"
  ) +
  geom_count() +
  geom_smooth()

```

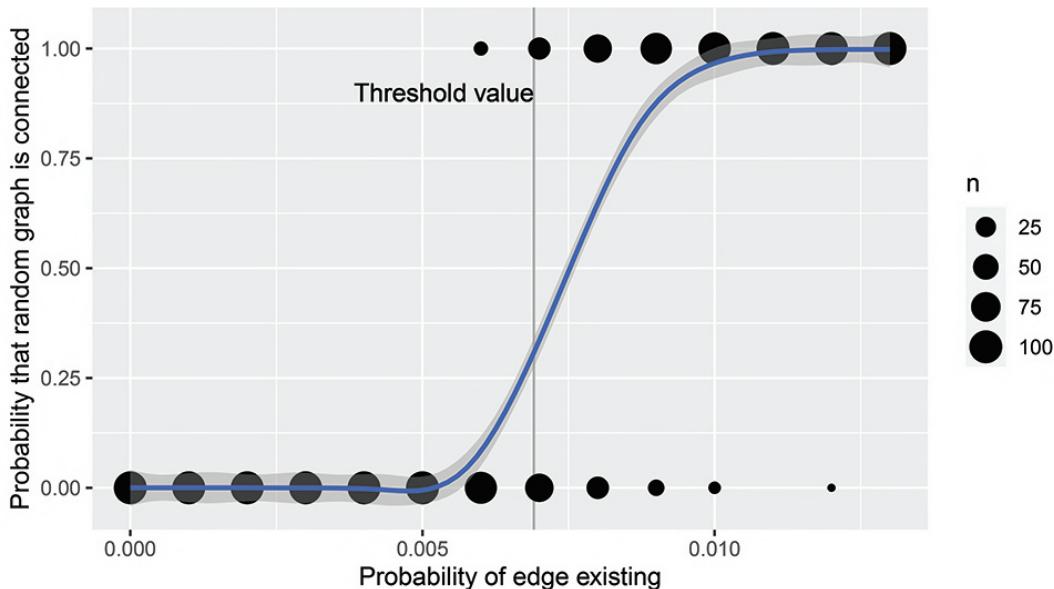


Figure 20.2: Simulation of connectedness of ER random graphs on 1,000 vertices.

This surprising discovery demonstrated that random graphs had interesting properties. Yet it was less clear whether the Erdős–Rényi random graph model could produce graphs whose properties were similar to those that we observe in reality. That is, while the Erdős–Rényi random graph model was interesting in its own right, did it model reality well?

The answer turned out to be “no,” or at least, “not really.” In particular, Watts and Strogatz (1998) identified two properties present in real-world networks that were not present in Erdős–Rényi random graphs: triadic closure and large hubs. As we saw above, triadic closure is the idea that two people with a friend in common are likely to be friends themselves. Real-world (not necessarily social) networks tend to have this property, but Erdős–Rényi random graphs do not. Similarly, real-world networks tend to have large hubs—individual nodes with many edges. More specifically, whereas the distribution of the degrees of vertices in Erdős–Rényi random graphs can be shown to follow a *Poisson distribution*, in real-world networks the distribution tends to be flatter. The Watts–Strogatz model provides a second random graph model that produces graphs more similar to those we observe in reality.

```
g <- play_smallworld(n_dim = 2, dim_size = 10, order = 5, p_rewire = 0.05)
```

In particular, many real-world networks, including not only social networks but also the World Wide Web, citation networks, and many others, have a degree distribution that follows a *power-law*. These are known as *scale-free* networks and were popularized by Albert–László Barabási in two widely-cited papers (Barabási and Albert, 1999, Albert and Barabási (2002)) and his readable book (Barabási and Frangos, 2014). Barabási and Albert proposed a third random graph model based on the notion of *preferential attachment*. Here, new nodes are connected to old nodes based on the existing degree distribution of the old nodes. Their model produces the power-law degree distribution that has been observed in many different real-world networks.

Here again, we can illustrate these properties using simulation. The `play_barabasi_albert()` function in `tidygraph` will allow us to simulate a Barabási–Albert random graph. Figure 20.3 compares the degree distribution between an Erdős–Rényi random graph and a Barabási–Albert random graph.

```
g1 <- play_erdos_renyi(n, p = log(n)/n, directed = FALSE)
g2 <- play_barabasi_albert(n, power = 1, growth = 3, directed = FALSE)
summary(g1)
```

```
IGRAPH 920eddc U--- 1000 3419 -- Erdos renyi (gnp) graph
+ attr: name (g/c), type (g/c), loops (g/l), p (g/n)
summary(g2)
```

```
IGRAPH 852ca74 U--- 1000 2994 -- Barabasi graph
+ attr: name (g/c), power (g/n), m (g/n), zero.appeal (g/n),
| algorithm (g/c)
d <- tibble(
  type = c("Erdos-Renyi", "Barabasi-Albert"),
  graph = list(g1, g2)
) %>%
  mutate(node_degree = map(graph, ~with_graph(., centrality_degree()))) %>%
  unnest(node_degree)

ggplot(data = d, aes(x = node_degree, color = type)) +
  geom_density(size = 2) +
  scale_x_continuous(limits = c(0, 25))
```

Network science is a very active area of research, with interesting unsolved problems for data scientists to investigate.

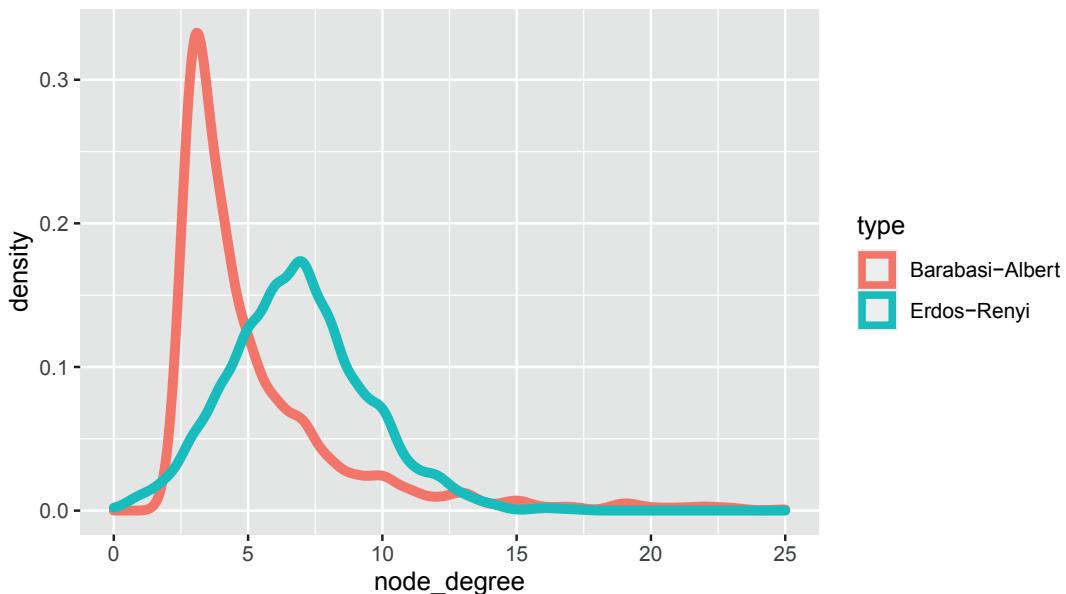


Figure 20.3: Degree distribution for two random graphs.

20.2 Extended example: Six degrees of Kristen Stewart

In this extended example, we will explore a fun application of network science to *Hollywood* movies. The notion of *Six Degrees of Separation* was conjectured by a Hungarian network theorist in 1929, and later popularized by a play (and movie starring Will Smith). Stanley Milgram's famous letter-mailing *small-world* experiment supposedly lent credence to the idea that all people are connected by relatively few "social hops" (Travers and Milgram, 1969). That is, we are all part of a social network with a relatively small diameter (as small as 6).

Two popular incarnations of these ideas are the notion of an *Erdős number* and the Kevin Bacon game. The question in each case is the same: How many hops are you away from Paul Erdős (or Kevin Bacon)? The former is popular among academics (mathematicians especially), where edges are defined by co-authored papers. Ben's Erdős number is three, since he has co-authored a paper with Amotz Bar-Noy, who has co-authored a paper with Noga Alon, who co-authored a paper with Erdős. According to MathSciNet, Nick's Erdős number is four (through Ben given (Baumer et al., 2014); but also through Nan Laird, Fred Mosteller, and Persi Diaconis), and Danny's is four (also through Ben). These data reflect the reality that Ben's research is "closer" to Erdős's, since he has written about network science (Bogdanov et al., 2013; Baumer et al., 2015; Basu et al., 2015; Baumer et al., 2011) and graph theory (Baumer et al., 2016). Similarly, the idea is that in principle, every actor in Hollywood can be connected to Kevin Bacon in at most six movie hops. We'll explore this idea using the *Internet Movie Database* (IMDB.com, 2013).

20.2.1 Collecting Hollywood data

We will populate a *Hollywood* network using actors in the IMDb. In this network, each actor is a node, and two actors share an edge if they have ever appeared in a movie together. Our goal will be to determine the centrality of Kevin Bacon.

First, we want to determine the edges, since we can then look up the node information based on the edges that are present. One caveat is that these networks can grow very rapidly (since the number of edges is $O(n^2)$, where n is the number of vertices). For this example, we will be conservative by including popular (at least 150,000 ratings) feature films (i.e., `kind_id` equal to 1) in 2012, and we will consider only the top-20 credited roles in each film.

To retrieve the list of edges, we need to consider all possible cast assignment pairs. To get this list, we start by forming all total pairs using the `CROSS JOIN` operation in MySQL (see Chapter 15), which has no direct `dplyr` equivalent. Thus, in this case we will have to actually write the SQL code. Note that we filter this list down to the unique pairs, which we can do by only including pairs where `person_id` from the first table is strictly less than `person_id` from the second table. The result of the following query will come into `R` as the object `E`.

```
library(mdsr)
db <- dbConnect_scidb("imdb")

SELECT a.person_id AS src, b.person_id AS dest,
       a.movie_id,
       a.nr_order * b.nr_order AS weight,
       t.title, idx.info AS ratings
FROM imdb.cast_info AS a
  CROSS JOIN imdb.cast_info AS b USING (movie_id)
  LEFT JOIN imdb.title AS t ON a.movie_id = t.id
  LEFT JOIN imdb.movie_info_idx AS idx ON idx.movie_id = a.movie_id
WHERE t.production_year = 2012 AND t.kind_id = 1
      AND info_type_id = 100 AND idx.info > 150000
      AND a.nr_order <= 20 AND b.nr_order <= 20
      AND a.role_id IN (1,2) AND b.role_id IN (1,2)
      AND a.person_id < b.person_id
GROUP BY src, dest, movie_id

E <- E %>%
  mutate(ratings = parse_number(ratings))
glimpse(E)
```

```
Rows: 10,223
Columns: 6
$ src      <int> 6388, 6388, 6388, 6388, 6388, 6388, 6388, 6388, ...
$ dest     <int> 405570, 445466, 688358, 722062, 830618, 838704, 960997...
$ movie_id <int> 4590482, 4590482, 4590482, 4590482, 4590482, 4590482, ...
$ weight   <dbl> 52, 13, 143, 234, 260, 208, 156, 247, 104, 130, 26, 18...
$ title    <chr> "Zero Dark Thirty", "Zero Dark Thirty", "Zero Dark Thi...
$ ratings  <dbl> 231992, 231992, 231992, 231992, 231992, 231992...
```

We have also computed a `weight` variable that we can use to weight the edges in the resulting graph. In this case, the `weight` is based on the order in which each actor appears in the

credits. So a ranking of 1 means that the actor had top billing. These weights will be useful because a higher order in the credits usually means more screen time.

```
E %>%
  summarize(
    num_rows = n(),
    num_titles = n_distinct(title)
  )
```

```
num_rows num_titles
1      10223          55
```

Our query resulted in 10,223 connections between 55 films. We can see that *Batman: The Dark Knight Rises* received the most user ratings on IMDb.

```
movies <- E %>%
  group_by(movie_id) %>%
  summarize(title = max(title), N = n(), numRatings = max(ratings)) %>%
  arrange(desc(numRatings))
movies
```

	movie_id	title	N	numRatings
	<int>	<chr>	<int>	<dbl>
1	4339115	The Dark Knight Rises	190	1258255
2	3519403	Django Unchained	190	1075891
3	4316706	The Avengers	190	1067306
4	4368646	The Hunger Games	190	750674
5	4366574	The Hobbit: An Unexpected Journey	190	681957
6	4224391	Silver Linings Playbook	190	577500
7	4231376	Skyfall	190	557652
8	4116220	Prometheus	190	504980
9	4300124	Ted	190	504893
10	3298411	Argo	190	493001
	# ... with 45 more rows			

Next, we should gather some information about the vertices in this graph. We could have done this with another JOIN in the original query, but doing it now will be more efficient. (Why? See the cross-join exercise.) In this case, all we need is each actor's name and IMDb identifier.

```
actor_ids <- unique(c(E$src, E$dest))
V <- db %>%
 tbl("name") %>%
  filter(id %in% actor_ids) %>%
  select(actor_id = id, actor_name = name) %>%
  collect() %>%
  arrange(actor_id) %>%
  mutate(id = row_number())
glimpse(V)
```

```
Rows: 1,010
Columns: 3
$ actor_id    <int> 6388, 6897, 8462, 16644, 17039, 18760, 28535, 33799,...
```

```
$ actor_name <chr> "Abkarian, Simon", "Aboutboul, Alon", "Abtahi, Omid"...
$ id <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 1...
```

20.2.2 Building the Hollywood network

To build a graph, we specify the edges, whether we want them to be directed, and add information about the vertices.

```
edges <- E %>%
  left_join(select(V, from = id, actor_id), by = c("src" = "actor_id")) %>%
  left_join(select(V, to = id, actor_id), by = c("dest" = "actor_id"))

g <- tbl_graph(nodes = V, directed = FALSE, edges = edges)
summary(g)
```

```
IGRAPH e2eb097 U-W- 1010 10223 --
+ attr: actor_id (v/n), actor_name (v/c), id (v/n), src (e/n), dest
| (e/n), movie_id (e/n), weight (e/n), title (e/c), ratings (e/n)
```

From the `summary()` command above, we can see that we have 1,010 actors and 10,223 edges between them. Note that we have associated metadata with each edge: namely, information about the movie that gave rise to the edge, and the aforementioned `weight` metric based on the order in the credits where each actor appeared. (The idea is that top-billed stars are likely to appear on screen longer, and thus have more meaningful interactions with more of the cast.)

With our network intact, we can visualize it. There are *many* graphical parameters that you may wish to set, and the default choices are not always good. In this case we have 1,010 vertices, so we'll make them small, and omit labels. [Figure 20.4](#) displays the results.

```
ggraph(g, 'drl') +
  geom_edge_fan(width = 0.1) +
  geom_node_point(color = "dodgerblue") +
  theme_void()
```

It is easy to see the clusters based on movies, but you can also see a few actors who have appeared in multiple movies, and how they tend to be more “central” to the network. If an actor has appeared in multiple movies, then it stands to reason that they will have more connections to other actors. This is captured by degree centrality.

```
g <- g %>%
  mutate(degree = centrality_degree())
g %>%
  as_tibble() %>%
  arrange(desc(degree)) %>%
  head()

# A tibble: 6 x 4
  actor_id actor_name          id degree
    <int> <chr>            <int>  <dbl>
1  502126 Cranston, Bryan     113      57
2  891094 Gordon-Levitt, Joseph 228      57
3  975636 Hardy, Tom        257      57
4  1012171 Hemsworth, Chris   272      57
```

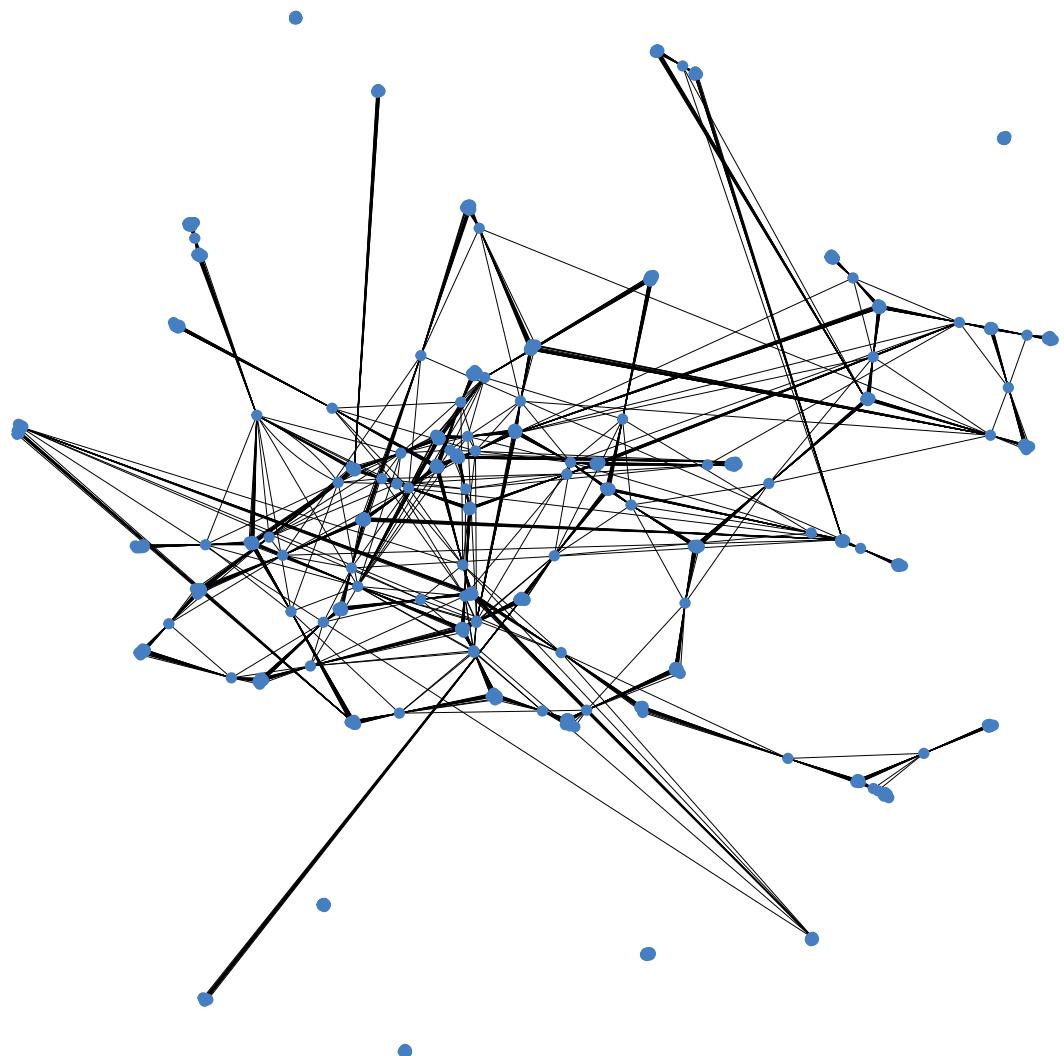


Figure 20.4: Visualization of Hollywood network for popular 2012 movies.

5	1713855	Neeson, Liam	466	57
6	1114312	Ivanek, Zeljko	304	56

There are a number of big name actors on this list who appeared in multiple movies in 2012. Why does Bryan Cranston have so many connections? The following quick function will retrieve the list of movies for a particular actor.

```
show_movies <- function(g, id) {
  g %>%
    activate(edges) %>%
    as_tibble() %>%
    filter(src == id | dest == id) %>%
    group_by(movie_id) %>%
    summarize(title = first(title), num_connections = n())
```

```
}
show_movies(g, 502126)
```

```
# A tibble: 3 x 3
  movie_id title      num_connections
    <int> <chr>              <int>
  1 3298411 Argo                19
  2 3780482 John Carter         19
  3 4472483 Total Recall        19
```

Cranston appeared in all three of these movies. Note however, that the distribution of degrees is not terribly smooth (see Figure 20.5). That is, the number of connections that each actor has appears to be limited to a few discrete possibilities. Can you think of why that might be?

```
ggplot(data = enframe(igraph::degree(g)), aes(x = value)) +
  geom_density(size = 2)
```

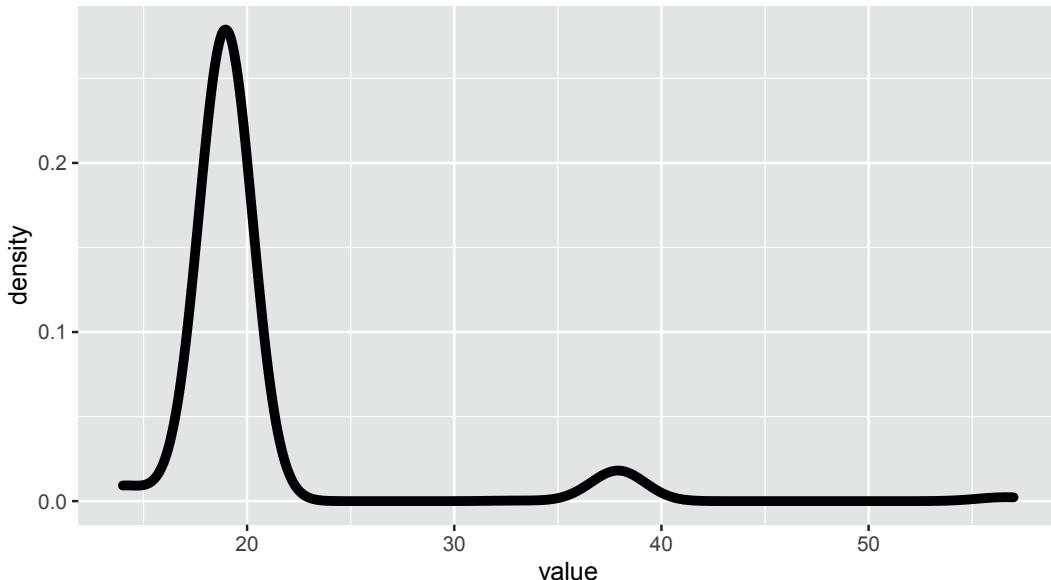


Figure 20.5: Distribution of degrees for actors in the Hollywood network of popular 2012 movies.

We use the **ggraph** package, which provides `geom_node_*`() and `geom_edge_*`() functions for plotting graphs directly with **ggplot2**. (Alternative plotting packages include **ggnetwork**, **geomnet**, and **GGally**)

```
hollywood <- ggraph(g, layout = 'drl') +
  geom_edge_fan(aes(alpha = weight), color = "lightgray") +
  geom_node_point(aes(color = degree), alpha = 0.6) +
  scale_edge_alpha_continuous(range = c(0, 1)) +
  scale_color_viridis_c() +
  theme_void()
```

We don't want to show vertex labels for everyone, because that would result in an unreadable mess. However, it would be nice to see the highly central actors. Figure 20.6 shows our

completed plot. The transparency of the edges is scaled relatively to the `weight` measure that we computed earlier.

The `ggnetwork()` function transforms our `igraph` object into a data frame, from which the `geom_nodes()` and `geom_edges()` functions can map variables to aesthetics. In this case, since there are so many edges, we use the `scale_size_continuous()` function to make the edges very thin.

```
hollywood +
  geom_node_label(
    aes(
      filter = degree > 40,
      label = str_replace_all(actor_name, " ", "\n")
    ),
    repel = TRUE
  )
```

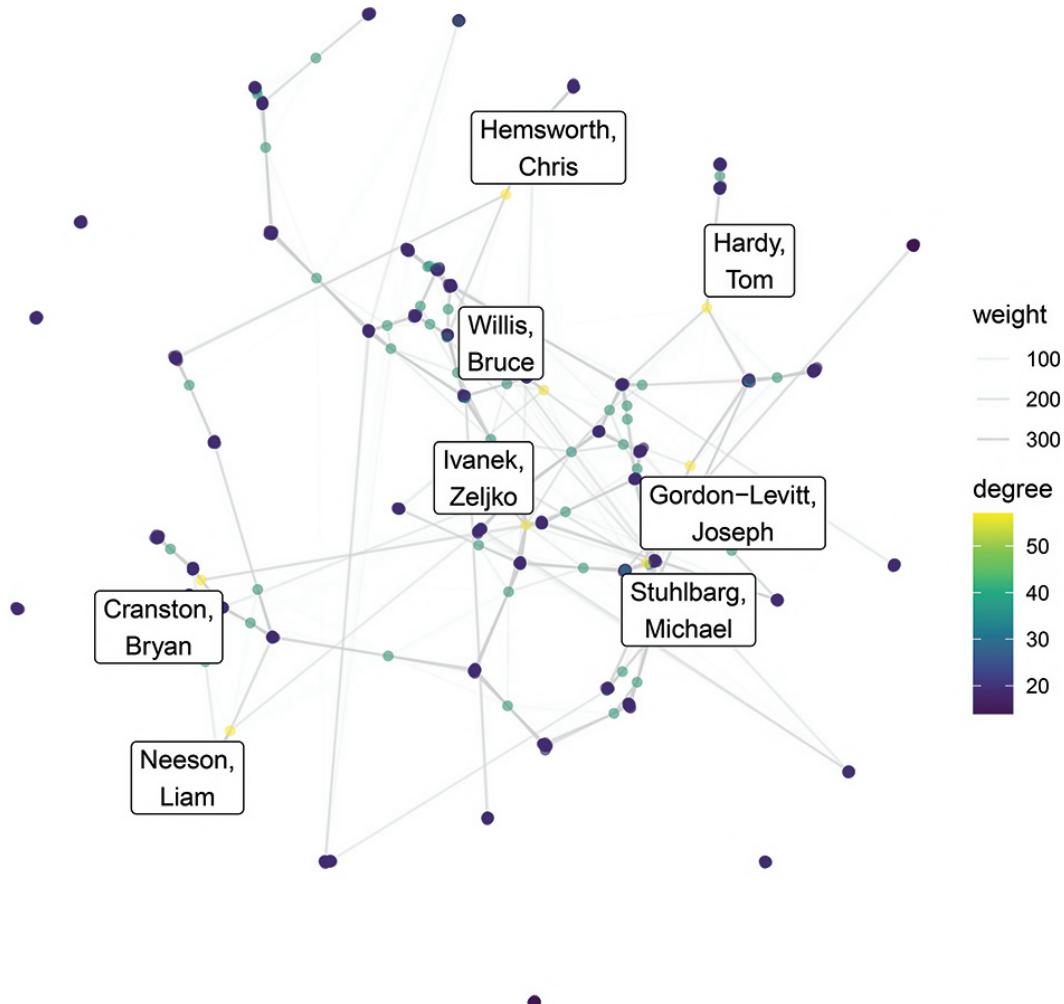


Figure 20.6: The Hollywood network for popular 2012 movies. Color is mapped to degree centrality.

20.2.3 Building a Kristen Stewart oracle

Degree centrality does not take into account the weights on the edges. If we want to emphasize the pathways through leading actors, we could consider *betweenness centrality*.

```
g <- g %>%
  mutate(btw = centrality_betweenness(weights = weight, normalized = TRUE))
g %>%
  as_tibble() %>%
  arrange(desc(btw)) %>%
  head(10)

# A tibble: 10 x 5
  actor_id actor_name      id degree   btw
  <int> <chr>        <int> <dbl> <dbl>
1 3945132 Stewart, Kristen    964     38 0.236
2 891094 Gordon-Levitt, Joseph  228     57 0.217
3 3346548 Kendrick, Anna     857     38 0.195
4 135422 Bale, Christian     27      19 0.179
5 76481 Ansari, Aziz        15      19 0.176
6 558059 Day-Lewis, Daniel   135     19 0.176
7 1318021 LaBeouf, Shia      363     19 0.156
8 2987679 Dean, Ester       787     38 0.152
9 2589137 Willis, Bruce     694     56 0.141
10 975636 Hardy, Tom        257     57 0.134

show_movies(g, 3945132)
```

```
# A tibble: 2 x 3
  movie_id title          num_connections
  <int> <chr>                <int>
1 4237818 Snow White and the Huntsman      19
2 4436842 The Twilight Saga: Breaking Dawn - Part 2 19
```

Notice that Kristen Stewart has the highest betweenness centrality, while Joseph Gordon-Levitt and Tom Hardy (and others) have the highest degree centrality. Moreover, Christian Bale has the third-highest betweenness centrality despite appearing in only one movie. This is because he played the lead in *The Dark Knight Rises*, the movie responsible for the most edges. Most shortest paths through *The Dark Knight Rises* pass through Christian Bale.

If Kristen Stewart (imdbId 3945132) is very central to this network, then perhaps instead of a Bacon number, we could consider a Stewart number. Charlize Theron's Stewart number is obviously 1, since they appeared in *Snow White and the Huntsman* together:

```
ks <- V %>%
  filter(actor_name == "Stewart, Kristen")
ct <- V %>%
  filter(actor_name == "Theron, Charlize")

g %>%
  convert(to_shortest_path, from = ks$id, to = ct$id)

# A tbl_graph: 2 nodes and 1 edges
#
# An unrooted tree
```

```

#
# Node Data: 2 x 6 (active)
#   actor_id actor_name      id degree    btw .tidygraph_node_index
#   <int>     <chr>      <int>  <dbl>  <dbl>                <int>
1  3945132 Stewart, Kristen    964     38  0.236                964
2  3990819 Theron, Charlize   974     38  0.0940               974
#
# Edge Data: 1 x 9
#   from      to    src   dest movie_id weight title  ratings .tidygraph_edge~
#   <int>  <int>  <int>  <int>  <dbl>  <chr>  <dbl>        <int>
1      1      2 3945132 3.99e6  4237818      3 Snow ~  243824            10198

```

On the other hand, her distance from Joseph Gordon-Levitt is 5. The interpretation here is that Joseph Gordon-Levitt was in *Batman: The Dark Knight Rises* with Tom Hardy, who was in *Lawless* with Guy Pearce, who was in *Prometheus* with Charlize Theron, who was in *Snow White and the Huntsman* with Kristen Stewart. We show this graphically in Figure 20.7.

```

set.seed(47)
jgl <- V %>%
  filter(actor_name == "Gordon-Levitt, Joseph")

h <- g %>%
  convert(to_shortest_path, from = jgl$id, to = ks$id, weights = NA)

h %>%
  ggplot('gem') +
  geom_node_point() +
  geom_node_label(aes(label = actor_name)) +
  geom_edge_fan2(aes(label = title)) +
  coord_cartesian(clip = "off") +
  theme(plot.margin = margin(6, 36, 6, 36))

```

Note, however, that these shortest paths are not unique. In fact, there are 9 shortest paths between Kristen Stewart and Joseph Gordon-Levitt, each having a length of 5.

```

igraph::all_shortest_paths(g, from = ks$id, to = jgl$id, weights = NA) %>%
  pluck("res") %>%
  length()

```

[1] 9

As we saw in Figure 20.6, our Hollywood network is not connected, and thus its diameter is infinite. However, the diameter of the largest connected component can be computed. This number (in this case, 10) indicates how many hops separate the two most distant actors in the network.

```

igraph::diameter(g, weights = NA)

[1] 10

g %>%
  mutate(eccentricity = node_eccentricity()) %>%
  filter(actor_name == "Stewart, Kristen")

```

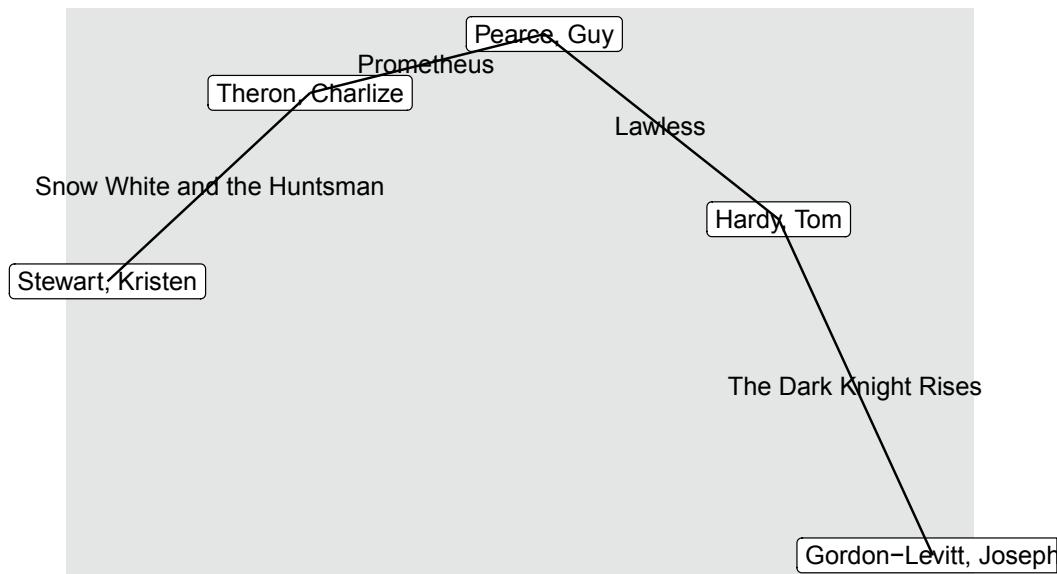


Figure 20.7: Subgraph showing a shortest path through the Hollywood network from Joseph Gordon-Levitt to Kristen Stewart.

```

# A tbl_graph: 1 nodes and 0 edges
#
# An unrooted tree
#
# Node Data: 1 x 6 (active)
  actor_id actor_name      id degree   btw eccentricity
    <int>  <chr>        <int>  <dbl>  <dbl>        <dbl>
1  3945132 Stewart, Kristen  964     38  0.236         6
#
# Edge Data: 0 x 8
# ... with 8 variables: from <int>, to <int>, src <int>, dest <int>,
#   movie_id <int>, weight <dbl>, title <chr>, ratings <dbl>

```

On the other hand, we note that Kristen Stewart's eccentricity is 6. This means that there is no actor in the connected part of the network who is more than 6 hops away from Kristen Stewart. Six degrees of separation indeed!

20.3 PageRank

For many readers, it may be difficult (or impossible) to remember what search engines on the Web were like before Google. Search engines such as *Altavista*, *Web Crawler*, *Excite*, and *Yahoo!* vied for supremacy, but none returned results that were of comparable use to the ones we get today. Frequently, finding what you wanted required sifting through pages of slow-to-load links.

Consider the search problem. A user types in a *search query* consisting of one or more

words or terms. Then the search engine produces an ordered list of Web pages ranked by their relevance to that search query. How would you instruct a computer to determine the relevance of a Web page to a query?

This problem is not trivial. Most pre-Google search engines worked by categorizing the words on every Web page, and then determining—based on the search query—which pages were most relevant to that query.

One problem with this approach is that it relies on each Web designer to have the words on its page accurately reflect the content. Naturally, advertisers could easily manipulate search engines by loading their pages with popular search terms, written in the same color as the background (making them invisible to the user), regardless of whether those words were related to the actual content of the page. So naïve search engines might rank these pages more highly, even though they were not relevant to the user.

Google conquered search by thinking about the problem in a fundamentally different way and taking advantage of the network structure of the World Wide Web. The web is a directed graph, in which each webpage (URL) is a node, and edges reflect links from one webpage to another. In 1998, Sergey Brin and Larry Page—while computer science graduate students at *Stanford University*—developed a centrality measure called *PageRank* that formed the basis of Google’s search algorithms (Page et al., 1999). The algorithm led to search results that were so much better than those of its competitors that Google quickly swallowed the entire search market, and is now one of the world’s largest companies. The key insight was that one could use the directed links on the Web as a means of “voting” in a way that was much more difficult to exploit. That is, advertisers could only control links on their pages, but not links to their pages from other sites.

20.3.1 Eigenvector centrality

Computing PageRank is a rather simple exercise in linear algebra. It is an example of a *Markov process*. Suppose there are n webpages on the Web. Let $\mathbf{v}_0 = \mathbf{1}/n$ be a vector that gives the initial probability that a randomly chosen Web surfer will be on any given page. In the absence of any information about this user, there is an equal probability that they might be on any page.

But for each of these n webpages, we also know to which pages it links. These are outgoing directed edges in the Web graph. We assume that a random surfer will follow each link with equal probability, so if there are m_i outgoing links on the i^{th} webpage, then the probability that the random surfer goes from page i to page j is $p_{ij} = 1/m_i$. Note that if the i^{th} page doesn’t link to the j^{th} page, then $p_{ij} = 0$. In this manner, we can form the $n \times n$ *transition matrix* \mathbf{P} , wherein each entry describes the probability of moving from page i to page j .

The product $\mathbf{P}\mathbf{v}_0 = \mathbf{v}_1$ is a vector where v_{1i} indicates the probability of being at the i^{th} webpage, after picking a webpage uniformly at random to start, and then clicking on one link chosen at random (with equal probability). The product $\mathbf{P}\mathbf{v}_1 = \mathbf{P}^2\mathbf{v}_0$ gives us the probabilities after two clicks, etc. It can be shown mathematically that if we continue to iterate this process, then we will arrive at a *stationary distribution* \mathbf{v}^* that reflects the long-term probability of being on any given page. Each entry in that vector then represents the popularity of the corresponding webpage— \mathbf{v}^* is the PageRank of each webpage.² Because \mathbf{v}^* is an eigenvector of the transition matrix (since $\mathbf{P}\mathbf{v}^* = \mathbf{v}^*$), this measure of centrality is

²As we will see below, this is not *exactly* true, but it is the basic idea.

known as *eigenvector centrality*. It was, in fact, developed earlier, but Page and Brin were the first to apply the idea to the World Wide Web for the purpose of search.

The success of PageRank has led to its being applied in a wide variety of contexts—virtually any problem in which a ranking measure on a network setting is feasible. In addition to the college team sports example below, applications of PageRank include: scholarly citations (e.g., eigenfactor.org), doctoral programs, protein networks, and lexical semantics.

Another metaphor that may be helpful in understanding PageRank is that of movable mass. That is, suppose that there is a certain amount of mass in a network. The initial vector \mathbf{v}_0 models a uniform distribution of that mass over the vertices. That is, $1/n$ of the total mass is located on each vertex. The transition matrix \mathbf{P} models that mass flowing through the network according to the weights on each edge. After a while, the mass will “settle” on the vertices, but in a non-uniform distribution. The node that has accumulated the most mass has the largest PageRank.

20.4 Extended example: 1996 men's college basketball

Every March (with exception in 2020 due to COVID-19), the attention of many sports fans and college students is captured by the NCAA basketball tournament, which pits 68 of the best teams against each other in a winner-take-all, single-elimination tournament. (A *tournament* is a special type of directed graph.) However, each team in the tournament is seeded based on their performance during the regular season. These seeds are important, since getting a higher seed can mean an easier path through the tournament. Moreover, a tournament berth itself can mean millions of dollars in revenue to a school's basketball program. Finally, predicting the outcome of the tournament has become something of a sport unto itself.

Kaggle has held a machine learning (see Chapters 11 and 12) competition each spring to solicit these predictions. We will use their data to build a PageRank metric for team strength for the 1995–1996 regular season (the best season in the history of the *University of Massachusetts*). To do this, we will build a directed graph whereby each team is a node, and each game creates a directed edge from the losing team to the winning team, which can be weighted based on the margin of victory. The PageRank in such a network is a measure of each team's strength.

First, we need to download the game-by-game results and a lookup table that translates the team IDs into school names. Note that Kaggle requires a sign-in, so the code below may not work for you without your using your Web browser to authenticate.

```
prefix <- "https://www.kaggle.com/c/march-machine-learning-mania-2015"
url_teams <- paste(prefix, "download/teams.csv", sep = "/")
url_games <- paste(
  prefix,
  "download/regular_season_compact_results.csv", sep = "/"
)
download.file(url_teams, destfile = "data/teams.csv")
download.file(url_games, destfile = "data/games.csv")
```

Next, we will load this data and `filter()` to select just the 1996 season.

```
library(mdsr)
teams <- readr::read_csv("data/teams.csv")
games <- readr::read_csv("data/games.csv") %>%
  filter(season == 1996)
dim(games)
```

```
[1] 4122 8
```

Since the basketball schedule is very unbalanced (each team does not play the same number of games against each other team), margin of victory seems like an important factor in determining how much better one team is than another. We will use the ratio of the winning team's score to the losing team's score as an edge weight.

```
E <- games %>%
  mutate(score_ratio = wscore/lscore) %>%
  select(lteam, wteam, score_ratio)
V <- teams %>%
  filter(team_id %in% unique(c(E$lteam, E$wteam)))
summary(g)
```

```
IGRAPH 40ff386 DN-- 305 4122 --
+ attr: name (v/c), team_name (v/c), team_id (v/n), score_ratio
| (e/n)
```

Our graph for this season contains 305 teams, who played a total of 4,122 games. The **igraph** package contains a **centrality_pagerank()** function that will compute PageRank for us. In the results below, we can see that by this measure, *George Washington University* was the highest-ranked team, followed by UMass and *Georgetown*. In reality, the 7th-ranked team, Kentucky, won the tournament by beating *Syracuse*, the 16th-ranked team. All four semifinalists (Kentucky, Syracuse, UMass, and Mississippi State) ranked in the top-16 according to PageRank, and all 8 quarterfinalists (also including Wake Forest, Kansas, Georgetown, and Cincinnati) were in the top-20. Thus, assessing team strength by computing PageRank on regular season results would have made for a high-quality prediction of the postseason results.

```
g <- g %>%
  mutate(pagerank = centrality_pagerank())
g %>%
  as_tibble() %>%
  arrange(desc(pagerank)) %>%
  head(20)
```

```
# A tibble: 20 x 4
  name   team_name     team_id pagerank
  <chr> <chr>        <dbl>    <dbl>
1 1203  G Washington  1203    0.0219
2 1269  Massachusetts 1269    0.0205
3 1207  Georgetown   1207    0.0164
4 1234  Iowa          1234    0.0143
```

5	1163	Connecticut	1163	0.0141
6	1437	Villanova	1437	0.0131
7	1246	Kentucky	1246	0.0127
8	1345	Purdue	1345	0.0115
9	1280	Mississippi St	1280	0.0114
10	1210	Georgia Tech	1210	0.0106
11	1112	Arizona	1112	0.0103
12	1448	Wake Forest	1448	0.0101
13	1242	Kansas	1242	0.00992
14	1336	Penn St	1336	0.00975
15	1185	E Michigan	1185	0.00971
16	1393	Syracuse	1393	0.00956
17	1266	Marquette	1266	0.00944
18	1314	North Carolina	1314	0.00942
19	1153	Cincinnati	1153	0.00940
20	1396	Temple	1396	0.00860

Note that these rankings are very different from simply assessing each team's record and winning percentage, since it implicitly considers *who beat whom*, and by how much. Using won-loss record alone, UMass was the best team, with a 31–1 record, while Kentucky was 4th at 28–2.

```
wins <- E %>%
  group_by(wteam) %>%
  summarize(W = n())
losses <- E %>%
  group_by(lteam) %>%
  summarize(L = n())

g <- g %>%
  left_join(wins, by = c("team_id" = "wteam")) %>%
  left_join(losses, by = c("team_id" = "lteam")) %>%
  mutate(win_pct = W / (W + L))

g %>%
  as_tibble() %>%
  arrange(desc(win_pct)) %>%
  head(20)
```

	name	team_name	team_id	pagerank	W	L	win_pct
1	1269	Massachusetts	1269	0.0205	31	1	0.969
2	1403	Texas Tech	1403	0.00548	28	1	0.966
3	1163	Connecticut	1163	0.0141	30	2	0.938
4	1246	Kentucky	1246	0.0127	28	2	0.933
5	1180	Drexel	1180	0.00253	25	3	0.893
6	1453	WI Green Bay	1453	0.00438	24	3	0.889
7	1158	Col Charleston	1158	0.00190	22	3	0.88
8	1307	New Mexico	1307	0.00531	26	4	0.867
9	1153	Cincinnati	1153	0.00940	25	4	0.862
10	1242	Kansas	1242	0.00992	25	4	0.862
11	1172	Davidson	1172	0.00237	22	4	0.846

```

12 1345  Purdue          1345  0.0115    25    5  0.833
13 1448  Wake Forest     1448  0.0101    23    5  0.821
14 1185  E Michigan       1185  0.00971   22    5  0.815
15 1439  Virginia Tech    1439  0.00633   22    5  0.815
16 1437  Villanova        1437  0.0131    25    6  0.806
17 1112  Arizona          1112  0.0103    24    6  0.8
18 1428  Utah             1428  0.00613   23    6  0.793
19 1265  Marist           1265  0.00260   22    6  0.786
20 1114  Ark Little Rock  1114  0.00429   21    6  0.778

```

```

g %>%
  as_tibble() %>%
  summarize(pr_wpct_cor = cor(pagerank, win_pct, use = "complete.obs"))

```

```

# A tibble: 1 x 1
pr_wpct_cor
<dbl>
1      0.639

```

While PageRank and winning percentage are moderately correlated, PageRank recognizes that, for example, *Texas Tech*'s 28-1 record did not even make them a top-20 team. Georgetown beat Texas Tech in the quarterfinals.

This particular graph has some interesting features. First, UMass beat Kentucky in their first game of the season.

```

E %>%
  filter(wteam == 1269 & lteam == 1246)

```

```

# A tibble: 1 x 3
lteam wteam score_ratio
<dbl> <dbl>      <dbl>
1 1246  1269      1.12

```

This helps to explain why UMass has a higher PageRank than Kentucky, since the only edge between them points to UMass. Sadly, Kentucky beat UMass in the semifinal round of the tournament—but that game is not present in this regular season data set.

Secondly, George Washington finished the regular season 21–7, yet they had the highest PageRank in the country. How could this have happened? In this case, George Washington was the only team to beat UMass in the regular season. Even though the two teams split their season series, this allows much of the mass that flows to UMass to flow to George Washington.

```

E %>%
  filter(lteam %in% c(1203, 1269) & wteam %in% c(1203, 1269))

```

```

# A tibble: 2 x 3
lteam wteam score_ratio
<dbl> <dbl>      <dbl>
1 1269  1203      1.13
2 1203  1269      1.14

```

The national network is large and complex, and therefore we will focus on the *Atlantic 10 conference* to illustrate how PageRank is actually computed. The A-10 consisted of 12 teams in 1996.

```
A_10 <- c("Massachusetts", "Temple", "G Washington", "Rhode Island",
       "St Bonaventure", "St Joseph's PA", "Virginia Tech", "Xavier",
       "Dayton", "Duquesne", "La Salle", "Fordham")
```

We can form an *induced subgraph* of our national network that consists solely of vertices and edges among the A-10 teams. We will also compute PageRank on this network.

```
a10 <- g %>%
  filter(team_name %in% A_10) %>%
  mutate(pagerank = centrality_pagerank())
summary(a10)
```

```
IGRAPH e11892b DN-- 12 107 --
+ attr: name (v/c), team_name (v/c), team_id (v/n), pagerank (v/n),
| W (v/n), L (v/n), win_pct (v/n), score_ratio (e/n)
```

We visualize this network in [Figure 20.8](#), where the size of the vertices are proportional to each team's PageRank, and the transparency of the edges is based on the ratio of the scores in that game. We note that George Washington and UMass are the largest nodes, and that all but one of the edges connected to UMass point towards it.

```
library(ggraph)
ggraph(a10, layout = 'kk') +
  geom_edge_arc(
    aes(alpha = score_ratio), color = "lightgray",
    arrow = arrow(length = unit(0.2, "cm")),
    end_cap = circle(1, 'cm'),
    strength = 0.2
  ) +
  geom_node_point(aes(size = pagerank, color = pagerank), alpha = 0.6) +
  geom_node_label(aes(label = team_name), repel = TRUE) +
  scale_alpha_continuous(range = c(0.4, 1)) +
  scale_size_continuous(range = c(1, 10)) +
  guides(
    color = guide_legend("PageRank"),
    size = guide_legend("PageRank")
  ) +
  theme_void()
```

Now, let's compute PageRank for this network using nothing but matrix multiplication. First, we need to get the transition matrix for the graph. This is the same thing as the *adjacency matrix*, with the entries weighted by the score ratios.

```
P <- a10 %>%
  igraph::as_adjacency_matrix(sparse = FALSE, attr = "score_ratio") %>%
  t()
```

However, entries in **P** need to be probabilities, and thus they need to be normalized so that each column sums to 1. We can achieve this using the `scale()` function.

```
P <- scale(P, center = FALSE, scale = colSums(P))
round(P, 2)
```

```
1173 1182 1200 1203 1247 1269 1348 1382 1386 1396 1439 1462
1173 0.00 0.09 0.00 0.00 0.09      0 0.14 0.11 0.00 0.00 0.00 0.16
```

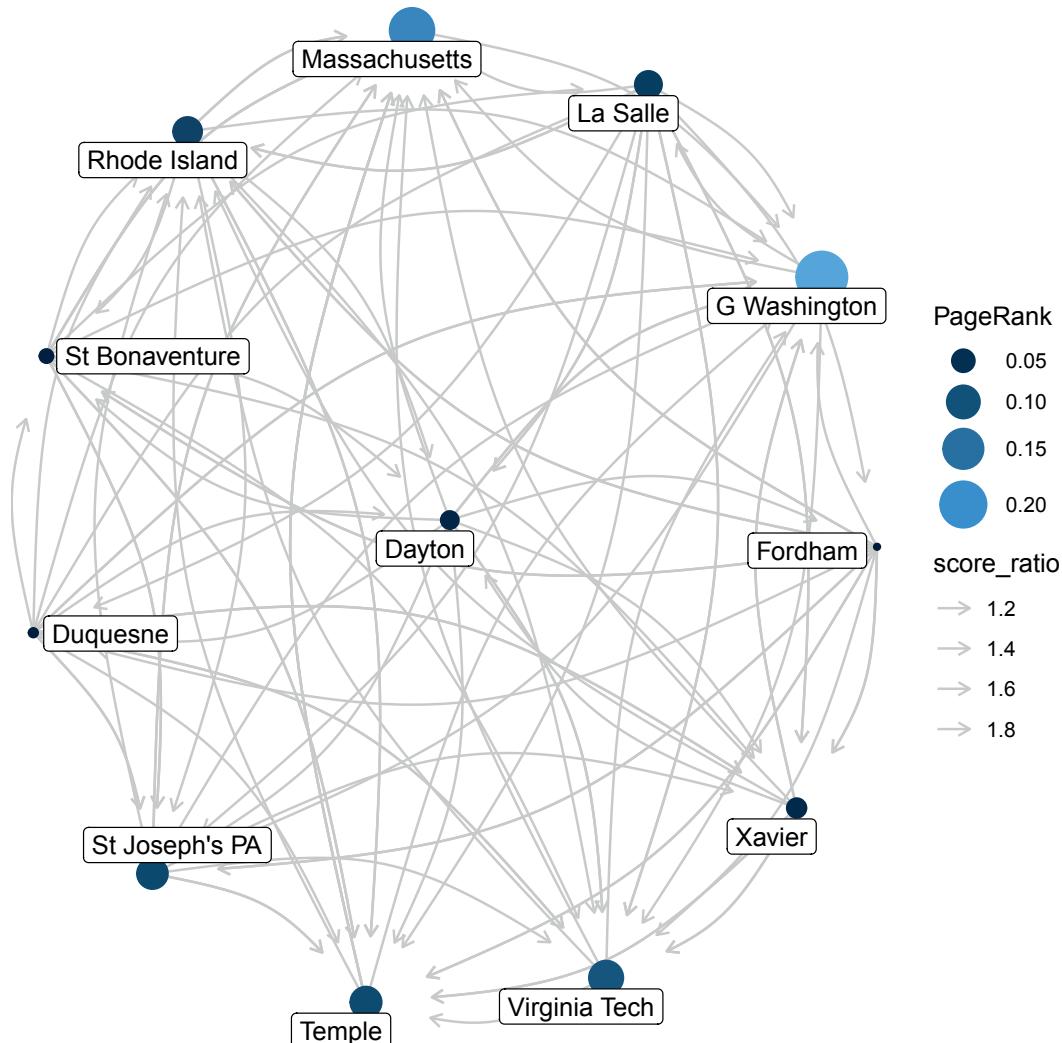


Figure 20.8: Atlantic 10 Conference network, NCAA men's basketball, 1995–1996.

1182 0.10 0.00 0.10 0.00 0.10	0 0.00 0.00 0.00 0.00 0.00 0.00
1200 0.11 0.00 0.00 0.00 0.09	0 0.00 0.00 0.00 0.00 0.00 0.00
1203 0.11 0.10 0.10 0.00 0.10	1 0.14 0.11 0.17 0.37 0.27 0.15
1247 0.00 0.09 0.00 0.25 0.00	0 0.00 0.12 0.00 0.00 0.00 0.00
1269 0.12 0.09 0.13 0.26 0.11	0 0.14 0.12 0.16 0.34 0.25 0.15
1348 0.00 0.11 0.11 0.00 0.12	0 0.00 0.12 0.16 0.29 0.21 0.18
1382 0.11 0.09 0.13 0.00 0.00	0 0.14 0.00 0.00 0.00 0.00 0.00
1386 0.11 0.10 0.10 0.24 0.09	0 0.14 0.11 0.00 0.00 0.00 0.00
1396 0.12 0.15 0.12 0.00 0.12	0 0.16 0.10 0.16 0.00 0.27 0.19
1439 0.12 0.09 0.12 0.25 0.09	0 0.14 0.11 0.17 0.00 0.00 0.17
1462 0.10 0.09 0.09 0.00 0.09	0 0.00 0.12 0.18 0.00 0.00 0.00
attr(,"scaled:scale")	
1173 1182 1200 1203 1247 1269 1348 1382 1386 1396 1439 1462	10.95 11.64 11.91 4.39 11.64 1.13 7.66 10.56 6.54 3.65 5.11 6.95

One shortcoming of this construction is that our graph has multiple edges between pairs of vertices, since teams in the same conference usually play each other twice. Unfortunately, the **igraph** function `as_adjacency_matrix()` doesn't handle this well:

If the graph has multiple edges, the edge attribute of an arbitrarily chosen edge (for the multiple edges) is included.

Even though UMass beat Temple twice, only one of those edges (apparently chosen arbitrarily) will show up in the adjacency matrix. Note also that in the transition matrix shown above, the column labeled 1269 contains a one and eleven zeros. This indicates that the probability of UMass (1269) transitioning to George Washington (1203) is 1—since UMass's only loss was to George Washington. This is not accurate, because the model doesn't handle multiple edges in a sufficiently sophisticated way. It is apparent from the matrix that George Washington is nearly equally likely to move to La Salle, UMass, St. Joseph's, and Virginia Tech—their four losses in the Atlantic 10.

Next, we'll define the initial vector with uniform probabilities—each team has an initial value of 1/12.

```
num_vertices <- nrow(as_tibble(a10))
v0 <- rep(1, num_vertices) / num_vertices
v0
```

[1] 0.0833 0.0833 0.0833 0.0833 0.0833 0.0833 0.0833 0.0833 0.0833 0.0833
[11] 0.0833 0.0833

To compute PageRank, we iteratively multiply the initial vector \mathbf{v}_0 by the transition matrix \mathbf{P} . We'll do 20 multiplications with a loop:

```
v <- v0
for (i in 1:20) {
  v <- P %*% v
}
as.vector(v)
```

[1] 0.02552 0.01049 0.00935 0.28427 0.07319 0.17688 0.08206 0.01612 0.09253
[10] 0.08199 0.11828 0.02930

We find that the fourth vertex—George Washington—has the highest PageRank. Compare these with the values returned by the built-in `page_rank()` function from **igraph**:

```
igraph::page_rank(a10)$vector
```

1173 1182 1200 1203 1247 1269 1348 1382 1386 1396
0.0346 0.0204 0.0193 0.2467 0.0679 0.1854 0.0769 0.0259 0.0870 0.0894
1439 1462
0.1077 0.0390

Why are they different? One limitation of PageRank as we've defined it is that there could be *sinks*, or *spider traps*, in a network. These are individual nodes, or even a collection of nodes, out of which there are no outgoing edges. (UMass is nearly—but not quite—a spider

trap in this network.) In this event, if random surfers find themselves in a spider trap, there is no way out, and all of the probability will end up in those vertices. In practice, PageRank is modified by adding a *random restart*. This means that every so often, the random surfer simply picks up and starts over again. The parameter that controls this in `page_rank()` is called `damping`, and it has a default value of 0.85. If we set the `damping` argument to 1, corresponding to the matrix multiplication we did above, we get a little closer.

```
igraph::page_rank(a10, damping = 1)$vector
```

```
1173     1182     1200     1203     1247     1269     1348     1382     1386
0.02290 0.00778 0.00729 0.28605 0.07297 0.20357 0.07243 0.01166 0.09073
1396     1439     1462
0.08384 0.11395 0.02683
```

Alternatively, we can do the random walk again, but allow for random restarts.

```
w <- v0
d <- 0.85
for (i in 1:20) {
  w <- d * P %*% w + (1 - d) * v0
}
as.vector(w)
```

```
[1] 0.0382 0.0231 0.0213 0.2453 0.0689 0.1601 0.0866 0.0302 0.0880 0.0872
[11] 0.1106 0.0407
```

```
igraph::page_rank(a10, damping = 0.85)$vector
```

```
1173     1182     1200     1203     1247     1269     1348     1382     1386     1396
0.0346 0.0204 0.0193 0.2467 0.0679 0.1854 0.0769 0.0259 0.0870 0.0894
1439     1462
0.1077 0.0390
```

Again, the results are not exactly the same due to the approximation of values in the adjacency matrix \mathbf{P} mentioned earlier, but they are quite close.

20.5 Further resources

There are two popular workhorse **R** packages for network analysis: **igraph** and **sna**. Both have large user bases and are actively developed. **igraph** also has bindings for Python and C, see Chapter 21.

For more sophisticated graph visualization software, see *Gephi*. In addition to **igraph**, the **ggnetwork**, **sna**, and **network** **R** packages are useful for working with graph objects.

Albert-László Barabási's book *Linked* is a popular introduction to network science (Barabási and Frangos, 2014). For a broader undergraduate textbook, see Easley and Kleinberg (2010).

20.6 Exercises

Problem 1 (Medium): The following problem considers the U.S. airport network as a graph.

- a. What information do you need to compute the PageRank of the U.S. airport network? Write an SQL query to retrieve this information for 2012. (Hint: use the `dbConnect_scidb` function to connect to the `airlines` database.)
- b. Use the data you pulled from SQL and build the network as a *weighted* `tidygraph` object, where the weights are proportional to the frequency of flights between each pair of airports.
- c. Compute the PageRank of each airport in your network. What are the top-10 “most central” airports? Where does Oakland International Airport `OAK` rank?
- d. Update the vertex attributes of your network with the geographic coordinates of each airport (available in the `airports` table).
- e. Use `ggraph` to draw the airport network. Make the thickness or transparency of each edge proportional to its weight.
- f. Overlay your airport network on a U.S. map (see the spatial data chapter).
- g. Project the map and the airport network using the Lambert Conformal Conic projection.
- h. Crop the map you created to zoom in on your local airport.

Problem 2 (Hard): Let’s reconsider the Internet Movie Database (IMDb) example.

- a. In the `CROSS JOIN` query in the movies example, how could we have modified the SQL query to include the actor’s and actresses’ names in the original query? Why would this have been less efficient from a computational and data storage point of view?
- b. Expand the Hollywood network by going further back in time. If you go back to 2000, which actor/actress has the highest degree centrality? Betweenness centrality? Eigenvector centrality?

Problem 3 (Hard): Use the `dbConnect_scidb` function to connect to the `airlines` database using the data from 2013 to answer the following problem. For a while, Edward Snowden was trapped in a Moscow airport. Suppose that you were trapped not in *one* airport, but in *all* airports. If you were forced to randomly fly around the United States, where would you be most likely to end up?

20.7 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-networks.html#networks-online-exercises>



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

21

Epilogue: Towards “big data”

The terms *data science* and *big data* are often used interchangeably, but this is not correct. Technically, “big data” is a part of data science: the part that deals with data that are so large that they cannot be handled by an ordinary computer. This book provides what we hope is a broad—yet principled—introduction to data science, but it does not specifically prepare the reader to work with big data. Rather, we see the concepts developed in this book as “precursors” to big data (Horton et al., 2015; Horton and Hardin, 2015). In this epilogue, we explore notions of big data and point the reader towards technologies that scale for truly big data.

21.1 Notions of big data

Big data is an exceptionally hot topic, but it is not so well-defined. Wikipedia states:

Big data is a field that treats ways to analyze, systematically extract information from, or otherwise deal with data sets that are too large or complex to be dealt with by traditional data-processing application software.

Relational database management systems, desktop statistics and software packages used to visualize data often have difficulty handling big data. The work may require “massively parallel software running on tens, hundreds, or even thousands of servers.” What qualifies as being “big data” varies depending on the capabilities of the users and their tools, and expanding capabilities make big data a moving target. “For some organizations, facing hundreds of gigabytes of data for the first time may trigger a need to reconsider data management options. For others, it may take tens or hundreds of terabytes before data size becomes a significant consideration” (retrieved December 2020).

Big data is often characterized by the three V’s: volume, velocity, and variety (Laney, 2001). Under this definition, the qualities that make big data different are its *size*, how *quickly* it grows as it is collected, and how many different *formats* it may come in. In big data, the size of tables may be too large to fit on an ordinary computer, the data and queries on it may be coming in too quickly to process, or the data may be distributed across many different systems. Randall Pruim puts it more concisely: “Big data is when your workflow breaks.”

Both relative and absolute definitions of big data are meaningful. The absolute definition

may be easier to understand: We simply specify a data size and agree that any data that are at least that large are “big”—otherwise they are not. The problem with this definition is that it is a moving target. It might mean *petabytes* (1,000 terabytes) today, but *exabytes* (1,000 petabytes) a few years from now. Regardless of the precise definition, it is increasingly clear that while many organizations like Google, Facebook, and Amazon are working with truly big data, most individuals—even data scientists like you and us—are not.

For us, the relative definition becomes more meaningful. A big data problem occurs when the workflow that you have been using to solve problems becomes infeasible due to the expansion in the size of your data. It is useful in this context to think about *orders of magnitude* of data. The evolution of baseball data illustrates how “big data problems” have arisen as the volume and variety of the data has increased over time.

- **Individual game data:** Henry Chadwick started collecting boxscores (a tabular summary of each game) in the early 1900s. These data (dozens or even hundreds of rows) can be stored on handwritten pieces of paper, or in a single spreadsheet. Each row might represent one *game*. Thus, a perfectly good workflow for working with data of this size is to store them on paper. A more sophisticated workflow would be to store them in a spreadsheet application.
- **Seasonal data:** By the 1970s, decades of baseball history were recorded in a seasonal format. Here, the data are aggregated at the *player-team-season* level. An example of this kind of data is the **Lahman** database we explored in [Chapter 4](#), which has nearly 100,000 rows in the **Batting** table. Note that in this seasonal format, we know how many home runs each player hit for each team, but we don’t know anything about *when* they were hit (e.g., in what month or what inning). *Excel* is limited in the number of rows that a single spreadsheet can contain. The original limit of $2^{14} = 16,384$ rows was bumped up to $2^{16} = 65,536$ rows in 2003, and the current limit is $2^{20} \approx 1$ million rows. Up until 2003, simply opening the **Batting** table in Excel would have been impossible. This is a big data problem, because your Excel workflow has broken due to the size of your data. On the other hand, opening the **Batting** table in **R** requires far less memory, since **R** does not try to display all of the data.
- **Play-by-play data:** By the 1990s, Retrosheet began collecting even more granular play-by-play data. Each row contains information about one *play*. This means that we know exactly when each player hit each home run—what date, what inning, off of which pitcher, which other runners were on base, and even which other players were in the field. As of this writing nearly 100 seasons occupying more than 10 million rows are available. This creates a big data problem for **R**—you would have a hard time loading these data into **R** on a typical personal computer. However, SQL provides a scalable solution for data of this magnitude, even on a laptop. Still, you will experience significantly better performance if these data are stored in an SQL cluster with lots of memory.
- **Camera-tracking data:** The Statcast data set contains (x, y, z) -coordinates for all fielders, baserunners, and the ball every $1/15^{\text{th}}$ of a second. Thus, each row is a moment in time. These data indicate not just the outcome of each play, but exactly where each of the players on the field and the ball were as the play evolved. These data are several gigabytes per game, which translates into many terabytes per season. Thus, some sort of distributed server system would be required just to store these data. These data are “big” in the relative sense for any individual, but they are still orders of magnitude away from being “big” in the absolute sense.

What does absolutely big data look like? For an individual user, you might consider the 13.5-terabyte data set of 110 billion events released in 2015 by Yahoo! for use in machine

learning research. The grand-daddy of data may be the *Large Hadron Collider* in Europe, which is generating 25 petabytes of data per year (CERN, 2008). However, only 0.001% of all of the data that is being generated by the supercollider is being saved, because to collect it all would mean capturing nearly 500 exabytes *per day*. This is clearly big data.

21.2 Tools for bigger data

By now, you have a working knowledge of both **R** and SQL. These are battle-tested, valuable tools for working with small and medium data. Both have large user bases, ample deployment, and continue to be very actively developed. Some of that development seeks to make **R** and SQL more useful for truly large data. While we don't have the space to cover these extensions in detail, in this section we outline some of the most important concepts for working with big data, and highlight some of the tools you are likely to see on this frontier of your working knowledge.

21.2.1 Data and memory structures for big data

An alternative to **dplyr**, **data.table** is a popular **R** package for fast SQL-style operations on very large data tables (many gigabytes of memory). It is not clear that **data.table** is faster or more efficient than **dplyr**, and it uses a different—but not necessarily better—syntax. Moreover, **dplyr** can use **data.table** itself as a backend. We have chosen to highlight **dplyr** in this book primarily because it fits so well syntactically with a number of other **R** packages we use herein (i.e., the **tidyverse**).

For some problems—more common in machine learning—the number of explanatory variables p can be large (not necessarily relative to the number of observations n). In such cases, the algorithm to compute a least-squares regression model may eat up quite a bit of memory. The **biglm** package seeks to improve on this by providing a memory-efficient **biglm()** function that can be used in place of **lm()**. In particular, **biglm** can fit generalized linear models with data frames that are larger than memory. The package accomplishes this by splitting the computations into more manageable chunks—updating the results iteratively as each chunk is processed. In this manner, you can write a drop-in replacement for your existing code that will scale to data sets larger than the memory on your computer.

```
library(tidyverse)
library(mdsr)
library(biglm)
library(bench)
n <- 20000
p <- 500
d <- rnorm(n * (p + 1)) %>%
  matrix(ncol = (p + 1)) %>%
  as_tibble(.name_repair = "unique")
expl_vars <- names(d) %>%
  tail(-1) %>%
  paste(collapse = " + ")
my_formula <- as.formula(paste("...1 ~ ", expl_vars))

system_time(lm(my_formula, data = d))
```

```

process    real
 4.18s   4.24s
system_time(biglm(my_formula, data = d))

process    real
 2.53s   2.56s

```

Here we see that the computation completed more quickly (and can be updated to incorporate more observations, unlike `lm()`). The **biglm** package is also useful in settings where there are many observations but not so many predictors. A related package is **bigmemory**. This package extends **R**’s capabilities to map memory to disk, allowing you to work with larger matrices.

21.2.2 Compilation

Python, SQL, and **R** are *interpreted programming languages*. This means that the code that you write in these languages gets translated into machine language on-the-fly as you execute it. The process is not altogether different than when you hear someone speaking in Russian on the news, and then you hear a halting English translation with a one- or two-second delay. Most of the time, the translation happens so fast that you don’t even notice.

Imagine that instead of translating the Russian speaker’s words on-the-fly, the translator took dictation, wrote down a thoughtful translation, and then re-recorded the segment in English. You would be able to process the English-speaking segment faster—because you are fluent in English. At the same time, the translation would probably be better, since more time and care went into it, and you would likely pick up subtle nuances that were lost in the on-the-fly translation. Moreover, once the English segment is recorded, it can be watched at any time without incurring the cost of translation again.

This alternative paradigm involves a one-time translation of the code called *compilation*. **R** code is not compiled (it is interpreted), but C++ code is. The result of compilation is a binary program that can be executed by the CPU directly. This is why, for example, you can’t write a desktop application in **R**, and executables written in C++ will be much faster than scripts written in **R** or Python. (To continue this line of reasoning, binaries written in assembly language can be faster than those written in C++, and binaries written in machine language can be faster than those written in assembly.)

If C++ is so much faster than **R**, then why write code in **R**? Here again, it is a trade-off. The code written in C++ may be faster, but when your programming time is taken into account you can often accomplish your task much faster by writing in **R**. This is because **R** provides extensive libraries that are designed to reduce the amount of code that you have to write. **R** is also interactive, so that you can keep a session alive and continue to write new code as you run the old code. This is fundamentally different from C++ development, where you have to re-compile every time you change a single line of code. The convenience of **R** programming comes at the expense of speed.

However, there is a compromise. **Rcpp** allows you to move certain pieces of your **R** code to C++. The basic idea is that **Rcpp** provides C++ data structures that correspond to **R** data structures (e.g., a `data.frame` data structure written in C++). It is thus possible to write functions in **R** that get compiled into faster C++ code, with minimal additional

effort on the part of the **R** programmer. The **dplyr** package makes extensive use of this functionality to improve performance.

21.2.3 Parallel and distributed computing

21.2.3.1 Embarrassingly parallel computing

How do you increase a program's capacity to work with larger data? The most obvious way is to add more memory (i.e., *RAM*) to your computer. This enables the program to read more data at once, enabling greater functionality with any additional programming. But what if the bottleneck is not the memory, but the processor (*CPU*)? A processor can only do one thing at a time. So if you have a computation that takes t units of time, and you have to do that computation for many different data sets, then you can expect that it will take many more units of time to complete.

For example, suppose we generate 20 sets of 1 million (x, y) random pairs and want to fit a regression model to each set.

```
n <- 1e6
k <- 20
d <- tibble(y = rnorm(n*k), x = rnorm(n*k), set = rep(1:k, each = n))

fit_lm <- function(data, set_id) {
  data %>%
    filter(set == set_id) %>%
    lm(y ~ x, data = .)
}
```

However long it takes to do it for the first set, it should take about 20 times as long to do it for all 20 sets. This is as expected, since the computation procedure was to fit the regression model for the first set, then fit it for the second set, and so on.

```
system_time(map(1:1, fit_lm, data = d))
```

```
process      real
  449ms   455ms

system_time(map(1:k, fit_lm, data = d))

process      real
  4.88s   4.92s
```

However, in this particular case, the data in each of the 20 sets has nothing to do with the data in any of the other sets. This is an example of an *embarrassingly parallel* problem. These data are ripe candidates for a *parallelized* computation. If we had 20 processors, we could fit one regression model on each CPU—all at the same time—and get our final result in about the same time as it takes to fit the model to *one* set of data. This would be a tremendous improvement in speed.

Unfortunately, we don't have 20 CPUs. Nevertheless, most modern computers have multiple cores.

```
library(parallel)
my_cores <- detectCores()
my_cores
```

```
[1] 8
```

The **parallel** package provides functionality for parallel computation in **R**. The **furrr** package extends the **future** package to allow us to express embarrassingly parallel computations in our familiar **purrr** syntax (Vaughan and Dancho, 2020). Specifically, it provides a function **future_map()** that works just like **map()** (see [Chapter 7](#)), except that it spreads the computations over multiple cores. The theoretical speed-up is a function of **my_cores**, but in practice this may be less for a variety of reasons (most notably, the overhead associated with combining the parallel results).

The **plan** function sets up a parallel computing environment. In this case, we are using the **multiprocess** mode, which will split computations across asynchronous separate **R** sessions. The **workers** argument to **plan()** controls the number of cores being used for parallel computation. Next, we fit the 20 regression models using the **future_map()** function instead of **map**. Once completed, set the computation mode back to **sequential** for the remainder of this chapter.

```
library(furrr)
plan(multiprocess, workers = my_cores)

system_time(
  future_map(1:k, fit_lm, data = d)
)

process    real
  9.98s  11.56s
plan(sequential)
```

In this case, the overhead associated with combining the results was larger than the savings from parallelizing the computation. But this will not always be the case.

21.2.3.2 GPU computing and CUDA

Another fruitful avenue to speed up computations is through use of a graphical processing unit (GPU). These devices feature a highly parallel structure that can lead to significant performance gains. *CUDA* is a parallel computing platform and application programming interface created by *NVIDIA* (one of the largest manufacturers of GPUs). The **OpenCL** package provides bindings for **R** to the open-source, general-purpose OpenCL programming language for GPU computing.

21.2.3.3 MapReduce

MapReduce is a programming paradigm for parallel computing. To solve a task using a MapReduce framework, two functions must be written:

1. **Map(key_0, value_0)**: The **Map()** function reads in the original data (which is stored in key-value pairs), and splits it up into smaller subtasks. It returns a **list** of key-value pairs ($key_1, value_1$), where the keys and values are not necessarily of the same type as the original ones.
2. **Reduce(key_1, list(value_1))**: The MapReduce implementation has a method for aggregating the key-value pairs returned by the **Map()** function by their keys (i.e., **key_1**). Thus, you only have to write the **Reduce()** function, which takes as input a particular **key_1**, and a list of all the **value_1**'s that correspond to **key_1**.

The `Reduce()` function then performs some operation on that list, and returns a list of values.

MapReduce is efficient and effective because the `Map()` step can be highly parallelized. Moreover, MapReduce is also fault tolerant, because if any individual `Map()` job fails, the controller can simply start another one. The `Reduce()` step often provides functionality similar to a `GROUP BY` operation in SQL.

Example

The canonical MapReduce example is to tabulate the frequency of each word in a large number of text documents (i.e., a *corpus* (see [Chapter 19](#))). In what follows, we show an implementation written in Python by Bill Howe of the *University of Washington* (Howe, 2014). Note that at the beginning, this bit of code calls an external `MapReduce` library that actually implements MapReduce. The user only needs to write the two functions shown in this block of code—not the MapReduce library itself.

```
import MapReduce
import sys

mr = MapReduce.MapReduce()

def mapper(record):
    key = record[0]
    value = record[1]
    words = value.split()
    for w in words:
        mr.emit_intermediate(w, 1)

def reducer(key, list_of_values):
    total = 0
    for v in list_of_values:
        total += v
    mr.emit((key, total))

if __name__ == '__main__':
    inputdata = open(sys.argv[1])
    mr.execute(inputdata, mapper, reducer)
```

We will use this MapReduce program to compile a word count for the issues raised on GitHub for the `ggplot2` package. These are stored in a *JSON* file (see [Chapter 6](#)) as a single JSON array. Since we want to illustrate how MapReduce can parallelize over many files, we will convert this single array into a JSON object for each issue. This will mimic the typical use case. The `jsonlite` package provides functionality for converting between JSON objects and native **R** data structures.

```
library(jsonlite)
url <- "https://api.github.com/repos/tidyverse/ggplot2/issues"
gg_issues <- url %>%
  fromJSON() %>%
  select(url, body) %>%
  group_split(url) %>%
```

```
map_chr(~toJSON(as.character(.x))) %>%
write(file = "code/map-reduce/issues.json")
```

For example, the first issue is displayed below. Note that it consists of two comma-separated character strings within brackets. We can think of this as having the format: [key, value].

```
readLines("code/map-reduce/issues.json") %>%
head(1) %>%
str_wrap(width = 70) %>%
cat()
```

```
["https://api.github.com/repos/tidyverse/ggplot2/issues/4019", "When
setting the limits of `scale_fill_steps()`, the fill brackets in
the legend becomes unevenly spaced. It's not clear why or how this
happens.\r\n\r\n``` r\r\nlibrary(tidyverse)\r\n\r\nr\nndf <- tibble(\r\n
crossing(\r\n  tibble(sample = paste(\"sample\", 1:4)),\r\n  tibble(pos
= 1:4)\r\n),\r\n  val = runif(16)\r\n)\r\nr\nr\nnggplot(df, aes(x =
pos, y = sample)) +\r\n  geom_line() +\r\n  geom_point(aes(fill =
val), pch = 21, size = 7) +\r\n  scale_fill_steps(low = \"white\",
high = \"black\")\r\n```\r\n!\n[] (https://i.imgur.com/6zoByUA.png)
\r\nr\nr\n``` r\r\nr\nr\nnggplot(df, aes(x = pos, y = sample)) +\r\n  geom_line() +\r\n  geom_point(aes(fill = val), pch = 21, size = 7) +
\r\n  scale_fill_steps(low = \"white\", high = \"black\", limits
= c(0, 1))\r\n```\r\n!\n[] (https://i.imgur.com/M0LWbjX.png)
\r\nr\nr\n<sup>Created on 2020-05-22 by the [reprex package] (https://reprex.tidyverse.org) (v0.3.0)</sup>"]
```

In the Python code written above (which is stored in the file `wordcount.py`), the `mapper()` function takes a `record` argument (i.e., one line of the `issues.json` file), and examines its first two elements—the `key` becomes the first argument (in this case, the URL of the GitHub issue) and the `value` becomes the second argument (the text of the issue). After splitting the `value` on each space, the `mapper()` function emits a `(key, value)` pair for each word. Thus, the first issue shown above would generate the pairs: `(When, 1)`, `(setting, 1)`, `(the, 1)`, etc.

The `MapReduce` library provides a mechanism for efficiently collecting all of the resulting pairs based on the `key`, which in this case corresponds to a single word. The `reducer()` function simply adds up all of the values associated with each key. In this case, these values are all `1`s, so the resulting pair is a word and the number of times it appears (e.g., `(the, 158)`, etc.).

Thanks to the `reticulate` package, we can run this Python script from within **R** and bring the results into **R** for further analysis. We see that the most common words in this corpus are short articles and prepositions.

```
library(mdsr)
cmd <- "python code/map-reduce/wordcount.py code/map-reduce/issues.json"
res <- system(cmd, intern = TRUE)
freq_df <- res %>%
  purrr::map(jsonlite::fromJSON) %>%
  purrr::map(set_names, c("word", "count")) %>%
  bind_rows() %>%
```

```

  mutate(count = parse_number(count))
glimpse(freq_df)

Rows: 1,605
Columns: 2
$ word <chr> "geom_point(aes(fill", "aliased", "desirable", "ggplot(ct...
$ count <dbl> 2, 1, 1, 1, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
freq_df %>%
  filter(str_detect(pattern = "[a-z]", word)) %>%
  arrange(desc(count)) %>%
  head(10)

# A tibble: 10 × 2
  word   count
  <chr> <dbl>
1 the     147
2 to      87 
3 a       63 
4 of      45 
5 is      43 
6 in      33 
7 and     32 
8 that    31 
9 it      28 
10 be     28 

```

MapReduce is popular and offers some advantages over SQL for some problems. When MapReduce first became popular, and Google used it to redo their webpage ranking system (see Chapter 20), there was great excitement about a coming “paradigm shift” in parallel and distributed computing. Nevertheless, advocates of SQL have challenged the notion that it has been completely superseded by MapReduce (Stonebraker et al., 2010).

21.2.3.4 Hadoop

As noted previously, MapReduce requires a software implementation. One popular such implementation is Hadoop MapReduce, which is one of the core components of *Apache Hadoop*. Hadoop is a larger software ecosystem for storing and processing large data that includes a distributed file system, Pig, Hive, Spark, and other popular open-source software tools. While we won’t be able to go into great detail about these items, we will illustrate how to interface with Spark, which has a particularly tight integration with **RStudio**.

21.2.3.5 Spark

One nice feature of *Apache Spark*—especially for our purposes—is that while it requires a distributed file system, it can implement a pseudo-distributed file system on a single machine. This makes it possible for you to experiment with Spark on your local machine even if you don’t have access to a cluster. For obvious reasons, you won’t actually see the performance boost that parallelism can bring, but you can try it out and debug your code. Furthermore, the **sparklyr** package makes it painless to install a local Spark cluster from within **R**, as well as connect to a local or remote cluster.

Once the **sparklyr** package is installed, we can use it to install a local Spark cluster.

```
library(sparklyr)
spark_install(version = "3.0") # only once!
```

Next, we make a connection to our local Spark instance from within **R**. Of course, if we were connecting to a remote Spark cluster, we could modify the `master` argument to reflect that. Spark requires *Java*, so you may have to install the *Java Development Kit* before using Spark.¹

```
# sudo apt-get install openjdk-8-jdk
sc <- spark_connect(master = "local", version = "3.0")
class(sc)
```

```
[1] "spark_connection"      "spark_shell_connection"
[3] "DBIConnection"
```

Note that `sc` has class `DBIConnection`—this means that it can do many of the things that other **dplyr** connections can do. For example, the `src_tbls()` function works just like it did on the MySQL connection objects we saw in [Chapter 15](#).

```
src_tbls(sc)
```

```
character(0)
```

In this case, there are no tables present in this Spark cluster, but we can add them using the `copy_to()` command. Here, we will load the `babynames` table from the **babynames** package.

```
babynames_tbl <- sc %>%
  copy_to(babynames::babynames, "babynames")
src_tbls(sc)
```

```
[1] "babynames"
```

```
class(babynames_tbl)
```

```
[1] "tbl_spark" "tbl_sql"   "tbl_lazy"  "tbl"
```

The `babynames_tbl` object is a `tbl_spark`, but also a `tbl_sql`. Again, this is analogous to what we saw in [Chapter 15](#), where a `tbl_MySQLConnection` was also a `tbl_sql`.

```
babynames_tbl %>%
  filter(name == "Benjamin") %>%
  group_by(year) %>%
  summarize(N = n(), total_births = sum(n)) %>%
  arrange(desc(total_births)) %>%
  head()
```

```
# Source:      spark<?> [?? x 3]
# Ordered by: desc(total_births)
  year      N total_births
  <dbl> <dbl>      <dbl>
1 1989      2       15785
2 1988      2       15279
3 1987      2       14953
4 2000      2       14864
```

¹Please check **sparklyr** for current information about which versions of the JDK are supported by which versions of Spark.

```
5 1990      2        14660
6 2016      2        14641
```

As we will see below with *Google BigQuery*, even though Spark is a parallelized technology designed to supersede SQL, it is still useful to know SQL in order to use Spark. Like BigQuery, **sparklyr** allows you to work with a Spark cluster using the familiar **dplyr** interface.

As you might suspect, because `babynames_tbl` is a `tbl_sql`, it implements SQL methods common in **DBI**. Thus, we can also write SQL queries against our Spark cluster.

```
library(DBI)
dbGetQuery(sc, "SELECT year, sum(1) as N, sum(n) as total_births
                 FROM babynames WHERE name == 'Benjamin'
                 GROUP BY year
                 ORDER BY total_births desc
                 LIMIT 6")
```

	year	N	total_births
1	1989	2	15785
2	1988	2	15279
3	1987	2	14953
4	2000	2	14864
5	1990	2	14660
6	2016	2	14641

Finally, because Spark includes not only a database infrastructure, but also a machine learning library, **sparklyr** allows you to fit many of the models we outlined in [Chapter 11](#) and 12 within Spark. This means that you can rely on Spark's big data capabilities without having to bring all of your data into **R**'s memory.

As a motivating example, we fit a multiple regression model for the amount of rainfall at the MacLeish field station as a function of the temperature, pressure, and relative humidity.

```
library(macleish)
weather_tbl <- copy_to(sc, whately_2015)
weather_tbl %>%
  ml_linear_regression(rainfall ~ temperature + pressure + rel_humidity) %>%
  summary()
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-0.041290	-0.021761	-0.011632	-0.000576	15.968356

Coefficients:

(Intercept)	temperature	pressure	rel_humidity
0.717754	0.000409	-0.000755	0.000438

R-Squared: 0.004824

Root Mean Squared Error: 0.1982

The most recent versions of **RStudio** include integrated support for management of Spark clusters.

21.2.4 Alternatives to SQL

Relational database management systems can be spread across multiple computers into what is called a *cluster*. In fact, it is widely acknowledged that one of the things that allowed Google to grow so fast was its use of the open-source (zero cost) MySQL RDBMS running as a cluster across many identical low-cost servers. That is, rather than investing large amounts of money in big machines, they built a massive MySQL cluster over many small, cheap machines. Both MySQL and PostgreSQL provide functionality for extending a single installation to a cluster.

Pro Tip 48. Use a cloud-based computing service, such as Amazon Web Services, Google Cloud Platform, or Digital Ocean, for a low-cost alternative to building your own server farm (many of these companies offer free credits for student and instructor use).

21.2.4.1 BigQuery

BigQuery is a Web service offered by Google. Internally, the BigQuery service is supported by *Dremel*, the open-source version of which is *Apache Drill*. The **bigrquery** package for **R** provides access to BigQuery from within **R**.

To use the BigQuery service, you need to sign up for an account with Google, but you won’t be charged unless you exceed the free limit of 10,000 requests per day (the BigQuery sandbox provides free access subject to certain limits).

If you want to use your own data, you have to upload it to Google Cloud Storage, but Google provides many data sets that you can use for free (e.g., COVID, Census, real-estate transactions). Here we illustrate how to query the *shakespeare* data set—which is a list of all of the words that appear in Shakespeare’s plays—to find the most common words. Note that BigQuery understands a recognizable dialect of SQL—what makes BigQuery special is that it is built on top of Google’s massive computing architecture.

```
library(bigrquery)
project_id <- "my-google-id"

sql <- "
SELECT word
, count(distinct corpus) AS numPlays
, sum(word_count) AS N
FROM [publicdata:samples.shakespeare]
GROUP BY word
ORDER BY N desc
LIMIT 10
"
bq_project_query(sql, project = project_id)
```

	word	numPlays	N
1	the	42	25568
2	I	42	21028
3	and	42	19649
4	to	42	17361
5	of	42	16438
6	a	42	13409
7	you	42	12527

```
8   my      42 11291
9   in      42 10589
10  is      42  8735
```

21.2.4.2 NoSQL

NoSQL refers not to a specific technology, but rather to a class of database architectures that are *not* based on the notion—so central to SQL (and `data.frames` in **R**)—that a table consists of a rectangular array of rows and columns. Rather than being built around tables, NoSQL databases may be built around columns, key-value pairs, documents, or graphs. Nevertheless NoSQL databases may (or may not) include an SQL-like query language for retrieving data.

One particularly successful NoSQL database is *MongoDB*, which is based on a document structure. In particular, MongoDB is often used to store JSON objects (see [Chapter 6](#)), which are not necessarily tabular.

21.3 Alternatives to R

Python is a widely-used general-purpose, high-level programming language. You will find adherents for both **R** and Python, and while there are ongoing debates about which is “better,” there is no consensus. It is probably true that—for obvious reasons—computer scientists tend to favor Python, while statisticians tend to favor **R**. We prefer the latter but will not make any claims about its being “better” than Python. A well-rounded data scientist should be competent in both environments.

Python is a modular environment (like **R**) and includes many libraries for working with data. The most **R**-like is `Pandas`, but other popular auxiliary libraries include `SciPy` for scientific computation, `NumPy` for large arrays, `matplotlib` for graphics, and `scikit-learn` for machine learning.

Other popular programming languages among data scientists include *Scala* and *Julia*. Scala supports a *functional programming* paradigm that has been promoted by Wickham (2019a) and other **R** users. Julia has a smaller user base but has nonetheless many strong adherents.

21.4 Closing thoughts

Advances in computing power and the internet have changed the field of statistics in ways that only the greatest visionaries could have imagined. In the 20th century, the science of extracting meaning from data focused on developing inferential techniques that required sophisticated mathematics to squeeze the most information out of small data. In the 21st century, the science of extracting meaning from data has focused on developing powerful computational tools that enable the processing of ever larger and more complex data. While the essential analytical language of the last century—mathematics—is still of great importance, the analytical language of this century is undoubtedly programming. The ability to write code is a necessary but not sufficient condition for becoming a data scientist.

We have focused on programming in **R**, a well-worn interpreted language designed by statisticians for computing with data. We believe that as an open-source language with a broad following, **R** has significant staying power. Yet we recognize that all technological tools eventually become obsolete. Nevertheless, by absorbing the lessons in this book, you will have transformed yourself into a competent, ethical, and versatile data scientist—one who possesses the essential capacities for working with a variety of data programmatically. You can build and interpret models, query databases both local and remote, make informative and interactive maps, and wrangle and visualize data in various forms. Internalizing these abilities will allow them to permeate your work in whatever field interests you, for as long as you continue to use data to inform.

21.5 Further resources

Tools for working with big data analytics are developing more quickly than any of the other topics in this book. A special issue of the *The American Statistician* addressed the training of students in statistics and data science (Horton and Hardin, 2015). The issue included articles on teaching statistics at “Google-Scale” (Chamandy et al., 2015) and on the teaching of data science more generally (Baumer, 2015b; Hardin et al., 2015). The board of directors of the American Statistical Association endorsed the *Curriculum Guidelines for Undergraduate Programs in Data Science* written by the Park City Math Institute (PCMI) Undergraduate Faculty Group (De Veaux et al., 2017). These guidelines recommended fusing statistical thinking into the teaching of techniques to solve big data problems.

A comprehensive survey of **R** packages for parallel computation and high-performance computing is available through the CRAN task view on that subject. The *Parallel R* book is another resource (McCallum and Weston, 2011).

More information about Google BigQuery can be found at their website. A tutorial for SparkR is available on Apache’s website.

Part IV

Part IV: Appendices



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

A

Packages used in this book

A.1 The mdsr package

The **mdsr** package contains most of the small data sets used in this book that are not available in other packages. To install it from CRAN, use `install.packages()`. To get the latest release, use the `install_github()` function from the **remotes** package. (See [Section B.4.1](#) for more comprehensive information about **R** package maintenance.)

```
# this command only needs to be run once
install.packages("mdsr")
# if you want the development version
remotes::install_github("mdsr-book/mdsr")
```

The list of data sets provided can be retrieved using the `data()` function.

```
library(mdsr)
data(package = "mdsr")
```

The **mdsr** package includes some functions that simplify a number of tasks. In particular, the `dbConnect_scidb()` function provides a shorthand for connecting to the public SQL server hosted by *Amazon Web Services*. We use this function extensively in [Chapter 15](#) and in our classes and projects.

In keeping with best practices, **mdsr** no longer loads any other packages. In every chapter in this book, a call to `library(tidyverse)` precedes a call to `library(mdsr)`. These two steps will set up an **R** session to replicate the code in the book.

A.2 Other packages

As we discuss in [Chapters 1](#) and [21](#), this book is not explicitly about “big data”—it is about mastering data science techniques for small and medium data with an eye towards big data. To that end, we need medium-sized data sets to work with. We have introduced several such data sets in this book, namely **airlines**, **fec12**, and **fec16**.

The **airlines** package, which was inspired by the **nycflights13** package, gives **R** users the ability to download the full 33 years (and counting) of flight data from the United States Bureau of Transportation Statistics and bring it seamlessly into SQL without actually having to write any SQL code. The **macleish** package also uses the **etl** framework for hourly-updated weather data from the MacLeish field station.

The full list of packages used in this book appears below in [Tables A.1](#) and [A.2](#).

Table A.1: List of CRAN packages used in this book.

Package	Citation	Title
alr3	Weisberg (2018)	Data to Accompany Applied Linear Regression 3rd Edition
ape	Paradis et al. (2020)	Analyses of Phylogenetics and Evolution
aRxiv	Ram and Broman (2019)	Interface to the arXiv API
assertthat	Wickham (2019b)	Easy Pre and Post Assertions
available	Ganz et al. (2019)	Check if the Title of a Package is Available, Appropriate and Interesting
babynames	Wickham (2019d)	US Baby Names 1880-2017
bench	Hester (2020)	High Precision Timing of R Expressions
biglm	Lumley (2020)	Bounded Memory Linear and Generalized Linear Models
bigrquery	Wickham and Bryan (2020a)	An Interface to Google's 'BigQuery' 'API'
bookdown	Xie (2020a)	Authoring Books and Technical Documents with R Markdown
broom	Robinson et al. (2020)	Convert Statistical Objects into Tidy Tibbles
caret	Kuhn (2020a)	Classification and Regression Training
DBI	R Special Interest Group on Databases (R-SIG-DB) et al. (2019)	R Database Interface
dbplyr	Wickham and Ruiz (2020)	A 'dplyr' Back End for Databases
discrim	Kuhn (2020c)	Model Wrappers for Discriminant Analysis
dplyr	Wickham et al. (2020b)	A Grammar of Data Manipulation
DT	Xie et al. (2021)	A Wrapper of the JavaScript Library 'DataTables'
dygraphs	Vanderkam et al. (2018)	Interface to 'Dygraphs' Interactive Time Series Charting Library
etl	Baumer (2020)	Extract-Transform-Load Framework for Medium Data
extrafont	Chang (2020)	Tools for Using Fonts
fec16	Tapal et al. (2020b)	Data Package for the 2016 United States Federal Elections
flexdashboard	Iannone et al. (2020)	R Markdown Format for Flexible Dashboards
forcats	Wickham (2020a)	Tools for Working with Categorical Variables (Factors)
fs	Hester and Wickham (2020)	Cross-Platform File System Operations Based on 'libuv'
furrr	Vaughan and Dancho (2020)	Apply Mapping Functions in Parallel using Futures
future	Bengtsson (2020)	Unified Parallel and Distributed Processing in R for Everyone
GGally	Schloerke et al. (2021)	Extension to 'ggplot2'
gganimate	Pedersen and Robinson (2020)	A Grammar of Animated Graphics
ggmosaic	Jeppson et al. (2020)	Mosaic Plots in the 'ggplot2' Framework

ggplot2	Wickham et al. (2020a)	Create Elegant Data Visualisations Using the Grammar of Graphics
ggraph	Pedersen (2020a)	An Implementation of Grammar of Graphics for Graphs and Networks
ggrepel	Slowikowski (2020)	Automatically Position Non-Overlapping Text Labels with ‘ggplot2’
ggspatial	Dunnington (2021)	Spatial Data Framework for ggplot2
ggthemes	Arnold (2019b)	Extra Themes, Scales and Geoms for ‘ggplot2’
glmnet	Friedman et al. (2020)	Lasso and Elastic-Net Regularized Generalized Linear Models
googlesheets4	Bryan (2020)	Access Google Sheets using the Sheets API V4
gutenbergr	Robinson (2020)	Download and Process Public Domain Works from Project Gutenberg
haven	Wickham and Miller (2020)	Import and Export ‘SPSS’, ‘Stata’ and ‘SAS’ Files
here	Müller (2020)	A Simpler Way to Find Your Files
Hmisc	Harrell (2020)	Harrell Miscellaneous
htmlwidgets	Vaidyanathan et al. (2020)	HTML Widgets for R
igraph	Csárdi et al. (2020)	Network Analysis and Visualization
janitor	Firke (2021)	Simple Tools for Examining and Cleaning Dirty Data
jsonlite	Ooms (2020a)	A Simple and Robust JSON Parser and Generator for R
kableExtra	Zhu (2020)	Construct Complex Table with ‘kable’ and Pipe Syntax
kknn	Schliep and Hechenbichler (2016)	Weighted k-Nearest Neighbors
knitr	Xie (2020b)	A General-Purpose Package for Dynamic Report Generation in R
Lahman	Friendly et al. (2020)	Sean ‘Lahman’ Baseball Database
lars	Hastie and Efron (2013)	Least Angle Regression, Lasso and Forward Stagewise
lattice	Sarkar (2020)	Trellis Graphics for R
lazyeval	Wickham (2019e)	Lazy (Non-Standard) Evaluation
leaflet	Cheng et al. (2021)	Create Interactive Web Maps with the JavaScript ‘Leaflet’ Library
lubridate	Spinu et al. (2020)	Make Dealing with Dates a Little Easier
macleish	Baumer et al. (2020)	Retrieve Data from MacLeish Field Station
magick	Ooms (2020b)	Advanced Graphics and Image-Processing in R
mapproj	McIlroy et al. (2020)	Map Projections
maps	Brownrigg (2018)	Draw Geographical Maps
mclust	Fraley et al. (2020)	Gaussian Mixture Modelling for Model-Based Clustering, Classification, and Density Estimation
mdsr	Baumer et al. (2021)	Complement to ‘Modern Data Science with R’
modelr	Wickham (2020c)	Modelling Functions that Work with the Pipe
mosaic	Pruim et al. (2020b)	Project MOSAIC Statistics and Mathematics Teaching Utilities
mosaicData	Pruim et al. (2020a)	Project MOSAIC Data Sets

network	Butts (2020a)	Classes for Relational Data
NeuralNetTools	Beck (2018)	Visualization and Analysis Tools for Neural Networks
NHANES	Pruim (2015)	Data from the US National Health and Nutrition Examination Study
nycflights13	Wickham (2019f)	Flights that Departed NYC in 2013
packrat	Ushey et al. (2018)	A Dependency Management System for Projects and their R Package Dependencies
palmerpenguins	Horst et al. (2020)	Palmer Archipelago (Antarctica) Penguin Data
parsnip	Kuhn and Vaughan (2020a)	A Common API to Modeling and Analysis Functions
partykit	Hothorn and Zeileis (2020)	A Toolkit for Recursive Partytioning
patchwork	Pedersen (2020b)	The Composer of Plots
plotly	Sievert et al. (2020)	Create Interactive Web Graphics via ‘plotly.js’
purrr	Henry and Wickham (2020a)	Functional Programming Tools
randomForest	Breiman et al. (2018)	Breiman and Cutler’s Random Forests for Classification and Regression
RColorBrewer	Neuwirth (2014)	ColorBrewer Palettes
Rcpp	Eddelbuettel et al. (2020)	Seamless R and C++ Integration
RCurl	Temple Lang (2020)	General Network (HTTP/FTP/...) Client Interface for R
readr	Wickham and Hester (2020)	Read Rectangular Text Data
readxl	Wickham and Bryan (2019)	Read Excel Files
remotes	Hester et al. (2020)	R Package Installation from Remote Repositories, Including ‘GitHub’ Project Environments
renv	Ushey (2021)	Interface to ‘Python’
reticulate	Ushey et al. (2020)	Bindings for the ‘Geospatial’ Data Abstraction Library
rgdal	Bivand et al. (2021)	Functions for Base Types and Core R and ‘Tidyverse’ Features
rlang	Henry and Wickham (2020b)	Dynamic Documents for R
rmarkdown	Allaire et al. (2020b)	Database Interface and ‘MySQL’ Driver for R
RMySQL	Ooms et al. (2020)	Recursive Partitioning and Regression Trees
rpart	Therneau and Atkinson (2019)	Deployment Interface for R Markdown Documents and Shiny Applications
rsconnect	Allaire (2019)	‘SQLite’ Interface for R
RSQLite	Müller et al. (2020)	Easily Harvest (Scrape) Web Pages
rvest	Wickham (2020d)	R/Weka Interface
RWeka	Hornik (2020)	Scale Functions for Visualization
scales	Wickham and Seidel (2020)	R Session Information
sessioninfo	Csárdi et al. (2018)	Simple Features for R
sf	Pebesma (2021)	Web Application Framework for R
shiny	Chang et al. (2020)	

shinybusy	Meyer and Perrier (2020)	Busy Indicator for ‘Shiny’ Applications
sna	Butts (2020b)	Tools for Social Network Analysis
sp	Pebesma and Bivand (2020)	Classes and Methods for Spatial Data
sparklyr	Luraschi et al. (2020)	R Interface to Apache Spark
stopwords	Benoit et al. (2020)	Multilingual Stopword Lists
stringr	Wickham (2019g)	Simple, Consistent Wrappers for Common String Operations
styler	Müller and Walthert (2020)	Non-Invasive Pretty Printing of R Code
testthat	Wickham (2020e)	Unit Testing for R
textdata	Hvitfeldt (2020)	Download and Load Various Text Datasets
tidycensus	Walker and Herman (2020)	Load US Census Boundary and Attribute Data as ‘tidyverse’ and ‘sf’-Ready Data Frames
tidygeocoder	Cambon (2020)	Geocoding Made Easy
tidygraph	Pedersen (2020c)	A Tidy API for Graph Manipulation
tidymodels	Kuhn and Wickham (2020)	Easily Install and Load the ‘Tidymodels’ Packages
tidyrr	Wickham (2020g)	Tidy Messy Data
tidytext	Robinson and Silge (2021)	Text Mining using ‘dplyr’, ‘ggplot2’, and Other Tidy Tools
tidyverse	Wickham (2019h)	Easily Install and Load the ‘Tidyverse’
tigris	Walker (2020a)	Load Census TIGER/Line Shapefiles
tm	Feinerer and Hornik (2020)	Text Mining Package
transformr	Pedersen (2020d)	Polygon and Path Transformations
twitteR	Gentry (2015)	R Based Twitter Client
units	Pebesma et al. (2020)	Measurement Units for R Vectors
usethis	Wickham and Bryan (2020b)	Automate Package and Project Setup
viridis	Garnier (2018a)	Default Color Maps from ‘matplotlib’
viridisLite	Garnier (2018b)	Default Color Maps from ‘matplotlib’ (Lite Version)
webshot	Chang (2019)	Take Screenshots of Web Pages
wordcloud	Fellows (2018)	Word Clouds
wru	Khanna and Imai (2020)	Who are You? Bayesian Prediction of Racial Category Using Surname and Geolocation
xaringanthememer	Aden-Buie (2020)	Custom ‘xaringan’ CSS Themes
xfun	Xie (2021)	Miscellaneous Functions by ’Yihui Xie’
xkcd	Torres-Manzanera (2018)	Plotting ggplot2 Graphics in an XKCD Style
yardstick	Kuhn and Vaughan (2020b)	Tidy Characterizations of Model Performance

Table A.2: List of GitHub packages used in this book.

Package	GitHub User	Citation	Title
etude	dtkaplan	Kaplan (2020)	Utilities for Handling Textbook Exercises with Knitr

fec12	baumer-lab	Tapal et al. (2020a)	Data Package for 2012 Federal Elections
openrouteservice	GIScience	Oleś (2020)	Openrouteservice API Client
streamgraph	hrbrmstr	Rudis (2019)	Build Streamgraph Visualizations

A.3 Further resources

More information on the **mdsr** package can be found at <http://www.github.com/mdsr-book/mdsr>.

B

Introduction to R and RStudio

This chapter provides a (brief) introduction to **R** and **RStudio**. The **R** language is a free, open-source software environment for statistical computing and graphics (Ihaka and Gentleman, 1996; R Core Team, 2020). **RStudio** is an open-source integrated development environment (IDE) for **R** that adds many features and productivity tools for **R** (RStudio, 2020). This chapter includes a short history, installation information, a sample session, background on fundamental structures and actions, information about help and documentation, and other important topics.

The **R** Foundation for Statistical Computing holds and administers the copyright of the **R** software and documentation. **R** is available under the terms of the Free Software Foundation's GNU General Public License in source code form.

RStudio facilitates use of **R** by integrating **R** help and documentation, providing a workspace browser and data viewer, and supporting syntax highlighting, code completion, and smart indentation. Support for reproducible analysis is made available with the **knitr** package and **R** Markdown (see Appendix D). It facilitates the creation of dynamic *web applications* using Shiny (see Chapter 14.4). It also provides support for multiple projects as well as an interface to source code control systems such as GitHub. It has become the default interface for many **R** users, and is our recommended environment for analysis.

RStudio is available as a client (standalone) for Windows, Mac OS X, and Linux. There is also a server version. Commercial products and support are available in addition to the open-source offerings (see <http://www.rstudio.com/ide> for details).

The first versions of **R** were written by Ross Ihaka and Robert Gentleman at the *University of Auckland*, New Zealand, while current development is coordinated by the **R** Development Core Team, a group of international volunteers.

The **R** language is quite similar to the **S** language, a flexible and extensible statistical environment originally developed in the 1980s at AT&T Bell Labs (now Alcatel-Lucent).

B.1 Installation

New users are encouraged to download and install **R** from the Comprehensive **R** Archive Network (CRAN, <http://www.r-project.org>) and install **RStudio** from <http://www.rstudio.com/download>. The sample session in the appendix of the *Introduction to R* documentation, also available from CRAN, is recommended reading.

The home page for the **R** project, located at <http://r-project.org>, is the best starting place for information about the software. It includes links to CRAN, which features pre-compiled binaries as well as source code for **R**, add-on packages, documentation (including

manuals, frequently asked questions, and the **R** newsletter) as well as general background information. Mirrored CRAN sites with identical copies of these files exist all around the world. Updates to **R** and packages are regularly posted on CRAN.

B.1.1 RStudio

RStudio for Mac OS X, Windows, or Linux can be downloaded from <https://rstudio.com/products/rstudio>. **RStudio** requires **R** to be installed on the local machine. A server version (accessible from Web browsers) is also available for download. Documentation of the advanced features is available on the **RStudio** website.

B.2 Learning R

The **R** environment features extensive online documentation, though it can sometimes be challenging to comprehend. Each command has an associated help file that describes usage, lists arguments, provides details of actions, gives references, lists other related functions, and includes examples of its use. The help system is invoked using either the `?` or `help()` commands.

```
?function  
help(function)
```

where `function` is the name of the function of interest. (Alternatively, the `Help` tab in **RStudio** can be used to access the help system.)

Some commands (e.g., `if`) are reserved, so `?if` will not generate the desired documentation. Running `?"if"` will work (see also `?Reserved` and `?Control`). Other reserved words include `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, and `NA`.

The `rSiteSearch()` function will search for key words or phrases in many places (including the search engine at <http://search.r-project.org>). The RSeek.org site can also be helpful in finding more information and examples. Examples of many functions are available using the `example()` function.

```
example(mean)
```

Other useful resources are `help.start()`, which provides a set of online manuals, and `help.search()`, which can be used to look up entries by description. The `apropos()` command returns any functions in the current search list that match a given pattern (which facilitates searching for a function based on what it does, as opposed to its name).

Other resources for help available from CRAN include the **R** help mailing list. The StackOverflow site for **R** provides a series of questions and answers for common questions that are tagged as being related to **R**. New users are also encouraged to read the **R** FAQ (frequently asked questions) list. **RStudio** provides a curated guide to resources for learning **R** and its extensions.

B.3 Fundamental structures and objects

Here we provide a brief introduction to **R** data structures.

B.3.1 Objects and vectors

Almost everything in **R** is an object, which may be initially confusing to a new user. An object is simply something stored in **R**'s memory. Common objects include vectors, matrices, arrays, factors, data frames (akin to data sets in other systems), lists, and functions. The basic variable structure is a vector. Vectors (and other objects) are created using the `<-` or `=` assignment operators (which assign the evaluated expression on the right-hand side of the operator to the object name on the left-hand side).

```
x <- c(5, 7, 9, 13, -4, 8) # preferred
x = c(5, 7, 9, 13, -4, 8) # equivalent
```

The above code creates a vector of length 6 using the `c()` function to concatenate scalars. The `=` operator is used in other contexts for the specification of arguments to functions. Other assignment operators exist, as well as the `assign()` function (see `help("<-")` for more information). The `exists()` function conveys whether an object exists in the workspace, and the `rm()` command removes it. In **RStudio**, the “Environment” tab shows the names (and values) of all objects that exist in the current workspace.

Since vector operations are so fundamental in **R**, it is important to be able to access (or index) elements within these vectors. Many different ways of indexing vectors are available. Here, we introduce several of these using the `x` as created above. The command `x[2]` returns the second element of `x` (the scalar 7), and `x[c(2, 4)]` returns the vector (7, 13). The expressions `x[c(TRUE, TRUE, TRUE, TRUE, FALSE)]`, `x[1:5]` and `x[-6]` all return a vector consisting of the first 5 elements in `x` (the last specifies all elements except the 6th).

```
x[2]
[1] 7
x[c(2, 4)]
[1] 7 13
x[c(TRUE, TRUE, TRUE, TRUE, FALSE)]
[1] 5 7 9 13 -4
x[1:5]
[1] 5 7 9 13 -4
x[-6]
[1] 5 7 9 13 -4
```

Vectors are *recycled* if needed; for example, when comparing each of the elements of a vector to a scalar.

```
x > 8
[1] FALSE FALSE TRUE TRUE FALSE FALSE
```

The above expression demonstrates the use of comparison operators (see [?Comparison](#)). Only the third and fourth elements of `x` are greater than 8. The function returns a logical value of either `TRUE` or `FALSE` (see [?Logic](#)).

A count of elements meeting the condition can be generated using the `sum()` function. Other comparison operators include `==` (equal), `>=` (greater than or equal), `<=` (less than or equal) and `!=` (not equal). Care needs to be taken in the comparison using `==` if noninteger values are present (see [all.equal\(\)](#)).

```
sum(x > 8)
```

```
[1] 2
```

B.3.2 Operators

There are many operators defined in **R** to carry out a variety of tasks. Many of these were demonstrated in the sample session (assignment, arithmetic) and previous examples (comparison). Arithmetic operations include `+`, `-`, `*`, `/`, `^` (exponentiation), `%%` (modulus), and `%/%` (integer division). More information about operators can be found using the help system (e.g., `?"+"`). Background information on other operators and precedence rules can be found using `help(Syntax)`.

Boolean operations (OR, AND, NOT, and XOR) are supported using the `|`, `||`, `&`, `!` operators and the `xor()` function. The `|` is an “or” operator that operates on each element of a vector, while the `||` is another “or” operator that stops evaluation the first time that the result is true (see [?Logic](#)).

B.3.3 Lists

Lists in **R** are very general objects that can contain other objects of arbitrary types. List members can be named, or referenced using numeric indices (using the `[[` operator).

```
newlist <- list(first = "hello", second = 42, Bob = TRUE)
is.list(newlist)
```

```
[1] TRUE
```

```
newlist
```

```
$first
```

```
[1] "hello"
```

```
$second
```

```
[1] 42
```

```
$Bob
```

```
[1] TRUE
```

```
newlist[[2]]
```

```
[1] 42
```

```
newlist$Bob
```

```
[1] TRUE
```

The `unlist()` function flattens (makes a vector out of) the elements in a list (see also `relist()`). Note that unlisted objects are coerced to a common type (in this case `character`).

```
unlisted <- unlist(newlist)
unlisted
```

```
first   second      Bob
"hello"    "42"    "TRUE"
```

B.3.4 Matrices

Matrices are like two-dimensional vectors: rectangular objects where all entries have the same type. We can create a 2×3 matrix, display it, and test for its type.

```
A <- matrix(x, 2, 3)
```

```
A
```

```
[,1] [,2] [,3]
[1,]    5     9    -4
[2,]    7    13     8
is.matrix(A)    # is A a matrix?
```

```
[1] TRUE
```

```
is.vector(A)
```

```
[1] FALSE
```

```
is.matrix(x)
```

```
[1] FALSE
```

Note that comments are supported within **R** (any input given after a `#` character is ignored).

Indexing for matrices is done in a similar fashion as for vectors, albeit with a second dimension (denoted by a comma).

```
A[2, 3]
```

```
[1] 8
```

```
A[, 1]
```

```
[1] 5 7
```

```
A[1, ]
```

```
[1] 5 9 -4
```

B.3.5 Dataframes and tibbles

Data sets are often stored in a `data.frame`, which is a special type of `list` that is more general than a `matrix`. This rectangular object, similar to a data table in other systems, can be thought of as a two-dimensional array with columns of vectors of the same length, but of possibly different types (as opposed to a matrix, which consists of vectors of the *same* type; or a list, whose elements needn't be of the same length). The function `read_csv()` in the `readr` package returns a `data.frame` object.

A simple `data.frame` can be created using the `data.frame()` command. Variables can be accessed using the `$` operator, as shown below (see also `help(Extract)`). In addition, operations can be performed by column (e.g., calculation of sample statistics). We can check to see if an object is a `data.frame` with `is.data.frame()`.

```
y <- rep(11, length(x))
```

```
y
```

```
[1] 11 11 11 11 11 11
```

```
ds <- data.frame(x, y)
```

```
ds
```

	x	y
1	5	11
2	7	11
3	9	11
4	13	11
5	-4	11
6	8	11

```
ds$x[3]
```

```
[1] 9
```

```
is.data.frame(ds)
```

```
[1] TRUE
```

Tibbles are a form of simple data frames (a modern interpretation) that are described as “lazy and surly” (<https://tibble.tidyverse.org>). They support multiple data technologies (e.g., SQL databases), make more explicit their assumptions, and have an enhanced print method (so that output doesn’t scroll so much). Many packages in the `tidyverse` create tibbles by default.

```
tbl <- as_tibble(ds)
```

```
is.data.frame(tbl)
```

```
[1] TRUE
```

```
is_tibble(ds)
```

```
[1] FALSE
```

```
is_tibble(tbl)
```

```
[1] TRUE
```

The use of `data.frame()` differs from the use of `cbind()`, which yields a `matrix` object (unless it is given data frames as inputs).

```
newmat <- cbind(x, y)
```

```
newmat
```

	x	y
[1,]	5	11
[2,]	7	11
[3,]	9	11
[4,]	13	11

```
[5,] -4 11
[6,]  8 11
is.data.frame(newmat)

[1] FALSE
is.matrix(newmat)

[1] TRUE
```

Data frames are created from matrices using `as.data.frame()`, while matrices are constructed from data frames using `as.matrix()`.

Although we strongly discourage its use, data frames can be attached to the workspace using the `attach()` command. The Tidyverse **R** Style guide (<https://style.tidyverse.org>) provides similar advice. Name conflicts are a common problem with `attach()` (see `conflicts()`, which reports on objects that exist with the same name in two or more places on the search path).

The `search()` function lists attached packages and objects. To avoid cluttering and confusing the name-space, the command `detach()` should be used once a data frame or package is no longer needed.

A number of **R** functions include a `data` argument to specify a data frame as a local environment. For functions without a `data` option, the `with()` and `within()` commands can be used to simplify reference to an object within a data frame without attaching.

B.3.6 Attributes and classes

Many objects have a set of associated attributes (such as names of variables, dimensions, or classes) that can be displayed or sometimes changed. For example, we can find the dimension of the matrix defined earlier.

```
attributes(A)

$dim
[1] 2 3
```

Other types of objects within **R** include `lists` (ordered objects that are not necessarily rectangular), regression models (objects of class `lm`), and formulae (e.g., `y ~ x1 + x2`). **R** supports *object-oriented programming* (see `help(UseMethod)`). As a result, objects in **R** have an associated `class` attribute, which changes the default behavior for some operations on that object. Many functions (called *generics*) have special capabilities when applied to objects of a particular class. For example, when `summary()` is applied to an `lm` object, the `summary.lm()` function is called. Conversely, `summary.aov()` is called when an `aov` object is given as argument. These class-specific implementations of generic functions are called *methods*.

The `class()` function returns the classes to which an object belongs, while the `methods()` function displays all of the classes supported by a generic function.

```
head(methods(summary))

[1] "summary,ANY-method"           "summary,DBIObject-method"
[3] "summary,MySQLConnection-method" "summary,MySQLDriver-method"
[5] "summary,MySQLResult-method"     "summary.aov"
```

Objects in **R** can belong to multiple classes, although those classes need not be nested. As noted above, generic functions are *dispatched* according the class attribute of each object. Thus, in the example below we create the `tbl` object, which belongs to multiple classes. When the `print()` function is called on `tbl`, **R** looks for a method called `print.tbl_df()`. If no such method is found, **R** looks for a method called `print.tbl()`. If no such method is found, **R** looks for a method called `print.data.frame()`. This process continues until a suitable method is found. If there is none, then `print.default()` is called.

```
tbl <- as_tibble(ds)
class(tbl)

[1] "tbl_df"      "tbl"        "data.frame"

print(tbl)

# A tibble: 6 x 2
#>   x     y
#>   <dbl> <dbl>
#> 1     5    11
#> 2     7    11
#> 3     9    11
#> 4    13    11
#> 5    -4    11
#> 6     8    11

print.data.frame(tbl)

  x  y
1 5 11
2 7 11
3 9 11
4 13 11
5 -4 11
6 8 11

print.default(tbl)

$x
[1] 5 7 9 13 -4 8

$y
[1] 11 11 11 11 11 11

attr(,"class")
[1] "tbl_df"      "tbl"        "data.frame"
```

There are a number of functions that assist with learning about an object in **R**. The `attributes()` command displays the attributes associated with an object. The `typeof()` function provides information about the underlying data structure of objects (e.g., logical, integer, double, complex, character, and list). The `str()` function displays the structure of an object, and the `mode()` function displays its storage mode. For data frames, the `glimpse()` function provides a useful summary of each variable.

A few quick notes on specific types of objects are worth relating here:

- A vector is a one-dimensional array of items of the same data type. There are six basic data

types that a vector can contain: `logical`, `character`, `integer`, `double`, `complex`, and `raw`. Vectors have a `length()` but not a `dim()`. Vectors can have—but needn’t have—`names()`.

- A `factor` is a special type of vector for categorical data. A factor has `level()`s. We change the reference level of a factor with `relevel()`. Factors are stored internally as integers that correspond to the id’s of the factor levels.

Pro Tip 49. *Factors can be problematic and their use is discouraged since they can complicate some aspects of data wrangling. A number of R developers have encouraged the use of the `stringsAsFactors = FALSE` option.*

- A `matrix` is a two-dimensional array of items of the same data type. A matrix has a `length()` that is equal to `nrow()` times `ncol()`, or the product of `dim()`.
- A `data.frame` is a `list` of vectors of the same length. This is like a matrix, except that columns can be of different data types. Data frames always have `names()` and often have `row.names()`.

Pro Tip 50. *Do not confuse a factor with a character vector.*

Note that data sets typically have class `data.frame` but are of type `list`. This is because, as noted above, **R** stores data frames as special types of lists—a list of several vectors having the same length, but possibly having different types.

```
class(mtcars)
```

```
[1] "data.frame"
```

```
typeof(mtcars)
```

```
[1] "list"
```

Pro Tip 51. *If you ever get confused when working with data frames and matrices, remember that a `data.frame` is a list (that can accommodate multiple types of objects), whereas a `matrix` is more like a vector (in that it can only support one type of object).*

B.3.7 Options

The `options()` function in **R** can be used to change various default behaviors. For example, the `digits` argument controls the number of digits to display in output.

The current options are returned when `options()` is called, to allow them to be restored. The command `help(options)` lists all of the settable options.

B.3.8 Functions

Fundamental actions within **R** are carried out by calling *functions* (either built-in or user defined—see [Appendix C](#) for guidance on the latter). Multiple *arguments* may be given, separated by commas. The function carries out operations using the provided arguments and returns values (an object such as a vector or list) that are displayed (by default) or which can be saved by assignment to an object.

Pro Tip 52. *It’s a good idea to name arguments to functions. This practice minimizes errors assigning unnamed arguments to options and makes code more readable.*

As an example, the `quantile()` function takes a numeric vector and returns the minimum, 25th percentile, median, 75th percentile, and maximum of the values in that vector. However, if an optional vector of quantiles is given, those quantiles are calculated instead.

```
vals <- rnorm(1000) # generate 1000 standard normal random variables
quantile(vals)
```

```
0%      25%      50%      75%      100%
-3.44520 -0.73199  0.00575  0.74008  3.47853
```

```
quantile(vals, c(.025, .975))
```

```
2.5% 97.5%
-1.98 1.92
```

```
# Return values can be saved for later use.
```

```
res <- quantile(vals, c(.025, .975))
res[1]
```

```
2.5%
-1.98
```

Arguments (options) are available for most functions. The documentation specifies the default action if named arguments are not specified. If not named, the arguments are provided to the function in order specified in the function call.

For the `quantile()` function, there is a `type` argument that allows specification of one of nine algorithms for calculating quantiles.

```
res <- quantile(vals, probs = c(.025, .975), type = 3)
res
```

```
2.5% 97.5%
-2.00 1.92
```

Some functions allow a variable number of arguments. An example is the `paste()` function. The calling sequence is described in the documentation as follows.

```
paste(..., sep = " ", collapse = NULL)
```

To override the default behavior of a space being added between elements output by `paste()`, the user can specify a different value for `sep`.

B.4 Add-ons: Packages

B.4.1 Introduction to packages

Additional functionality in R is added through packages, which consist of functions, data sets, examples, vignettes, and help files that can be downloaded from CRAN. The function `install.packages()` can be used to download and install packages. Alternatively, RStudio provides an easy-to-use Packages tab to install and load packages.

Throughout the book, we assume that the `tidyverse` and `mdsr` packages are loaded. In

many cases, additional add-on packages (see [Appendix A](#)) need to be installed prior to running the examples in this book.

Packages that are not on CRAN can be installed using the `install_github()` function in the `remotes` package.

```
install.packages("mdsr")      # CRAN version
remotes::install_github("mdsr-book/mdsr")    # development version
```

The `library()` function will load an installed package.

For example, to install and load Frank Harrell's `Hmisc()` package, two commands are needed:

```
install.packages("Hmisc")
library(Hmisc)
```

If a package is not installed, running the `library()` command will yield an error. Here we try to load the `xaringanthemer` package (which has not been installed):

```
> library(xaringanthemer)
Error in library(xaringanthemer) : there is no package called 'xaringanthemer'
```

To rectify the problem, we install the package from CRAN.

```
> install.packages("xaringanthemer")
trying URL 'https://cloud.r-project.org/src/contrib/xaringanthemer_0.3.0.tar.gz'
Content type 'application/x-gzip' length 1362643 bytes (1.3 MB)
=====
downloaded 1.3 Mb

library(xaringanthemer)
```

The `require()` function will test whether a package is available—this will load the library if it is installed, and generate a warning message if it is not (as opposed to `library()`, which will return an error).

The names of all variables within a given data set (or more generally for sub-objects within an object) are provided by the `names()` command. The names of all objects defined within an **R** session can be generated using the `objects()` and `ls()` commands, which return a vector of character strings. **RStudio** includes an `Environment` tab that lists all the objects in the current environment.

The `print()` and `summary()` functions return the object or summaries of that object, respectively. Running `print(object)` at the command line is equivalent to just entering the name of the object, i.e., `object`.

B.4.2 Packages and name conflicts

Different package authors may choose the same name for functions that exist within base **R** (or within other packages). This will cause the other function or object to be *masked*. This can sometimes lead to confusion, when the expected version of a function is not the one that is called. The `find()` function can be used to determine where in the environment (workspace) a given object can be found.

```
find("mean")
```

```
[1] "package:base"
```

Sometimes it is desirable to remove a package from the workspace. For example, a package might define a function with the same name as an existing function. Packages can be detached using the syntax `detach(package:PKGNAME)`, where `PKGNAME` is the name of the package. Objects with the same name that appear in multiple places in the environment can be accessed using the `location::objectname` syntax. As an example, to access the `mean()` function from the `base` package, the user would specify `base::mean()` instead of `mean()`. It is sometimes preferable to reference a function or object in this way rather than loading the package.

As an example where this might be useful, there are functions in the `base` and `Hmisc` packages called `units()`. The `find` command would display both (in the order in which they would be accessed).

```
library(Hmisc)
find("units")
```

```
[1] "package:Hmisc" "package:base"
```

When the `Hmisc` package is loaded, the `units()` function from the `base` package is masked and would not be used by default. To specify that the version of the function from the `base` package should be used, prefix the function with the package name followed by two colons: `base::units()`. The `conflicts()` function reports on objects that exist with the same name in two or more places on the search path.

Running the command `library(help = "PKGNAME")` will display information about an installed package. Alternatively, the **Packages** tab in **RStudio** can be used to list, install, and update packages.

The `session_info()` function from the `sessioninfo` package provides improved reporting version information about **R** as well as details of loaded packages.

```
sessioninfo::session_info()
```

```
- Session info -----
  setting  value
  version  R version 4.0.2 (2020-06-22)
  os        macOS Catalina 10.15.7
  system   x86_64, darwin17.0
  ui        X11
  language (EN)
  collate  en_US.UTF-8
  ctype    en_US.UTF-8
  tz       America/New_York
  date     2021-01-09

- Packages -----
  package      * version date      lib source
  assertthat    0.2.1   2019-03-21 [1] CRAN (R 4.0.0)
  backports     1.2.1   2020-12-09 [1] CRAN (R 4.0.2)
  base64enc     0.1-3   2015-07-28 [1] CRAN (R 4.0.0)
  bookdown      0.21    2020-10-13 [1] CRAN (R 4.0.2)
  broom         0.7.3   2020-12-16 [1] CRAN (R 4.0.2)
  cellranger    1.1.0   2016-07-27 [1] CRAN (R 4.0.0)
  checkmate     2.0.0   2020-02-06 [1] CRAN (R 4.0.0)
```

cli	2.2.0	2020-11-20	[1]	CRAN	(R 4.0.2)
cluster	2.1.0	2019-06-19	[1]	CRAN	(R 4.0.2)
colorspace	2.0-0	2020-11-11	[1]	CRAN	(R 4.0.2)
crayon	1.3.4	2017-09-16	[1]	CRAN	(R 4.0.0)
data.table	1.13.6	2020-12-30	[1]	CRAN	(R 4.0.2)
DBI	* 1.1.0	2019-12-15	[1]	CRAN	(R 4.0.0)
dbplyr	2.0.0	2020-11-03	[1]	CRAN	(R 4.0.2)
digest	0.6.27	2020-10-24	[1]	CRAN	(R 4.0.2)
dplyr	* 1.0.2	2020-08-18	[1]	CRAN	(R 4.0.2)
ellipsis	0.3.1	2020-05-15	[1]	CRAN	(R 4.0.0)
evaluate	0.14	2019-05-28	[1]	CRAN	(R 4.0.0)
fansi	0.4.1	2020-01-08	[1]	CRAN	(R 4.0.0)
forcats	* 0.5.0	2020-03-01	[1]	CRAN	(R 4.0.0)
foreign	0.8-81	2020-12-22	[1]	CRAN	(R 4.0.2)
Formula	* 1.2-4	2020-10-16	[1]	CRAN	(R 4.0.2)
fs	1.5.0	2020-07-31	[1]	CRAN	(R 4.0.2)
generics	0.1.0	2020-10-31	[1]	CRAN	(R 4.0.2)
ggplot2	* 3.3.3	2020-12-30	[1]	CRAN	(R 4.0.2)
glue	1.4.2	2020-08-27	[1]	CRAN	(R 4.0.2)
gridExtra	2.3	2017-09-09	[1]	CRAN	(R 4.0.0)
gttable	0.3.0	2019-03-25	[1]	CRAN	(R 4.0.0)
haven	2.3.1	2020-06-01	[1]	CRAN	(R 4.0.0)
Hmisc	4.4-2	2020-11-29	[1]	CRAN	(R 4.0.2)
hms	0.5.3	2020-01-08	[1]	CRAN	(R 4.0.0)
htmlTable	2.1.0	2020-09-16	[1]	CRAN	(R 4.0.2)
htmltools	0.5.0	2020-06-16	[1]	CRAN	(R 4.0.0)
htmlwidgets	1.5.3	2020-12-10	[1]	CRAN	(R 4.0.2)
httr	1.4.2	2020-07-20	[1]	CRAN	(R 4.0.2)
jpeg	0.1-8.1	2019-10-24	[1]	CRAN	(R 4.0.0)
jsonlite	1.7.2	2020-12-09	[1]	CRAN	(R 4.0.2)
knitr	1.30	2020-09-22	[1]	CRAN	(R 4.0.2)
lattice	* 0.20-41	2020-04-02	[1]	CRAN	(R 4.0.2)
latticeExtra	0.6-29	2019-12-19	[1]	CRAN	(R 4.0.0)
lifecycle	0.2.0	2020-03-06	[1]	CRAN	(R 4.0.0)
lubridate	1.7.9.2	2020-11-13	[1]	CRAN	(R 4.0.2)
magrittr	2.0.1	2020-11-17	[1]	CRAN	(R 4.0.2)
Matrix	1.3-2	2021-01-06	[1]	CRAN	(R 4.0.2)
mdsr	* 0.2.4	2021-01-06	[1]	CRAN	(R 4.0.2)
modelr	0.1.8	2020-05-19	[1]	CRAN	(R 4.0.0)
mosaicData	* 0.20.1	2020-09-13	[1]	CRAN	(R 4.0.2)
munsell	0.5.0	2018-06-12	[1]	CRAN	(R 4.0.0)
nnet	7.3-14	2020-04-26	[1]	CRAN	(R 4.0.2)
pillar	1.4.7	2020-11-20	[1]	CRAN	(R 4.0.2)
pkgconfig	2.0.3	2019-09-22	[1]	CRAN	(R 4.0.0)
png	0.1-7	2013-12-03	[1]	CRAN	(R 4.0.0)
purrr	* 0.3.4	2020-04-17	[1]	CRAN	(R 4.0.0)
R6	2.5.0	2020-10-28	[1]	CRAN	(R 4.0.2)
RColorBrewer	1.1-2	2014-12-07	[1]	CRAN	(R 4.0.0)
Rcpp	1.0.5	2020-07-06	[1]	CRAN	(R 4.0.2)
readr	* 1.4.0	2020-10-05	[1]	CRAN	(R 4.0.2)
readxl	1.3.1	2019-03-13	[1]	CRAN	(R 4.0.0)

repr	1.1.0	2020-01-28	[1]	CRAN	(R 4.0.0)
reprex	0.3.0	2019-05-16	[1]	CRAN	(R 4.0.0)
rlang	0.4.10	2020-12-30	[1]	CRAN	(R 4.0.2)
rmarkdown	2.6	2020-12-14	[1]	CRAN	(R 4.0.2)
RMySQL	0.10.21	2020-12-15	[1]	CRAN	(R 4.0.2)
rpart	4.1-15	2019-04-12	[1]	CRAN	(R 4.0.2)
rstudioapi	0.13	2020-11-12	[1]	CRAN	(R 4.0.2)
rvest	0.3.6	2020-07-25	[1]	CRAN	(R 4.0.2)
scales	1.1.1	2020-05-11	[1]	CRAN	(R 4.0.0)
sessioninfo	1.1.1	2018-11-05	[1]	CRAN	(R 4.0.0)
showtext	0.9-1	2020-11-14	[1]	CRAN	(R 4.0.2)
showtextdb	3.0	2020-06-04	[1]	CRAN	(R 4.0.2)
skimr	2.1.2	2020-07-06	[1]	CRAN	(R 4.0.0)
stringi	1.5.3	2020-09-09	[1]	CRAN	(R 4.0.2)
stringr	* 1.4.0	2019-02-10	[1]	CRAN	(R 4.0.0)
survival	* 3.2-7	2020-09-28	[1]	CRAN	(R 4.0.2)
sysfonts	0.8.2	2020-11-16	[1]	CRAN	(R 4.0.2)
tibble	* 3.0.4	2020-10-12	[1]	CRAN	(R 4.0.2)
tidyverse	* 1.1.2	2020-08-27	[1]	CRAN	(R 4.0.2)
tidyselect	1.1.0	2020-05-11	[1]	CRAN	(R 4.0.0)
tidyverse	* 1.3.0	2019-11-21	[1]	CRAN	(R 4.0.0)
utf8	1.1.4	2018-05-24	[1]	CRAN	(R 4.0.0)
vctrs	0.3.6	2020-12-17	[1]	CRAN	(R 4.0.2)
withr	2.3.0	2020-09-22	[1]	CRAN	(R 4.0.2)
xaringanthemer	* 0.3.0	2020-05-04	[1]	CRAN	(R 4.0.2)
xfun	0.20	2021-01-06	[1]	CRAN	(R 4.0.2)
xml2	1.3.2	2020-04-23	[1]	CRAN	(R 4.0.0)
yaml	2.2.1	2020-02-01	[1]	CRAN	(R 4.0.0)

```
[1] /Library/Frameworks/R.framework/Versions/4.0/Resources/library
```

The `update.packages()` function should be run periodically to ensure that packages are up-to-date

As of December 2020, there were more than 16,800 packages available from CRAN. This represents a tremendous investment of time and code by many developers (Fox, 2009). While each of these has met a minimal standard for inclusion, it is important to keep in mind that packages in **R** are created by individuals or small groups, and not endorsed by the **R** core group. As a result, they do not necessarily undergo the same level of testing and quality assurance that the core **R** system does.

B.4.3 CRAN task views

The “Task Views” on CRAN are a very useful resource for finding packages. These are curated listings of relevant packages within a particular application area (such as multivariate statistics, psychometrics, or survival analysis). [Table B.1](#) displays the task views available as of January 2021.

Table B.1: A complete list of CRAN task views.

Task View	Subject
Bayesian	Bayesian Inference
ChemPhys	Chemometrics and Computational Physics
ClinicalTrials	Clinical Trial Design, Monitoring, and Analysis
Cluster	Cluster Analysis and Finite Mixture Models
Databases	Databases with R
DifferentialEquations	Differential Equations
Distributions	Probability Distributions
Econometrics	Econometrics
Environmetrics	Analysis of Ecological and Environmental Data
ExperimentalDesign	Design of Experiments (DoE) and Analysis of Experimental Data
ExtremeValue	Extreme Value Analysis
Finance	Empirical Finance
FunctionalData	Functional Data Analysis
Genetics	Statistical Genetics
gR	gRaphical Models in R
Graphics	Graphic Displays and Dynamic Graphics and Graphic Devices and Visualization
HighPerformanceComputing	High-Performance and Parallel Computing with R
Hydrology	Hydrological Data and Modeling
MachineLearning	Machine Learning and Statistical Learning
MedicalImaging	Medical Image Analysis
MetaAnalysis	Meta-Analysis
MissingData	Missing Data
ModelDeployment	Model Deployment with R
Multivariate	Multivariate Statistics
NaturalLanguageProcessing	Natural Language Processing
NumericalMathematics	Numerical Mathematics
OfficialStatistics	Official Statistics and Survey Methodology
Optimization	Optimization and Mathematical Programming
Pharmacokinetics	Analysis of Pharmacokinetic Data
Phylogenetics	Phylogenetics, Especially Comparative Methods
Psychometrics	Psychometric Models and Methods
ReproducibleResearch	Reproducible Research
Robust	Robust Statistical Methods
SocialSciences	Statistics for the Social Sciences
Spatial	Analysis of Spatial Data
SpatioTemporal	Handling and Analyzing Spatio-Temporal Data
Survival	Survival Analysis
TeachingStatistics	Teaching Statistics
TimeSeries	Time Series Analysis
Tracking	Processing and Analysis of Tracking Data
WebTechnologies	Web Technologies and Services

B.5 Further resources

Advanced R is an excellent source for learning more about how **R** works (Wickham, 2019a). Extensive resources and documentation about **R** can be found at the Comprehensive R Archive Network (CRAN).

The **forcats** package, included in the **tidyverse**, is designed to facilitate data wrangling with factors.

More information regarding tibbles can be found at <https://tibble.tidyverse.org>.

JupyterLab and JupyterHub are alternative environments that support analysis via sophisticated notebooks for multiple languages including Julia, Python, and **R**.

B.6 Exercises

Problem 1 (Easy): The following code chunk throws an error.

```
mtcars %>%  
  select(mpg, cyl)
```

```
Error in select(., mpg, cyl): could not find function "select"
```

What is the problem?

Problem 2 (Easy): Which of these kinds of names should be wrapped with quotation marks when used in R?

- function name
- file name
- the name of an argument in a named argument
- object name

Problem 3 (Easy): A user has typed the following commands into the RStudio console.

```
obj1 <- 2:10  
obj2 <- c(2, 5)  
obj3 <- c(TRUE, FALSE)  
obj4 <- 42
```

What values are returned by the following commands?

```
obj1 * 10  
obj1[2:4]  
obj1[-3]  
obj1 + obj2  
obj1 * obj3  
obj1 + obj4  
obj2 + obj3  
sum(obj2)  
sum(obj3)
```

Problem 4 (Easy): A user has typed the following commands into the RStudio console:

```
mylist <- list(x1 = "sally", x2 = 42, x3 = FALSE, x4 = 1:5)
```

What values do each of the following commands return?

```
is.list(mylist)  
names(mylist)  
length(mylist)  
mylist[[2]]  
mylist[["x1"]]  
mylist$x2  
length(mylist[["x4"]])  
class(mylist)  
typeof(mylist)  
class(mylist[[4]])  
typeof(mylist[[3]])
```

Problem 5 (Easy): What's wrong with this statement?

```
help(NHANES, package <- "NHANES")
```

Problem 6 (Easy): Consult the documentation for `cps85` in the `mosaicData` package to determine the meaning of CPS.

Problem 7 (Easy): The following code chunk throws an error. Why?

```
library(tidyverse)
mtcars %>%
  filter(cylinders == 4)
```

```
Error in filter(., cylinders == 4): object 'cylinders' not found
```

What is the problem?

Problem 8 (Easy): The `date` function returns an indication of the current time and date. What arguments does `date` take? What kind of object is the result from `date`? What kind of object is the result from `sys.time`?

Problem 9 (Easy): A user has typed the following commands into the RStudio console.

```
a <- c(10, 15)
b <- c(TRUE, FALSE)
c <- c("happy", "sad")
```

What do each of the following commands return? Describe the class of the object as well as its value.

```
data.frame(a, b, c)
cbind(a, b)
rbind(a, b)
cbind(a, b, c)
list(a, b, c)[[2]]
```

Problem 10 (Easy): For each of the following assignment statements, describe the error (or note why it does not generate an error).

```
result1 <- sqrt 10
result2 <-- "Hello to you!"
3result <- "Hello to you"
result4 <- "Hello to you
result5 <- date()
```

Problem 11 (Easy): The following code chunk throws an error.

```
library(tidyverse)
mtcars %>%
  filter(cyl = 4)
```

```
Error in filter(., cyl = 4): unused argument (cyl = 4)
```

The error suggests that you need to use `==` inside of `filter()`. Why?

Problem 12 (Medium): The following code undertakes some data analysis using the HELP (Health Evaluation and Linkage to Primary Care) trial.

```
library(mosaic)
ds <- 
  read.csv("http://nhorton.people.amherst.edu/r2/datasets/helpmiss.csv")
summarise(group_by(
  select(filter(mutate(ds,
    sex = ifelse(female == 1, "F", "M")
  ), !is.na(pcs)), age, pcs, sex),
  sex
), meanage = mean(age), meanpcs = mean(pcs), n = n())
```

Describe in words what computations are being done. Using the pipe notation, translate this code into a more readable version.

Problem 13 (Medium): The following concepts should have some meaning to you: package, function, command, argument, assignment, object, object name, data frame, named argument, quoted character string.

Construct an example of R commands that make use of at least four of these. Label which part of your example R command corresponds to each.

B.7 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/appR.html#datavizI-online-exercises>



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

C

Algorithmic thinking

C.1 Introduction

Algorithmic thinking can be defined as a set of abilities that are related to constructing and understanding algorithms (Futschek, 2006):

1. the ability to analyze a given problem
2. the ability to precisely specify a problem
3. the ability to find the basic actions that are adequate to solve a problem
4. the ability to construct a correct algorithm to a given problem using basic actions
5. the ability to think about all possible special and normal cases of a problem
6. the ability to improve the efficiency of an algorithm

These important capacities are a necessary but not sufficient component of “computational thinking” and data science.

It is critical that data scientists have the skills to break problems down and code solutions in a flexible and powerful computing environment using *functions*. We focus on the use of **R** for this task (although other environments such as Python have many adherents and virtues). In this appendix, we presume a basic background in **R** to the level of [Appendix B](#).

C.2 Simple example

We begin with an example that creates a simple function to complete a statistical task (calculate a confidence interval for an estimate). In **R**, a new *function* is defined by the syntax shown below, using the keyword `function`. This creates a new object in **R** called `new_function()` in the workspace that takes two arguments (`argument1` and `argument2`). The body is made up of a series of commands (or expressions), typically separated by line breaks and enclosed in curly braces.

```
library(tidyverse)
library(mdsr)
new_function <- function(argument1, argument2) {
  R expression
  another R expression
}
```

Here, we create a function to calculate the estimated confidence interval (CI) for a mean, using the formula $\bar{X} \pm t^*s/\sqrt{n}$, where t^* is the appropriate t-value for that particular con-

fidence level. As an example, for a 95% interval with 50 degrees of freedom (equivalent to $n = 51$ observations) the appropriate value of t^* can be calculated using the `cdist()` function from the **mosaic** package. This computes the quantiles of the t-distribution between which 95% of the distribution lies. A graphical illustration is shown in [Figure C.1](#).

```
library(mosaic)
mosaic::cdist(dist = "t", p = 0.95, df = 50)

[1] -2.01  2.01

mosaic::xqt(p = c(0.025, 0.975), df = 50)
```

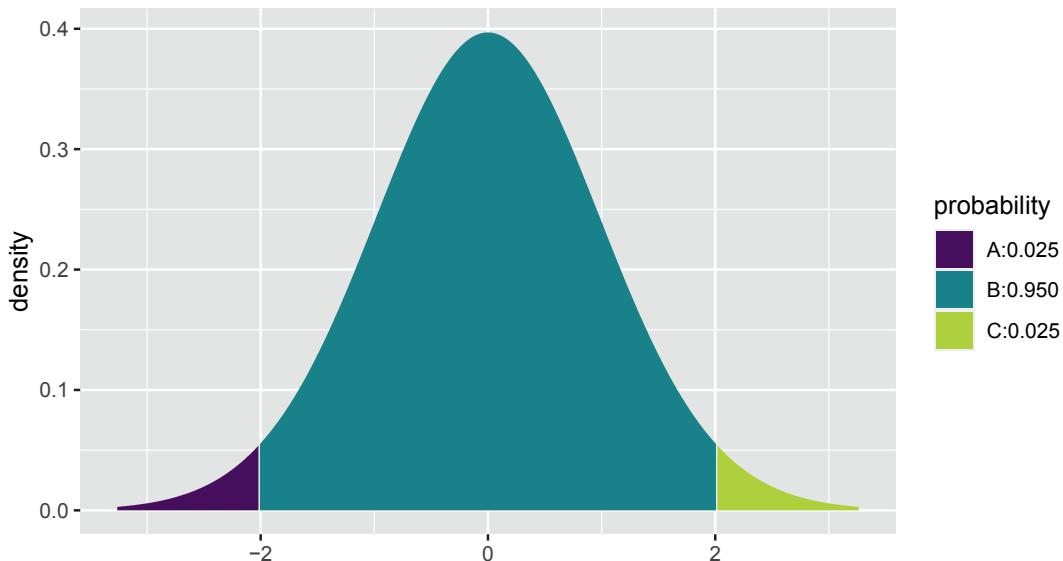


Figure C.1: Illustration of the location of the critical value for a 95% confidence interval for a mean. The critical value of 2.01 corresponds to the location in the t-distribution with 50 degrees of freedom, for which 2.5% of the distribution lies above it.

```
[1] -2.01  2.01
```

We see that the value is slightly larger than 2. Note that since by construction our confidence interval will be centered around the mean, we want the critical value that corresponds to having 95% of the distribution in the middle.

We will write a function to compute a t-based confidence interval for a mean from scratch. We'll call this function `ci_calc()`, and it will take a numeric vector `x` as its first argument, and an optional second argument `alpha`, which will have a default value of `0.95`.

```
# calculate a t confidence interval for a mean
ci_calc <- function(x, alpha = 0.95) {
  samp_size <- length(x)
  t_star <- qt(1 - ((1 - alpha)/2), df = samp_size - 1)
  my_mean <- mean(x)
  my_sd <- sd(x)
  se <- my_sd/sqrt(samp_size)
  me <- t_star * se
  return(
```

```

list(
  ci_vals = c(my_mean - me, my_mean + me),
  alpha = alpha
)
)
}
}

```

Here the appropriate quantile of the t-distribution is calculated using the `qt()` function, and the appropriate confidence interval is calculated and returned as a list. In this example, we explicitly `return()` a list of values. If no return statement is provided, the result of the last expression evaluation is returned by default. The tidyverse style guide (see [Section 6.3](#)) encourages that default, but we prefer to make the return explicit.

The function has been stored in the object `ci_calc()`. Once created, it can be used like any other built-in function. For example, the expression below will print the CI and confidence level for the object `x1` (a set of 100 random normal variables with mean 0 and standard deviation 1).

```

x1 <- rnorm(100, mean = 0, sd = 1)
ci_calc(x1)

```

```

$ci_vals
[1] -0.0867  0.2933

$alpha
[1] 0.95

```

The order of arguments in **R** matters if arguments to a function are not named. When a function is called, the arguments are assumed to correspond to the order of the arguments as the function is defined. To see that order, check the documentation, use the `args()` function, or look at the code of the function itself.

```

?ci_calc # won't work because we haven't written any documentation
args(ci_calc)
ci_calc

```

Pro Tip 53. Consider creating an **R** package for commonly used functions that you develop so that they can be more easily documented, tested, and reused.

Since we provided only one unnamed argument (`x1`), **R** passed the value `x1` to the argument `x` of `ci_calc()`. Since we did not specify a value for the `alpha` argument, the default value of `0.95` was used.

User-defined functions nest just as pre-existing functions do. The expression below will return the CI and report that the confidence limit is `0.9` for 100 normal random variates.

```
ci_calc(rnorm(100), 0.9)
```

To change the confidence level, we need only change the `alpha` option by specifying it as a *named argument*.

```
ci_calc(x1, alpha = 0.90)
```

```

$ci_vals
[1] -0.0557  0.2623

```

```
$alpha
[1] 0.9
```

The output is equivalent to running the command `ci_calc(x1, 0.90)` with two unnamed arguments, where the arguments are matched in order. Perhaps less intuitive but equivalent would be the following call.

```
ci_calc(alpha = 0.90, x = x1)
```

```
$ci_vals
[1] -0.0557  0.2623
```

```
$alpha
[1] 0.9
```

The key take-home message is that the order of arguments is not important *if all of the arguments are named*.

Using the pipe operator introduced in [Chapter 4](#) can avoid nesting.

```
rnorm(100, mean = 0, sd = 1) %>%
  ci_calc(alpha = 0.9)
```

```
$ci_vals
[1] -0.0175  0.2741
```

```
$alpha
[1] 0.9
```

Pro Tip 54. The `testthat` package can help to improve your functions by writing testing routines to check that the function does what you expect it to.

C.3 Extended example: Law of large numbers

The *law of large numbers* concerns the convergence of the arithmetic average of a sample to the expected value of a random variable, as the sample size increases. What this means is that with a sufficiently large unbiased sample, we can be pretty confident about the true mean. This is an important result in statistics, described in [Section 9.2.1](#). The convergence (or lack thereof, for certain distributions) can easily be visualized.

We define a function to calculate the running average for a given vector, allowing for variates from many distributions to be generated.

```
runave <- function(n, gendist, ...) {
  x <- gendist(n, ...)
  avex <- numeric(n)
  for (k in 1:n) {
    avex[k] <- mean(x[1:k])
  }
  return(tibble(x, avex, n = 1:length(avex)))
}
```

The `runave()` function takes at a minimum two arguments: a sample size `n` and function (see B.3.8) denoted by `gendist` that is used to generate samples from a distribution.

Pro Tip 55. Note that there are more efficient ways to write this function using vector operations (see for example the `cummean()` function).

Other options for the function can be specified, using the ... (dots) syntax. This syntax allows additional options to be provided to functions that might be called downstream. For example, the dots are used to specify the degrees of freedom for the samples generated for the t-distribution in the next code block.

The Cauchy distribution is symmetric and has heavy tails. It is useful to know that because the expectation of a Cauchy random variable is undefined (Romano and Siegel, 1986), the sample average does not converge to the center (see related discussion in [Section 9.2.1](#)). The variance of a Cauchy random variable is also infinite (does not exist). Such a distribution arises when ratios are calculated. Conversely, a t-distribution with more than 1 degree of freedom (a distribution with less of a heavy tail) does converge to the center. For comparison, the two distributions are displayed in [Figure C.2](#).

```
mosaic::plotDist(
  "t",
  params = list(df = 4),
  xlim = c(-5, 5),
  lty = 2,
  lwd = 3
)
mosaic::plotDist("cauchy", xlim = c(-10, 10), lwd = 3, add = TRUE)
```

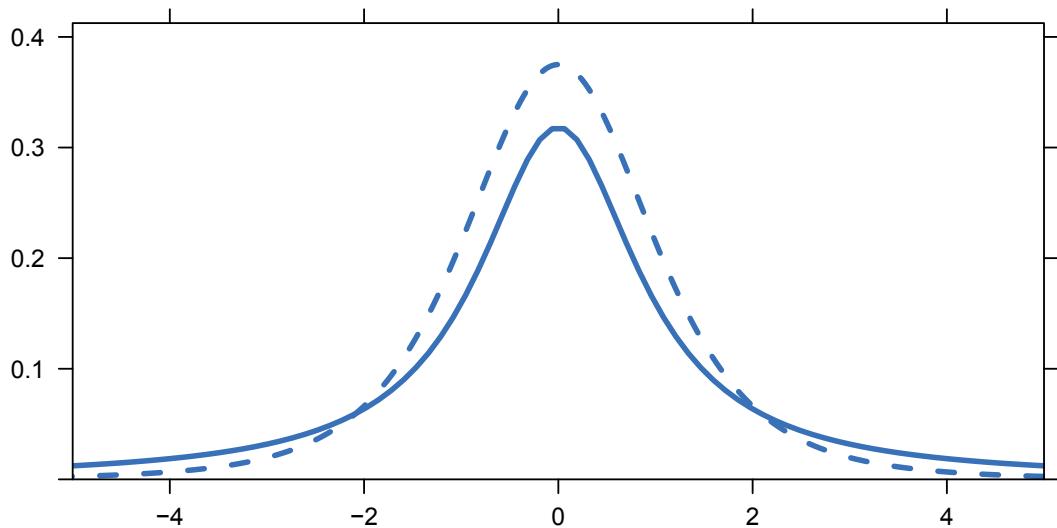


Figure C.2: Cauchy distribution (solid line) and t-distribution with 4 degrees of freedom (dashed line).

To make sure we can replicate our results for this simulation, we first set a fixed seed (see [Section 13.6.3](#)). Next, we generate some data, using our new `runave()` function.

```
nvals <- 1000
set.seed(1984)
sims <- bind_rows(
  runave(nvals, rt, 4),
  runave(nvals, rcauchy)
) %>%
  mutate(dist = rep(c("t4", "cauchy"), each = nvals))
```

In this example, the value 4 is provided to the `rt()` function using the ... mechanism. This is used to specify the `df` argument to `rt()`. The results are plotted in [Figure C.3](#). While the running average of the t-distribution converges to the true mean of zero, the running average of the Cauchy distribution does not.

```
ggplot(
  data = sims,
  aes(x = n, y = avex, color = dist)) +
  geom_hline(yintercept = 0, color = "black", linetype = 2) +
  geom_line() +
  geom_point() +
  labs(color = "Distribution", y = "running mean", x = "sample size") +
  xlim(c(0, 600))
)
```

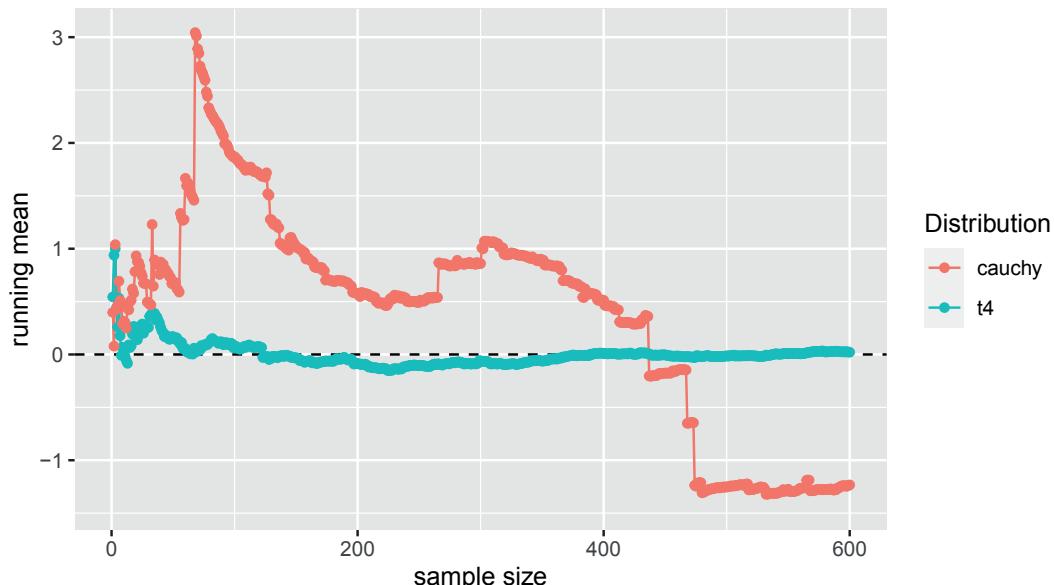


Figure C.3: Running average for t-distribution with 4 degrees of freedom and a Cauchy random variable (equivalent to a t-distribution with 1 degree of freedom). Note that while the former converges, the latter does not.

C.4 Non-standard evaluation

When evaluating expressions, **R** searches for objects in an *environment*. The most general environment is the *global environment*, the contents of which are displayed in the environment tab in **RStudio** or through the `ls()` command. When you try to access an object that cannot be found in the global environment, you get an error.

We will use a subset of the **NHANES** data frame from the **NHANES** package to illustrate a few of these subtleties. This data frame has a variety of data types.

```
library(NHANES)
nhanes_small <- NHANES %>%
  select(ID, SurveyYr, Gender, Age, AgeMonths, Race1, Poverty)
glimpse(nhanes_small)
```

```
Rows: 10,000
Columns: 7
$ ID      <int> 51624, 51624, 51624, 51625, 51630, 51638, 51646, 5164...
$ SurveyYr <fct> 2009_10, 2009_10, 2009_10, 2009_10, 2009_10, 2009_10, ...
$ Gender    <fct> male, male, male, male, female, male, male, female, f...
$ Age       <int> 34, 34, 34, 4, 49, 9, 8, 45, 45, 45, 66, 58, 54, 10, ...
$ AgeMonths <int> 409, 409, 409, 49, 596, 115, 101, 541, 541, 541, 795, ...
$ Race1     <fct> White, White, White, Other, White, White, White, Whit...
$ Poverty   <dbl> 1.36, 1.36, 1.36, 1.07, 1.91, 1.84, 2.33, 5.00, 5.00, ...
```

Consider the differences between trying to access the `ID` variable each of the three ways shown below. In the first case, we are simply creating a character vector of length one that contains the single string `ID`. The second command causes **R** to search the global environment for an object called `ID`—which does not exist. In the third command, we correctly access the `ID` variable within the `nhanes_small` data frame, which *is* accessible in the global environment. These are different examples of how **R** uses *scoping* to identify objects.

```
"ID"    # string variable
[1] "ID"
ID      # generates an error

Error in eval(expr, envir, enclos): object 'ID' not found
nhanes_small %>%
  pull(ID) %>%    # access within a data frame
  summary()
```

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
51624	51624	51624	51625	51630	51638	51646

How might this be relevant? Notice that several of the variables in `nhanes_small` are factors. We might want to convert each of them to type `character`. Typically, we would do this using the `mutate()` command that we introduced in [Chapter 4](#).

```
nhanes_small %>%
  mutate(SurveyYr = as.character(SurveyYr)) %>%
```

```
select(ID, SurveyYr) %>%
glimpse()
```

Rows: 10,000
 Columns: 2
\$ ID <int> 51624, 51624, 51624, 51625, 51630, 51638, 51646, 51647...
\$ SurveyYr <chr> "2009_10", "2009_10", "2009_10", "2009_10", "2009_10", ...

Note however, that in this construction we have to know the name of the variable we wish to convert (i.e., SurveyYr) and list it explicitly. This is unfortunate if the goal is to automate our data wrangling (see [Chapter 7](#)).

If we tried instead to set the name of the column (i.e., SurveyYr) to a variable (i.e., varname) and use that variable to change the names, it would not work as intended. In this case, rather than changing the data type of SurveyYr, we have created a new variable called varname that is a character vector of the values SurveyYr.

```
varname <- "SurveyYr"
nhanes_small %>%
  mutate(varname = as.character(varname)) %>%
  select(ID, SurveyYr, varname) %>%
  glimpse()
```

Rows: 10,000
 Columns: 3
\$ ID <int> 51624, 51624, 51624, 51625, 51630, 51638, 51646, 51647...
\$ SurveyYr <fct> 2009_10, 2009_10, 2009_10, 2009_10, 2009_10, 2009_10, ...
\$ varname <chr> "SurveyYr", "SurveyYr", "SurveyYr", "SurveyYr", "SurveyYr", ...

This behavior is a consequence of a feature of the R language called *non-standard evaluation* (NSE). The **rlang** package provides a principled way to work with expressions in the **tidyverse** and is used extensively in the **dplyr** package. The **dplyr** functions use a form of non-standard evaluation called *tidy evaluation*. Tidy evaluation allows R to locate SurveyYr, even though there is no object called SurveyYr in the global environment. Here, **mutate()** knows to look for SurveyYr within the nhanes_small data frame.

In this case, we can solve our problem using the **across()** and **where()** adverbs and **mutate()**. Namely, we can use **is.factor()** in conjunction with **across()** to find all of the variables that are factors and convert them to character using **as.character()**.

```
nhanes_small %>%
  mutate(across(where(is.factor), as.character))
```

```
# A tibble: 10,000 x 7
  ID SurveyYr Gender Age AgeMonths Race1 Poverty
  <int> <chr>   <chr> <int>    <int> <chr>   <dbl>
1 51624 2009_10 male     34      409 White    1.36
2 51624 2009_10 male     34      409 White    1.36
3 51624 2009_10 male     34      409 White    1.36
4 51625 2009_10 male      4      49 Other    1.07
5 51630 2009_10 female    49      596 White   1.91
6 51638 2009_10 male      9      115 White   1.84
7 51646 2009_10 male      8      101 White   2.33
8 51647 2009_10 female    45      541 White    5
```

```

9 51647 2009_10 female    45      541 White     5
10 51647 2009_10 female   45      541 White     5
# ... with 9,990 more rows

```

When you come to a problem that involves programming with **tidyverse** functions that you aren't sure how to solve, consider these approaches:

1. Use `across()` and/or `where()`: This approach is outlined above and in [Section 7.2](#). If this suffices, it will usually be the cleanest solution.
2. Pass the dots: Sometimes, you can avoid having to hard-code variable names by passing the dots through your function to another **tidyverse** function. For example, you might allow your function to take an arbitrary list of arguments that are passed directly to `select()` or `filter()`.
3. Use tidy evaluation: A full discussion of this is beyond the scope of this book, but we provide pointers to more material in [Section C.6](#).
4. Use base R syntax: If you are comfortable working in this dialect, the code can be relatively simple, but it has two obstacles for **tidyverse** learners: 1) the use of selectors like `[]` may be jarring and break pipelines; and 2) the quotation marks around variable names can get cumbersome. We show a simple example below.

A base R implementation

In base R, you can do this:

```

var_to_char_base <- function(data, varname) {
  data[[varname]] <- as.character(data[[varname]])
  data
}

var_to_char_base(nhanes_small, "SurveyYr")

```

```

# A tibble: 10,000 x 7
  ID SurveyYr Gender Age AgeMonths Race1 Poverty
  <int> <chr>    <fct> <int>    <int> <fct>   <dbl>
1 51624 2009_10 male    34      409 White    1.36
2 51624 2009_10 male    34      409 White    1.36
3 51624 2009_10 male    34      409 White    1.36
4 51625 2009_10 male     4      49 Other    1.07
5 51630 2009_10 female   49      596 White   1.91
6 51638 2009_10 male     9      115 White   1.84
7 51646 2009_10 male     8      101 White   2.33
8 51647 2009_10 female   45      541 White    5
9 51647 2009_10 female   45      541 White    5
10 51647 2009_10 female   45      541 White    5
# ... with 9,990 more rows

```

Note that the variable names have quotes when they are put into the function.

C.5 Debugging and defensive coding

It can be challenging to identify issues and problems with code that is not working.

Both **R** and **RStudio** include support for debugging functions and code. Calling the `browser()` function in the body of a function will cause execution to stop and set up an **R** interpreter. Once at the browser prompt, the analyst can enter either commands (such as `c` to continue execution, `f` to finish execution of the current function, `n` to evaluate the next statement (without stepping into function calls), `s` to evaluate the next statement (stepping into function calls), `Q` to exit the browser, or `help` to print this list). Other commands entered at the browser are interpreted as **R** expressions to be evaluated (the function `ls()` lists available objects). Calls to the browser can be set using the `debug()` or `debugonce()` functions (and turned off using the `undebug()` function). **RStudio** includes a debugging mode that is displayed when `debug()` is called.

Adopting *defensive coding* techniques is always recommended: They tend to identify problems early and minimize errors. The `try()` function can be used to evaluate an expression while allowing for error recovery. The `stop()` function can be used to stop evaluation of the current expression and execute an error action (typically displaying an error message). More flexible testing is available in the **assertthat** package.

Let's revisit the `ci_calc()` function we defined to calculate a confidence interval. How might we make this more robust? We can begin by confirming that the calling arguments are sensible.

```
library(assertthat)
# calculate a t confidence interval for a mean
ci_calc <- function(x, alpha = 0.95) {
  if (length(x) < 2) {
    stop("Need to provide a vector of length at least 2.\n")
  }
  if (alpha < 0 | alpha > 1) {
    stop("alpha must be between 0 and 1.\n")
  }
  assert_that(is.numeric(x))
  samp_size <- length(x)
  t_star <- qt(1 - ((1 - alpha)/2), df = samp_size - 1)
  my_mean <- mean(x)
  my_sd <- sd(x)
  se <- my_sd / sqrt(samp_size)
  me <- t_star * se
  return(list(ci_vals = c(my_mean - me, my_mean + me),
             alpha = alpha))
}
ci_calc(1)  # will generate error
```

```
Error in ci_calc(1): Need to provide a vector of length at least 2.
```

```
ci_calc(1:3, alpha = -1)  # will generate error
```

```
Error in ci_calc(1:3, alpha = -1): alpha must be between 0 and 1.
```

```
ci_calc(c("hello", "goodbye"))  # will generate error
```

```
Error: x is not a numeric or integer vector
```

C.6 Further resources

More examples of functions can be found in [Chapter 13](#). The American Statistical Association's *Guidelines for Undergraduate Programs in Statistics* American Statistical Association Undergraduate Guidelines Workgroup (2014) stress the importance of algorithmic thinking (see also Nolan and Temple Lang (2010)). Rizzo (2019) and Wickham (2019a) provide useful reviews of statistical computing. A variety of online resources are available to describe how to create **R** packages and to deploy them on GitHub (see for example http://kbroman.org/pkg_primer). Wickham (2015) is a comprehensive and accessible guide to writing **R** packages. The **testthat** package is helpful in structuring more extensive unit tests for functions. The **dplyr** package documentation includes a vignette detailing its use of the **lazyeval** package for performing non-standard evaluation. Henry (2020) explains the most recent developments surrounding tidy evaluation.

Wickham (2019a) includes a fuller discussion of passing the dots.

C.7 Exercises

Problem 1 (Easy): Consider the following function definition, and subsequent call of that function.

```
library(tidyverse)

summarize_species <- function(pattern = "Human") {
  x <- starwars %>%
    filter(species == pattern) %>%
    summarize(
      num_people = n(),
      avg_height = mean(height, na.rm = TRUE)
    )
}

summarize_species("Wookiee")
x
```

```
Error in eval(expr, envir, enclos): object 'x' not found
```

What causes this error?

Problem 2 (Medium): Write a function called `count_name` that, when given a name as an argument, returns the total number of births by year from the `babynames` data frame in the `babynames` package that match that name. The function should return one row per year that matches (and generate an error message if there are no matches). Run the function once with the argument `Ezekiel` and once with `Ezze`.

Problem 3 (Medium):

- a. Write a function called `count_na` that, when given a vector as an argument, will count the number of NA's in that vector.

Count the number of missing values in the `SEXRISK` variable in the `HELPfull` data frame in the `mosaicData` package.

- b. Apply `count_na` to the columns of the `Teams` data frame from the `Lahman` package. How many of the columns have missing data?

Problem 4 (Medium): Write a function called `map_negative` that takes as arguments a data frame and the name of a variable and returns that data frame with the negative values of the variable replaced by zeroes. Apply this function to the `cyl` variable in the `mtcars` data set.

Problem 5 (Medium): Write a function called `grab_name` that, when given a name and a year as an argument, returns the rows from the `babynames` data frame in the `babynames` package that match that name for that year (and returns an error if that name and year combination does not match any rows). Run the function once with the arguments `Ezekiel` and `1883` and once with `Ezekiel` and `1983`.

Problem 6 (Medium): Write a function called `prop_cancel` that takes as arguments a month number and destination airport and returns the proportion of flights missing arrival delay for each day to that destination. Apply this function to the `nycflights13` package for February and Atlanta airport `ATL` and again with an invalid month number.

Problem 7 (Medium): Write a function called `cum_min()` that, when given a vector as an argument, returns the cumulative minimum of that vector. Compare the result of your function to the built-in `cummin()` function for the vector `c(4, 7, 9, -2, 12)`.

Problem 8 (Hard): Benford's law concerns the frequency distribution of leading digits from numerical data. Write a function that takes a vector of numbers and returns the empirical distribution of the first digit. Apply this function to data from the `corporate.payment` data set in the `benford.analysis` package.

C.8 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-algorithmic.html#algorithmic-online-exercises>

D

Reproducible analysis and workflow

The notion that scientific findings can be confirmed repeatedly through replication is fundamental to the centuries-old paradigm of science. The underlying logic is that if you have identified a truth about the world, that truth should persist upon further investigation by other observers. In the physical sciences, there are two challenges in replicating a study: *replicating* the experiment itself, and *reproducing* the subsequent data analysis that led to the conclusion. More concisely, replicability means that different people get the same results with *different* data. Reproducibility means that the same person (or different people) get the same results with the *same* data.

It is easy to imagine why replicating a physical experiment might be difficult, and not being physical scientists ourselves, we won't tackle those issues here. On the other hand, the latter challenge of reproducing the data analysis is most certainly our domain. It seems like a much lower hurdle to clear—isn't this just a matter of following a few steps? Upon review, for a variety of reasons many scientists are in fact tripping over even this low hurdle.

To further explicate the distinction between *replicability* and *reproducibility*, recall that scientists are legendary keepers of lab notebooks. These notebooks are intended to contain all of the information needed to carry out the study again (i.e., replicate): reagents and other supplies, equipment, experimental material, etc. Modern software tools enable scientists to carry this same ethos to data analysis: Everything needed to repeat the analysis (i.e., reproduce) should be recorded in one place.

Even better, modern software tools allow the analysis to be repeated at the push of a button. This provides a proof that the analysis being documented is in fact exactly the same as the analysis that was performed. Moreover, this capability is a boon to those generating the analysis. It enables them to draft and redraft the analysis until they get it exactly right. Even better, when the analysis is written appropriately, it's straightforward to apply the analysis to new data. Spreadsheet software, despite its popularity, is not suitable for this. Spreadsheet software references specific rows and columns of data, and so the analysis commands themselves need to be updated to conform to new data.

The *replication crisis* is a very real problem for modern science. More than 15 years ago, Ioannidis (2005) argued that “most published research findings are false.” More recently, the journal *Nature* ran a series of editorials bemoaning the lack of replicability in published research (Editorial, 2013). It now appears that even among peer-reviewed, published scientific articles, many of the findings—which are supported by experimental and statistical evidence—do not hold up under the scrutiny of replication. That is, when other researchers try to do the same study, they don't reliably reach the same conclusions.

Some of the issues leading to irreproducibility are hard to understand, let alone solve. Much of the blame involves multiplicity and the “garden of forking paths” introduced in [Chapter 9](#). While we touch upon issues related to null hypothesis testing in [Chapter 9](#), the focus of this chapter is on modern workflows for *reproducible analysis*, since the ability to regenerate a set

of results at a later point in time is a necessary but not sufficient condition for reproducible results.

The National Academies report on undergraduate data science included workflow and reproducibility as an important part of *data acumen* (National Academies of Science, Engineering, and Medicine, 2018). They described key components including workflows and workflow systems, reproducible analysis, documentation and code standards, version control systems, and collaboration.

Reproducible workflows consist of three components: a fully scriptable statistical programming environment (such as **R** or Python), reproducible analysis (first described as literate programming), and version control (commonly implemented using GitHub).

D.1 Scriptable statistical computing

In order for data analysis to be reproducible, all of the steps taken in the analysis have to be recorded in a linear fashion. Scriptable applications like Python, **R**, SAS, and Stata do this by default. Even when graphical user interfaces to these programs are used, they add the automatically-generated code to the history so that it too can be recorded. Thus, the full series of commands that make up the data analysis can be recorded, reviewed, and transmitted. Contrast this with the behavior of spreadsheet applications like *Microsoft Excel* and *Google Sheets*, where it is not always possible to fully retrace one's steps.

D.2 Reproducible analysis with R Markdown

The concept of *literate programming* was introduced by Knuth decades ago (Knuth, 1992). His advice was:

“Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.”

Central to this prescription is the idea that the relevant documentation for the code—which is understandable not just to the programmer, but to other human beings as well—occurs alongside the code itself. In data analysis, this is manifest as the need to have three kinds of things in one document: the code, the results of that code, and the written analysis. We belong to a growing group of people who find the **rmarkdown** (Allaire et al., 2014) and **knitr** packages (Xie, 2014) to be an environment that is ideally suited to support a reproducible analysis workflow (Baumer et al., 2014).

The **rmarkdown** and **knitr** packages use a source file and output file paradigm. This

approach is common in programming but is fundamentally different than a “what-you-see-is-what-you-get” editor like *Microsoft Word* or *Google Docs*. Code is typed into the source document, which is then rendered into an output format that is readable by anyone. The principles of literate programming stipulate that the source file should *also* be readable by anyone.

We favor the simple document markup language **R** Markdown (Allaire et al., 2020a) for most applications. An **R** Markdown source file can be rendered (by **knitr**, leveraging **pandoc**) into PDF, HTML, and Microsoft Word formats. The resulting document will contain the **R** code, the results of that code, and the analyst’s written analysis.

Markdown is well-integrated with **RStudio**, and both L^AT_EX and Markdown source files can be rendered via a single-click mechanism. More details can be found in Xie (2014) and Gandrud (2014) as well as the CRAN reproducible research task view (Kuhn, 2020b). See also (<http://yihui.name/knitr>).

As an example of how these systems work, we demonstrate a document written in the Markdown format using data from the **SwimRecords** data frame. Within **RStudio**, a new template **R** Markdown file can be generated by selecting **R Markdown** from the **New File** option on the **File** menu. This generates the dialog box displayed in [Figure D.1](#). The default output format is HTML, but other options (PDF or Microsoft Word) are available.

Pro Tip 56. *The Markdown templates included with some packages (e.g., **mosaic**) are useful to set up more appropriate defaults for figure and font size. These can be accessed using the “From Template” option when opening a new Markdown file.*

```
---
```

```
title: "Sample R Markdown example"
author: "Sample User"
date: "November 8, 2020"
output:
  html_document: default
  fig_height: 2.8
  fig_width: 5
---
```

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
library(tidyverse)
library(mdsr)
library(mosaic)
```

## R Markdown
```

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the ****Knit**** button a document will be generated that includes both content as well as the output of any embedded R code chunks

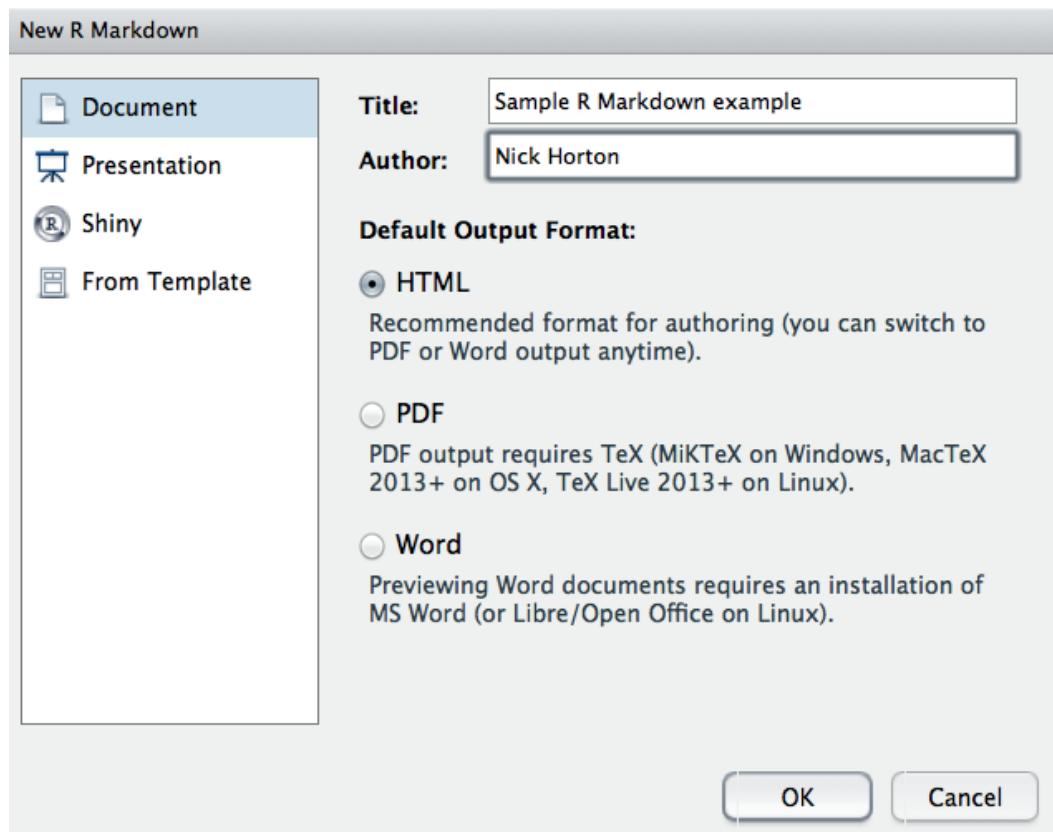


Figure D.1: Generating a new R Markdown file in RStudio.

within the document. You can embed an R code chunk like this:

```
```{r display}
glimpse(SwimRecords)
```
```

```
## Including Plots
```

You can also embed plots, for example:

```
```{r scatplot, echo=FALSE, message = FALSE}
ggplot(
 data = SwimRecords,
 aes(x = year, y = time, color = sex)
) +
 geom_point() +
 geom_smooth(method = loess, se = FALSE) +
 theme(legend.position = "right") +
 labs(title = "100m Swimming Records over time")
```
```

```
There are n=`r nrow(SwimRecords)` rows in the Swim records dataset.
```

Note that the `echo = FALSE` option was added to the code chunk to prevent printing of the R code that generated the plot.

Figure D.2: Sample Markdown input file.

Figure D.2 displays a modified version of the default **R** Markdown input file. The file is given a title (`Sample R Markdown example`) with output format set by default to HTML. Simple markup (such as bolding) is added through use of the `**` characters before and after the word `Help`. Blocks of code are begun using a ````{r}` line and closed with a ````` line (three back quotes in both cases).

The formatted output can be generated and displayed by clicking the `Knit HTML` button in RStudio, or by using the commands in the following code block, which can also be used when running **R** without the benefit of **RStudio**.

```
library(rmarkdown)
render("filename.Rmd")    # creates filename.html
browseURL("filename.html")
```

The `render()` function extracts the **R** commands from a specially formatted **R** Markdown input file (`filename.Rmd`), evaluates them, and integrates the resulting output, including text and graphics, into an output file (`filename.html`). A screenshot of the results of performing these steps on the `.Rmd` file displayed in **Figure D.2** is displayed in **Figure D.3**. `render()` uses the value of the `output:` option to determine what format to generate. If the `.Rmd` file specified `output: word_document`, then a Microsoft Word document would be created.

Alternatively, a PDF or Microsoft Word document can be generated in **RStudio** by selecting `New` from the `R Markdown` menu, then clicking on the PDF or Word options. **RStudio** also supports the creation of **R** Presentations using a variant of the **R** Markdown syntax. Instructions and an example can be found by opening a new `R presentations` document in **RStudio**.

D.3 Projects and version control

Projects are a useful feature of **RStudio**. A project provides a separate workspace. Selecting a project also reorients your **RStudio** environment to a specified directory, in the process reorienting the Files tab, the working directory, etc. Once you start working on multiple projects, being able to switch back and forth becomes very helpful.

Given that data science has been called a “team sport,” the ability to track changes to files and discuss issues in a collaborative manner is an important prerequisite to reproducible analysis. Projects can be tied to a version control system, such as *Subversion* or *GitHub*. These systems help you and your collaborators keep track of changes to files, so that you can go back in time to review changes to previous pieces of code, compare versions, and retrieve older versions as needed.

Pro Tip 57. While critical for collaboration with others, source code version control systems

Sample R Markdown example

Sample User

November 8, 2020

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

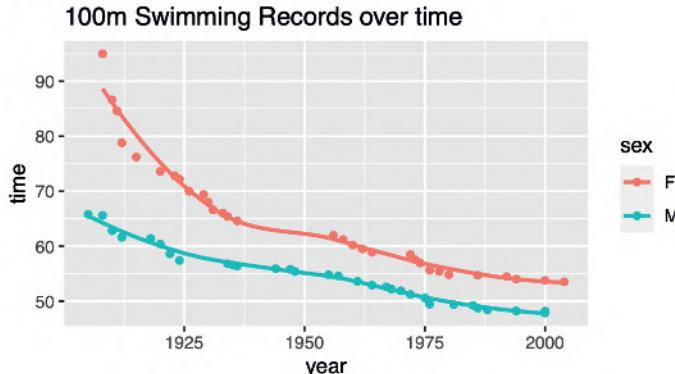
When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
glimpse(SwimRecords)
```

```
## Rows: 62
## Columns: 3
## $ year <int> 1905, 1908, 1910, 1912, 1918, 1920, 1922, 1924, 1934, 1935, 19...
## $ time <dbl> 65.80, 65.60, 62.80, 61.60, 61.40, 60.40, 58.60, 57.40, 56.80, ...
## $ sex <fct> M, ...
```

Including Plots

You can also embed plots, for example:



There are n=62 rows in the Swim records dataset.

Note that the `echo = FALSE` option was added to the code chunk to prevent printing of the R code that generated the plot.

Figure D.3: Formatted output from R Markdown example.

are also useful for individual projects because they document changes and maintain version histories. In such a setting, the collaboration is with your future self!

GitHub is a cloud-based implementation of Git that is tightly integrated into **RStudio**. It works efficiently, without cluttering your workspace with duplicate copies of old files or compressed archives. **RStudio** users can collaborate on projects hosted on GitHub without having to use the command line. This has proven to be an effective way of ensuring a consistent, reproducible workflow, even for beginners. This book was written collaboratively through a private repository on GitHub, just as the **mdsr** package is maintained in a public repository.

Finally, random number seeds (see section 13.6.3) are an important part of a reproducible

workflow. It is a good idea to set a seed to allow others (or yourself) to be able to reproduce a particular analysis that has a stochastic component to it.

D.4 Further resources

Project TIER is an organization at *Haverford College* that has developed a protocol (Ball and Medeiros, 2012) for reproducible research. Their efforts originated in the social sciences using *Stata* but have since expanded to include **R**.

R Markdown is under active development. For the latest features see the **R** Markdown authoring guide at (<http://rmarkdown.rstudio.com>). The **RStudio** cheat sheet serves as a useful reference.

Broman and Woo (2018) describe best practices for organizing data in spreadsheets.

GitHub can be challenging to learn but is now the default in many data science research settings. Jenny Bryan's resources on *Happy Git and GitHub for the userR* are particularly relevant for new data scientists beginning to use GitHub (Bryan et al., 2018).

Another challenge for reproducible analysis is ever-changing versions of **R** and other **R** packages. The **packrat** and **renv** packages help ensure that projects can maintain a particular version of **R** and set of packages. The **reproducible** package provides a set of tools to enhance reproducibility.

D.5 Exercises

Problem 1 (Easy): Insert a chunk in an R Markdown document that generates an error. Set the options so that the file renders even though there is an error. (Note: Some errors are easier to diagnose if you can execute specific R statements during rendering and leave more evidence behind for forensic examination.)

Problem 2 (Easy): Why does the **mosaic** package plain R Markdown template include the code chunk option `message=FALSE` when the **mosaic** package is loaded?

Problem 3 (Easy): Consider an R Markdown file that includes the following code chunks. What will be output when this file is rendered?

```
```{r}
x <- 1:5
...``
```

```
```{r}
x <- x + 1
...``
```

```
```{r}
x
...``
```

**Problem 4 (Easy):** Consider an R Markdown file that includes the following code chunks. What will be output when the file is rendered?

```
```{r echo = FALSE}
x <- 1:5
...```

```

```
```{r echo = FALSE}
x <- x + 1
...```

```

```
```{r include = FALSE}
x
...```

```

Problem 5 (Easy): Consider an R Markdown file that includes the following code chunks. What will be output when this file is rendered?

```
```{r echo = FALSE}
x <- 1:5
...```

```

```
```{r echo = FALSE}
x <- x + 1
...```

```

```
```{r echo = FALSE}
x
...```

```

**Problem 6 (Easy):** Consider an R Markdown file that includes the following code chunks. What will be output when the file is rendered?

```
```{r echo = FALSE}
x <- 1:5
...```

```

```
```{r echo = FALSE, eval = FALSE}
x <- x + 1
...```

```

```
```{r echo = FALSE}
x
...```

```

Problem 7 (Easy): Describe in words what the following excerpt from an R Markdown file will display when rendered.

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 \cdot x + \epsilon$$

Problem 8 (Easy): Create an RMarkdown file that uses an inline call to R to display the value of an object that you have created previously in that file.

Problem 9 (Easy): Describe the implications of changing `warning=TRUE` to `warning=FALSE` in the following code chunk.

```
sqrt(-1)
```

Warning in sqrt(-1): NaNs produced

```
[1] NaN
```

Problem 10 (Easy): Describe how the `fig.width` and `fig.height` chunk options can be used to control the size of graphical figures. Generate two versions of a figure with different options.

Problem 11 (Medium): The `knitr` package allows the analyst to display nicely formatted tables and results when outputting to pdf files. Use the following code chunk as an example to create a similar display using your own data.

```
library(mdsr)
library(mosaicData)
mod <- broom::tidy(lm(cesd ~ mcs + sex, data = HELPrc))
knitr::kable(
  mod,
  digits = c(0, 2, 2, 2, 4),
  caption = "Regression model from HELP clinical trial.",
  longtable = TRUE
)
```

Table D.1: Regression model from HELP clinical trial.

term	estimate	std.error	statistic	p.value
(Intercept)	55.79	1.31	42.62	0.0000
mcs	-0.65	0.03	-19.48	0.0000
sexmale	-2.95	1.01	-2.91	0.0038

Problem 12 (Medium): Explain what the following code chunks will display and why this might be useful for technical reports from a data science project.

```
```{r chunk1, eval = TRUE, include = FALSE}
x <- 15
cat("assigning value to x.\n")
```
```

```
```{r chunk2, eval = TRUE, include = FALSE}
x <- x + 3
cat("updating value of x.\n")
```
```

```
```{r chunk3, eval = FALSE, include = TRUE}
cat("x =", x, "\n")
```
```

```
```{r chunk1, eval = FALSE, include = TRUE}
```
```

```
```{r chunk2, eval = FALSE, include = TRUE}
```
```

D.6 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-reproducible.html#reproducible-online-exercises>

E

Regression modeling

Regression analysis is a powerful and flexible framework that allows an analyst to model an outcome (the *response variable*) as a function of one or more *explanatory variables* (or predictors). Regression forms the basis of many important statistical models described in Chapters 9 and 11. This appendix provides a brief review of linear and logistic regression models, beginning with a single predictor, then extending to multiple predictors.

E.1 Simple linear regression

Linear regression can help us understand how values of a quantitative (numerical) outcome (or response) are associated with values of a quantitative explanatory (or predictor) variable. This technique is often applied in two ways: to generate predicted values or to make inferences regarding associations in the dataset.

In some disciplines the outcome is called the dependent variable and the predictor the independent variable. We avoid such usage since the words dependent and independent have many meanings in statistics.

A simple linear regression model for an outcome y as a function of a predictor x takes the form:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i, \text{ for } i = 1, \dots, n,$$

where n represents the number of observations (rows) in the data set. For this model, β_0 is the population parameter corresponding to the *intercept* (i.e., the predicted value when $x = 0$) and β_1 is the true (population) *slope* coefficient (i.e., the predicted increase in y for a unit increase in x). The ϵ_i 's are the *errors* (these are assumed to be random noise with mean 0).

We almost never know the true values of the population parameters β_0 and β_1 , but we estimate them using data from our sample. The `lm()` function finds the “best” coefficients $\hat{\beta}_0$ and $\hat{\beta}_1$ where the *fitted values* (or expected values) are given by $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$. What is left over is captured by the *residuals* ($\hat{\epsilon}_i = y_i - \hat{y}_i$). The model almost never fits perfectly—if it did there would be no need for a model.

The best-fitting regression line is usually determined by a *least squares* criteria that minimizes the sum of the squared residuals (ϵ_i^2). The least squares regression line (defined by the values of $\hat{\beta}_0$ and $\hat{\beta}_1$) is unique.

E.1.1 Motivating example: Modeling usage of a rail trail

The *Pioneer Valley* Planning Commission (PVPC) collected data north of Chestnut Street in *Florence, Massachusetts* for a 90-day period. Data collectors set up a laser sensor that

recorded when a rail-trail user passed the data collection station. The data are available in the `RailTrail` data set in the **mosaicData** package.

```
library(tidyverse)
library(mdsr)
library(mosaic)
glimpse(RailTrail)

Rows: 90
Columns: 11
$ hightemp <int> 83, 73, 74, 95, 44, 69, 66, 66, 80, 79, 78, 65, 41, ...
$ lowtemp <int> 50, 49, 52, 61, 52, 54, 39, 38, 55, 45, 55, 48, 49, ...
$ avgtemp <dbl> 66.5, 61.0, 63.0, 78.0, 48.0, 61.5, 52.5, 52.0, 67.5...
$ spring <int> 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1...
$ summer <int> 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0...
$ fall <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0...
$ cloudcover <dbl> 7.6, 6.3, 7.5, 2.6, 10.0, 6.6, 2.4, 0.0, 3.8, 4.1, 8...
$ precip <dbl> 0.00, 0.29, 0.32, 0.00, 0.14, 0.02, 0.00, 0.00, 0.00...
$ volume <int> 501, 419, 397, 385, 200, 375, 417, 629, 533, 547, 43...
$ weekday <lgl> TRUE, TRUE, TRUE, FALSE, TRUE, TRUE, FALSE, FA...
$ dayType <chr> "weekday", "weekday", "weekday", "weekend", "weekday..."
```

The PVPC wants to understand the relationship between daily ridership (i.e., the number of riders and walkers who use the bike path on any given day) and a collection of explanatory variables, including the temperature, rainfall, cloud cover, and day of the week.

In a simple linear regression model, there is a single quantitative explanatory variable. It seems reasonable that the high temperature for the day (`hightemp`, measured in degrees Fahrenheit) might be related to ridership, so we will explore that first. Figure E.1 shows a scatterplot between ridership (`volume`) and high temperature (`hightemp`), with the simple linear regression line overlaid. The fitted coefficients are calculated through a call to the `lm()` function.

We will use functions from the **broom** package to display model results in a tidy fashion.

```
mod <- lm(volume ~ hightemp, data = RailTrail)
library(broom)
tidy(mod)

# A tibble: 2 x 5
  term      estimate std.error statistic     p.value
  <chr>     <dbl>     <dbl>     <dbl>       <dbl>
1 (Intercept) -17.1      59.4    -0.288  0.774
2 hightemp      5.70      0.848     6.72  0.000000000171

ggplot(RailTrail, aes(x = hightemp, y = volume)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```

The first coefficient is $\hat{\beta}_0$, the estimated y -intercept. The interpretation is that if the high temperature was 0 degrees Fahrenheit, then the estimated ridership would be about -17 riders. This is doubly non-sensical in this context, since it is impossible to have a negative number of riders and this represents a substantial extrapolation to far colder temperatures than are present in the data set (recall the *Challenger* discussion from Chapter 2). It turns

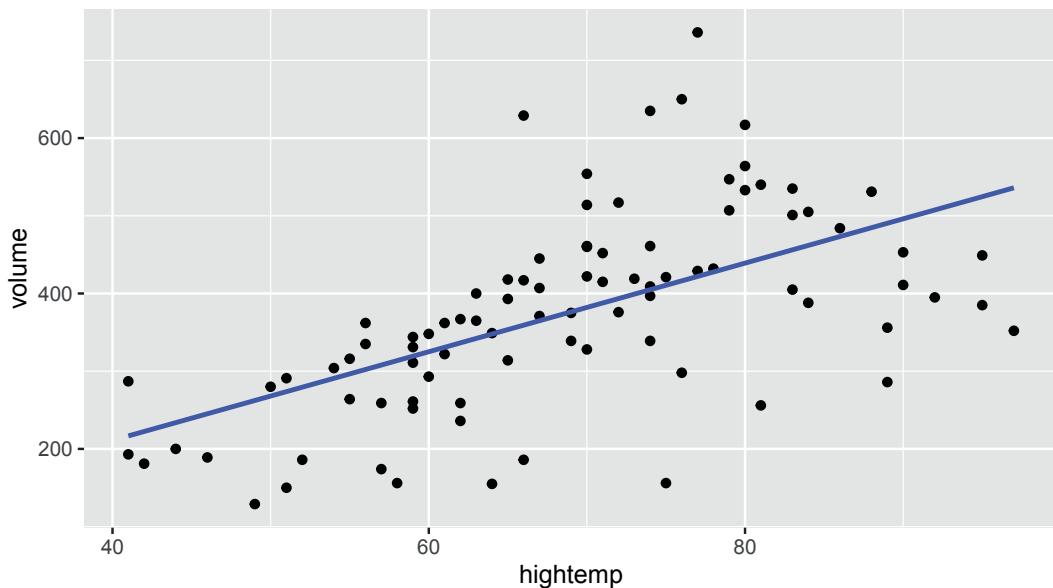


Figure E.1: Scatterplot of number of trail crossings as a function of highest daily temperature (in degrees Fahrenheit).

out that the monitoring equipment didn't work when it got too cold, so values for those days are unavailable.

Pro Tip 58. *In this case, it is not appropriate to simply multiply the average number of users on the observed days by the number of days in a year, since cold days that are likely to have fewer trail users are excluded due to instrumentation issues. Such missing data can lead to selection bias.*

The second coefficient (the slope) is usually more interesting. This coefficient ($\hat{\beta}_1$) is interpreted as the predicted increase in trail users for each additional degree in temperature. We expect to see about 5.7 additional riders use the rail trail on a day that is 1 degree warmer than another day.

E.1.2 Model visualization

Figure E.1 allows us to visualize our model in the data space. How does our model compare to a null model? That is, how do we know that our model is useful?

```
library(broom)
mod_avg <- RailTrail %>%
  lm(volume ~ 1, data = .) %>%
  augment(RailTrail)
mod_temp <- RailTrail %>%
  lm(volume ~ hightemp, data = .) %>%
  augment(RailTrail)
mod_data <- bind_rows(mod_avg, mod_temp) %>%
  mutate(model = rep(c("null", "slr"), each = nrow(RailTrail)))
```

In Figure E.2, we compare the least squares regression line (right) with the null model

that simply returns the average for every input (left). That is, on the left, the average temperature of the day is ignored. The model simply predicts an average ridership every day, regardless of the temperature. However, on the right, the model takes the average ridership into account, and accordingly makes a different prediction for each input value.

```
ggplot(data = mod_data, aes(x = hightemp, y = volume)) +
  geom_smooth(
    data = filter(mod_data, model == "null"),
    method = "lm", se = FALSE, formula = y ~ 1,
    color = "dodgerblue", size = 0.5
  ) +
  geom_smooth(
    data = filter(mod_data, model == "slr"),
    method = "lm", se = FALSE, formula = y ~ x,
    color = "dodgerblue", size = 0.5
  ) +
  geom_segment(
    aes(xend = hightemp, yend = .fitted),
    arrow = arrow(length = unit(0.1, "cm")),
    size = 0.5, color = "darkgray"
  ) +
  geom_point(color = "dodgerblue") +
  facet_wrap(~model)
```

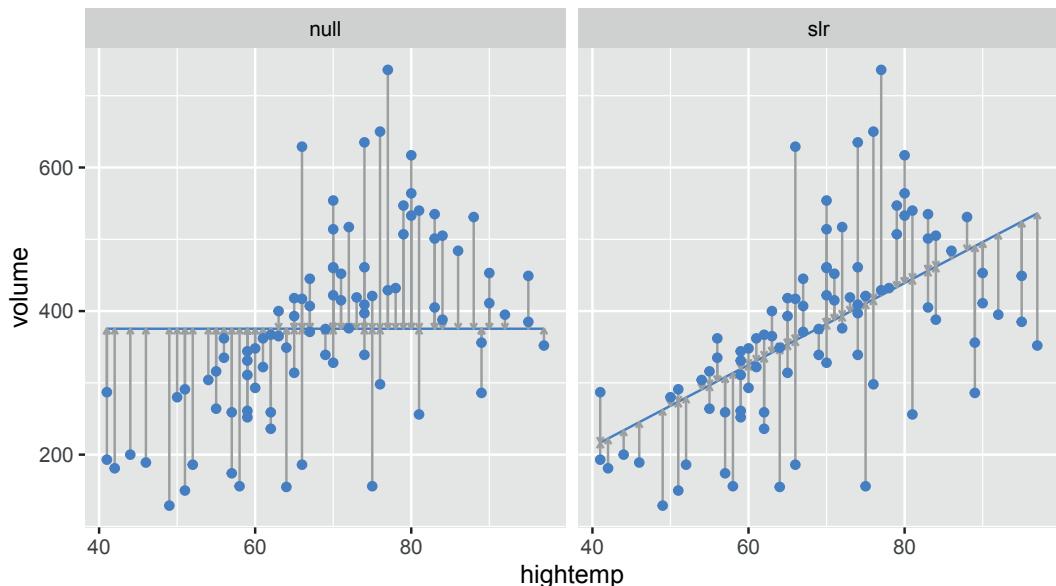


Figure E.2: At left, the model based on the overall average high temperature. At right, the simple linear regression model.

Obviously, the regression model works better than the null model (that forces the slope to be zero), since it is more flexible. But how much better?

E.1.3 Measuring the strength of fit

The correlation coefficient, r , is used to quantify the strength of the linear relationship between two variables. We can quantify the proportion of variation in the response variable (y) that is explained by the model in a similar fashion. This quantity is called the *coefficient of determination* and is denoted R^2 . It is a common measure of goodness-of-fit for regression models. Like any proportion, R^2 is always between 0 and 1. For simple linear regression (one explanatory variable), $R^2 = r^2$. The definition of R^2 is given by the following expression.

$$\begin{aligned} R^2 &= 1 - \frac{SSE}{SST} = \frac{SSM}{SST} \\ &= 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2} \\ &= 1 - \frac{SSE}{(n-1)Var(y)}, \end{aligned}$$

Here, \hat{y} is the predicted value, $Var(y)$ is the observed variance, SSE is the sum of the squared residuals, SSM is the sum of the squares attributed to the model, and SST is the total sum of the squares. We can calculate these values for the rail trail example.

```
n <- nrow(RailTrail)
SST <- var(pull(RailTrail, volume)) * (n - 1)
SSE <- var(residuals(mod)) * (n - 1)
1 - SSE / SST

[1] 0.339
glance(mod)

# A tibble: 1 x 12
#>   r.squared adj.r.squared sigma statistic p.value    df logLik     AIC     BIC
#>   <dbl>        <dbl>    <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl>
1     0.339       0.332   104.      45.2 1.71e-9      1  -545. 1096. 1103.
# ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

In Figure E.2, the null model on the left has an R^2 of 0, because $\hat{y}_i = \bar{y}$ for all i , and so $SSE = SST$. On the other hand, the R^2 of the regression model on the right is 0.339. We say that the regression model based on average daily temperature explained about 34% of the variation in daily ridership.

E.1.4 Categorical explanatory variables

Suppose that instead of using temperature as our explanatory variable for ridership on the rail trail, we only considered whether it was a weekday or not a weekday (e.g., weekend or holiday). The indicator variable `weekday` is *binary* (or dichotomous) in that it only takes on the values 0 and 1. (Such variables are sometimes called *indicator* variables or more pejoratively *dummy* variables.)

This new linear regression model has the form:

$$volume = \hat{\beta}_0 + \hat{\beta}_1 \cdot \text{weekday},$$

where the fitted coefficients are given below.

```
coef(lm(volume ~ weekday, data = RailTrail))

(Intercept) weekdayTRUE
        430.7      -80.3
```

Note that these coefficients could have been calculated from the means of the two groups (since the regression model has only two possible predicted values). The average ridership on weekdays is 350.4 while the average on non-weekdays is 430.7.

```
RailTrail %>%
  group_by(weekday) %>%
  summarize(mean_volume = mean(volume))
```

| | weekday | mean_volume |
|---|---------|-------------|
| 1 | FALSE | 431. |
| 2 | TRUE | 350. |

In the coefficients listed above, the `weekdayTRUE` variable corresponds to rows in which the value of the `weekday` variable was `TRUE` (i.e., weekdays). Because this value is negative, our interpretation is that 80 fewer riders are expected on a weekday as opposed to a weekend or holiday.

To improve the readability of the output we can create a new variable with more mnemonic values.

```
RailTrail <- RailTrail %>%
  mutate(day = ifelse(weekday == 1, "weekday", "weekend/holiday"))
```

Pro Tip 59. *Care was needed to recode the `weekday` variable because it was a factor. Avoid the use of factors unless they are needed.*

```
coef(lm(volume ~ day, data = RailTrail))

(Intercept) dayweekend/holiday
            350.4          80.3
```

The model coefficients have changed (although they still provide the same interpretation). By default, the `lm()` function will pick the alphabetically lowest value of the categorical predictor as the *reference group* and create indicators for the other levels (in this case `dayweekend/holiday`). As a result the intercept is now the predicted number of trail crossings on a weekday. In either formulation, the interpretation of the model remains the same: On a weekday, 80 fewer riders are expected than on a weekend or holiday.

E.2 Multiple regression

Multiple regression is a natural extension of simple linear regression that incorporates multiple explanatory (or predictor) variables. It has the general form:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p + \epsilon, \text{ where } \epsilon \sim N(0, \sigma_\epsilon).$$

The estimated coefficients (i.e., $\hat{\beta}_i$'s) are now interpreted as “conditional on” the other variables—each β_i reflects the *predicted* change in y associated with a one-unit increase in x_i , conditional upon the rest of the x_j 's. This type of model can help to disentangle more complex relationships between three or more variables. The value of R^2 from a multiple regression model has the same interpretation as before: the proportion of variability explained by the model.

Pro Tip 60. *Interpreting conditional regression parameters can be challenging. The analyst needs to ensure that comparisons that hold other factors constant do not involve extrapolations beyond the observed data.*

E.2.1 Parallel slopes: Multiple regression with a categorical variable

Consider first the case where x_2 is an *indicator* variable that can only be 0 or 1 (e.g., `weekday`). Then,

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2.$$

In the case where x_1 is quantitative but x_2 is an indicator variable, we have:

$$\begin{aligned} \text{For weekends, } \hat{y}|_{x_1,x_2=0} &= \hat{\beta}_0 + \hat{\beta}_1 x_1 \\ \text{For weekdays, } \hat{y}|_{x_1,x_2=1} &= \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 \cdot 1 \\ &= (\hat{\beta}_0 + \hat{\beta}_2) + \hat{\beta}_1 x_1. \end{aligned}$$

This is called a *parallel slopes* model (see Figure E.3), since the predicted values of the model take the geometric shape of two parallel lines with slope $\hat{\beta}_1$: one with y -intercept $\hat{\beta}_0$ for weekends, and another with y -intercept $\hat{\beta}_0 + \hat{\beta}_2$ for weekdays.

```
mod_parallel <- lm(volume ~ hightemp + weekday, data = RailTrail)
tidy(mod_parallel)
```

```
# A tibble: 3 x 5
  term      estimate std.error statistic   p.value
  <chr>     <dbl>    <dbl>     <dbl>      <dbl>
1 (Intercept)  42.8     64.3     0.665  0.508
2 hightemp      5.35    0.846     6.32  0.0000000109
3 weekdayTRUE -51.6    23.7     -2.18  0.0321
```

```
glance(mod_parallel)
```

```
# A tibble: 1 x 12
  r.squared adj.r.squared sigma statistic p.value    df logLik    AIC    BIC
  <dbl>        <dbl>    <dbl>     <dbl>      <dbl>    <dbl>    <dbl>    <dbl>
1 0.374       0.359  102.      25.9  1.46e-9      2 -542. 1093. 1103.
# ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

```
mod_parallel %>%
  augment() %>%
  ggplot(aes(x = hightemp, y = volume, color = weekday)) +
  geom_point() +
  geom_line(aes(y = .fitted)) +
  labs(color = "Is it a\nweekday?")
```

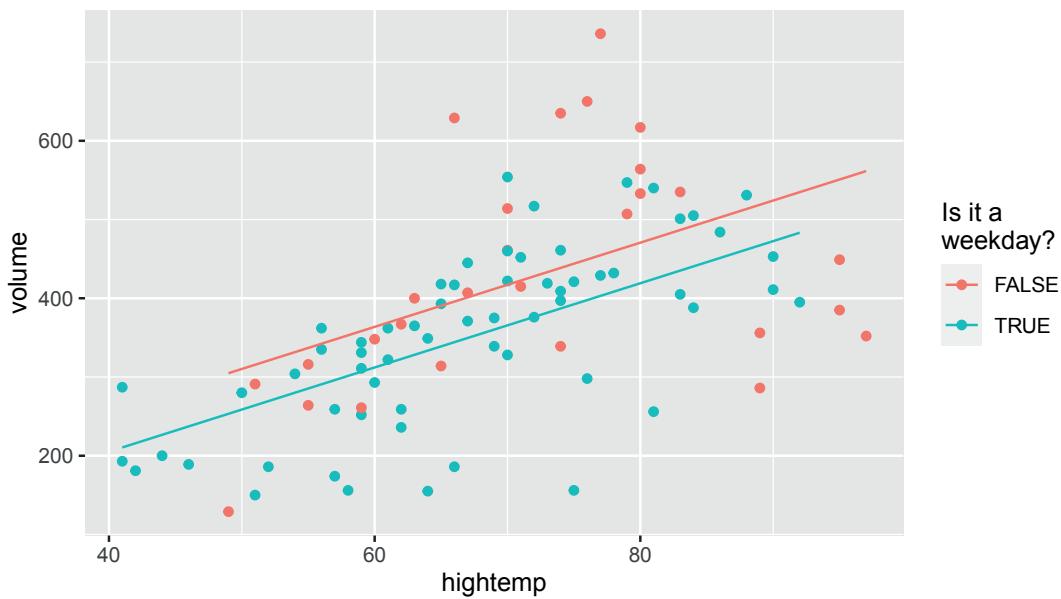


Figure E.3: Visualization of parallel slopes model for the rail trail data.

E.2.2 Parallel planes: Multiple regression with a second quantitative variable

If x_2 is a quantitative variable, then we have:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2.$$

Notice that our model is no longer a line, rather it is a *plane* that exists in three dimensions.

Now suppose that we want to improve our model for ridership by considering not only the average temperature, but also the amount of precipitation (rain or snow, measured in inches). We can do this in R by simply adding this variable to our regression model.

```
mod_plane <- lm(volume ~ hightemp + precip, data = RailTrail)
tidy(mod_plane)
```

| # A tibble: 3 x 5 | term | estimate | std.error | statistic | p.value |
|-------------------|-------------|----------|-----------|-----------|----------|
| | <chr> | <dbl> | <dbl> | <dbl> | <dbl> |
| 1 | (Intercept) | -31.5 | 55.2 | -0.571 | 5.70e- 1 |
| 2 | hightemp | 6.12 | 0.794 | 7.70 | 1.97e-11 |
| 3 | precip | -153. | 39.3 | -3.90 | 1.90e- 4 |

Note that the coefficient on `hightemp` (6.1 riders per degree) has changed from its value in the simple linear regression model (5.7 riders per degree). This is due to the moderating effect of precipitation. Our interpretation is that for each additional degree in temperature, we expect an additional 6.1 riders on the rail trail, after controlling for the amount of precipitation.

As you can imagine, the effect of precipitation is strong—some people may be less likely to bike or walk in the rain. Thus, even after controlling for temperature, an inch of rainfall is associated with a drop in ridership of about 153.

Pro Tip 61. Note that since the median precipitation on days when there was precipitation was only 0.15 inches, a predicted change for an additional inch may be misleading. It may be better to report a predicted difference of 0.15 additional inches or replace the continuous term in the model with a dichotomous indicator of any precipitation.

If we added all three explanatory variables to the model we would have parallel *planes*.

```
mod_p_planes <- lm(volume ~ hightemp + precip + weekday, data = RailTrail)
tidy(mod_p_planes)
```

| # A tibble: 4 x 5 | term | estimate | std.error | statistic | p.value |
|-------------------|-------------|----------|-----------|-----------|----------|
| | <chr> | <dbl> | <dbl> | <dbl> | <dbl> |
| 1 | (Intercept) | 19.3 | 60.3 | 0.320 | 7.50e- 1 |
| 2 | hightemp | 5.80 | 0.799 | 7.26 | 1.59e-10 |
| 3 | precip | -146. | 38.9 | -3.74 | 3.27e- 4 |
| 4 | weekdayTRUE | -43.1 | 22.2 | -1.94 | 5.52e- 2 |

E.2.3 Non-parallel slopes: Multiple regression with interaction

Let's return to a model that includes `weekday` and `hightemp` as predictors. What if the parallel lines model doesn't fit well? Adding an additional term into the model can make it more flexible and allow there to be a different slope on the two different types of days.

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \hat{\beta}_3 x_1 x_2.$$

The model can also be described separately for weekends/holidays and weekdays.

$$\begin{aligned} \text{For weekends, } \hat{y}|_{x_1,x_2=0} &= \hat{\beta}_0 + \hat{\beta}_1 x_1 \\ \text{For weekdays, } \hat{y}|_{x_1,x_2=1} &= \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 \cdot 1 + \hat{\beta}_3 \cdot x_1 \\ &= (\hat{\beta}_0 + \hat{\beta}_2) + (\hat{\beta}_1 + \hat{\beta}_3) x_1. \end{aligned}$$

This is called an *interaction model* (see [Figure E.4](#)). The predicted values of the model take the geometric shape of two non-parallel lines with different slopes.

```
mod_interact <- lm(volume ~ hightemp + weekday + hightemp * weekday,
                     data = RailTrail)
tidy(mod_interact)
```

| # A tibble: 4 x 5 | term | estimate | std.error | statistic | p.value |
|-------------------|----------------------|----------|-----------|-----------|---------|
| | <chr> | <dbl> | <dbl> | <dbl> | <dbl> |
| 1 | (Intercept) | 135. | 108. | 1.25 | 0.215 |
| 2 | hightemp | 4.07 | 1.47 | 2.78 | 0.00676 |
| 3 | weekdayTRUE | -186. | 129. | -1.44 | 0.153 |
| 4 | hightemp:weekdayTRUE | 1.91 | 1.80 | 1.06 | 0.292 |

```
glance(mod_interact)
```

| # A tibble: 1 x 12 | r.squared | adj.r.squared | sigma | statistic | p.value | df | logLik | AIC | BIC | <dbl> | <dbl> |
|--------------------|-----------|---------------|-------|-----------|---------|-------|--------|-------|-------|-------|-------|
| | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> |
| 1 | 0.382 | 0.360 | 102. | 17.7 | 4.96e-9 | 3 | -542. | 1094. | 1106. | | |

```
# ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
mod_interact %>%
  augment() %>%
  ggplot(aes(x = hightemp, y = volume, color = weekday)) +
  geom_point() +
  geom_line(aes(y = .fitted)) +
  labs(color = "Is it a\nweekday?")
```

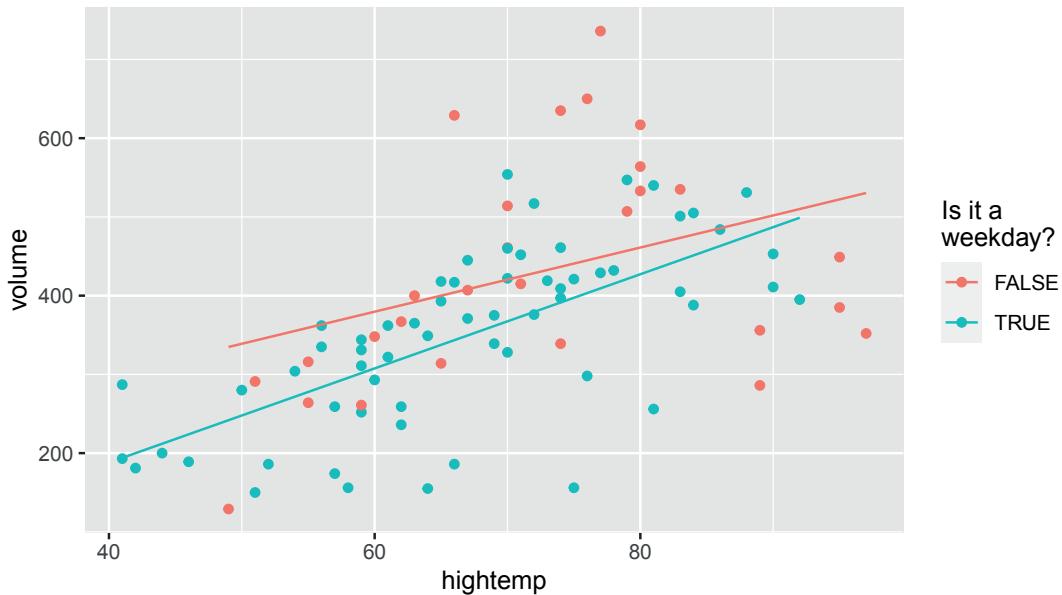


Figure E.4: Visualization of interaction model for the rail trail data.

We see that the slope on weekdays is about two riders per degree higher than on weekends and holidays. This may indicate that trail users on weekends and holidays are less concerned about the temperature than on weekdays.

E.2.4 Modeling non-linear relationships

A linear model with a single parameter fits well in many situations but is not appropriate in others. Consider modeling height (in centimeters) as a function of age (in years) using data from a subset of female subjects included in the *National Health and Nutrition Examination Study* (from the **NHANES** package) with a linear term. Another approach uses a *smoother* instead of a linear model. Unlike the straight line, the smoother can bend to better fit the points when modeling the functional form of a relationship (see Figure E.5).

```
library(NHANES)
NHANES %>%
  sample(300) %>%
  filter(Gender == "female") %>%
  ggplot(aes(x = Age, y = Height)) +
  geom_point() +
  geom_smooth(method = lm, se = FALSE) +
  geom_smooth(method = loess, se = FALSE, color = "green") +
```

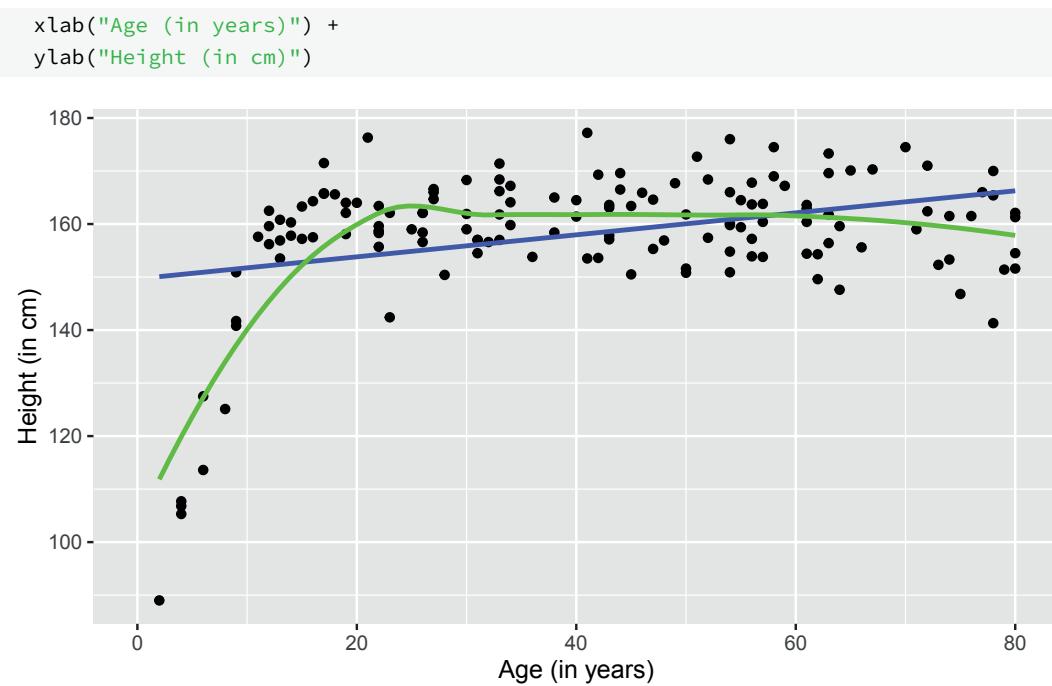
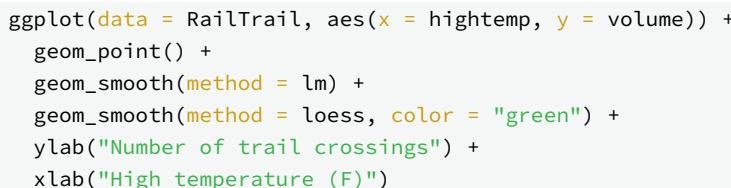


Figure E.5: Scatterplot of height as a function of age with superimposed linear model (blue) and smoother (green).

The fit of the linear model (denoted in blue) is poor: A straight line does not account for the dramatic increases in height during puberty to young adulthood or for the gradual decline in height for older subjects. The smoother (in green) does a much better job of describing the functional form.

The improved fit does come with a cost. Compare the results for linear and smoothed models in [Figure E.6](#). Here the functional form of the relationship between high temperature and volume of trail use is closer to linear (with some deviation for warmer temperatures).



The width of the confidence bands (95% confidence interval at each point) for the smoother tend to be wider than that for the linear model. This is one of the costs of the additional flexibility in modeling. Another cost is interpretation: It is more complicated to explain the results from the smoother than to interpret a slope coefficient (straight line).

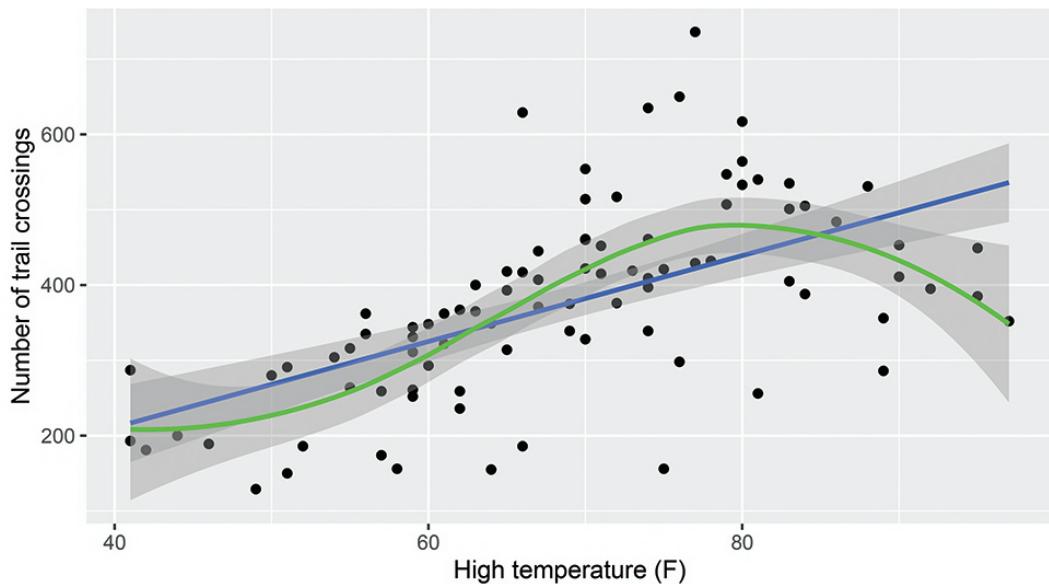


Figure E.6: Scatterplot of volume as a function of high temperature with superimposed linear and smooth models for the rail trail data.

E.3 Inference for regression

Thus far, we have fit several models and interpreted their estimated coefficients. However, with the exception of the confidence bands in Figure E.6, we have only made statements about the estimated coefficients (i.e., the $\hat{\beta}$'s)—we have made no statements about the true coefficients (i.e., the β 's), the values of which of course remain unknown.

However, we can use our understanding of the t -distribution to make *inferences* about the true value of regression coefficients. In particular, we can test a hypothesis about β_1 (most commonly that it is equal to zero) and find a confidence interval (range of plausible values) for it.

```
tidy(mod_p_planes)
```

```
# A tibble: 4 x 5
  term      estimate std.error statistic p.value
  <chr>      <dbl>     <dbl>      <dbl>    <dbl>
1 (Intercept)  19.3      60.3      0.320  7.50e- 1
2 hightemp      5.80      0.799     7.26   1.59e-10
3 precip     -146.       38.9     -3.74   3.27e- 4
4 weekdayTRUE -43.1      22.2     -1.94   5.52e- 2
```

```
glance(mod_p_planes)
```

```
# A tibble: 1 x 12
  r.squared adj.r.squared sigma statistic p.value    df logLik    AIC    BIC
  <dbl>        <dbl> <dbl>      <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl>
1     0.461      0.443  95.2      24.6  1.44e-11      3  -536.  1081. 1094.
```

```
# ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

In the output above, the p-value that is associated with the `hightemp` coefficient is displayed as 1.59e-10 (or nearly zero). That is, if the true coefficient (β_1) was in fact zero, then the probability of observing an association on ridership due to average temperature as large or larger than the one we actually observed in the data, after controlling for precipitation and day of the week, is essentially zero. This output suggests that the hypothesis that β_1 was in fact zero is dubious based on these data. Perhaps there is a real association between ridership and average temperature?

Pro Tip 62. Very small p-values should be rounded to the nearest 0.0001. We suggest reporting this p-value as $p < 0.0001$.

Another way of thinking about this process is to form a confidence interval around our estimate of the slope coefficient $\hat{\beta}_1$. Here we can say with 95% confidence that the value of the true coefficient β_1 is between 4.21 and 7.39 riders per degree. That this interval does not contain zero confirms the result from the hypothesis test.

```
confint(mod_p_planes)
```

| | 2.5 % | 97.5 % |
|-------------|---------|---------|
| (Intercept) | -100.63 | 139.268 |
| hightemp | 4.21 | 7.388 |
| precip | -222.93 | -68.291 |
| weekdayTRUE | -87.27 | 0.976 |

E.4 Assumptions underlying regression

The inferences we made above were predicated upon our assumption that the slope follows a t -distribution. This follows from the assumption that the errors follow a normal distribution (with mean 0 and standard deviation σ_ϵ , for some constant σ_ϵ). Inferences from the model are only valid if the following assumptions hold:

- **Linearity:** The functional form of the relationship between the predictors and the outcome follows a linear combination of regression parameters that are correctly specified (this assumption can be verified by bivariate graphical displays).
- **Independence:** Are the errors uncorrelated? Or do they follow a pattern (perhaps over time or within clusters of subjects)?
- **Normality of residuals:** Do the residuals follow a distribution that is approximately normal? This assumption can be verified using univariate displays.
- **Equal variance of residuals:** Is the variance in the residuals constant across the explanatory variables (*homoscedastic errors*)? Or does the variance in the residuals depend on the value of one or more of the explanatory variables (*heteroscedastic errors*)? This assumption can be verified using residual diagnostics.

These conditions are sometimes called the “LINE” assumptions. All but the independence assumption can be assessed using diagnostic plots.

How might we assess the `mod_p_planes` model? [Figure E.7](#) displays a scatterplot of residuals

versus fitted (predicted) values. As we observed in [Figure E.6](#), the number of crossings does not increase as much for warm temperatures as it does for more moderate ones. We may need to consider a more sophisticated model with a more complex model for temperature.

```
mpplot(mod_p_planes, which = 1, system = "ggplot2")
```

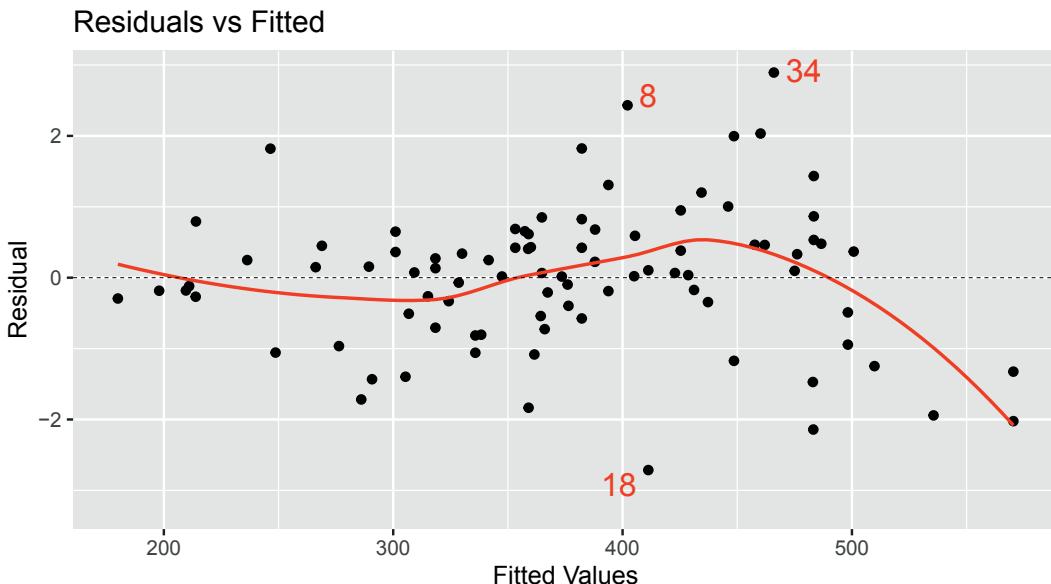


Figure E.7: Assessing linearity using a scatterplot of residuals versus fitted (predicted) values.

[Figure E.8](#) displays the quantile-quantile plot for the residuals from the regression model. The plot deviates from the straight line: This indicates that the residuals have heavier tails than a normal distribution.

```
mpplot(mod_p_planes, which = 2, system = "ggplot2")
```

[Figure E.9](#) displays the scale-location plot for the residuals from the model: The results indicate that there is evidence of heteroscedasticity (the variance of the residuals increases as a function of predicted value).

```
mpplot(mod_p_planes, which = 3, system = "ggplot2")
```

When performing model diagnostics, it is important to identify any outliers and understand their role in determining the regression coefficients.

- We call observations that don't seem to fit the general pattern of the data *outliers* [Figures E.7, E.8, and E.9](#) mark three points (8, 18, and 34) as values that large negative or positive residuals that merit further exploration.
- An observation with an extreme value of the explanatory variable is a point of high *leverage*.
- A high leverage point that exerts disproportionate influence on the slope of the regression line is an *influential point*.

[Figure E.10](#) displays the values for *Cook's distance* (a common measure of influential points in a regression model).

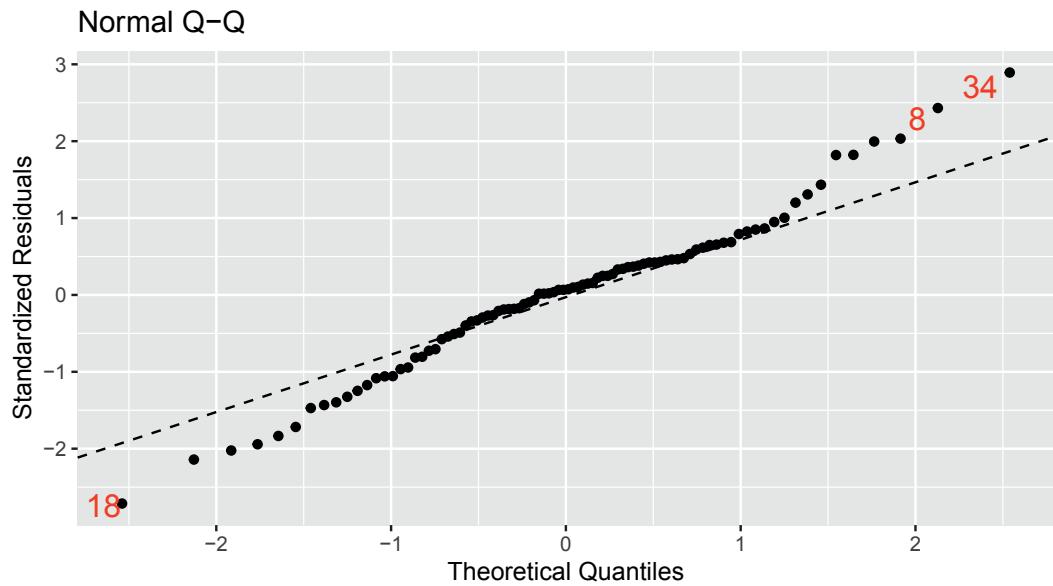


Figure E.8: Assessing normality assumption using a Q–Q plot.

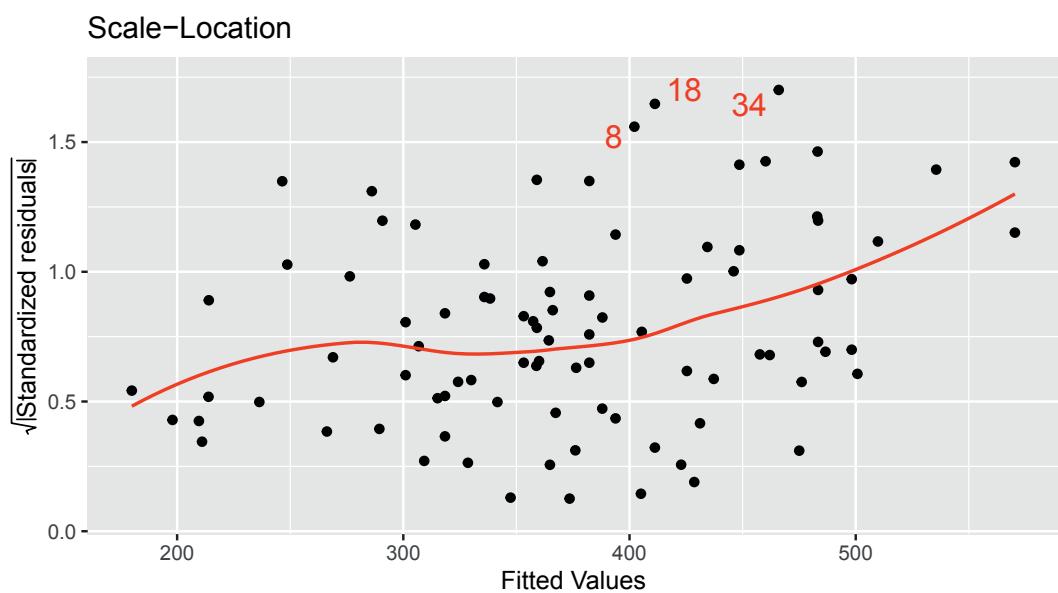


Figure E.9: Assessing equal variance using a scale–location plot.

```
mplot(mod_p_planes, which = 4, system = "ggplot2")
```

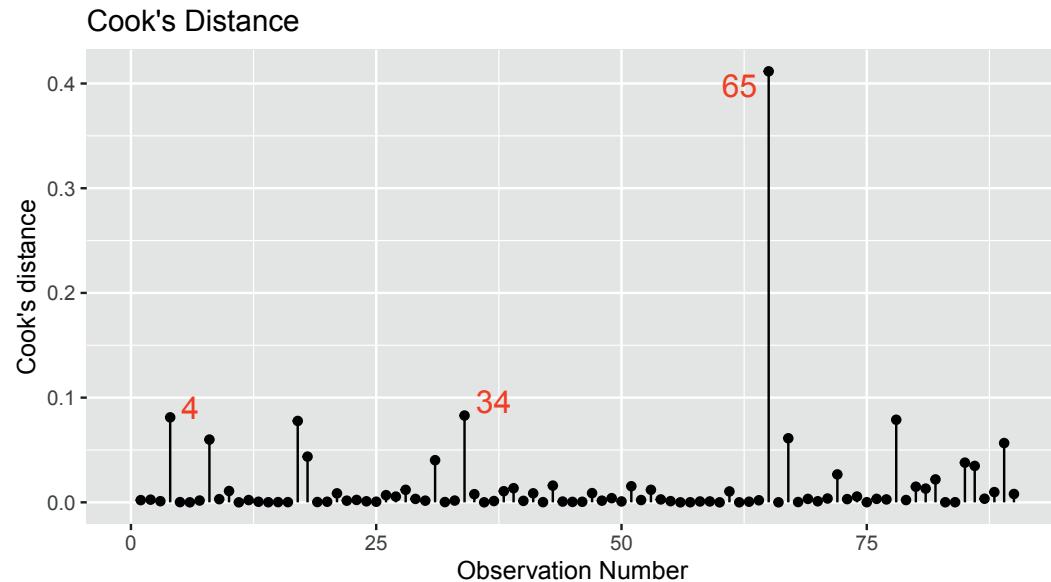


Figure E.10: Cook's distance for rail trail model.

We can use the `augment()` function from the **broom** package to calculate the value of this statistic and identify the most extreme Cook's distance.

```
library(broom)
augment(mod_p_planes) %>%
  mutate(row_num = row_number()) %>%
  select(-std.resid, -sigma) %>%
  filter(.cooksdi > 0.4)

# A tibble: 1 x 9
  volume hightemp precip weekday .fitted .resid .hat .cooksdi row_num
    <int>     <int>   <dbl>    <lgsl>     <dbl>   <dbl> <dbl>    <dbl>    <int>
1     388        84     1.49    TRUE      246.    142.  0.332   0.412      65
```

Observation 65 has the highest Cook's distance. It references a day with nearly one and a half inches of rain (the most recorded in the dataset) and a high temperature of 84 degrees. This data point has high leverage and is influential on the results. Observations 4 and 34 also have relatively high Cook's distances and may merit further exploration.

E.5 Logistic regression

Our previous examples had quantitative (or continuous) outcomes. What happens when we are interested in modeling a dichotomous outcome? For example, we might model the probability of developing diabetes as a function of age and BMI (we explored this question further in [Chapter 11](#)). [Figure E.11](#) displays the scatterplot of diabetes status as a function of age, while [Figure E.12](#) displays the scatterplot of diabetes as a function of BMI (body

mass index). Note that each subject can either have diabetes or not, so all of the points are displayed at 0 or 1 on the y -axis.

```
NHANES <- NHANES %>%
  mutate(has_diabetes = as.numeric(Diabetes == "Yes"))

log_plot <- ggplot(data = NHANES, aes(x = Age, y = has_diabetes)) +
  geom_jitter(alpha = 0.1, height = 0.05) +
  geom_smooth(method = "glm", method.args = list(family = "binomial")) +
  ylab("Diabetes status") +
  xlab("Age (in years)")

log_plot
```

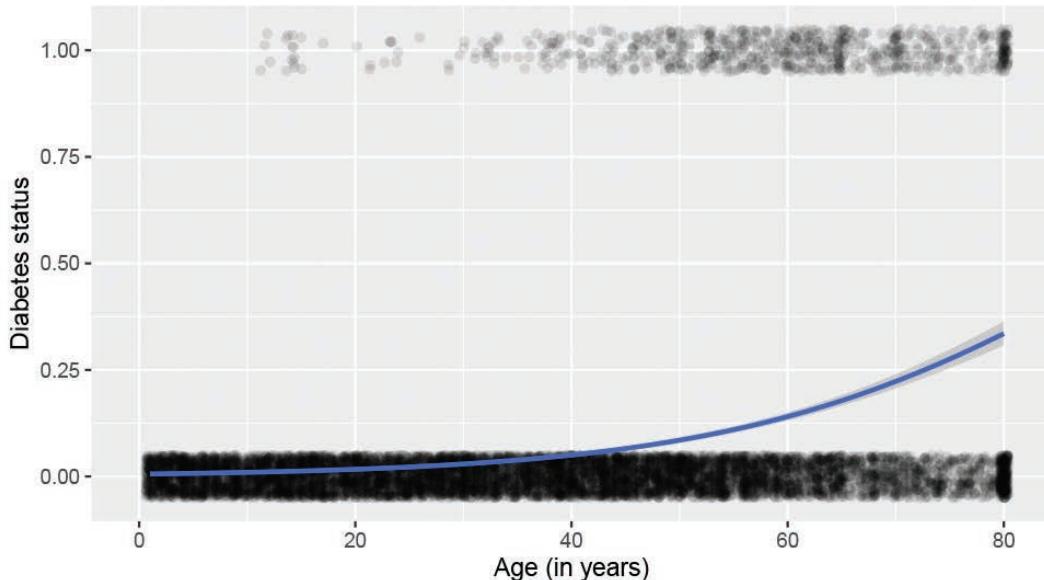


Figure E.11: Scatterplot of diabetes as a function of age with superimposed smoother.

```
log_plot + aes(x = BMI) + xlab("BMI (body mass index)")
```

We see that the probability that a subject has diabetes tends to increase as both a function of age and of BMI.

Which variable is more important: Age or BMI? We can use a multiple logistic regression model to model the probability of diabetes as a function of both predictors.

```
logreg <- glm(has_diabetes ~ BMI + Age, family = "binomial", data = NHANES)
tidy(logreg)
```

| term | estimate | std.error | statistic | p.value |
|-------------|----------|-----------|-----------|-----------|
| (Intercept) | -8.08 | 0.244 | -33.1 | 1.30e-239 |
| BMI | 0.0943 | 0.00552 | 17.1 | 1.74e-65 |
| Age | 0.0573 | 0.00249 | 23.0 | 2.28e-117 |

The answer is that both predictors seem to be important (since both p-values are very small).

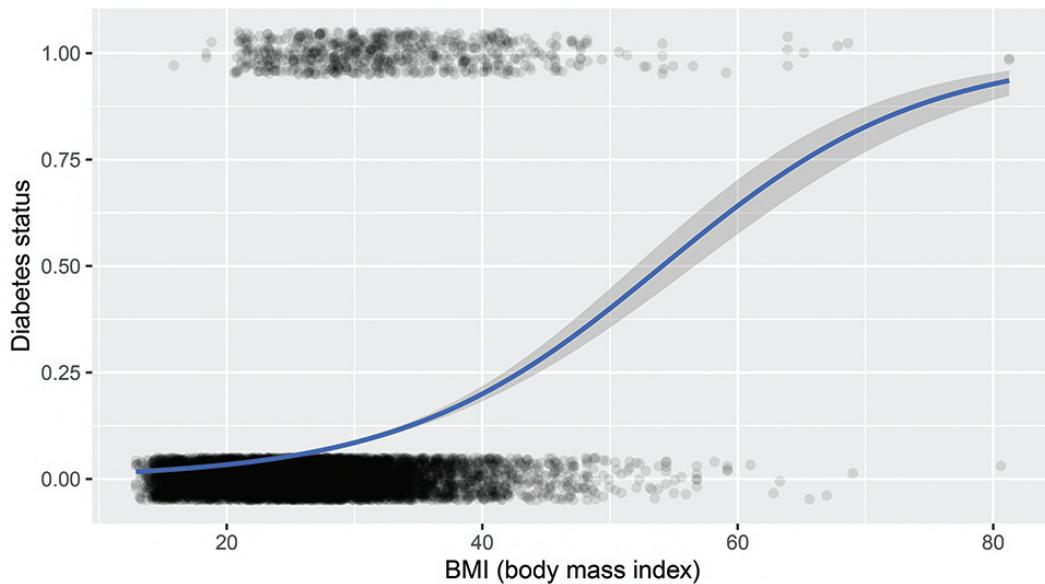


Figure E.12: Scatterplot of diabetes as a function of BMI with superimposed smoother.

To interpret the findings, we might consider a visual display of predicted probabilities as displayed in [Figure E.13](#) (compare with [Figure 11.11](#)).

```
library(modelr)
fake_grid <- data_grid(
  NHANES,
  Age = seq_range(Age, 100),
  BMI = seq_range(BMI, 100)
)
y_hats <- fake_grid %>%
  mutate(y_hat = predict(logreg, newdata = ., type = "response"))
head(y_hats, 1)

# A tibble: 1 x 3
  Age    BMI   y_hat
  <dbl> <dbl>   <dbl>
1     0    12.9 0.00104
```

The predicted probability from the model is given by:

$$\pi_i = \text{logit}(P(y_i = 1)) = \frac{e^{\beta_0 + \beta_1 \text{Age}_i + \beta_2 \text{BMI}_i}}{1 + e^{\beta_0 + \beta_1 \text{Age}_i + \beta_2 \text{BMI}_i}} \quad \text{for } i = 1, \dots, n.$$

Let's consider a hypothetical 0 year old with a BMI of 12.9 (corresponding to the first entry in the `y_hats` datafram). Their predicted probability of having diabetes would be calculated as a function of the regression coefficients.

```
linear_component <- c(1, 12.9, 0) %*% coef(logreg)
exp(linear_component) / (1 + exp(linear_component))
```

```
[,1]
[1,] 0.00104
```

The predicted probability is very small: about 1/10th of 1%.

But what about a 60 year old with a BMI of 25?

```
linear_component <- c(1, 25, 60) %*% coef(logreg)
exp(linear_component) / (1 + exp(linear_component))
```

```
[,1]
[1,] 0.0923
```

The predicted probability is now 9.2%.

```
ggplot(data = NHANES, aes(x = Age, y = BMI)) +
  geom_tile(data = y_hats, aes(fill = y_hat), color = NA) +
  geom_count(aes(color = factor(has_diabetes)), alpha = 0.8) +
  scale_fill_gradient(low = "white", high = "red") +
  scale_color_manual("Diabetes", values = c("gold", "black")) +
  scale_size(range = c(0, 2)) +
  labs(fill = "Predicted\nprobability")
```

Figure E.13 displays the predicted probabilities for each of our grid points. We see that very few young adults have diabetes, even if they have moderately high BMI scores. As we look at older subjects while holding BMI fixed, the predicted probability of diabetes increases.

E.6 Further resources

Regression is described in many books. An introduction is found in most introductory statistics textbooks, including *Open Intro Statistics* (Diez et al., 2019). For a deeper but still accessible treatment, we suggest Cannon et al. (2019). James et al. (2013) and Hastie et al. (2009) also cover regression from a modeling and machine learning perspective. Hoaglin (2016) provides guidance on how to interpret conditional regression parameters. Cook (1982) comprehensively reviews regression diagnostics. An accessible introduction to smoothing can be found in Ruppert et al. (2003).

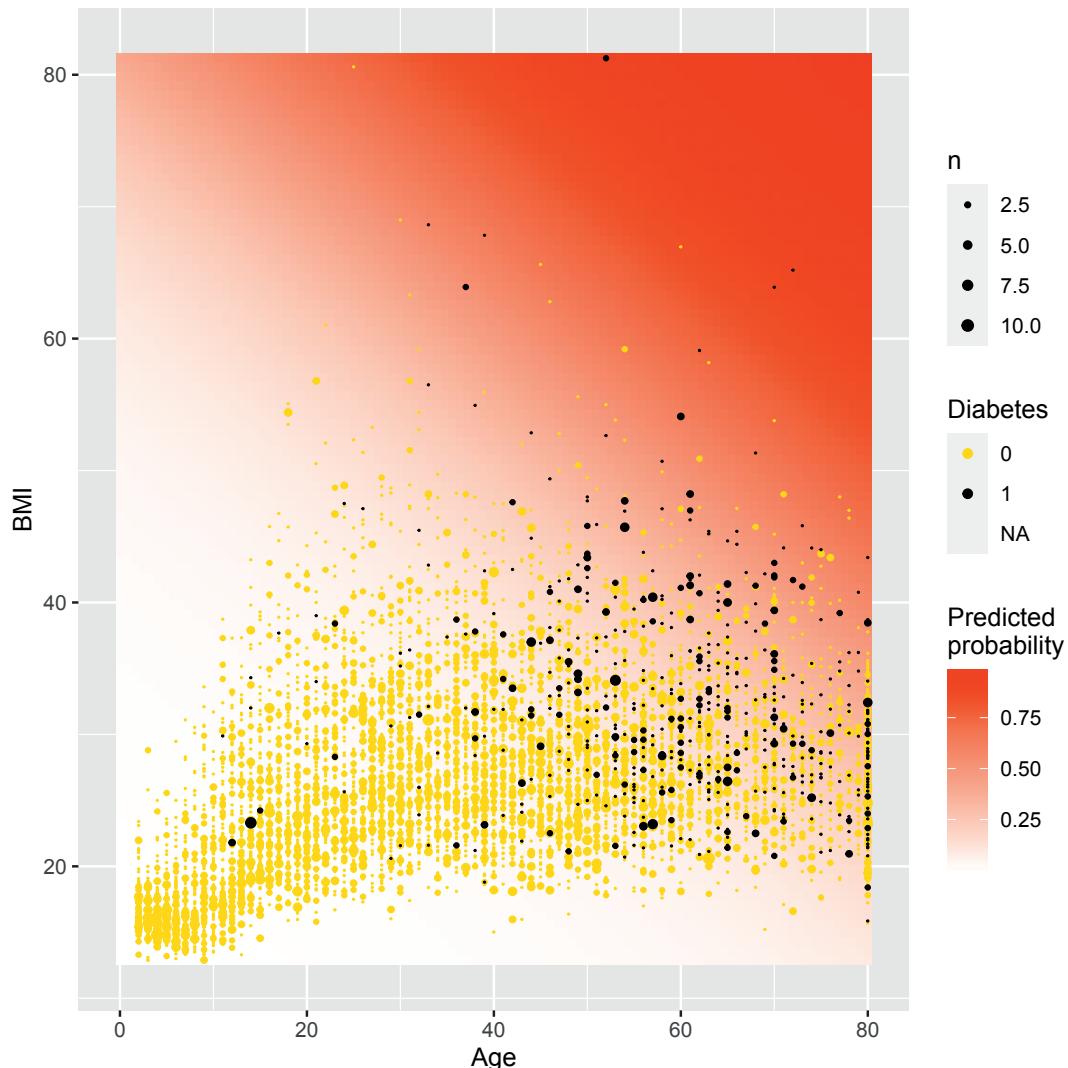


Figure E.13: Predicted probabilities for diabetes as a function of BMI and age.

E.7 Exercises

Problem 1 (Easy): In 1966, Cyril Burt published a paper called *The genetic determination of differences in intelligence: A study of monozygotic twins reared apart*. The data consist of IQ scores for [an assumed random sample of] 27 identical twins, one raised by foster parents, the other by the biological parents.

Here is the regression output for using `Biological` IQ to predict `Foster` IQ:

```
library(mdsr)
library(faraway)
mod <- lm(Foster ~ Biological, data = twins)
coef(mod)

(Intercept)  Biological
         9.208        0.901

rsquared(mod)
```

```
[1] 0.778
```

Which of the following is **FALSE**? Justify your answers.

- Alice and Beth were raised by their biological parents. If Beth's IQ is 10 points higher than Alice's, then we would expect that her foster twin Bernice's IQ is 9 points higher than the IQ of Alice's foster twin Ashley.
- Roughly 78% of the foster twins' IQs can be accurately predicted by the model.
- The linear model is $\widehat{Foster} = 9.2 + 0.9 \times Biological$.
- Foster twins with IQs higher than average are expected to have biological twins with higher than average IQs as well.

Problem 2 (Medium): The `atus` package includes data from the American Time Use Survey (ATUS). Use the `atusresp` dataset to model `hourly_wage` as a function of other predictors in the dataset.

Problem 3 (Medium): The `Gestation` data set in `mdsr` contains birth weight, date, and gestational period collected as part of the Child Health and Development Studies. Information about the baby's parents—age, education, height, weight, and whether the mother smoked is also recorded.

- a. Fit a linear regression model for `birthweight` (`wt`) as a function of the mother's age (`age`).
- b. Find a 95% confidence interval and p-value for the slope coefficient.
- c. What do you conclude about the association between a mother's age and her baby's birthweight?

Problem 4 (Medium): The Child Health and Development Studies investigate a range of topics. One study, in particular, considered all pregnancies among women in the Kaiser Foundation Health Plan in the San Francisco East Bay area. The goal is to model the weight of the infants (`bwt`, in ounces) using variables including length of pregnancy in days (`gestation`), mother's age in years (`age`), mother's height in inches (`height`), whether the child was the first born (`parity`), mother's pregnancy weight in pounds (`weight`), and whether the mother was a smoker (`smoke`). The summary table that follows shows the results

of a regression model for predicting the average birth weight of babies based on all of the variables included in the data set.

```
library(mdsr)
library(mosaicData)
babies <- Gestation %>%
  rename(bwt = wt, height = ht, weight = wt.1) %>%
  mutate(parity = parity == 0, smoke = smoke > 0) %>%
  select(id, bwt, gestation, age, height, weight, parity, smoke)
mod <- lm(bwt ~ gestation + age + height + weight + parity + smoke,
          data = babies
)
coef(mod)
```

| | (Intercept) | gestation | age | height | weight | parityTRUE |
|-----------|-------------|-----------|--------|--------|--------|------------|
| smokeTRUE | -85.7875 | 0.4601 | 0.0429 | 1.0623 | 0.0653 | -2.9530 |
| NA | | | | | | |

Answer the following questions regarding this linear regression model.

- The coefficient for `parity` is different than if you fit a linear model predicting weight using only that variable. Why might there be a difference?
- Calculate the residual for the first observation in the data set.
- This data set contains missing values. What happens to these rows when we fit the model?

Problem 5 (Medium): Investigators in the HELP (Health Evaluation and Linkage to Primary Care) study were interested in modeling predictors of being `homeless` (one or more nights spent on the street or in a shelter in the past six months vs. housed) using baseline data from the clinical trial. Fit and interpret a parsimonious model that would help the investigators identify predictors of homelessness.

E.8 Supplementary exercises

Available at <https://mdsr-book.github.io/mdsr2e/ch-regression.html#regression-online-exercises>

F

Setting up a database server

Setting up a local or remote database server is neither trivial nor difficult. In this chapter, we provide instructions as to how to set up a local database server on a computer that you control. While everything that is done in this chapter can be accomplished on any modern operating system, many tools for data science are designed for *UNIX-like* operating systems, and can be a challenge to set up on *Windows*. This is no exception. In particular, comfort with the command line is a plus and the material presented here will make use of *UNIX shell* commands. On *Mac OS X* and other Unix-like operating systems (e.g., *Ubuntu*), the command line is accessible using a Terminal application. On Windows, some of these shell commands might work at a DOS prompt, but others will not.¹ Unfortunately, providing Windows-specific setup instructions is outside the scope of this book.

Three open-source SQL database systems are most commonly encountered. These include SQLite, MySQL, and PostgreSQL. While MySQL and PostgreSQL are full-featured relational database systems that employ a strict client-server model, SQLite is a lightweight program that runs only locally and requires no initial configuration. However, while SQLite is certainly the easiest system to set up, it has far fewer functions, lacks a caching mechanism, and is not likely to perform as well under heavy usage. Please see the official documentation for appropriate uses of SQLite for assistance with choosing the right SQL implementation for your needs.

Both MySQL and PostgreSQL employ a *client-server* architecture. That is, there is a server program running on a computer somewhere, and you can connect to that server from any number of client programs—from either that same machine or over the internet. Still, even if you are running MySQL or PostgreSQL on your local machine, there are always two parts: the client and the server. This chapter provides instructions for setting up the server on a machine that you control—which for most analysts, is your local machine.

F.1 SQLite

For SQLite, there is nothing to configure, but it must be installed. The **RSQLite** package embeds the database engine and provides a straightforward interface. On Linux systems, `sqlite` is likely already installed, but the source code, as well as pre-built binaries for Mac OS X and Windows, is available at SQLite.org.

¹Note that *Cygwin* provides a Unix-like shell for Windows.

F.2 MySQL

We will focus on the use of MySQL (with brief mention of PostgreSQL in the next section). The steps necessary to install a PostgreSQL server will follow similar logic, but the syntax will be importantly different.

F.2.1 Installation

If you are running Mac OS X or a Linux-based operating system, then you probably already have a MySQL server installed and running on your machine. You can check to see if this is the case by running the following from your operating system's shell (i.e., the command line or using the “Terminal” application).

```
ps aux | grep "mysql"
```

```
bbaumer@bbaumer-Precision-Tower-7810:~$ ps aux | grep "mysql"  
mysql      1269  0.9 2092204 306204 ?        Ssl May20  5:22 /usr/sbin/mysqld
```

If you see anything like the first line of this output (i.e., containing `mysqld`), then MySQL is already running. (If you don't see anything like that, then it is not.)

If MySQL is not installed, then you can install it by downloading the relevant version of the *MySQL Community Server* for your operating system at dev.mysql.com. If you run into trouble, please consult the installation instructions.

For Mac OS X, there are more specific instructions available. After installation, you will need to install the Preference Pane and start the server.

It is also helpful to add the `mysql` binary directory to your *PATH environment variable*, so you can launch `mysql` easily from the shell. To do this, execute the following command in your shell:

```
export PATH=$PATH:/usr/local/mysql/bin  
echo $PATH
```

You may have to modify the path to the `mysql/bin` directory to suit your local setup. If you don't know where the directory is, you can try to find it using the `which` program provided by your operating system.

```
which mysql
```

```
/usr/local/mysql/bin
```

F.2.2 Access

In most cases, the installation process will result in a server process being launched on your machine, such as the one that we saw above in the output of the `ps` command. Once the server is running, we need to configure it properly for our use. The full instructions for post-installation provide great detail on this process. However, in our case, we will mostly stick with the default configuration, so there are only a few things to check.

The most important thing is to gain access to the server. MySQL maintains a set of user accounts just like your operating system. After installation, there is usually only one account created: `root`. In order to create other accounts, we need to log into MySQL as `root`. Please

read the documentation on *Securing the Initial MySQL Accounts* for your setup. From that documentation:

Some accounts have the user name `root`. These are superuser accounts that have all privileges and can do anything. If these root accounts have empty passwords, anyone can connect to the MySQL server as root without a password and be granted all privileges.

If this is your first time accessing MySQL, typing this into your shell might work:

```
mysql -u root
```

If you see an `Access denied` error, it means that the `root` MySQL user has a password, but you did not supply it. You may have created a password during installation. If you did, try:

```
mysql -u root -p
```

and then enter that password (it may well be blank). If you don't know the `root` password, try a few things that might be the password. If you can't figure it out, contact your system administrator or re-install MySQL.

You might—on Windows especially—get an error that says something about “command not found.” This means that the program `mysql` is not accessible from your shell. You have two options: 1) you can specify the full path to the MySQL application; or 2) you can append your `PATH` variable to include the directory where the MySQL application is. The second option is preferred and is illustrated above.

On Linux or Mac OS X, it is probably in `/usr/bin/` or `/usr/local/mysql/bin` or something similar, and on Windows, it is probably in `\Applications\MySQL Server 5.6\bin` or something similar. Once you find the path to the application and the password, you should be able to log in. You will know when it works if you see a `mysql` prompt instead of your usual one.

```
bbaumer@bbaumer-Precision-Tower-7810:~$ mysql -u root -p
```

Enter password:

```
Welcome to the MySQL monitor. Commands end with ; or \g.
```

```
Your MySQL connection id is 47
```

```
Server version: 5.7.31-0ubuntu0.18.04.1 (Ubuntu)
```

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Type `'help;'` or `'\h'` for help. Type `'\c'` to clear the current input statement.

```
mysql>
```

Once you are logged into MySQL, try running the following command at the `mysql>` prompt (do not forget the trailing semi-colon):².

```
SELECT User, Host, Password FROM mysql.user;
```

This command will list the users on the MySQL server, their encrypted passwords, and the hosts from which they are allowed to connect. Next, if you want to change the root password, set it to something else (in this example `mypass`).

```
UPDATE mysql.user SET Password = PASSWORD('mypass') WHERE User = 'root';
FLUSH PRIVILEGES;
```

The most responsible thing to do now is to create a new account for yourself. You should probably choose a different password than the one for the `root` user. Do this by running:

```
CREATE USER 'r-user'@'localhost' IDENTIFIED BY 'mypass';
```

It is important to understand that MySQL's concept of users is really a `{user, host}` pair. That is, the user `'bbaumer'@'localhost'` can have a different password and set of privileges than the user `'bbaumer'@'%'`. The former is only allowed to connect to the server from the machine on which the server is running. (For most of you, that is your computer.) The latter can connect from anywhere ('%' is a wildcard character). Obviously, the former is more secure. Use the latter only if you want to connect to your MySQL database from elsewhere.

You will also want to make yourself a superuser.

```
GRANT ALL PRIVILEGES ON *.* TO 'r-user'@'localhost' WITH GRANT OPTION;
```

Again, flush the privileges to reload the tables.

```
FLUSH PRIVILEGES;
```

Finally, log out by typing `quit`. You should now be able to log in to MySQL as yourself by typing the following into your shell:

```
mysql -u r-user -p
```

F.2.2.1 Using an option file

A relatively safe and convenient method of connecting to MySQL servers (whether local or remote) is by using an option file. This is a simple text file located at `~/.my.cnf` that may contain various connection parameters. Your entire file might look like this:

```
[client]
user=r-user
password="mypass"
```

These options will be read by MySQL automatically anytime you connect from a client program. Thus, instead of having to type:

```
mysql -u yourusername -p
```

you should be automatically logged on with just `mysql`. Moreover, you can have `dplyr` read your MySQL option file using the `default.file` argument (see [Section F.4.3.1](#)).

²NB: As of version 5.7, the `mysql.user` table include the field `authentication_string` instead of `password`.

F.2.3 Running scripts from the command line

MySQL will run SQL scripts contained in a file via the command line client. If the file `myscript.sql` is a text file containing MySQL commands, you can run it using the following command from your shell:

```
mysql -u yourusername -p dbname < myscript.sql
```

The result of each command in that script will be displayed in the terminal. Please see [Section 16.3](#) for an example of this process in action.

F.3 PostgreSQL

Setting up a PostgreSQL server is logically analogous to the procedure demonstrated above for MySQL. The default user in a PostgreSQL installation is `postgres` and the default password is either `postgres` or blank. Either way, you can log into the PostgreSQL command line client—which is called `psql`—using the `sudo` command in your shell.

```
sudo -u postgres psql
```

This means: “Launch the `psql` program as if I was the user `postgres`.” If this is successful, then you can create a new account for yourself from inside PostgreSQL. Here again, the procedure is similar to the procedure demonstrated above for MySQL in section F.2.2.

You can list all of the PostgreSQL users by typing at your `postgres` prompt:

```
\du
```

You can change the password for the `postgres` user:

```
ALTER USER postgres PASSWORD 'some_pass';
```

Create a new account for yourself:

```
CREATE USER yourusername SUPERUSER CREATEDB PASSWORD 'some_pass';
```

Create a new database called `airlines`:

```
CREATE DATABASE airlines;
```

Quit the `psql` client by typing:

```
\q
```

Now that your user account is created, you can log out and back in with the shell command:

```
psql -U yourusername -W
```

If this doesn’t work, it is probably because the client authentication is set to `ident` instead of `md5`. Please see the documentation on client authentication for instructions on how to correct this on your installation, or simply continue to use the `sudo` method described above.

F.4 Connecting to SQL

There are many different options for connecting to and retrieving data from an SQL server. In all cases, you will need to specify at least four pieces of information:

- **host**: the name of the SQL server. If you are running this server locally, that name is `localhost`
- **dbname**: the name of the database on that server to which you want to connect (e.g., `airlines`)
- **user**: your username on the SQL server
- **password**: your password on the SQL server

F.4.1 The command line client

From the command line, the syntax is:

```
mysql -u username -p -h localhost dbname
```

After entering your password, this will bring you to an interactive MySQL session, where you can bounce queries directly off of the server and see the results in your terminal. This is often useful for debugging because you can see the error messages directly, and you have the full suite of MySQL directives at your disposal. On the other hand, it is a fairly cumbersome route to database development, since you are limited to text-editing capabilities of the command line.

Command-line access to PostgreSQL is provided via the `psql` program described above.

F.4.2 GUIs

The *MySQL Workbench* is a graphical user interface (GUI) that can be useful for configuration and development. This software is available on Windows, Linux, and Mac OS X. The analogous tool for PostgreSQL is *pgAdmin*, and it is similarly cross-platform. *sqlitebrowser* is another cross-platform GUI for SQLite databases.

These programs provide full-featured access to the underlying database system, with many helpful and easy-to-learn drop-down menus. We recommend developing queries and databases in these programs, especially when learning SQL.

F.4.3 R and RStudio

The downside to the previous approaches is that you don't actually capture the data returned by your queries, so you can't do anything with them. Using the GUIs, you can of course save the results of any query to a CSV. But a more elegant solution is to pull the data directly into **R**. This functionality is provided by the **RMySQL**, **RPostgreSQL**, and **RSQLite** packages. The **DBI** package provides a common interface to all three of the SQL back-ends listed above, and the **dbplyr** package provides a slicker interface to **DBI**. A schematic of these dependencies is displayed in [Figure F.1](#). We recommend using either the **dplyr** or the **DBI** interfaces whenever possible, since they are implementation agnostic.

For most purposes (e.g., `SELECT` queries) there may be significant convenience to using the **dplyr** interface. However, the functionality of this construction is limited to `SELECT` queries.

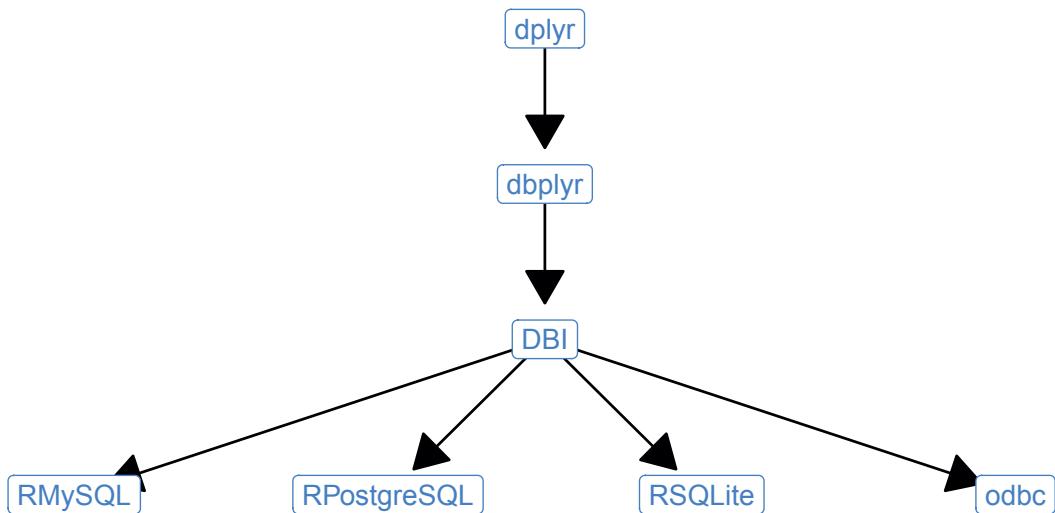


Figure F.1: Schematic of SQL-related R packages and their dependencies.

Thus, other SQL directives (e.g., EXPLAIN, INSERT, UPDATE, etc.) will not work in the **dplyr** construction. This functionality must be accessed using **DBI**.

In what follows, we illustrate how to connect to a MySQL backend using **dplyr** and **DBI**. However, the instructions for connecting to a PostgreSQL and SQLite are analogous. First, you will need to load the relevant package.

```
library(RMySQL)
```

F.4.3.1 Using dplyr and `tbl()`

To set up a connection to a MySQL database using **dplyr**, we must specify the four parameters outlined above, and save the resulting object using the `dbConnect()` function.

```
library(dplyr)
db <- dbConnect(
  RMySQL::MySQL(),
  dbname = "airlines", host = "localhost",
  user = "r-user", password = "mypass"
)
```

If you have a MySQL option file already set up (see [Section F.2.2.1](#)), then you can alternatively connect using the `default.file` argument. This enables you to connect without having to type your password, or save it in plaintext in your R scripts.

```
db <- dbConnect(
  RMySQL::MySQL(),
  dbname = "airlines", host = "localhost",
  default.file = "~/.my.cnf"
)
```

Next, we can retrieve data using the `tbl()` function and the `sql()` command.

```
res <- tbl(db, sql("SELECT faa, name FROM airports"))
res
```

```
# Source: SQL [?? x 2]
# Database: mysql 5.6.40-log
# [mdsr_public@mdsr.cdc7tgkkqd0n.us-east-1.rds.amazonaws.com:/airlines]
faa    name
<chr> <chr>
1 04G  Lansdowne Airport
2 06A  Moton Field Municipal Airport
3 06C  Schaumburg Regional
4 06N  Randall Airport
5 09J  Jekyll Island Airport
6 0A9  Elizabethton Municipal Airport
7 0G6  Williams County Airport
8 0G7  Finger Lakes Regional Airport
9 0P2  Shoestring Aviation Airfield
10 0S9  Jefferson County Intl
# ... with more rows
```

Note that the resulting object has class `tbl_sql`.

```
class(res)
```

```
[1] "tbl_SQLConnection" "tbl_dbi"           "tbl_sql"
[4] "tbl_lazy"          "tbl"
```

Note also that the derived table is described as having an unknown number of rows (indicated by `??`). This is because `dplyr` is smart (and lazy) about evaluation. It hasn't actually pulled all of the data into **R**. To force it to do so, use `collect()`.

```
collect(res)
```

```
# A tibble: 1,458 x 2
faa    name
<chr> <chr>
1 04G  Lansdowne Airport
2 06A  Moton Field Municipal Airport
3 06C  Schaumburg Regional
4 06N  Randall Airport
5 09J  Jekyll Island Airport
6 0A9  Elizabethton Municipal Airport
7 0G6  Williams County Airport
8 0G7  Finger Lakes Regional Airport
9 0P2  Shoestring Aviation Airfield
10 0S9  Jefferson County Intl
# ... with 1,448 more rows
```

F.4.3.2 Writing SQL queries

In [Section F.4.3.1](#), we used the `tbl()` function to interact with a table stored in a SQL database. This enabled us to use `dplyr` functions directly, without having to write any SQL queries.

In this section, we use the `dbGetQuery()` function from the **DBI** package to send an SQL command to the server and retrieve the results.

```
dbGetQuery(db, "SELECT faa, name FROM airports LIMIT 0,5")
```

| | faa | name |
|---|-----|-------------------------------|
| 1 | 04G | Lansdowne Airport |
| 2 | 06A | Moton Field Municipal Airport |
| 3 | 06C | Schaumburg Regional |
| 4 | 06N | Randall Airport |
| 5 | 09J | Jekyll Island Airport |

Unlike the `tbl()` function from `dplyr`, `dbGetQuery()` can execute arbitrary SQL commands, not just `SELECT` statements. So we can also run `EXPLAIN`, `DESCRIBE`, and `SHOW` commands.

```
dbGetQuery(db, "EXPLAIN SELECT faa, name FROM airports")
```

| | id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
|---|----|-------------|----------|------|---------------|------|---------|------|------|-----------|
| 1 | 1 | SIMPLE | airports | ALL | | <NA> | <NA> | <NA> | <NA> | 1458 <NA> |

```
dbGetQuery(db, "DESCRIBE airports")
```

| | Field | Type | Null | Key | Default | Extra |
|---|---------|---------------|------|-----|---------|-------|
| 1 | faa | varchar(3) | NO | PRI | | |
| 2 | name | varchar(255) | YES | | <NA> | |
| 3 | lat | decimal(10,7) | YES | | <NA> | |
| 4 | lon | decimal(10,7) | YES | | <NA> | |
| 5 | alt | int(11) | YES | | <NA> | |
| 6 | tz | smallint(4) | YES | | <NA> | |
| 7 | dst | char(1) | YES | | <NA> | |
| 8 | city | varchar(255) | YES | | <NA> | |
| 9 | country | varchar(255) | YES | | <NA> | |

```
dbGetQuery(db, "SHOW DATABASES")
```

| | Database |
|---|--------------------|
| 1 | information_schema |
| 2 | airlines |
| 3 | fec |
| 4 | imdb |
| 5 | lahman |
| 6 | nyctaxi |

F.4.4 Load into SQLite database

A process similar to the one we exhibit in [Section 16.3](#) can be used to create a SQLite database, although in this case it is not even necessary to specify the table schema in advance. Launch `sqlite3` from the command line using the shell command:

```
sqlite3
```

Create a new database called `babynames` in the current directory using the `.open` command:

```
.open babynamesdata.sqlite3
```

Next, set the `.mode` to `csv`, import the two tables, and exit.

```
.mode csv
.import babynames.csv babynames
```

```
.import births.csv births
.exit
```

This should result in an SQLite database file called `babynamesdata.sqlite3` existing in the current directory that contains two tables. We can connect to this database and query it using `dplyr`.

```
db <- dbConnect(RSQLite::SQLite(), "babynamesdata.sqlite3")
babynames <- tbl(db, "babynames")
babynames %>%
  filter(name == "Benjamin")

# Source:  lazy query [?? x 5]
# Database: sqlite 3.30.1
#   [/home/bbaumer/Dropbox/git/mdsr2e/babynamesdata.sqlite3]
#   year  sex    name    n      prop
#   <chr> <chr> <chr> <chr> <chr>
# 1 1976   F    Benjamin 53  3.37186805943904e-05
# 2 1976   M    Benjamin 10680 0.0065391571834601
# 3 1977   F    Benjamin 63  3.83028784917178e-05
# 4 1977   M    Benjamin 12112 0.00708409319279004
# 5 1978   F    Benjamin 73  4.44137806835342e-05
# 6 1978   M    Benjamin 11411 0.00667764880752091
# 7 1979   F    Benjamin 79  4.58511127310548e-05
# 8 1979   M    Benjamin 12516 0.00698620342042644
# 9 1980   F    Benjamin 80  4.49415983928884e-05
# 10 1980  M    Benjamin 13630 0.00734980487697031
# ... with more rows
```

Alternatively, the **RSQLite** package includes a vignette describing how to set up a database from within R.

Bibliography

- Aden-Buie, G. (2020). *xaringantherem: Custom xaringan CSS Themes*. R package version 0.3.0.
- Alberani, D. (2014). Imdbpy. <http://imdbpy.sourceforge.net>.
- Albert, J. (2003). *Teaching statistics using baseball*. Mathematical Association of America: Washington, DC.
- Albert, J. and Hu, J. (2019). *Probability and Bayesian Modeling*. CRC Press.
- Albert, J., Marchi, M., and Baumer, B. S. (2018). *Analyzing Baseball Data with R*. CRC Press: Boca Raton, FL, 2nd edition.
- Albert, R. and Barabási, A.-L. (2002). Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47.
- Allaire, J. (2019). *rsconnect: Deployment Interface for R Markdown Documents and Shiny Applications*. R package version 0.8.16.
- Allaire, J., Cheng, J., Xie, Y., McPherson, J., Chang, W., Allen, J., Wickham, H., Atkins, A., and Hyndman, R. (2020a). *rmarkdown: Dynamic Documents for R*. R package version 2.3.
- Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W., and Iannone, R. (2020b). *rmarkdown: Dynamic Documents for R*. R package version 2.6.
- Allaire, J. J., Horner, J., Marti, V., and Porte, N. (2014). *markdown: Markdown rendering for R*. R package version 0.7.4.
- American Statistical Association Undergraduate Guidelines Workgroup (2014). *2014 Curriculum Guidelines for Undergraduate Programs in Statistical Science*. <http://www.amstat.org/education/curriculumguidelines.cfm>.
- Angwin, J., Larson, J., Mattu, S., and Kirchner, L. (2016). Machine bias. ProPublica. <https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing>.
- Arnold, J. B. (2019a). *ggthemes: Extra Themes, Scales and Geoms for ggplot2*. R package version 4.2.0.
- Arnold, J. B. (2019b). *ggthemes: Extra Themes, Scales and Geoms for ggplot2*. R package version 4.2.0.
- Ball, R. and Medeiros, N. (2012). Teaching integrity in empirical research: A protocol for documenting data management and analysis. *The Journal of Economic Education*, 43(2):182–189.

- Barabási, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286(5439):509–512.
- Barabási, A.-L. and Frangos, J. (2014). *Linked: The New Science of Networks*. Basic Books: New York.
- Basu, P., Baumer, B. S., Bar-Noy, A., Chau, C.-K., and City, M. (2015). Social-communication composite networks. In Wu, J. and Wang, Y., editors, *Opportunistic Mobile Social Networks*, chapter 1, pages 1–36. CRC Press: Boca Raton.
- Baumer, B. (2015a). In a Moneyball world, a number of teams remain slow to buy into sabermetrics. In Webb, R., editor, *The Great Analytics Rankings*. ESPN.com. http://espn.go.com/espn/feature/story/_/id/12331388/the-great-analytics-rankings#!mlb.
- Baumer, B. (2020). *etl: Extract-Transform-Load Framework for Medium Data*. R package version 0.3.9.
- Baumer, B., Basu, P., and Bar-Noy, A. (2011). Modeling and analysis of composite network embeddings. In Helmy, A., Landfeldt, B., and Bononi, L., editors, *MSWiM*, pages 341–350. ACM.
- Baumer, B. S. (2015b). A data science course for undergraduates: Thinking with data. *The American Statistician*, 69(4):334–342.
- Baumer, B. S., Çetinkaya Rundel, M., Bray, A., Loi, L., and Horton, N. J. (2014). R mark-down: Integrating a reproducible analysis tool into introductory statistics. *Technology Innovations in Statistics Education*, 8(1).
- Baumer, B. S. and Gjekmarkaj, E. (2017). *fec: Campaign finance for Federal Elections*. R package version 0.0.0.9015.
- Baumer, B. S., Goueth, R., Li, W., Zhang, W., and Horton, N. (2020). *macleish: Retrieve Data from MacLeish Field Station*. R package version 0.3.6.
- Baumer, B. S., Horton, N., and Kaplan, D. (2021). *mdsr: Complement to Modern Data Science with R*. R package version 0.2.4.
- Baumer, B. S., Rabanca, G., Bar-Noy, A., and Basu, P. (2015). Star search: Effective subgroups in collaborative social networks. In Pei, J., Silvestri, F., and Tang, J., editors, *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2015, Paris, France, August 25–28, 2015*, pages 729–736. ACM.
- Baumer, B. S., Wei, Y., and Bloom, G. S. (2016). The smallest non-autograph. *Discussiones Mathematicae Graph Theory*, 36(3):577–602.
- Baumer, B. S. and Zimbalist, A. (2014). *The Sabermetric Revolution: Assessing the Growth of Analytics in Baseball*. University of Pennsylvania Press: Philadelphia, PA.
- Beck, M. W. (2018). *NeuralNetTools: Visualization and Analysis Tools for Neural Networks*. R package version 1.5.2.
- Bengtsson, H. (2020). *future: Unified Parallel and Distributed Processing in R for Everyone*. R package version 1.21.0.
- Benoit, K., Muhr, D., and Watanabe, K. (2020). *stopwords: Multilingual Stopword Lists*. R package version 2.1.

- Bivand, R., Keitt, T., and Rowlingson, B. (2021). *rgdal: Bindings for the Geospatial Data Abstraction Library*. R package version 1.5-19.
- Bivand, R. S., Pebesma, E., and Gómez-Rubio, V. (2013). *Applied Spatial Data Analysis with R (second edition)*. Springer Verlag: New York, NY.
- Bogdanov, P., Baumer, B. S., Basu, P., Bar-Noy, A., and Singh, A. K. (2013). As strong as the weakest link: Mining diverse cliques in weighted graphs. In Blockeel, H., Kersting, K., Nijssen, S., and Zelezný, F., editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23–27, 2013, Proceedings, Part I*, volume 8188 of *Lecture Notes in Computer Science*, pages 525–540. Springer Verlag: New York, NY.
- Bombardier, C., Laine, L., Reicin, A., Shapiro, D., Burgos-Vargas, R., Davis, B., Day, R., Ferraz, M. B., Hawkey, C. J., Hochberg, M. C., Kvien, T. K., and Schnitzer, T. J. (2000). Comparison of upper gastrointestinal toxicity of rofecoxib and naproxen in patients with rheumatoid arthritis. *New England Journal of Medicine*, 343:1520–1528.
- Breiman, L. (2001). Statistical modeling: The two cultures. *Statistical Science*, 16(3):199–215. <http://www.jstor.org/stable/2676681>.
- Breiman, L., Cutler, A., Liaw, A., and Wiener, M. (2018). *randomForest: Breiman and Cutler's Random Forests for Classification and Regression*. R package version 4.6-14.
- Brewer, C. A. (1994). Color use guidelines for mapping and visualization. *Visualization in Modern Cartography*, 2:123–148.
- Brewer, C. A. (1999). Color use guidelines for data representation. In *Proceedings of the Section on Statistical Graphics, American Statistical Association*, pages 55–60.
- Bridgeford, L. C. (2014). Q&A: Statistical proof of discrimination isn't static. <http://www.bna.com/qa-statistical-proof-b17179891425>.
- Broman, K. W. and Woo, K. H. (2018). Data organization in spreadsheets. *The American Statistician*, 72(1):2–10.
- Brownrigg, R. (2018). *maps: Draw Geographical Maps*. R package version 3.3.0.
- Bryan, J. (2020). *googlesheets4: Access Google Sheets using the Sheets API V4*. R package version 0.2.0.
- Bryan, J., the STAT 545 TAs, and Hester, J. (2018). *Happy Git and GitHub for the useR*. GitHub.
- Butts, C. T. (2020a). *network: Classes for Relational Data*. R package version 1.16.1.
- Butts, C. T. (2020b). *sna: Tools for Social Network Analysis*. R package version 2.6.
- Cambon, J. (2020). *tidygeocoder: Geocoding Made Easy*. R package version 1.0.1.
- Cannon, A. R., Cobb, G. W., Hartlaub, B. A., Legler, J. M., Lock, R. H., Moore, T. L., Rossman, A. J., and Witmer, J. (2019). *STAT2: Building Models for a World of Data (second edition)*. W. H. Freeman and Company: New York, NY.
- CERN (2008). LHC Guide: A collection of facts and figures about the Large Hadron Collider (LHC) in the form of questions and answers. <http://cds.cern.ch/record/1092437/files/CERN-Brochure-2008-001-Eng.pdf?version=1>.
- Chamandy, N., Muraldharan, O., and Wager, S. (2015). Teaching statistics at ‘Google-Scale’. *The American Statistician*, 69(4):283–291.

- Chang, W. (2019). *webshot: Take Screenshots of Web Pages*. R package version 0.5.2.
- Chang, W. (2020). *extrafont: Tools for Using Fonts*. R package version 0.17.0.9000.
- Chang, W., Cheng, J., Allaire, J., Xie, Y., and McPherson, J. (2020). *shiny: Web Application Framework for R*. R package version 1.5.0.
- Cheng, J., Karambelkar, B., and Xie, Y. (2021). *leaflet: Create Interactive Web Maps with the JavaScript Leaflet Library*. R package version 2.0.4.1.
- Cleveland, W. S. (2001). Data science: An action plan for expanding the technical areas of the field of statistics. *International statistical review*, 69(1):21–26. <http://www.jstor.org/stable/1403527>.
- Cleveland, W. S. and McGill, R. (1984). Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):531–554.
- Cobb, G. W. (2007). The introductory statistics course: A Ptolemaic curriculum? *Technology Innovations in Statistics Education (TISE)*, 1(1). <http://escholarship.org/uc/item/6hb3k0nz>.
- Cobb, G. W. (2015). Mere renovation is too little too late: We need to rethink our undergraduate curriculum from the ground up. *The American Statistician*, 69(4):266–282.
- Columbus, L. (2019). Data scientist leads 50 best jobs in America for 2019 according to Glassdoor. Forbes.com. <https://www.forbes.com/sites/louis columbus/2019/01/23/data-scientist-leads-50-best-jobs-in-america-for-2019-according-to-glassdoor/>.
- Committee on Professional Ethics (1999). *Ethical Guidelines for Statistical Practice*. <http://www.amstat.org/about/ethicalguidelines.cfm>.
- Cook, R. D. (1982). *Residuals and Influence in Regression*. Chapman & Hall, London.
- Cressie, N. (1993). *Statistics for Spatial Data*. John Wiley & Sons: Hoboken, NJ.
- Csárdi, G., Nepusz, T., Horvát, S., Traag, V., and Zanini, F. (2020). *igraph: Network Analysis and Visualization*. R package version 1.2.6.
- Csárdi, G., R Core Team, Wickham, H., Chang, W., Flight, R. M., Müller, K., and Hester, J. (2018). *sessioninfo: R Session Information*. R package version 1.1.1.
- De Veaux, R. D., Agarwal, M., Averett, M., Baumer, B. S., Bray, A., Bressoud, T. C., Bryant, L., Cheng, L. Z., Francis, A., Gould, R., et al. (2017). Curriculum guidelines for undergraduate programs in data science. *Annual Review of Statistics and Its Application*, 4:15–30.
- Diez, D. M., Barr, C. D., and Çetinkaya Rundel, M. (2019). *OpenIntro Statistics (fourth edition)*. OpenIntro.org.
- D'Ignazio, C. and Klein, L. F. (2020). *Data Feminism*. MIT Press: Cambridge, MA.
- Donoho, D. (2017). 50 years of data science. *Journal of Computational and Graphical Statistics*, 26(4):745–766.
- Dunnington, D. (2021). *ggspatial: Spatial Data Framework for ggplot2*. R package version 1.1.5.
- Easley, D. and Kleinberg, J. (2010). *Networks, Crowds, and Markets: Reasoning about a Highly Connected World*. Cambridge University Press: Cambridge, UK.

- Eddelbuettel, D., Francois, R., Allaire, J., Ushey, K., Kou, Q., Russell, N., Bates, D., and Chambers, J. (2020). *Rcpp: Seamless R and C++ Integration*. R package version 1.0.5.
- Editorial (2013). Announcement: Reducing our irreproducibility. *Nature*, 496.
- Efron, B. (2020). Prediction, estimation, and attribution. *Journal of the American Statistical Association*, 115(530):636–655.
- Efron, B. and Hastie, T. (2016). *Computer Age Statistical Inference: Algorithms, Evidence, and Data Science*. Cambridge University Press: Cambridge, UK.
- Efron, B. and Tibshirani, R. J. (1993). *An Introduction to the Bootstrap*. Chapman & Hall: London.
- Ellenberg, J. H. (1983). Ethical guidelines for statistical practice: A historical perspective. *The American Statistician*, 37(1):1–4.
- Engel, C. A. (2019). *R for Geospatial Analysis and Mapping*. The Geographic Information Science & Technology Body of Knowledge. 1st Quarter 2019 Edition.
- Engstrom, R. L. and Wildgen, J. K. (1977). Pruning thorns from the thicket: An empirical test of the existence of racial gerrymandering. *Legislative Studies Quarterly*, pages 465–479.
- Erdős, P. and Rényi, A. (1959). On random graphs. *Publicationes Mathematicae Debrecen*, 6:290–297.
- Euler, L. (1953). Leonhard Euler and the Königsberg bridges. *Scientific American*, 189(1):66–70.
- Feinerer, I. and Hornik, K. (2020). *tm: Text Mining Package*. R package version 0.7-8.
- Fellows, I. (2018). *wordcloud: Word Clouds*. R package version 2.6.
- Finzer, W. (2013). The data science education dilemma. *Technology Innovations in Statistics Education*, 7(2). <http://escholarship.org/uc/item/7gv0q9dc.pdf>.
- Firke, S. (2021). *janitor: Simple Tools for Examining and Cleaning Dirty Data*. R package version 2.1.0.
- Fox, J. (2009). Aspects of the social organization and trajectory of the R Project. *The R Journal*, 1(2):5–13.
- Fraley, C., Raftery, A. E., and Scrucca, L. (2020). *mclust: Gaussian Mixture Modelling for Model-Based Clustering, Classification, and Density Estimation*. R package version 5.4.7.
- Friedman, J., Hastie, T., Tibshirani, R., Narasimhan, B., Tay, K., and Simon, N. (2020). *glmnet: Lasso and Elastic-Net Regularized Generalized Linear Models*. R package version 4.0-2.
- Friendly, M., Dalzell, C., Monkman, M., and Murphy, D. (2020). *Lahman: Sean Lahman Baseball Database*. R package version 8.0-0.
- Futschek, G. (2006). Algorithmic thinking: the key for understanding computer science. In *International conference on informatics in secondary schools-evolution and perspectives*, pages 159–168. Springer.
- Gandrud, C. (2014). *Reproducible Research with R and RStudio*. CRC Press: Boca Raton, FL.

- Ganz, C., Csárdi, G., Hester, J., Lewis, M., and Tatman, R. (2019). *available: Check if the Title of a Package is Available, Appropriate and Interesting*. R package version 1.0.4.
- Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company: New York, NY.
- Garnier, S. (2018a). *viridis: Default Color Maps from matplotlib*. R package version 0.5.1.
- Garnier, S. (2018b). *viridisLite: Default Color Maps from matplotlib (Lite Version)*. R package version 0.3.0.
- Gelman, A. (2011). Ethics and statistics: Open data and open methods. *Chance*, 24(4):51–53.
- Gelman, A. (2012). Ethics and statistics: Ethics and the statistical use of prior information. *Chance*, 25(4):52–54.
- Gelman, A. (2020). Ethics and statistics: Statistics as squid ink. *Chance*, 33(2):25–27.
- Gelman, A., Fagan, J., and Kiss, A. (2007). An analysis of the New York City police department’s “stop-and-frisk” policy in the context of claims of racial bias. *Journal of the American Statistical Association*, 102(479):813–823.
- Gelman, A. and Loken, E. (2012). Ethics and statistics: Statisticians: When we teach, we don’t practice what we preach. *Chance*, 25(1):47–48.
- Gelman, A., Pasarica, C., and Dodhia, R. (2002). Let’s practice what we preach: Turning tables into graphs. *The American Statistician*, 56(2):121–130.
- Gentry, J. (2015). *twitteR: R Based Twitter Client*. R package version 1.1.9.
- Glickman, M., Brown, J., and Song, R. (2019). (A) Data in the life: Authorship attribution in Lennon-McCartney songs. *Harvard Data Science Review*, 1(1). <https://hdsr.mitpress.mit.edu/pub/xcq8a1v1>.
- Good, P. I. and Hardin, J. W. (2012). *Common Errors in Statistics (and How to Avoid Them)*. John Wiley & Sons: Hoboken, NJ.
- Graphodatsky, A. S., Trifonov, V. A., and Stanyon, R. (2011). The genome diversity and karyotype evolution of mammals. *Molecular cytogenetics*, 4(1):1.
- Green, J. L. and Blankenship, E. E. (2015). Fostering conceptual understanding in mathematical statistics. *The American Statistician*, 69(4):315–325.
- Hardin, J., Hoerl, R., Horton, N. J., Nolan, D., Baumer, B. S., Hall-Holt, O., Murrell, P., Peng, R., Roback, P., Temple Lang, D., and Ward, M. D. (2015). Data science in statistics curricula: Preparing students to ‘think with data’. *The American Statistician*, 69(4):343–353.
- Harrell, Jr., F. E. (2020). *Hmisc: Harrell Miscellaneous*. R package version 4.4-2.
- Hastie, T. and Efron, B. (2013). *lars: Least Angle Regression, Lasso and Forward Stagewise*. R package version 1.2.
- Hastie, T., Tibshirani, R., and Friedman, J. J. H. (2009). *The elements of statistical learning*. Springer Verlag: New York, NY, 2nd edition. <http://www-stat.stanford.edu/~tibs/ElemStatLearn/>.
- Henry, L. (2020). Interactivity and programming in the tidyverse. RStudio::conf 2020.

- Henry, L. and Wickham, H. (2020a). *purrr: Functional Programming Tools*. R package version 0.3.4.
- Henry, L. and Wickham, H. (2020b). *rlang: Functions for Base Types and Core R and Tidyverse Features*. R package version 0.4.10.
- Herndon, T., Ash, M., and Pollin, R. (2014). Does high public debt consistently stifle economic growth? A critique of Reinhart and Rogoff. *Cambridge Journal of Economics*, 38(2):257–279.
- Hester, J. (2020). *bench: High Precision Timing of R Expressions*. R package version 1.1.1.
- Hester, J., Csárdi, G., Wickham, H., Chang, W., Morgan, M., and Tenenbaum, D. (2020). *remotes: R Package Installation from Remote Repositories, Including GitHub*. R package version 2.2.0.
- Hester, J. and Wickham, H. (2020). *fs: Cross-Platform File System Operations Based on libuv*. R package version 1.5.0.
- Hesterberg, T. (2015). What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum. *The American Statistician*, 69(4):371–386.
- Hesterberg, T. C., Moore, D. S., Monaghan, S., Clipson, A., and Epstein, R. (2005). *Bootstrap Methods and Permutation Tests*. W. H. Freeman and Company: New York, NY.
- Hoaglin, D. C. (2016). Regressions are commonly misinterpreted. *Stata Journal*, 16(1):5–22.
- Hodge, J. K., Marshall, E., and Patterson, G. (2010). Gerrymandering and convexity. *The College Mathematics Journal*, 41(4):312–324.
- Hornik, K. (2020). *RWeka: R/Weka Interface*. R package version 0.4-43.
- Horst, A., Hill, A., and Gorman, K. (2020). *palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data*. R package version 0.1.0.
- Horton, N. J. (2013). I hear, I forget. I do, I understand: A modified Moore-method mathematical statistics course. *The American Statistician*, 67(3):219–228.
- Horton, N. J. (2015). Challenges and opportunities for statistics and statistical education: looking back, looking forward. *The American Statistician*, 69(2):138–145.
- Horton, N. J., Baumer, B. S., and Wickham, H. (2015). Setting the stage for data science: integration of data management skills in introductory and second courses in statistics. *Chance*, 28(2). <http://chance.amstat.org/2015/04/setting-the-stage/>.
- Horton, N. J., Brown, E. R., and Qian, L. (2004). Use of R as a toolbox for mathematical statistics exploration. *The American Statistician*, 58(4):343–357.
- Horton, N. J. and Hardin, J. S. (2015). Teaching the next generation of statistics students to “think with data”: special issue on statistics and the undergraduate curriculum. *The American Statistician*, 69(4):259–265.
- Horton, N. J. and Kleinman, K. P. (2007). Much ado about nothing: A comparison of missing data methods and software to fit incomplete data regression models. *The American Statistician*, 61:79–90.
- Hothorn, T. and Zeileis, A. (2020). *partykit: A Toolkit for Recursive Partitioning*. R package version 1.2-11.

- Howe, B. (2014). Data manipulation at scale: Systems and algorithms. Coursera. public materials at https://github.com/uwescience/datasci_course_materials.
- Hubert, L. and Wainer, H. (2012). *A Statistical Guide for the Ethically Perplexed*. CRC Press: Boca Raton, FL.
- Huff, D. (1954). *How to Lie with Statistics*. W.W. Norton & Company: New York, NY.
- Hvitfeldt, E. (2020). *textdata: Download and Load Various Text Datasets*. R package version 0.4.1.
- Hyafil, L. and Rivest, R. L. (1976). Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17.
- Iannone, R., Allaire, J., and Borges, B. (2020). *flexdashboard: R Markdown Format for Flexible Dashboards*. R package version 0.5.2.
- Ihaka, R. and Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314.
- Imai, K. and Khanna, K. (2016). Improving ecological inference by predicting individual ethnicity from voter registration records. *Political Analysis*, pages 263–272.
- IMDB.com (2013). Internet movie database. http://www.imdb.com/help/show_article?conditions.
- Ioannidis, J. P. (2005). Why most published research findings are false. *Chance*, 18(4):40–47. <http://journals.plos.org/plosmedicine/article?id=10.1371/journal.pmed.0020124>.
- James, B. (1986). *The Bill James Historical Baseball Abstract*. Random House: New York, NY.
- James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An introduction to statistical learning*. Springer Verlag: New York, NY. <http://www-bcf.usc.edu/~gareth/ISL/>.
- Jeppson, H., Hofmann, H., and Cook, D. (2020). *ggmosaic: Mosaic Plots in the ggplot2 Framework*. R package version 0.3.0.
- Kaplan, D. (2020). *etude: Utilities for Handling Textbook Exercises with Knitr*. R package version 0.3.0.9001.
- Kern, S., Skoog, I., Börjesson-Hanson, A., Blennow, K., Zetterberg, H., Östling, S., Kern, J., Gudmundsson, P., Marlow, T., Rosengren, L., et al. (2014). Higher CSF interleukin-6 and CSF interleukin-8 in current depression in older women. results from a population-based sample. *Brain, Behavior, and Immunity*, 41:55–58.
- Kern, S., Skoog, I., Börjesson-Hanson, A., Östling, S., Kern, J., Gudmundsson, P., Marlow, T., Waern, M., Blennow, K., Zetterberg, H., et al. (2013). Retraction notice to “lower CSF interleukin-6 predicts future depression in a population-based sample of older women followed for 17 years”. *Brain, Behavior, and Immunity*, 32:153–158.
- Khanna, K. and Imai, K. (2020). *wru: Who are You? Bayesian Prediction of Racial Category Using Surname and Geolocation*. R package version 0.1-10.
- Kim, A. Y. and Escobedo-Land, A. (2015). Okcupid data for introductory statistics and data science courses. *Journal of Statistics Education*, 23(2).
- Kirkegaard, E. O. and Bjerrekær, J. D. (2016). The okcupid dataset: A very large public dataset of dating site users. *Open Differential Psychology*, 46:1–10.

- Kline, K. E., Kline, D., Hunt, B., and Heymann-Reder, D. (2008). *SQL in a Nutshell*. O'Reilly Media: Sebastopol, CA. 3rd edition.
- Knuth, D. (1992). Literate programming. *CSLI Lecture Notes, Stanford University*, 27.
- Kuhn, M. (2020a). *caret: Classification and Regression Training*. R package version 6.0-86.
- Kuhn, M. (2020b). Cran Task View: Reproducible research. Accessed September 6, 2020.
- Kuhn, M. (2020c). *discrim: Model Wrappers for Discriminant Analysis*. R package version 0.1.1.
- Kuhn, M. and Vaughan, D. (2020a). *parsnip: A Common API to Modeling and Analysis Functions*. R package version 0.1.4.
- Kuhn, M. and Vaughan, D. (2020b). *yardstick: Tidy Characterizations of Model Performance*. R package version 0.0.7.
- Kuhn, M. and Wickham, H. (2020). *tidymodels: Easily Install and Load the Tidymodels Packages*. R package version 0.1.2.
- Kuiper, S. and Sklar, J. (2012). *Practicing Statistics: Guided Investigations for the Second Course*. Pearson Education: New York, NY.
- Laney, D. (2001). 3D data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6:70.
- Lewis, M. (2003). *Moneyball: The Art of Winning an Unfair Game*. W.W. Norton & Company: New York, NY.
- Little, R. J. A. and Rubin, D. B. (2002). *Statistical Analysis With Missing Data (second edition)*. John Wiley & Sons: Hoboken, NJ.
- Lovelace, R., Nowosad, J., and Muenchow, J. (2019). *Geocomputation with R*. CRC Press: Boca Raton, FL.
- Ludwig, L. (2012). Technically speaking. http://techspeaking.denison.edu/Technically_Speaking.
- Lumley, T. (2020). *biglm: Bounded Memory Linear and Generalized Linear Models*. R package version 0.9-2.1.
- Lunzer, A. and McNamara, A. (2017). Introduction to the "exploring histograms" online essay. Technical report, Viewpoints Research Institute. Research Note RN-2017-003.
- Luraschi, J., Kuo, K., Ushey, K., Allaire, J., Falaki, H., Wang, L., Zhang, A., Li, Y., and The Apache Software Foundation (2020). *sparklyr: R Interface to Apache Spark*. R package version 1.5.2.
- Mackenzie, J. (2009). Gerrymandering and legislator efficiency. Technical report, University of Delaware. <https://www.udel.edu/johnmack/research/gerrymandering.pdf>.
- Marchi, M. and Albert, J. (2013). *Analyzing Baseball Data with R*. CRC Press: Boca Raton, FL.
- McCallum, E. and Weston, S. (2011). *Parallel R*. O'Reilly Media: Sebastopol, CA.
- McIlroy, D., Brownrigg, R., Minka, T. P., and Bivand, R. (2020). *mapproj: Map Projections*. R package version 1.2.7.

- Meyer, F. and Perrier, V. (2020). *shinybusy: Busy Indicator for Shiny Applications*. R package version 0.2.2.
- Mosteller, F. (1987). *Fifty Challenging Problems in Probability with Solutions*. Dover Publications: Mineola, NY.
- Mosteller, F. and Wallace, D. L. (1963). Inference in an authorship problem: A comparative study of discrimination methods applied to the authorship of the disputed federalist papers. *Journal of the American Statistical Association*, 58(302):275–309.
- Müller, K. (2020). *here: A Simpler Way to Find Your Files*. R package version 1.0.1.
- Müller, K. and Walthert, L. (2020). *styler: Non-Invasive Pretty Printing of R Code*. R package version 1.3.2.
- Müller, K., Wickham, H., James, D. A., and Falcon, S. (2020). *RSQlite: SQLite Interface for R*. R package version 2.2.1.
- National Academies of Science, Engineering, and Medicine (2018). Data science for undergraduates: Opportunities and options. <https://nas.edu/envisioningds>.
- Neuwirth, E. (2014). *RColorBrewer: ColorBrewer Palettes*. R package version 1.1-2.
- Nielsen, F. Å. (2011). A new ANEW: evaluation of a word list for sentiment analysis in microblogs. *CoRR*, abs/1103.2903.
- Niemi, R. G., Grofman, B., Carlucci, C., and Hofeller, T. (1990). Measuring compactness and the role of a compactness standard in a test for partisan and racial gerrymandering. *The Journal of Politics*, 52(4):1155–1181.
- Nolan, D. and Perrett, J. (2016). Teaching and learning data visualization: Ideas and assignments. *The American Statistician*, 70(3):260–269.
- Nolan, D. and Speed, T. P. (1999). Teaching statistics theory through applications. *The American Statistician*, 53:370–375.
- Nolan, D. and Temple Lang, D. (2010). Computing in the statistics curricula. *The American Statistician*, 64(2):97–107.
- Nuzzo, R. (2014). Scientific method: Statistical errors. *Nature*, 506:150–152.
- Ohm, P. (2010). Broken promises of privacy: Responding to the surprising failure of anonymization. *UCLA Law Review*, 57:1701.
- Oleś, A. (2020). *openrouteservice: Openrouteservice API Client*. R package version 0.4.0.
- Olford, R. and Cherry, W. (2003). Picturing probability: The poverty of Venn diagrams, the richness of Eikosograms. <http://sas.uwaterloo.ca/~rwoldfor/papers/venn/eikosograms/paper.pdf>.
- O’Neil, C. (2016). *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*. Crown Publishing, New York, NY.
- Ooms, J. (2020a). *jsonlite: A Simple and Robust JSON Parser and Generator for R*. R package version 1.7.2.
- Ooms, J. (2020b). *magick: Advanced Graphics and Image-Processing in R*. R package version 2.5.2.
- Ooms, J., James, D., DebRoy, S., Wickham, H., and Horner, J. (2020). *RMySQL: Database Interface and MySQL Driver for R*. R package version 0.10.21.

- Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The PageRank citation ranking: bringing order to the web. Technical report, Stanford University InfoLab. <http://ilpubs.stanford.edu:8090/422>.
- Paradis, E., Blomberg, S., Bolker, B., Brown, J., Claramunt, S., Claude, J., Cuong, H. S., Desper, R., Didier, G., Durand, B., Dutheil, J., Ewing, R., Gascuel, O., Guillerme, T., Heibl, C., Ives, A., Jones, B., Krah, F., Lawson, D., Lefort, V., Legendre, P., Lemon, J., Louvel, G., Marcon, E., McCloskey, R., Nylander, J., Opgen-Rhein, R., Popescu, A.-A., Royer-Carenzi, M., Schliep, K., Strimmer, K., and de Vienne, D. (2020). *ape: Analyses of Phylogenetics and Evolution*. R package version 5.4-1.
- Pebesma, E. (2021). *sf: Simple Features for R*. R package version 0.9-7.
- Pebesma, E. and Bivand, R. (2020). *sp: Classes and Methods for Spatial Data*. R package version 1.4-4.
- Pebesma, E., Mailund, T., and Kalinowski, T. (2020). *units: Measurement Units for R Vectors*. R package version 0.6-7.
- Pedersen, T. L. (2020a). *ggraph: An Implementation of Grammar of Graphics for Graphs and Networks*. R package version 2.0.4.
- Pedersen, T. L. (2020b). *patchwork: The Composer of Plots*. R package version 1.1.1.
- Pedersen, T. L. (2020c). *tidygraph: A Tidy API for Graph Manipulation*. R package version 1.2.0.
- Pedersen, T. L. (2020d). *transformr: Polygon and Path Transformations*. R package version 0.1.3.
- Pedersen, T. L. and Robinson, D. (2020). *gganimate: A Grammar of Animated Graphics*. R package version 1.0.7.
- Pierson, S. (2016). Jordan urges both computational and inferential thinking in data science. *Amstat News*. <http://magazine.amstat.org/blog/2016/03/01/jordan16>.
- Provost, F. and Fawcett, T. (2013). Data science and its relationship to big data and data-driven decision making. *Big Data*, 1(1):51–59.
- Pruim, R. (2015). *NHANES: Data from the US National Health and Nutrition Examination Study*. R package version 2.1.0.
- Pruim, R., Kaplan, D., and Horton, N. (2020a). *mosaicData: Project MOSAIC Data Sets*. R package version 0.20.1.
- Pruim, R., Kaplan, D. T., and Horton, N. J. (2020b). *mosaic: Project MOSAIC Statistics and Mathematics Teaching Utilities*. <https://github.com/ProjectMOSAIC/mosaic>, <https://projectmosaic.github.io/mosaic/>.
- R Core Team (2020). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.
- R Special Interest Group on Databases (R-SIG-DB), Wickham, H., and Müller, K. (2019). *DBI: R Database Interface*. R package version 1.1.0.
- Raghunathan, T. E. (2004). What do we do with missing data? Some options for analysis of incomplete data. *Annual Review of Public Health*, 25:99–117.
- Ram, K. and Broman, K. (2019). *aRxiv: Interface to the arXiv API*. R package version 0.5.19.

- Ram, K. and Wickham, H. (2018). *wesanderson: A Wes Anderson Palette Generator*. R package version 0.3.6.
- Rice, J. A. (2006). *Mathematical Statistics and Data Analysis (third edition)*. Cengage Learning: Boston, MA.
- Rizzo, M. L. (2019). *Statistical Computing with R (second edition)*. CRC Press: Boca Raton, FL.
- Robinson, D. (2020). *gutenbergr: Download and Process Public Domain Works from Project Gutenberg*. R package version 0.2.0.
- Robinson, D., Hayes, A., and Couch, S. (2020). *broom: Convert Statistical Objects into Tidy Tibbles*. R package version 0.7.3.
- Robinson, D. and Silge, J. (2021). *tidytext: Text Mining using dplyr, ggplot2, and Other Tidy Tools*. R package version 0.3.0.
- Rogoff, K. and Reinhart, C. (2010). Growth in a time of debt. *American Economic Review*, 100(2):573–8.
- Romano, J. P. and Siegel, A. F. (1986). *Counterexamples in Probability and Statistics*. Cengage Learning: Boston, MA.
- Roose, K. (2013). Meet the 28-year-old grad student who just shook the global austerity movement. New York Magazine.
- Rosling, O., Rönnlund, A. R., and Rosling, H. (2005). Gapminder. <http://gapminder.org>.
- RStudio, P. (2020). *RStudio: Integrated Development Environment for R*. RStudio, PBC, 250 Northern Ave, Boston, MA, 02210. Version 1.3.1056.
- Rudis, B. (2019). *streamgraph: streamgraph is an htmlwidget for building streamgraph visualizations*. R package version 0.9.0.
- Ruppert, D., Wand, M. P., and Carroll, R. J. (2003). *Semiparametric Regression*. Cambridge University Press: Cambridge, UK.
- Samet, J. H., Larson, M. J., Horton, N. J., Doyle, K., Winter, M., and Saitz, R. (2003). Linking alcohol and drug dependent adults to primary medical care: A randomized controlled trial of a multidisciplinary health intervention in a detoxification unit. *Addiction*, 98(4):509–516.
- Sarkar, D. (2020). *lattice: Trellis Graphics for R*. R package version 0.20-41.
- Schliep, K. and Hechenbichler, K. (2016). *kknn: Weighted k-Nearest Neighbors*. R package version 1.3.1.
- Schloerke, B., Cook, D., Larmarange, J., Briatte, F., Marbach, M., Thoen, E., Elberg, A., and Crowley, J. (2021). *GGally: Extension to ggplot2*. R package version 2.1.0.
- Schwarz, A. (2005). *The Numbers Game: Baseball's Lifelong Fascination With Statistics*. St. Martin's Press: New York, NY.
- Sievert, C., Parmer, C., Hocking, T., Chamberlain, S., Ram, K., Corvellec, M., and Despouy, P. (2020). *plotly: Create Interactive Web Graphics via plotly.js*. R package version 4.9.2.2.
- Silge, J. and Robinson, D. (2016). tidytext: Text mining and analysis using tidy data principles in R. *Journal of Open Source Software*, 1(3).

- Silge, J. and Robinson, D. (2017). *Text Mining with R: A Tidy Approach*. O'Reilly Media: Sebastopol, CA.
- Slowikowski, K. (2020). *ggrepel: Automatically Position Non-Overlapping Text Labels with ggplot2*. R package version 0.9.0.
- Spinu, V., Golemund, G., and Wickham, H. (2020). *lubridate: Make Dealing with Dates a Little Easier*. R package version 1.7.9.2.
- Stonebraker, M., Abadi, D., DeWitt, D. J., Madden, S., Paulson, E., Pavlo, A., and Rasin, A. (2010). MapReduce and parallel DBMSs: friends or foes? *Communications of the ACM*, 53(1):64–71.
- Tan, P.-N., Steinbach, M., and Kumar, V. (2006). *Introduction to Data Mining*. Pearson Education: New York, NY, 1st edition. <http://www-users.cs.umn.edu/~kumar/dmbook/index.php>.
- Tapal, M., Gahwagy, R., Ryan, I., and Baumer, B. S. (2020a). *fec12: Data Package for 2012 Federal Elections*. R package version 0.0.0.9011.
- Tapal, M., Gahwagy, R., Ryan, I., and Baumer, B. S. (2020b). *fec16: Data Package for the 2016 United States Federal Elections*. R package version 0.1.3.
- Temple Lang, D. (2020). *RCurl: General Network (HTTP/FTP/...) Client Interface for R*. R package version 1.98-1.2.
- Therneau, T. and Atkinson, B. (2019). *rpart: Recursive Partitioning and Regression Trees*. R package version 4.1-15.
- Torres-Manzanera, E. (2018). *xkcd: Plotting ggplot2 Graphics in an XKCD Style*. R package version 0.0.6.
- Travers, J. and Milgram, S. (1969). An experimental study of the small world problem. *Sociometry*, pages 425–443.
- Tufte, E. R. (1990). *Envisioning Information*. Graphics Press: Cheshire, CT.
- Tufte, E. R. (1997). *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press: Cheshire, CT.
- Tufte, E. R. (2001). *Visual Display of Quantitative Information (second edition)*. Graphics Press: Cheshire, CT.
- Tufte, E. R. (2003). *The Cognitive Style of PowerPoint*. Graphics Press: Cheshire, CT.
- Tufte, E. R. (2006). *Beautiful Evidence*. Graphics Press: Cheshire, CT.
- Tukey, J. W. (1990). Data-based graphics: visual display in the decades to come. *Statistical Science*, 5(3):327–339.
- Ushey, K. (2021). *renv: Project Environments*. R package version 0.12.5.
- Ushey, K., Allaire, J., and Tang, Y. (2020). *reticulate: Interface to Python*. R package version 1.18.
- Ushey, K., McPherson, J., Cheng, J., Atkins, A., and Allaire, J. (2018). *packrat: A Dependency Management System for Projects and their R Package Dependencies*. R package version 0.5.0.
- Vaidyanathan, R., Xie, Y., Allaire, J., Cheng, J., Sievert, C., and Russell, K. (2020). *htmlwidgets: HTML Widgets for R*. R package version 1.5.3.

- van Belle, G. (2008). *Statistical Rules of Thumb (second edition)*. John Wiley & Sons: Hoboken, NJ.
- Vanderkam, D., Allaire, J., Owen, J., Gromer, D., and Thieurnel, B. (2018). *dygraphs: Interface to Dygraphs Interactive Time Series Charting Library*. R package version 1.1.1.6.
- Vaughan, D. and Dancho, M. (2020). *furrr: Apply Mapping Functions in Parallel using Futures*. R package version 0.2.1.
- Vinten-Johansen, P., Brody, H., Paneth, N., Rachman, S., Rip, M., and Zuck, D. (2003). *Cholera, Chloroform, and the Science of Medicine: A Life of John Snow*. Oxford University Press.
- Walker, K. (2020a). *tigris: Load Census TIGER/Line Shapefiles*. R package version 1.0.
- Walker, K. (2020b). *tigris: Load Census TIGER/Line Shapefiles into R*. R package version 1.0.
- Walker, K. and Herman, M. (2020). *tidycensus: Load US Census Boundary and Attribute Data as tidyverse and sf-Ready Data Frames*. R package version 0.11.
- Wang, V. (2006). The OBP/SLG ratio: What does history say? *By the Numbers*, 16(3):3.
- Wang, Y. and Kosinski, M. (2018). Deep neural networks are more accurate than humans at detecting sexual orientation from facial images. *Journal of Personality and Social Psychology*, 114(2):246.
- Wasserstein, R. L. and Lazar, N. A. (2016). The ASA's statement on p-values: Context, process, and purpose. *The American Statistician*, 70(2):129–133.
- Wasserstein, R. L., Schirm, A. L., and Lazar, N. A. (2019). Moving to a world beyond $p < 0.05$. *The American Statistician*, 73(sup1):1–19.
- Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of ‘small-world’ networks. *Nature*, 393(6684):440–442.
- Weisberg, S. (2018). *alr3: Data to Accompany Applied Linear Regression 3rd Edition*. R package version 2.0.8.
- Wickham, H. (2011). Asa 2009 data expo. *Journal of Computational and Graphical Statistics*, 20(2):281–283.
- Wickham, H. (2014). Tidy data. *The Journal of Statistical Software*, 59(10). <http://vita.had.co.nz/papers/tidy-data.html>.
- Wickham, H. (2015). *R Packages: Organize, Test, Document, and Share your Code*. O'Reilly Media: Sebastopol, CA.
- Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer Verlag: New York, NY.
- Wickham, H. (2019a). *Advanced R (second edition)*. CRC Press: Boca Raton, FL.
- Wickham, H. (2019b). *assertthat: Easy Pre and Post Assertions*. R package version 0.2.1.
- Wickham, H. (2019c). *babynames: US Baby Names 1880-2014*. R package version 1.0.0.
- Wickham, H. (2019d). *babynames: US Baby Names 1880-2017*. R package version 1.0.0.
- Wickham, H. (2019e). *lazyeval: Lazy (Non-Standard) Evaluation*. R package version 0.2.2.

- Wickham, H. (2019f). *nycflights13: Flights that Departed NYC in 2013*. R package version 1.0.1.
- Wickham, H. (2019g). *stringr: Simple, Consistent Wrappers for Common String Operations*. R package version 1.4.0.
- Wickham, H. (2019h). *tidyverse: Easily Install and Load the Tidyverse*. R package version 1.3.0.
- Wickham, H. (2020a). *forecats: Tools for Working with Categorical Variables (Factors)*. R package version 0.5.0.
- Wickham, H. (2020b). *Mastering Shiny*. O'Reilly Media: Sebastopol, CA.
- Wickham, H. (2020c). *modelr: Modelling Functions that Work with the Pipe*. R package version 0.1.8.
- Wickham, H. (2020d). *rvest: Easily Harvest (Scrape) Web Pages*. R package version 0.3.6.
- Wickham, H. (2020e). *testthat: Unit Testing for R*. R package version 3.0.1.
- Wickham, H. (2020f). *tidyr: Easily Tidy Data with ‘spread()’ and ‘gather()’ Functions*. R package version 1.1.2.
- Wickham, H. (2020g). *tidyverse: Tidy Messy Data*. R package version 1.1.2.
- Wickham, H. and Bryan, J. (2019). *readxl: Read Excel Files*. R package version 1.3.1.
- Wickham, H. and Bryan, J. (2020a). *bigrquery: An Interface to Google's BigQuery 'API'*. R package version 1.3.2.
- Wickham, H. and Bryan, J. (2020b). *usethis: Automate Package and Project Setup*. R package version 2.0.0.
- Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., Yutani, H., and Dunnington, D. (2020a). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.3.3.
- Wickham, H., Cook, D., and Hofmann, H. (2015). Visualizing statistical models: Removing the blindfold. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 8(4):203–225.
- Wickham, H. and Francois, R. (2020). *dplyr: a grammar of data manipulation*. R package version 1.0.2.
- Wickham, H., François, R., Henry, L., and Müller, K. (2020b). *dplyr: A Grammar of Data Manipulation*. R package version 1.0.2.
- Wickham, H. and Hester, J. (2020). *readr: Read Rectangular Text Data*. R package version 1.4.0.
- Wickham, H. and Miller, E. (2020). *haven: Import and Export SPSS, Stata and SAS Files*. R package version 2.3.1.
- Wickham, H. and Ruiz, E. (2020). *dbplyr: A dplyr Back End for Databases*. R package version 2.0.0.
- Wickham, H. and Seidel, D. (2020). *scales: Scale Functions for Visualization*. R package version 1.1.1.
- Wikipedia (2016). Hippocratic oath. https://en.wikipedia.org/wiki/Hippocratic_oath.

- Wilkinson, L., Wills, D., Rope, D., Norton, A., and Dubbs, R. (2005). *The Grammar of Graphics (second edition)*. Springer Verlag: New York, NY.
- Xie, Y. (2014). *Dynamic Documents with R and knitr*. CRC Press: Boca Raton, FL.
- Xie, Y. (2020a). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.21.
- Xie, Y. (2020b). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.30.
- Xie, Y. (2021). *xfun: Miscellaneous Functions by Yihui Xie*. R package version 0.20.
- Xie, Y., Cheng, J., and Tan, X. (2021). *DT: A Wrapper of the JavaScript Library DataTables*. R package version 0.17.
- Yau, N. (2011). *Visualize this: the Flowing Data guide to design, visualization, and statistics*. John Wiley & Sons: Hoboken, NJ.
- Yau, N. (2013). *Data points: visualization that means something*. John Wiley & Sons: Hoboken, NJ.
- Zaslavsky, A. M. and Horton, N. J. (1998). Balancing disclosure risk against the loss of nonpublication. *Journal of Official Statistics*, 14(4):411–419.
- Zhu, H. (2020). *kableExtra: Construct Complex Table with kable and Pipe Syntax*. R package version 1.3.1.

Indices

Separate indices are provided for subject (concept or task) and R command. References to the examples are denoted in *italics*.

Subject index

- 100-meter freestyle, 20
 1960s, 445
 2012 federal election cycle, 9
 3NF, 348
 8th district in Ohio, 15
- A Boy Named Sue, 118
 A/B tests, 5, 6, 199
 Aadam, 119
 Aaron, Hank, 92, 93
 Abrams, Stacey, 162
 access
 databases, 568
 elements, 501
 variables, 526
 accidental patterns, 183
 accountability, 159, 172, 176
 accuracy, 211, 245
 ACM, 176
 acres, 412
 actors, 456, 457, 459, 462
 actuarial tables, 54
 acumen, data, 4, 8, 532
 add rownames, 238
 additional factors, 12, 198
 adjacency matrix, 471, 473
 administration, database, 363
 Adobe Illustrator, 317
 Advanced R, 514
 adverb, 142
 adversarial system, 170
 advertisers, 466
 aesthetic, 35, 36
 Africa, 384
 afternoon, 196
 age, 223
 aggregating, 74, 237, 338
 Aguadilla, 349
 airline delays, *see flight delays*
 airport, 184
 codes, 345
 Reagan National, 348, 369
 AirTran, 354
 Airy ellipsoid, 387
 Akten, Memo, 29
 Alaska, 350, 401
 Albers equal projection, 385
 Albers equal-area conic, 385
- Albert, Jim, xv, 99
 Albert, Réka, 455
 Alderson, Sandy, 81
 algebra, 108
 algorithmic bias, 171, 176
 algorithmic techniques, 207
 algorithmic thinking, 4, 519, 529
 alias, table, 346
 alive, 54
 all or nothing thinking, 202
 Allaire, J.J., xv
 Alon, Noga, 456
 alpha level, 202, 203
 Altavista, 465
 alter database, 365
 ALTER statement, *see SQL*
 Amazon Web Services, 332, 477, 488, 493
 ambient temperature, 25
 American Airlines, 89, 354, 356
 American League, 94, 149
 American National Standards Institute, 331
 American Political Association, 177
 American Statistical Association, 174–176, 529
 statement on p-values, 203
American Statistician, 490
 Amherst College, xiv, 407
 Anaheim Angels, 144
 analogy, cooking, 17
 analysis
 reproducible, 104, 532
 univariate, 25
 analysis of ecological and environmental data task view, 512
 analysis of pharmacokinetic data task view, 512
 analysis of spatial data task view, 512
 analytic (exact) result, 287
 analytics, 67, 490
 predictive, 8, 207, 209, 258, 263
 anchoring, 428
 AND operator, *see SQL*
 Anderson, Wes, 31, 32
 Android, 331
 anecdotal evidence, 25
 angle, 21, 23, 383

- animation, 18
Anna, 62
annotated text, 56
ANOVA, 5
ANSI standard, 331
Aotearoa (New Zealand), 499
Apache Drill, 488
Apache Hadoop, 485
Apache Spark, 485, 490
APIs, *see* application programming interface
application programming interface, 122, 125, 135, 169, 173, 407, 408, 431
apportionment, 392
ArcGIS, 402
ArcView, 380
argument, 68
arguments, 507, 508, 520, 521
array
 extract elements, 503
 indexing, 140
 rectangular, 105
 two-dimensional, 103
arrow, 56
arrows, sorting, 302
art, 29, 31
artificial neural network, 243
artistic talent, 29
arXiv, 431
as the crow files, 408
ascending order, 74
aspect ratio, 104
assembly language, 480
assertions, 528
assessing models, 172
assignment operators, 501
Association for Computing Machinery, 176
associations, 196, 541
assumption, normality, 554
athletes, 29
Atlanta, GA, 266
Atlantic 10 conference, 470
ATOM proposal, 203
attack ads, 10
attributes, 36, 105
 R, 505
attribution, 25
Auckland, University of, 499
audience, intended, xi
audits, data, 173
austerity measures, 166
authorship, 25, 445
average (running), 522
avoiding for loops, 141
axis, 40
axis labels, 28
axis scaling, 13
aye votes, 276
baby names, 7, 53, 60, 105, 118, 302, 371, 486, 571
backticks, 218
Bacon, Kevin, 456, 457
bacteria, 389
Bale, Christian, 463
ballot, 107, 271
bandwidth, 44
Banquo, 429
bar graph, 11, 20, 44, 45
Bar–Noy, Amotz, 456
Barabási, Albert-László, 455, 474
barrels of oil, 35
base graphics, 35
baseball, 7, 76, 92, 99, 139, 144, 478
 analytics, 6, 77
 camera-tracking data, 478
 offensive statistics, 92
 play-by-play data, 478
 rules, 6, 92
Baseball Prospectus, 365
Baseball-Reference, 365
basketball, 467, 468
bathrooms, 126
Batman, 118
Batman: The Dark Knight Rises, 458, 463, 464
batter value, 6
batting, 92
Baumer, Ben, 76
Bayes
 naïve, 241–243, 255
 Theorem, 241
Bayesian inference task view, 512
Bayesian statistics, 296
Beane, Billy, 7
Bears, Chicago, 29
Beatles, 302, 304, 308, 316, 445, 447
bedrooms, 126
beginning of line, 428
bell-shaped, 190
Bella, 53

- Belmont Report, 176
benchmarking, 141, 212
Bentley College, 172
Bergstrom, Carl, 176
Berkeley Initiative for Transparency in the Social Sciences, 177
best job, xi
BETWEEN operator, *see* SQL
betweenness centrality, 452, 463
Beveridge, Andrew, 270
bias, 216
 - algorithmic, 171
 - publication, 175
 - selection, 543
 - statistical, 204
 - variance tradeoff, 216, 255, 551
bichromatic choropleth map, 397
big data, 4, 183, 477, 490
big memory, 479
Big O notation, 231, 232, 366
BigQuery, 487, 488
bigram, 439, 440
bike trail, 409, 541
billing, top, 457
bills of mortality, 3
Bills, Buffalo, *see* Buffalo Bills
binary, 480, 545
binwidth, 43
biologists, 263
birds (Ordway), 126, 129
births, 105
Bivand, Roger, 403
bivariate, 154
black-box, 125
blinding, 205
blindness, color, 17, 19
blocks of code, 535
blood pressure, 112
BMI, 50, *see* Body Mass Index
Body Mass Index, 153, 223, 556
Boehner, John, 15
bold font, 535
Bonds, Barry, 93
Bonferroni correction, 203, 204
book website, xii
Boolean
 - operators, 428, 502
 - pairs, 31
bootstrap, 5, 152, 190
 - decision trees, 237
 - distribution, 192
bootstrap distribution, 192
borough, 310
Bostock, Mike, 301
Boston Red Sox, 93, 94
Boston, MA, 266
botulism, 159
bounding box, 382, 388
box, bounding, 388
box-and-whisker, 50
boxplot, 48, 196
Boynton, Nancy, xv
BP_narrow, 112
BP_wide, 112
Bradley International Airport, 336, 338
Brain, Behavior, and Immunity, 174
branches, 235, 263
breach, ethical, 165
break down problems, 519
breast cancer, 52
Brewer, Cynthia, 19, 31
Bridges of Königsberg, 451
Brin, Sergey, 466
British invasion, 445
British National Grid, 387
Broad Street pump, 378, 403
Broadwick (formerly Broad) Street, 382
Brooklyn, 410
Brooklyn Dodgers, 6
Brooklyn-Queens Expressway, 410
Brown, Scott, 16
browser, 124, 301
 - debugging, 527
brushing, 302
Bryan, Jenny, 155, 537
budgeting time, 28
Buffalo Bills, 19
bulk updates, 370
Bureau of Labor Statistics, 288
Bureau of Transportation Statistics, 89, 493
C++, 480
calculate running average, 522
calculate score, 444
California Angels, 144
calling functions, 507
Calvin Coolidge Bridge, 409
CamelCase, 120
cancelled flights, 353
cancer, 282
cancer genetics, 52

- canonical graphics, 50
capital gains, 212, 231
capitalization, 120
captions, 28
Caris, Jon, xv
cars, 264
CART, 209, 230
Cartesian plane, 18, 21, 44
cartographic imperialism, 384
Cary, 399
cascading delays, 196
case, 105, 109
 in codebook, 110
 special, 519
 studies, xii
 unique, 109
Cash, Johnny, 118
categorical, 20, 23
 outcome, 556
 predictor, 545
 scale, 18
 variable, 17, 58, 108
causal inference, 204, 433
causality, 48
causation, 199, 209
cell lines, 52
Census Bureau, 3, 209, 229, 392, 403
Census tracts, 23
Center for Open Science, 177
Centers for Disease Control, 122
centrality, 451, 452, 466
centroid, 412
CEO, 163, 168
cesspit, 389
Çetinkaya-Rundel, Mine, xv
Chadwick, Henry, 6, 478
chaining operations, 70, 77
Challenger, 26, 27
Challenger, 24, 31, 542
CHANCE magazine, 176
change database, 365
change factor levels, 48
changes, tracking, 535
changing font size, 533
changing graphic size, 533
Channing Laboratory, 174
character data, 425
character sets, 428
character string, 131, 425
character vector, 525
charges, medical, 41
Charlotte, 399
Che, Jonathan, xv
cheat sheets, 62, 84, 322, 537
checkboxes, 309
chefs, great, 16
chemometrics and computational physics
 task view, 512
Cherry Blossom Race, 109
Chestnut Street, 541
Chicago Bears, 29, 31
Chicago White Sox, 93, 94
Chicago, IL, 266
chicken pox, 159
child abuse, 159
cholera, 159, 377, 389, 403
choosing names, 53
choropleth, 399
choropleth map, 23, 52, 397, 399
Citi Field, 409, 410
Citizens United v. Federal Election Com-
 mission, 9
civil rights, 172
class, 505
class label, 230
classic books, 448
classification, 258
classification and regression trees
 (CART), *see* CART
classification rate, 245
classifiers, 209, 229, 241, 243
 perfect, 219
cleaning data, 121, 126
Cleveland Hopkins International Airport,
 342
Cleveland Indians, 93, 99
Cleveland, OH, 342
click-and-drag, 104, 169
client, 170
client-server, 563
client-server model, 332
client-side web applications, 301
climate change, 160
clinical trial design, monitoring, and anal-
 ysis task view, 512
clinical trials, 5, 203, 204
 registries, 176
clobbering, 70
closure, triadic, 454
cloud storage, 488
cluster, 375, 488
 hierarchical, 264, 267

- cluster analysis and finite mixture models
 - task view, 512
- clustering, 263, 270, 275
- clusters, 375, 452
- clusters, Spark, 487
- Cobb, Ty, 93
- code
 - airport, 345
 - blocks, 535
 - completion, 499
 - control systems, 532
 - defensive, 527
 - ethical, 176
 - examples, xii
 - review, 174
 - solutions, 519
- codebook, 109, 110, 126, 174
- coding errors, 166
- coefficient
 - of determination, *see R²*
 - regression, 554
 - true, 552
- coefficient of determination, 220, 545
- coercion, 503, 525
 - data frames into matrices, 505
 - matrices into data frames, 505
- coffee, 199
- cognitive overload, xiii
- cognitive skills, 183
- collaboration, 535
- collapse, 74, 338
- college
 - basketball, 467
 - entry, 199
 - liberal arts, xiv
- colon cancer, 52, 53
- color, 10, 19, 23, 61, 315
 - blindness, 17, 19
 - diverging palette, 397
 - gradient, 559
 - scheme, 397
- ColorBrewer, 19
- Columbia University, 175, 204
- column names, 68
- columns, 105, 489
 - list, 114
- combining tables, 89
- comma separated, 329, 365
- command line, 564, 567
- commander-in-chief, 9
- commands, 519
- comments, 503
- common sense, 163
- Commonwealth Games, 29
- communication skills, xi
- communications, privileged, 171
- commuting routes, 409
- compare versions, 535
- comparison operators, 502
- competing interests, 176
- compilation, 480
- complexity parameter, 235
- Comprehensive R archive network, 499
- compromise, 480
- computational linguistics, 425
- computational thinking, xi
- computer programming, 4
- computer science, 5, 466
- computing, 3
 - parallel, 481
 - scriptable, 532
 - statistical, 281, 296
- CONCAT function, *see SQL*
- concept index, xii
- conclusions, 204
- conditional
 - inference, 258, 281
 - probability, 241
 - regression parameter, 547, 559
- conference, Atlantic 10, 470
- confidence band, 551
- confidence interval, 153, 188, 193, 197, 519, 552
- confidentiality, 163
- conflicts, 505, 509
- confound, 199
- confounding, 198–200
- confusion matrix, 212, 217
- congressional districts, 391, 395, 397, 402
- connected, 451
- connected component, 464
- consensus statement, 159
- consent, 169
- Conservative and Unionist Party, 275
- conservative, socially, 276
- Consolidated Standards of Reporting Trials (CONSORT), 177, 205
- constraints, 255
- context, xii, 17, 18, 28, 378
- continents, 269
- continue execution, 527
- contributions, 9

- control structures, 500
- control, version, 532, 535
- convenience, 480
- conventions, naming, 120
- converged, 294
- convergence, 294, 522
- convert
 - character string to date, 129
 - character to numeric, 128
 - tall to wide, 113
 - wide to tall, 113
- Cook's distance, 554
- cooking analogy, 17
- Coolidge Bridge, 409
- coordinate reference system, 385
- coordinate systems, 17, 18
 - radial, 23
- coordinates, 382
- copyrights expired, 448
- cores, 481
- corpora, 425
- corpus, 425, 434, 483, 484
- correct algorithm, 519
- correction for multiple comparisons, 175, 284
- correlation, 13, 220, 444, 545
 - does not imply causation, 199
- correlation does not imply causation, 199
- counting patterns, 426
- countries, 35
- courts, 168, 170
- COVID, 161, 184, 467, 488
- CPU, 481
- CRAN (Comprehensive R Archive Network), 499, 514
- CRAN task views, *see* task views
- Cranston, Bryan, 460
- create databases, 363, 364
- CREATE statement, *see* SQL
- create table, 364
- creating presentations, 28
- credits, 457
- critical analysis, 17
- cross join, 344
- cross-validation, 216, 217, 240, 241
- crossings, trail, 542
- CRS, 385
- CSV, 122
- CSV file, 122, 329
- CUDA, 482
- cues, visual, 17, 35, 36, 60
- cumulative sum, 523
- curated guide to learning R, 500
- current population, 105
- customizing graphics, 35
- Cygwin, 563
- D3.js, 301, 317
- damage, O-ring, 25
- damages, 167
- damping, 473
- dashboard, 306, 308
- data acumen, 4, 8, 532
- data analysis, 207, 531
- data arrange, 144
- data art, 29, 31
- data big, 477, 490
- data cleaning, 121, 126
- data clustering, 263
- data definition, 183
- data demographic, 403
- data disclosure, 172
- data documentation, 109
- data encoding, 17
- data ethics, 176
- data extracting meaning, 4, 183, 489
- data fabrication, 175
- data feminism, 176
- data frames, 67, 105, 326, 501, 503–505
- data friendly formats, 122
- data gathering, 114
- data generation, 296
- data geospatial, 377
- data graphics, 9, 35
- data habits of mind, 84
- data idioms, 67
- data indexing, 140
- data ingesting, 125
- data intake, 121
- data integration, 379
- data interpretation, 183
- data journalist, 99
- data marshaling, xi
- data medium, 325, 479
- data meta, 174, 388
- data microarray, 282
- data mining, 4
- data missing, 204, 401, 543
- data modeling, 207
- data network, 6, 52, 451
- data observational, 199, 202
- data omitted, 25

- data provenance, 105, 121
- data reading from a URL, 123
- data real-world, 281
- data reduction, 263
- data release, 478
- data safeguards, 172
- data science, xi, xiv, 4, 8, 281, 379, 477
 - ethics, 176
 - papers, 431
- Data Science Association, 176
- data scientist, xi
- data scraping, 165
- data sets, xii, 7, 493
- data sharing, 176
- data spatial, 6, 403, 407
- data spreading, 113
- data storage, 173
- data structure, 377, 501
- data tables, 105, 302
- data technologies, xi
- data test, 210, 217
- data text, 6, 377, 425
- data tidy, 84
- data to ink ratio, 45
- data to understanding, 281
- data training, 210, 217
- data transforming, 105
- data unstructured, 425
- data verbs, 67, 106
 - arrange, 112
 - group by, 118
 - mutate, 107
 - pivot_longer, 113
 - pivot_wider, 113
 - summarize, 118
- data viewer, 499
- data visualization, 4
- data weather, 7
- data wide and narrow, 112
- data wrangling, xi, 4, 67, 84, 106, 526
 - cheat sheet, 84
- data-to-ink ratio, 45
- database, 325
 - access, R, 568
 - administration, 363
 - baseball, 76, 478
 - design, 348
 - Internet movie, 456
 - keys, 92, 365
 - load, 571
 - MySQL, 563
- PostgreSQL, 563
- real estate, 173
- relational, 110, 122, 330
- schema, 365, 372, 571
- server, 332, 563
- SQL, 563
- SQLite, 563
- database integrity, 345
- databases with R task view, 512
- date variables, 70, 123, 129, 133
- dates and times, 336
- datetime, 130
- datum, 387
- datums, 403
- dbf files, 380
- DBIConnection, 486
- de Veaux, Richard, 126
- Death of a Salesman*, 29, 31
- debt crisis, 166
- debugging, 527
- deception, 281
- decile, 24
- decimal places, 364
- decision tree, 230
- decisions, xi, 204
- decomposition, singular value, 273
- default reference category, 546
- default value, 520, 521
- defensive coding, 527, 528
- definition
 - big data, 477
 - data science, 5, 8
 - statistics, 5, 186
- degree, 452
- degree distribution, 455
- deidentified data, 165
- delays, airline, *see* flight delays
- delete objects, 501
- Delta Airlines, 196
- Democrat, 391
- Democratic, 10, 394
- demographic data, 403
- dendrogram, 263, 267
- density curves, 13
- density plot, 43
- Department of Energy, 264
- departure time, 196
- dependency management, 512
- dependent variable, 541
- DePodesta, Paul, 7
- depression, 174

- descending order, 74
describe database, 364
DESCRIBE statement, *see* SQL
design of experiments (doe) and analysis of experimental data task view, 512
design of simulations, 293
designated hitter, 150
detach packages, 510
deterministic, 151
diabetes, 50, 208, 209, 223, 250, 556
diabetes mellitus, 223
Diaconis, Persi, 456
diagnostics, residual, 554
dialects, SQL, 331
diameter, 451
dichotomous, 199
dichotomous outcome, 198, 556
differential equations task view, 512
Digital Ocean, 488
dimension reduction, 263, 270
directed, 451
directed edges, 451
directed graph, 243, 466
direction, 21
disclosure avoidance, 172
discrete, 6
discrimination, employment, 164, 168
disease, 223, 377
disk, hard, 329
dispatched, 506
displaying objects, 509
dissecting graphics, 20
distance, 265, 409
distance measure, 451
distributed architectures, 375
distributed computing, 481
distributed file system, 485
distribution
 degree, 455
 donations, 13
 normal, 190
 probability, 286
 sampling, 186
 stationary, 466
 uniform, 473
districts, congressional, 391
diverging palette, 19, 397
DNA, 207
document term matrix, 440
documentation
 codebook, 109
 ethics, 174
 R, 500, 514
Dodgers
 Brooklyn, 6
 Los Angeles, 93
domain knowledge, 5–7, 25
downloading
 code examples, xii
doxing, 166
dplyr package, 4
 issues on GitHub, 483
Dremel, 488
DROP TABLE statement, *see* SQL
drug company, 166
dummy, 545
dummy variables, 545
Duncan, 429
duplicates, 191
Duquette, Jim, 81
Durham, 399
dygraphs, 304
dynamic maps, 389
dynamic web applications, 308, 499
earth, 267
Eastern White Pine, 419
eating hot dogs, 317
eBay, 29
eccentricity, 451, 465
ecological disruption, 161
econometrics task view, 512
economic indicator, 288
economic productivity, 35
economically conservative, 276
economics, 166
edge, 52, 293, 451, 459, 467
 centrality, 452
 effect, 292
 multiple, 473
 weight, 468
edges, 52
effect size, 203
effective decisions, xi
effective presentations, 24, 28
efficient, 367
 algorithms, 519
 databases, 363
 SQL, 343
Efron, Bradley, 204
Eigenvector centrality, 452

- eigenvector centrality, 452, 466, 467
 eikosogram, 50
 El Guerrouj, Hicham, 125
 elections, 7, 9, 10, 106, 399
 - ballot, 107
 - campaign, 391
 - rank choice, 109
 electronic computers, 3
 electronic repository, 431
 ellipsoids, 403
 Elmer, 53
 else statement, 500
 Elynor, 105
 embarrassingly parallel, 481
 empirical finance task view, 512
 empirical research, 537
 employee selection, 164, 168
 employment discrimination, 164, 168, 170
 encoding data, 17
 encoding variables, 126
 end of line, 425, 428
 Energy Department, 264
 ensemble methods, 243
 entropy, 230
 environment, 509, 525
 environment variable, 564
 Environmental Systems Research Institute, 380
 Envoy Air, 354
 ephemeral table, 351
 Epi, 122
 epidemiology, 205, 377
 EPSG, 385, 387
 epsilon, 541
 equal area projection, 385
 equal variance, 553
 Erdős
 - number, 456
 - Rényi random graphs, 453
 Erdős number, 456
 Erdős, Paul, 452, 456
 error matrix, 212, 217
 error rate, 203
 errors, 541
 - coding, 166
 - minimizing, 528
 - prediction, 219
 Esri, 380
 estimation, 5, 191
 ethical principles, 159, 163, 199
 ethics, 176
 ETL, 371
 Ettinger, Caroline, 270
 Euclidean distance, 239, 265
 Euler, Leonhard, 451
 Europe, 350
 European debt crisis, 166
 European Petroleum Survey Group, 387
 evaluate expression, 527
 evaluating models, 216
 evaluation, lazy, 529
 evaluation, non-standard, 526
 evening, 196
 evidence, 203
 evolutionary
 - biologists, 263
 - tree, 207
 exabytes, 477, 478
 exact result, 287
 exam, high-stakes, *see SAT scores*
 example code, downloading, xii
 Excel, 103, 169, 174, 265, 329, 478
 Excel format, 122
 Excite, 465
 execution, 527
 exercises, xii
 expected, 522
 - flight delay, 196
 expected winning percentage, 77, 147
 expert witness, 171
 expired copyrights, 448
 EXPLAIN statement, *see SQL*
 explaining to humans, 532
 explanatory variables, 198, 209, 541
 exploratory data analysis, 207
 exponent, 147
 exponential random variable, 285
 expression, gene, 52
 expressions, 501, 519, 526, 527
 - regular, 316, 317, 425, 448
 ExpressJet, 347, 354, 356
 extended case studies, xii
 extract from objects, 503
 extract information, xi, 183
 extract number, 133
 extract values, 444
 extract-transform-load, 371
 extracting meaning from data, 4, 489
 extrapolation, 25, 547
 extreme value analysis task view, 512
 FAA, 92, 345

- fabrication of data, 175
Facebook, 135, 331, 451, 477
facet, 18, 40, 46, 58
factor, 131
factors, 131, 514, 546
 reorder levels, 48
Fahrenheit, 28, 160, 542
fail to reject null, 203
fairness, 172, 176
false discovery rate, 284
false findings, 531
false positive, 219
falsehoods, 160
Fangraphs, 365
FAQ
 R, 500
Fayetteville, 399
FDA, 166
features, 208, 267
FEC, 9
fecal bacteria, 389
Federal Election Commission, 7, 9, 392, 493
federal elections, 9
female names, 58
feminism, 176
Feynman, Richard, 24, 25
field station, 493
file
 list, 371
 source, 532
file-drawer problem, 175
filename extension, 122
files
 loading, 121
 readable, 532
 saving, 121
fill
 aesthetic, 36
 scale, 559
finding packages, 512
finish execution, 527
firearm murders, 160
first base, 6
fit, 216
fitted values, 541, 553
Five Colleges, 407
five idioms, 67
five-number summary, 50
FiveThirtyEight, 53, 60
FiveThirtyEight.com, 316, 352
flat-file database, 329
flexdashboard, 306, 322
Flickr, 135
flight delays, 7, 89, 184, 196, 325, 338, 352, 364, 366, 493
Florence, Massachusetts, 541
Florida legislature, 160
flowchart, 230
Flushing, NY, 410
font size, changing, 533
fonts, 316
Food and Drug Administration (FDA), 166
food handling, 290
fool's errand, 104
for loops, 139, 500
Ford, Gerald, 71
foreign key, 95, 365, 366
FOREIGN KEY definition, *see SQL*
forest
 random, 237
forking paths, garden of, 204, 531
format
 data table, 122
 date variables, 129
 output, 535
 PDF, 533
forms, normal, 348
formula, 208, 224, 254, 542
found data, 199, 202
Foundation for Statistical Computing, R, 499
framework for graphics, 12, 16
Frenett, Patrick, xv
frequency inverse document, 440
frequently asked questions, *see FAQ*
Friday, first, 288
Friendly, Michael, 99
fuel economy, 264, 267
Fukushima Daichi, 133
full text, 448
function, 208, 519, 526
 call, 507, 527
 examples, 500
 name conflicts, 509
functional data analysis task view, 512
functional form, 550
functional programming, 155, 489
functionals, 155
functions, 507, 519
funding network, 16

- fundraising, 9
- Gall-Peters projection, 384
- Gapminder, 103, 105
- garden of forking paths, 204, 531
- gastrointestinal events, 166
- gather data, 112, 114
- Gaussian distribution, 286
- Gaussian random variable, 285
- GCP, 488
- GDP, 35
- Gelman, Andrew, 31, 176, 204
- gender, 58, 105
 - neutral names, 118
- gene expression, 52
- general manager, 81
- general-purpose programming languages, 140
- generate random numbers, 294
- generic functions, 72
- generics, 505
- genetics, cancer, 52, 282
- Gentleman, Robert, 499
- geocoding, 407
- geodesic, 408, 451
- geographic coordinate system, 18, 383
- geographic information, 172
- Geographic Information System, 402
- geographic positioning systems, 387
- geography, 52, 377
- geom, 36, 55
 - with new data, 41
- geometry, 287
- George Washington University, 468, 470, 473
- Georgetown, 468
- Georgia, 161
- geospatial, 378
- geospatial coordinates, 383
- Gephi, 474
- gerrymandering, 399, 403
- ggplot2 package, 4
 - themes, 313
- ggplot2 plotting commands, 50
- Gibb, Robin, 118
- GIF, 306
- Gilman, Scott, xv
- Gini coefficient, 230–232
- Gîrjău, Maria-Cristiana, xv
- GitHub, 165, 485, 498, 499, 529, 532, 535, 536
- install from, 509
- issues, 483
- learning, 537
- version control, 532, 535, 537
- global environment, 525
- global temperature, 160
- glyph, 35, 61
 - changing, 36
- glyph-ready data, 39
- goodness of fit, 545
- Google, 331, 477
 - BigQuery, 487, 488, 490
 - Cloud Storage, 488
 - Drive, 532
 - Earth, 387, 402
 - Maps, 387, 402
 - n-grams, 448
 - page rank, 452, 465, 467
 - R style guide, 505
 - scale, 490
 - sheets, 532
 - spreadsheet, 103, 122, 329
- Google BigQuery, 487
- Google Cloud Platform (GCP), 488
- Google Docs, 533
- Google Earth, 402
- Google Sheets, 103, 122, 329, 532
- Gordon-Levitt, Joseph, 463, 464
- Gorman, Michele, 31
- governor, 172
- GPS systems, 387
- GPU computing, 482
- grades, 290
- gradient, scale, 559
- grammar
 - data wrangling, 67
 - of graphics, 31, 35, 62
 - regular expressions, 425
- Grand Central Parkway, 409, 410
- graph, 451
 - random, 293, 452
 - theory, 16, 293, 451
 - visualization, 459
- graphic displays and dynamic graphics and graphic devices and visualization task view, 512
- graphic size, changing, 533
- graphical
 - displays, 9
 - elements, 17
 - framework, 12

- perception, 17
ploys, 160
processing unit (GPU), 482
themes, 313
user interface, 499
- graphical models in R task view, 512
- graphics
aesthetic, 35, 36
annotated text, 56
arrow, 56
bar graph, 44
boxplot, 48, 196
cheat sheets, 62
eikosogram, 50
grammar, 31, 35, 62
grid of plots, 40
interactive, 301
layers, 41
legend, 40
mosaic plot, 50
multivariate, 46
pie charts, 45
scale, 18, 39
scatterplot, 46
taxonomy, 50
title, 56
univariate, 43
- graphs
versus tables, 28
- Graunt, John, 3
- great chefs, 16
- greedy strategies, 230
- Greenland, 384
- Greensboro, 399
- grid computing, 481
- grid of plots, 40
- gross domestic product (GDP), 35
- GROUP BY clause, *see* SQL
- grouping, 74
- groups, iteration over, 146
- GUI, 332
- guide to packages, 512
- guidelines
design of simulations, 293
ethical conduct, 176
for data science programs, 490
for statistics programs, 529
- guides, 40
- gun control, 160
- Gutenberg Project, 425, 448
- hacking, 173
- Hadoop, 485
- Hadron Collider, 478
- handling and analyzing spatio-temporal data task view, 512
- Hanna, Maya, xv
- Hansen, Mark, 29, 31
- hard disk, 329
- Hardin, Johanna, xv
- Hardy, Tom, 463, 464
- Harrell, Frank, 509
- Harrison, George, *see* Beatles
- Harvard University, 166, 174, 177
- harvesting, 173
- Hastie, Trevor, 204
- have our cake and eat it too, 345
- Haverford College, 537
- HAVING clause, *see* SQL
- Hawaii, 350
- haystack, 426
- head node, 375
- health
care, 209
records, 172
study, 550
violations, 7, 290, 310
- Health Evaluation and Linkage to Primary Care, 23
- Health Insurance Portability and Accountability Act, 172
- heart attacks, 166
- heat maps, 17
- heat type, 126
- heavy tail, 554
- hedging bets, 245
- height, 48, 551
- HELP study, 21, 45, 110
- help system
R, 500
R packages, 510
- hemisphere, 209
- Herndon, Thomas, 166
- Hesterberg, Tim, 204
- heteroscedastic errors, 553
- heteroscedasticity, 553
- Hewlett-Packard, 331
- hidden layer, 243
- hierarchical clustering, 264, 267
- high-earner, 209, 229
- high-performance and parallel computing with R

- task view, 490
- high-performance and parallel computing
 - with R task view, 512
- high-stakes exam, *see* SAT scores
- HIPAA, 172
- Hippocratic Oath, 159
- histogram, 12, 43
- history
 - R, 499
 - version, 535
- HIV, 103
- Hive, 485
- Hoaglin, David, 559
- hold-out sample, 210
- Hollerith, Herman, 3
- Hollywood, 456, 457, 459, 464
- home run, 6, 92, 96
- homeless, 45
- homogeneous subsets, 230
- homoscedastic errors, 553
- homoscedasticity, 553
- Horton, Alana, xv
- Horton, John, xv
- Horton, Kinari, xv
- hospital, 377
- hospitalization, 172
- hot dog eating, 317
- hour, minute, second format, 129
- House of Representatives, 7, 13
- house prices, 126, 173
- how to lie with statistics, 160
- Howe, Bill, 483
- HTML, 122
 - files, 122, 389
 - format, 445, 533
 - tables, 124
- htmlwidgets, 301, 308, 322, 389
- Huff, Darrell, 160
- humidity, 487
- Hunt's algorithm, 230
- Hvitfeldt, Emil, 448
- hybrids, 265
- hydrological data and modeling task view, 512
- hypothesis, 209, 281, 553
 - null, 204, 283
- hypothesis testing, 202
- IBM Corporation, 3
- ice cubes, 24
- IDE, 499
- idioms for data, 67
- if statement, 500
- IGNORE clause, *see* SQL
- Ihaka, Ross, 499
- imaging, medical, 218
- IMDb, *see* Internet Movie Database, *see also* Internet Movie Database
- immune response, 174
- implementation, SQL, 563
- importance, 238
 - of captions, 28
- IN operator, *see* SQL
- in operator, 103
- in statement, 500
- in-sample, 217
- inches, 548
- income, 209, 229
- independence, 553
- independent variable, 541
- indexing, 140
 - lists, 502
 - SQL, 337
 - vector, 426, 501
- indicator, 545, 547
- indicator, economic, 288
- indices, 365, 366
 - R, xii
 - subject, xii
- induced subgraph, 471
- infarction, myocardial, 166
- inference, 183, 541, 552
 - bootstrap, 190
 - causal, 204
 - conditional, 281
 - modern, 204
 - perils, 202
 - trees, 258
- inferences, 552
- inflation, 209
- influential point, 554
- information, 230
- information gain, 230
- ingesting data, 121, 125
- ingesting text, 445
- Inkscape, 317
- inner join, 344
- insert data, 568
- INSERT statement, *see* SQL
- insights from outliers, 194
- inspections, restaurant, 290, 310
- Instagram, 135, 331

- installing
 packages, 508
 R, 499
 RStudio, 500
 SQL server, 363
- Institute for Quantitative Social Science, 177
- instructing computers, 532
- insurance records, 172
- integer codes, 127
- integrated development environment, 499
- integrity, 177
 ethical, 159, 537
 referential, 92, 366
- intellectual traditions, 4
- intended audience, xi
- interaction model, 549
- interactive
 graphics, 301
 maps, 389
 tables, 302
 web applications, 308
- intercept, 541
- International Amateur Athletics Federation, 125
- International Business Machines Corporation, 3
- Internet Movie Database, 456
- internet use, 35
- interpretation
 regression model, 547
- interpreted languages, 480
- interpreted programming language, 480
- intervals
 confidence, 153, 188, 519
 presidential terms, 70
- introduction
 R, 499
 RStudio, 499
- Ioannidis, John, 531
- irreproducibility, 531
- issues, tracking, 535
- iteration, 139, 151
 over groups, 146
- iterative process, xiv
- James, Bill, 7, 78, 147
- Japanese nuclear reactors, 132
- Java, 486
- Javaid, Azka, xv
- JavaScript, 301, 302, 317, 389
- Jessie, 58
- Jets, New York, *see* New York Jets
- jitter points, 557
- job title, 170
- job, best, xi
- jobs report, 288
- Johnson, Lyndon, 71
- join, 174
 left, 348
 tables, 89, 95, 98, 110, 127, 174, 344, 368, 397
- JOIN clause, *see* SQL
- Jordan, Michael, xi, 4
- Joseph, 54
- Josephine, 57
- journalism, 99, 161
- JSON, 122, 483
 files, 122
 objects, 483, 489
- Julia, 489
- jurisdiction, 159
- k-fold cross-validation, 217
- k-means, 267
- k-nearest neighbor, 239, 255
- Kadir, 105
- Kaggle, 467
- Keisha Lance Bottoms, 162
- Kemp, Brian, 162
- Kennedy, John, 71
- Kentucky University, 468, 470
- kernel smoother, 43
- Kevin Bacon number, 456, 463
- key, 89, 92, 95, 365
 primary, 364
- key-value pairs, 489
- Keyhole Markup Language, 402
- keys, 89, 92, 365
- Kim, Albert, xv
- Kim, Albert Y., 173
- Kim, Eunice, xv
- Kimmel, John, xv
- King of Scotland, 429
- King, Stephen, 448
- King-Hall, Stephen, 448
- Kirkegaard, Emil, 177
- Kleinman, Ken, xv
- KML, 380, 402
- knitr, 532
- knowledge
 domain, 6

- Knuth, Donald, 532
 Kusiak, Caroline, xv
 Königsberg bridges, 451
 La Salle, 473
 lab notebooks, 531
 label, class, 230
 labels, 208
 Labour Party, 275, 277
 lack of fit, 236
 Lady Macbeth, 429
 LaGuardia Airport, 410
 Lahman, Sean, 7, 76, 99
 Laird, Nan, 456
 Lambert conformal conic, 385
 Lambert projection, 385
 Lambert, Diane, xi
 Lancet, 166
 language processing, 425
 language, structured query, 325
 languages
 - interpreted, 480
 Large Hadron Collider, 478, 479
 large networks, 451
 large numbers, law of, 522
 laser sensor, 541
 LASSO, 216, 255
 lasso, 216
 latitude, 24, 269, 377, 382
 lattice graphics, 35
 law of large numbers, 522
 law, stand your ground, 160
Lawless, 464
 lawyer, 163
 layer, 46
 layers, 18, 41, 243, 381, 389
 lazy evaluation, 529
 lazy learners, 239
 leaflet, 302, 389
 learning from data, xi
 learning R, 500
 learning, statistical, 207, 258, 263
 Least Absolute Shrinkage and Selection Operator (LASSO), 255
 least squares, 541
 left join, 344, 348
 LEFT JOIN clause, *see* SQL
 legal negotiations, 167, 170
 legal system, 170
 legends, 40
 legislature, Florida, 160
 length, 20
 length of office, 70
 Lennon, John, *see* Beatles
 less volume, more creativity, xiii
 lethal force, 160
 levels, 108
 leverage, 554
 lexicon, 436, 448
 Lexington Street, 382
 Li, Priscilla, xv
 liberal arts colleges, xiv
 Liberal Democrats, 275
 libraries, 480
 library
 - help, 510
 - R, 508
 life tables, 54
 LIMIT clause, *see* SQL
 linear algebra, 273, 466
 linear models, 196–199, 541
 linear regression, *see* regression
 linear relationship, 545
 linearity, 553
 linguistics, 425
 LinkedIn, 135, 331
 links, 466
 Linux, 563
 - installation, 500
 list columns, 114
 list files, 265, 371
 list object, in R, 125
 list of data sets, 493
 list-columns, 115
 lists, 124, 502
 - extract elements, 503
 literate programming, 532
 literature, 31
 LOAD DATA statement, *see* SQL
 load files, 121
 load into SQLite, 571
 loading data, 370
 local regression, 154
 locally optimal strategies, 230
 locations, 409
 log scale, 13, 231
 logic of science, 531
 logical operator, 502
 logical vectors, 430
 logistic regression, 198, 199, 209, 217, 219, 221, 225, 246, 556
 logit, 215

- London, England, 3, 377
long format, 104
Long Island Expressway, 409, 410
longitude, 24, 269, 377, 382
loop, 139, 140
loops, 139, 141
Lopez, Robin, 118
Los Angeles Angels, 144
Los Angeles Angels of Anaheim, 144
Los Angeles Dodgers, 93, 94
Los Angeles, CA, 266
love, 447
low-rank, 271
lung cancer, 52, 53, 199, 282
lurking variables, 200
- Mac OS X, 563
Macalester College, 270, 407
Macbeth, 425
MacDuff, 426
machine learning, 4, 171, 207, 218, 263, 278, 478
 library, 487
machine learning and statistical learning
 task view, 278, 512
MacLeish field station, 7, 48, 487, 493
Maclester College, xiv
Madness, March, 467
MAE, 220
magic numbers, 140
Major League Baseball, 76, 92, 93, 139
majority rule, 237, 239
making up data, 281
male names, 58
malware, 173
managing data, 106
managing packages, 537
map family, 143, 481
map projections, 407
mapped, 36
MapReduce, 482, 484, 485
maps, 18, 52, 377, 382
 choropleth, 52, 397
 dynamic, 389
 projections, 403
 proportional symbol, 400
 road, 266
March Madness, 467
Marchi, Max, 99
margin of victory, 468
MariaDB, 331
- mark, 35
markdown, xiv, 174, 302, 532
 cheat sheets, 537
 output format, 535
 templates, 533
Markov Chain Monte Carlo, 296
Markov process, 466
Martin, Paul, 277
Martin, Trayvon, 162
masked, 509
masking, 505, 509
mass, movable, 467
Massachusetts, 16, 23, 48
 governor, 172
Massachusetts Group Insurance Commission, 172
mastering data science, 493
matching characters, 427
math SAT scores, 20, 44, 46
mathematical statistics, 204
mathematics, 5, 489
MathSciNet, 456
MATLAB, 122
matplotlib, 20, 32
matrix, 209, 271
 adjacency, 471, 473
 algebra, 273
 confusion, 212, 217
 distance, 266
 document term, 440
 extract elements, 503
 indexing, 503
 multiplication, 473, 502
 R, 503
 sparse, 444
 transition, 466
 voting, 277
maximum delay, 184
mayoral election, 106
McCartney, Paul, *see* Beatles
McIntyre, Mike, 394
McNamara, Amelia, xv
mdsr package, 493
mean, 186, 188
Mean absolute error, 220
mean absolute error (MAE), 220
measure of evidence, 203
measure, distance, 451
mechanism, 281
median, 186
medical image analysis task view, 512

- medical imaging, 218
 medical procedures, 41
 medical records, 172
 Medicare, 41
 medium data, 325, 479, 493
 meeting times, 286
 melanoma, 52, 53
 memory (RAM), 329
 memory, big, 479
 Mercator projection, 384, 401
 Merck, 166
 merge tables, 174, 368, 397
 merged cells, 104
 mergers, airline, 354
 merging tables, 89, 344
 Mescon, Cory, xv
 meta-analysis task view, 512
 metacharacter, 427
 metadata, 174, 388, 431, 505
 method overloading, 72
 methodology, statistical, 183
 methods, 505
 Mets, New York, *see* New York Mets
 Mexico, 350
 Michael I. Jordan, xi, 5
 microarray, 282
 microarrays, 282
 microcomputer, 5
 Microsoft, 331
 Microsoft Excel, 103, 122, 169, 265, 478, 532
 Microsoft SQL Server, 331
 Microsoft Word, 532, 533
 format, 533, 535
 Middlebury College, 173
 miles, 266
 miles per gallon, 264
 Milgram, Stanley, 456
 Miller, Arthur, 29, 31
 Minaya, Omar, 81
 minimizing errors, 528
 minimum, 119
 mining text, 31, 173, 426
 Minitab, 122, 329
 Minneapolis, 106–108
 Minneapolis–St. Paul, 342
 Minnesota Democratic–Farmer–Labor Party, 11
 misleading graphics, 160
 misnomer, 380
 missing data, 92, 99, 140, 204, 401, 543
 missing data task view, 512
 mistakes, 176
 MIT, 172
 MLB, 76
 MLBAM, 365
 mode of storage, 505
 model
 assessment, 172
 assumptions, 553
 ensemble, 243
 evaluation, 216, 221, 246
 formula, 208
 interaction, 549
 linear, 198
 logistic, 198, 209, 221, 246, 556
 non-linear, 550
 null, 212, 221, 236, 246, 283, 543
 overfitting, 216, 235
 predictive, 209
 regression, 46, 197, 199, 207, 541
 regularization, 255
 selection, 255
 statistical, 196, 389
 to generate data, 281
 visualization, 253
 model deployment with R task view, 512
 modern inference, 204
 modesty, 203
 modularity, 293
 MonetDB, 331
 MonetDBLite, 331
Moneyball, 7
 MongoDB, 331, 489
 monitor, MySQL, 565
 month, 50
 Montreal, 350
 more creativity, less volume, xiii
 morning, 196, 339
 Morocco, 125
 mortality, 3
 Morton Thiokol, 24
 mosaic plot, 50
 Mosteller, Fred, 456
 mouse-over, 302
 movable mass, 467
 movement of athletes, 29
 movies, 31, 456
 multidimensional data, 67
 multiple comparisons, 175, 203, 204, 284
 multiple edges, 473
 multiple regression, 487, 541, 546

- multiple tables, 110
multiples, small, 18
multiplication, matrix, 473
multivariate displays, 46, 207
multivariate statistics task view, 512
murders, 160
myocardial infarction, 166
MySQL, 331, 360, 457, 486, 488, 563
 monitor, 565
 server, 326
MySQL Community Server, 564
MySQL Workbench, 332, 568
- N-gram, 439, 448
naïve Bayes, 241–243, 255
name conflicts, 505, 509
named arguments, 508, 521
named lists, 502
names
 baby, 7, 53, 60, 105, 302, 371, 486
 gender neutral, 118
 special characters, 133
naming conventions, 120
naproxen, 166, 170
narrow data, 112, 113
NASA, 24
National Academy of Sciences, 167, 176
National Cancer Institute, 52
National Football League, 19
National Health and Nutrition Examination Study, 550
National Health and Nutrition Examination Study (NHANES), *see NHANES*
National Health and Nutrition Examination Survey, 154
National League, 77
National Party, 275, 277
National Review, 160
native file format, 121
natural language processing, 425
natural language processing task view, 512
nay votes, 276
NCAA, 467
nearest neighbor, 239
needle, 426
needle in haystack, 426
negative results, 174
negligence, 176
negotiations, legal, 167
- neighbor, nearest, 239
nested, 69
nesting, 77
nesting function calls, 521
network, 52, 451
network science, 4, 6, 52, 53, 243, 293, 451
 history, 452
networks, 16
neural network, 243
neutral assessment, 163
neutral names, 118
New England Journal of Medicine, 166, 170
New Jersey, 41, 42
New York City, 91, 135, 184, 196, 290, 310
New York Jets, 19
New York Mets, 76, 80, 409
New York Times, 16, 288, 301, 317
New York Yankees, 149
New Zealand (Aotearoa), 499
Newark, 198
Newark Liberty International airport, 347
next statement, 500, 527
NHANES study, 47, 208, 223, 550
Nixon, Richard, 71
node, 243
nodes, 124, 293
Nolan, Deborah, 31
non-farm payroll, 288
non-linear models, 550
non-standard evaluation, 526, 529
nondisclosure, 172
normal distribution, 190, 285
normal forms, 348
normal random variables, 288
normality assumption, 554
normality of residuals, 553
normalization, 401
North Carolina, 391, 393, 397, 402
Northampton, MA, 208, 407
Norwottuck rail trail, 409
NoSQL, 489
notebooks, lab, 531
nouns, 29
NP-complete, 230
NSE (non-standard evaluation), 526
nuclear reactors, 132
null hypothesis, 202, 281, 553
null model, 211, 212, 221, 236, 246, 283, 543
number, parsing, 133

- numeric, 17
 - scale, 18
- numeric vector, 520
- numerical mathematics task view, 512
- NumPy, 489
- nutrition study, 550
- Nuzzo, Regina, 205
- NVIDIA, 482
- NYC Open Data, 135
- O(n), 231, 366
- O-rings, 24
- Oakland A's, 7
- oath, Hippocratic, 159, 176
- Obama, Barack, 10, 391, 452
- object-oriented programming, 505
- objects
 - coercion, 503
 - displaying, 509
 - list, 125
 - R, 501
 - reactive, 308
 - remove, 501
 - rules for naming, 120
- oblate spheroid, 383
- obligations, 159
- observation, 105, 109
- observational data, 5, 202, 205
- ocean levels, 161
- Octave, 122
- Odum Institute for Research in Social Science, 174
- OFCCP, 164, 168
- offensive statistics, 92
- Office of Civil Rights, 172
- Office of Federal Contract Compliance Programs (OFCCP), 164, 168
- official statistics and survey methodology task view, 512
- Ohio, 15
- oil, number of barrels, 35
- OkCupid, 165, 169, 173
- OLAP, 67
- omitted data, 25
- On-Base Percentage, 96
- On-Base Plus Slugging Percentage, 96
- online library, 448
- Open Data NYC, 135
- Open Science, 177
- open source, 485
- Open Street Maps, 387
- OpenCL, 482
- OpenIntro Statistics, 186, 559
- OpenPsych Forum, 165
- OpenStreetMaps, 302
- operations research, 218
- operations, vector, 139
- OPS+, 97
- optimization and mathematical programming task view, 512
- option file, 566
- optional argument, 520
- options in R, 507
- OR operator, *see SQL*
- or operator, 502
- Oracle, 331
- oracle, Kristen Stewart, 463
- ORDER BY clause, *see SQL*
- order, descending, 74
- orders of magnitude, 478
- ordinal variable, 73
- Ordnance Survey, 387
- Ordway birds, 126
- other factors, 198, 199
- Ott, Mel, 6
- out-of-sample, 217
- outcome variable, 541
 - measured, 208
 - unmeasured, 207
- outer join, 344
- outliers, 194, 554
- output
 - file, 532, 533
 - format, 535
 - formatted, 535
- ovarian cancer, 52, 282
- overfitting, 216, 235, 251, 255
- overload, cognitive, xiii
- overloading, 72
- Owen, Melody, xv
- p-values, 170, 175, 197, 203, 238, 553
 - perils, 202
 - reporting, 203
- PAC, 9
- Pacific Time Zone, 91, 351
- packages
 - conflicts, 509
 - guide, 512
 - help, 510
 - R, 508
 - remove from workspace, 510

- used in book, 493
- versions, 510, 537
- packrat, 537
- Page, Larry, 466
- PageRank, 452, 465–467, 470, 473
- pairwise relationships, 52
- palettes, color, 19, 397
- Pandas, 489
- Pandoc, 533
- panel background, 314
- panel grid, 314
- papers in data science, 431
- paradigm of science, 531
- paradigm shift, 485
- paradox, Simpson's, 201
- parallel computation and high-performance computing
 - task view, 490
- parallel computing, 479, 481, 482
- parallel planes, 548
- parallel slopes, 547
- parallelized, 481
- parameter tuning, 245
- parameters
 - conditional regression, 547, 559
 - regression, 541
 - tuning, 236
- parametric functions, 229
- parentheses, 445
- parents, 53
- park factor, 142
- parking lot, 409
- parse number, 133
- PARTITION defintion, *see* SQL
- partitioning, 217, 231, 369
 - recursive, 230, 253
- party affiliation, 11, 270
- passwords, 565
- path, 123, 451
- pathological variable names, 445
- paths, 451
 - forking, 204, 531
- patterns, 183, 184
 - counting, 426
- patterns in data, 9
- payroll, 288
- PCA, 273, 278
- pdf
 - creating, 533, 535
 - format, 533, 535
- Pearce, Guy, 464
- Pearson correlation, 220
- peer review, 531
- perceived threats, 160
- percentages, 13
- percentile, 185, 193
- perception, graphical, 17
- performance, 480
 - query, 363
- perils of extrapolation, 25
- perils of p-values, 202
- permission, read, 374
- permutation tests, 5, 6, 283, 292
- Perrett, Jamis, 31
- Perry, William (Refrigerator), 29
- persistent truth, 531
- personal hygiene, 290
- petabytes, 477, 478
- Peters, Arno, 384
- Pew Research, 162
- pgAdmin, 568
- phase transition, 453
- PHI, 172
- PHP, 301
- phylogenetics, especially comparative methods task view, 512
- physical experiment, 531
- physician, 159, 377
- π , 120
- pie charts, 17, 21, 45
- Pig, 485
- Pioneer Valley, 541
- Pioneer Valley Planning Commission, 541
- pipe operator, 4, 70, 77, 103, 522
- pipeline, 70, 326
- piping, 77
- pivot data, 112
- plaintiff, 167
- planes, 548, 549
- plays, 426, 488
- Plot.ly, 302
- plotting text, 36
- plurality, 239
- Poisson distribution, 286, 455
- Poisson random variable, 285, 454
- Polar coordinates, 18
- police, 171
- political action committees, 9
- political parties, 275
- political science, 174, 403
- polygons, 380, 401
- polynomial time solution, 230

- Ponce, 349
 Poole, Jane, 109
 population, 35, 105, 183, 185, 392
 population parameters, 541
 popups, 390
 Porta, Rose, xv
 position, 21, 40
 PostgreSQL, 331, 342, 360, 488, 563
 POTUS, 452
 power-law, 455
 PowerPoint, 31
 precinct, 106
 precipitation, 548
 precursors to data science, 477
 predicate, 413
 predicate function, 142
 predicted values, 541, 553
 prediction error, 219
 predictions, 207, 216
 predictive analytics, 8, 207, 209, 216, 258, 263
 predictive model, 6, 208, 209
 predictor variable, 541
 predictors, categorical, 545
 preferential attachment, 455
 preprints, 431
 presentations, 28, 31
 in RStudio, 535
 President Obama, *see* Obama, Barack
 presidential elections, 9, 391
 presidents, 7, 10, 67
 pressure
 air, 487
 blood, 112
 prices, home, 173
 primary key, 95, 365
 PRIMARY KEY definition, *see* SQL
 principal component analysis, 273, 278
 principles, 159
 privacy, 159, 169
 policies, 172
 private repositories, 536
 privileged communications, 171
 privileges, SQL, 566
 probability, 50, 184, 209, 218, 557
 conditional, 241
 probability distributions task view, 512
 probes, 282
 probes, genetic, 52, 282
 problem-solving, xi, 519
 processing and analysis of tracking data
 task view, 512
 processors, 481
 product-moment correlation, 220
 productivity, xiii
 economic, 35
 professional ethics, 159, 176
 programming, 4
 defensive, 528
 programming interfaces, 122
 programming, literate, 532
 PROJ.4, 385, 387
 Project Gutenberg, 425, 448
 Project TIER, 537
 projection, 383, 389
 Albers, 385
 Gall-Peters, 384
 Lambert, 385
 Mercator, 384, 401
 projections, 377, 403
 projects, 499, 535
Prometheus, 464
 proportional symbol map, 400
 proportions, 45
 prostate cancer, 282
 protected health information (PHI), 172
 protocol, 170
 clinical trial, 176
 CONSORT, 177, 205
 protocols, 204
 provenance of data, 105, 121
 proxy, 171
 Pruijm, Randall, xv, 477
 pruning, 235
 pseudo-random number seeds, 293, 294
 psychometric models and methods task view, 512
 puberty, 551
 public data set, 165
 public health, 103, 389
 public policy, 169
 public repositories, 536
 publication bias, 175
 published studies, 531
 Puerto Rico, 349, 370, 392
 pump, 378, 389, 403
 punch cards, 3
 purity, 230
 Pythagorean Winning Percentage, 147
 Python, 20, 140, 480, 483, 484, 489

- QGIS, 402
qq plot, 554
qualitative palette, 19
quantifying patterns, 183
quantiles, 192, 193
plot, 554
t-distribution, 521
quantitative
 outcome, 556
 variable, 108
Quayola, 29
Queens, 410
query language (SQL), 325
query optimization, 327
query performance, 363
query, search, 465
quotation marks, 445
- R, xiii, 480, 489, 499
 accessing databases, 568
 accessing variables, 526
 data structures, 501
 Development Core Team, 499
 documentation, 514
 environment, 525
 FAQ, 500
 Foundation for Statistical Computing, 499
 help system, 500
 history, 499
 index, xii
 installation, 499
 introduction, 499
 libraries, 508
 Linux installation, 500
 Markdown, 174, 302, 532, 535
 Markdown cheat sheets, 537
 objects, 501
 packages, 508, 510, 521
 questions, 500
 style guide, 120, 135, 505
 task views, 512
- R^2 , 220, 545
race and crime in the United States, 171
race, running, 109
radial, 23
radial coordinate system, 23
rail trail, 409, 541
rain, 548
raison d'etre, 5
Raleigh, 399
RAM, 329, 481
Ramirez, Manny, 92, 95
random chance, 203
random classifier, 219
random forest, 237, 255
random graph, 293, 452, 453
 Erdős–Rényi, 453
random noise, 183, 289
random number, 330
 exponential, 285
 Gaussian, 285
 normal, 288
 Poisson, 285
 seed, 293, 294, 523
 uniform, 285
random restart, 474
random sample, 183, 210, 221, 246, 285
random selection, 184
randomization procedure, 292
randomized trials, 199, 203, 204
rank, 271
rank choice voting, 109
rank correlation, 220
rate, false discovery, 284
ratio
 data to ink, 45
 of name frequency, 119
RBI, 92
reactive objects, 308, 309
read files, 121
read permission, 374
readable files, 532
Reagan National Airport, 348, 369
Reagan, Ronald, 370
real estate database, 173
real-world case studies, xii
receiver operating characteristic, 217, 218
recidivism, 171
records, world, 20, 124
rectangular array, 105
recursion, 230
recursive partitioning, 230, 253
recycled, 501
Red Sox, 93
Reddit, 331
reduction, dimension, 270
reference category, 546
reference group, 546
reference system, coordinate, 385
referencing variables, 218
referential integrity, 92, 366

- regional airline, 354
- registries, clinical trial, 176
- regression, 46, 197, 199, 207, 208
 - assumptions, 553
 - coefficients, 554
 - inference, 552
 - interaction, 549
 - logistic, 198, 209, 217, 219, 221, 225, 246, 556, 557
 - model, xii, 543
 - multiple, 487, 541, 546
 - parameters, 541
 - regularization, 255
 - residuals, 541
 - ridge, 216, 255
 - simple linear, 541
 - tree, 209, 230
- regular expression, 426, 434
- regular expressions, 31, 316, 317, 425, 426, 448
- regularization, 216, 255
- rehearse, 28
- reidentification, 169, 172, 173
- Reinhart, Carmen, 166, 169
- reject null, 203
- relational calculus, tuple, 331
- relational database, 67, 89, 110, 122, 325, 330, 365
- relationships between variables, 13
- relationships, pairwise, 52
- relative humidity, 487
- relevance, 465
- reliability, 169, 189, 193
- remove
 - data frame from workspace, 505
 - objects, 501
 - package from workspace, 510
- Renacci, Jim, 15
- renaming variables, 72, 218
- reorder factor levels, 48
- repeat statement, 500
- repeated measures, 112
- repetitions, 429
- REPLACE statement, *see* SQL
- replicability, 531
- replication
 - crisis, 531
- replication crisis, 531
- reporting p-values, 203
- repositories, *see* GitHub
- reproducibility, 293, 531, 537
- reproducible analysis, xii, xiv, 104, 174, 175, 177, 499, 531, 532, 535
- packages, 512
- spreadsheets, 166, 169
- task view, 533
- workflow, 121, 536
- reproducible research, 174, 205, 531, 537
- reproducible research task view, 512
- reproducible results, 204
- Republican, 10, 69, 391, 394
- resampling, 6, 152, 190, 283, 285
- research
 - reproducible, 537
 - responsible conduct, 176
- reserved commands, 500
- reset values, 369
- reshape, 113, 114
- residual diagnostics, 554
- residuals, 541, 553
- response variable, 198, 208, 263, 264, 541
- responsible conduct of research, 176
- restaurant inspections, 7, 290, 310
- result, exact, 287
- retaliation, 172
- retraction, 175, 177
- retrieve old versions, 535
- Retrosheet, 478
- return value, 521
- Reuters, 160
- revision control, *see* GitHub
- rho, 220
- Rickey, Branch, 6
- ridge regression, 216, 255
- right join, 344
- right-skewed, 24
- Ripken, Cal, 98
- Riseman, Julia, xv
- Riseman, Tanya, xv
- rising ocean levels, 161
- Rmd file, 535
- RMSE, 219
- road maps, 266
- Robinson, David, 448
- Robinson, Jackie, 6
- robust, 186
- robust statistical methods task view, 512
- ROC curves, 217
- Rogers Commission, 24
- Rogoff, Kenneth, 166, 169
- Rome, 125
- Romney, Mitt, 10, 16, 391

- Ronald Reagan Washington National, 348, 370
Roosevelt Avenue, 410
root mean squared error (RMSE), 219
root privilege, 565
Root-mean-square error, 220
rOpenSci, 135
Rosling, Hans, 103
Roth, Allan, 6, 7
rounding, 218, 553
routes, commuting, 409
Rouzer, David, 394
row, 109
rownames, add, 238
RSeek, 500
RStudio, xiii, xv, 301, 499, 535
 cheat sheets, 62, 84, 322, 537
 curated guide to learning R, 500
 developers, xv
 Environment tab, 525
 installation, 500
 presentations, 535
 reproducible analysis, 535
Rubin, Ben, 31
Ruby, 301
rules for naming, 120
rules of baseball, 6, 92
run ratio, 147
running average, 522
running race, 109
 world record, 124
runs batted in (RBI), 92
runs scored and allowed, 77, 147
Ruth, Babe, 6
Ryan, Paul, 166
Rényi, Alfréd, 452

sabermetrics, 6, 7, 147
sacrifice flies, 142
safety of patrons, 290
salary, teacher, 46, 199
sales, houses, 126, 173
sample, 183, 185, 285
 size, 184, 188, 294
 statistics, 186
 with replacement, 190
sample size, 188
sample with replacement, 190
sampling, 285
sampling distribution, 186, 188
San Francisco, 173, 184, 196
San Juan, 349
Saratoga, NY, 126
SAS, 122
SAT, 20
SAT scores, 20, 44, 199
 percentage taking, 46
save
 files, 121
Scala, 489
scalability, 375
scalar, 140
scale, 17, 18, 39
 fill, 559
 log, 13
 vs guide, 18
scale-free, 455
scaled percentages, 13
scatterplot, 15, 46, 199
 jitter points, 557
 smoother, 154, 550
scheduled departure time, 196
schema, 365, 372, 571
scheme, color, 397
Scholastic Aptitude Test, 199, *see* SAT scores
science
 network, 451
 of learning from data, xi
scientific consensus on climate change, 162
scientific method, 531
scikit-learn, 489
scoping, 525
score, 468
Scotland
 King of , 429
 Parliament, 270, 271
 political parties, 275
Scottish Conservative and Unionist Party, 275
Scottish Labour, 275
Scottish Liberal Democrats, 275
Scottish National Party, 275
Scottish play, 425
scraping text, 445
scriptable computing, 532
scripts, 567
search box, 302
search engines, 440
search query, 465
search, Web, 465

second, 96
 security, root privilege, 565
 seed, 294
 seeds, random number, 293, 294
 seeds, tournament, 467
SELECT statement, *see SQL*
 selection bias, 543
 self-intersections, 451
 Senate, 7
 sensitivity, 218
 sensor
 laser, 541
 network, 4
 sentiment analysis, 436, 448
 sequential palette, 19
 server
 database, 563
 file, 308
 installation, 363
 MySQL, 326
 version of RStudio, 499
 session information, 510
 SET clause, *see SQL*
 set inclusion operator, 103
 setting up database server, 363, 563
 Seven Bridges of Königsberg, 451
 sewer type, 126
 SFO, 184
 Shakespeare, 31, 425, 426, 488
 Machine, 31
 Shakespeare, William, 425
 Shalizi, Cosma, 204
 shape, 21, 188, 383
 shapefile, 380
 shapefiles, 377, 380, 395, 403, 407
 sharing results, 176
 Shea Stadium, 409
 shell, 564
 Shenandoah National Park, 420
 Shiny, 308, 390, 499
 cheat sheets, 322
 deploying, 310
 gallery, 322
 shortcomings of PowerPoint, 31
 shortest path, 451
SHOW INDEXES statement, *see SQL*
 show indices, 367
 show keys, 365
SHOW KEYS statement, *see SQL*
 shp files, 380
 shrinkage, 255, 256
 shuffling, 283, 285
 shx files, 380
 significant result, 175
 Silge, Julia, 448
 Silver, Nate, 352, 353
 similarity, 263
 simple features, 381
 Simpson's paradox, 201, 202
 simulation, 184, 281, 296
 null hypothesis, 202
 principles, 293
 singer, 445
 single, 96
 singular value decomposition, 273
 singular value decomposition (SVD), 273
 sinks, 473
 six degrees, 456
 Six Degrees of Separation, 456
 size, 38
 effect, 203
 skills
 communication, xi
 SkyWest, 354, 356
 Slate.com, 173
 slides in RStudio, 535
 slope, 17, 541, 543, 547, 551
 different, 549
 parallel, 547
 Slugging Percentage, 96
 small multiples, 18
 small world, 456
 small-world, 456
 Smith College, xiv, 7, 8, 173, 407, 487, 493
 Smith, Will, 456
 smoother, 43, 46, 154, 550
 snake case, 72, 120
 snake_case, 72, 121
 snow, 548
Snow White and the Huntsman, 463, 464
 Snow, John, 31, 377, 403
 social hops, 456
 social network, 451
 social science, 537
 Social Security Administration, 53, 105
 socially conservative, 276
 society, 176
 Socrata, 135
 software tools, 531
 song
 titles, 445
 writers, 446

- sorting, 73, 367
sorting arrows, 302
Sosa, Gabriel, xv
source code control, *see* GitHub
source file, 532
Southwest, 354
space shuttle, 24
Spark, 485, 490
sparse matrix, 444
spatial data, 6, 377, 403
spatial projection, 383
spatial statistics, 377
Speaker of the House, 15
speaking line, 426
Spearman correlation, 220
special cases, 519
special characters in variable names, 133
specificity, 218
specifying problems, 519
speculation to data, 281
spheroid, 383
spider trap, 473
spider traps, 473
spidering, 173
split string, 425
splitting, 231
spread, 186
spreading data, 112, 113
spreadsheet, 7, 104
spreadsheets, 7, 329, 478, 532
 data organization, 135
 Excel, 103, 122, 169, 265
 Google, 103, 122
 perils of, 531
 reproducibility, 166, 169
SPSS, 122, 329
SQL, xii, 67, 325, 326, 480, 488, 493, 563
 administration, 363
 ALTER TABLE, 365
 ALTER TABLE statement, 365
 AND, 338
 AND operator, 338
 and R, 360
 AS, 335, 346
 AVG(), 338, 360
 baseball, 478
 BETWEEN, 337, 338
 BETWEEN operator, 337
 CHANGE clause, 365
 CONCAT function, 328, 337
 CONCAT(), 328
COUNT, 338
COUNT(), 338
CREATE DATABASE, 364
CREATE DATABASE statement, 364, 374
CREATE TABLE, 364, 365, 372
CREATE TABLE statement, 364, 372
 creating databases, 363
CROSS JOIN, 345
DESCRIBE, 364
DESCRIBE statement, 332, 364
dialects, 331
DISTINCT, 337
DROP TABLE, 372
DROP TABLE statement, 373
efficient, 343
errors, 374
EXPLAIN, 367
EXPLAIN statement, 367, 368, 568
FOREIGN KEY definition, 365
FROM, 334, 346
 FROM clause, 327, 333, 368
GROUP BY, 338, 358
 GROUP BY clause, 327, 333, 338
HAVING, 342, 343
 HAVING clause, 327, 333, 342
IF EXISTS clause, 373
IN, 338
 in a nutshell, 375
IN operator, 338
indexing, 337
INSERT, 369, 370
 INSERT IGNORE, 370
 INSERT INGORE, 370
 INSERT statement, 370
 IGNORE clause, 370
install, 363
JOIN, 344–346, 348, 349, 351
 JOIN clause, 327, 333, 344, 346
 CROSS JOIN clause, 344, 457
 LEFT JOIN clause, 344, 348, 368, 457
 OUTER JOIN clause, 344
 RIGHT JOIN clause, 344
KEY definition, 365
keys, 365
LEFT JOIN, 345, 349
LIMIT, 344
LIMIT clause, 333, 343
LOAD DATA, 370–373

- LOCAL INFILE clause, 373
- LOAD DATA statement, 370, 373
- MAKETIME(), 340
- MAX(), 338
- Microsoft, 331
- MIN(), 340
- MongoDB, 331
- normal forms, 348
- NULL, 345, 364, 366
- OR, 338
- OR operator, 338
- Oracle, 331
- ORDER BY, 341
- ORDER BY clause, 327, 333, 341
- OUTER JOIN, 344
- PARTITION definition, 369
- PostgreSQL, 331
- PRIMARY KEY definition, 365, 372
- privileges, 566
- REPLACE, 370
- REPLACE statement, 370
- RIGHT JOIN, 345, 349
- scalability, 375
- SD(), 360
- SELECT, 334, 363, 369, 374
- SELECT statement, 327, 333, 334, 368, 457, 568
- SET clause, 370
- SHOW
 - CREATE TABLE statement, 365
 - DATABASES statement, 363
 - INDEXES statement, 367
 - KEYS statement, 365
 - TABLES statement, 332
 - WARNINGS statement, 373
- SHOW DATABASES, 363
- SHOW INDEXES, 367
- SHOW KEYS, 367
- SHOW WARNINGS, 373
- SQLite, 331
- STR_TO_DATE function, 337
- STR_TO_DATE(), 336
- subqueries, 350
- SUM(), 338
- table alias, 346
- TERMINATED BY clause, 373
- TO_DATE(), 336
- translation, 327
- UNION, 349
- UNION statement, 349
- UNIQUE KEY definition, 365
- universe, 331
- UPDATE, 369, 370
- UPDATE statement, 369, 370, 568
- USE, 364, 372
- USE statement, 364
- VALUES clause, 370
- WHERE, 327, 336–338, 342, 343, 351
- WHERE clause, 327, 333, 336, 368
- SQLite, 331, 360, 563
- squared residuals, 541, 545
- SSE (sum of squared residuals), 545
- SSM (sum of model squared deviations), 545
- St. Clair, Katie, xv
- St. Joseph's, 473
- stack overflow, 500
- stacked bar plot, 45
- stakeholders, 169, 175
- stand your ground law, 160
- standard deviation, 168, 186, 239
- standard error, 188, 192
- standard evaluation, 526
- standards, ethical, 159, 176
- Stanford University, 301, 466
- Starr, Ringo, *see* Beatles
- Stata, 122, 329, 537
- static graphics, 29
- stationary distribution, 466
- statistic, 186
- statistical computing, 296
- statistical genetics task view, 512
- statistical learning, 171, 207, 263
- statistical methodology, 183
- statistical methods, 3
- statistical modeling, 196
- statistical programming, 529
- statistical significance, 175, 197, 202
- statistics, xi, 5, 183, 186
 - definition, 186
 - how to lie with, 160
 - theoretical, 204
- statistics for the social sciences task view, 512
- step into function call, 527
- steroid era, 97
- Stewart number, 463
- Stewart, Kristen, 456, 463–465
- Stonebraker, Michael, 331
- stop words, 435
- storage mode, 505
- storage, data, 173

- storms, 161
straight line, 554
stratification, 202
streamgraph, 304
strength of association, 183
Strengthening the Reporting of Observational Studies in Epidemiology (STROBE) statement, 205
string, 131
 character, 425
 data, 425
 split, 425
 to date, 337
STROBE statement, 205
Strogatz, Steven, 454
structured query language (SQL), 325
studies, published, 531
Sturgeon, Nicola, 277
style guide, 120, 135
subgroups, iteration, 146
subject index, xii
subquery, 350
subsets, 230
Subversion, 535
Sugar Maple, 419
summarizing, 74
 rows, 338
summary functions, 141
super PAC, 9
superuser, 566
supervised learning, 171, 207–209, 229, 258, 263
Supreme Court, 9, 168
survival analysis task view, 512
Sutton, Betty, 15
SVD, 273
Sweave, *see* knitr
swim records, 20
swimmer, 29
swing state, 391
symmetry, 276
syntax highlighting, 499
Syracuse, 468
systemic racism, 165
systolic blood pressure, 112
- t-distribution, 552
 quantile, 521
t-test, 5, 188
table scan, 368
tables
- alias, 346
data, 302
ephemeral, 351
HTML, 124
life, 54
multiple, 110
tabulate, 185
 versus graphs, 28
tabular content, 124
tabulate, 3, 185
talent, artistic, 29
tall data, 104, 112, 113
Tampa Bay Rays, 93, 94
target time, 356
tartan, 271
task view, 135, 512
 analysis of ecological and environmental data, 512
 analysis of pharmacokinetic data, 512
 analysis of spatial data, 512
 Bayesian inference, 512
 chemometrics and computational physics, 512
 clinical trial design, monitoring, and analysis, 512
 cluster analysis and finite mixture models, 512
 databases with R, 512
 design of experiments (doe) and analysis of experimental data, 512
 differential equations, 512
 econometrics, 512
 empirical finance, 512
 extreme value analysis, 512
 functional data analysis, 512
 graphic displays and dynamic graphics and graphic devices and visualization, 512
 graphical models in R, 512
 handling and analyzing spatio-temporal data, 512
 high-performance and parallel computing with R, 490, 512
 hydrological data and modeling, 512
 machine learning and statistical learning, 278, 512
 medical image analysis, 512
 meta-analysis, 512
 missing data, 512
 model deployment with R, 512
 multivariate statistics, 512

- natural language processing, 512
- numerical mathematics, 512
- official statistics and survey methodology, 512
- optimization and mathematical programming, 512
- parallel computation and high-performance computing, 490
- phylogenetics, especially comparative methods, 512
- probability distributions, 286, 512
- processing and analysis of tracking data, 512
- psychometric models and methods, 512
- reproducible research, 512, 533
- robust statistical methods, 512
- statistical genetics, 512
- statistics for the social sciences, 512
- survival analysis, 512
- teaching statistics, 512
- time series analysis, 512
- Web technologies and services, 512
- tau, 220
- taxonomy of graphics, 9, 17
- teacher salary, 46, 199
- teaching
 - data science, 490
 - integrity in empirical research (Project TIER), 537
 - reproducible research, 537
 - statistics, 99, 490
- teaching statistics task view, 512
- team sport, xiv, 535
- technical ability, 29
- technical review, 174
- technologies, data, xi
- temperature, 25, 50, 160, 208, 487, 542
- templates, markdown, 533
- Temple University, 473
- terabytes, 477
- term frequency, 444
- term frequency-inverse document frequency, 440
- term matrices, 440
- terms, 466
 - of presidents, 70
 - of use, 169, 173
- terms of use, 169
- test
 - characteristics, 218
 - data, 210, 217
 - testing, 216
 - tests, 5
 - hypothesis, 202, 553
 - reporting, 203
 - standard deviation, 168
 - unit, 529
 - Texas, 402
 - Texas Tech, 470
 - text
 - as data, 425
 - corpus, 434, 484
 - data, 6
 - ingesting, 445
 - mining, 29, 31, 173, 425, 426, 448
 - tf-idf, 440
 - The Weather Underground, 173
 - theft, 173
 - theme, 313
 - Theorem, Bayes, 241
 - theoretical statistics, 204
 - Theron, Charlize, 463, 464
 - thinking
 - algorithmic, 4, 519
 - computational, xi
 - statistical, 183
 - third normal form, 348
 - Thorpe, Jer, 29
 - thoughtful, 203
 - three V's, 477
 - threshold, 203, 291
 - threshold function, 454
 - tibble, 115, 238
 - tick marks, 40
 - tidy data, xiii, 103, 105, 108, 135
 - tidy data format, 122
 - tidy data multiple tables, 110
 - tidy data rules, 106
 - tidy data rules for, 105
 - tidy evaluation, 155, 525, 526, 529
 - tidyverse, 67, 84, 479
 - style guide, *see* style guide
 - tilde, 208
 - tile, 389
 - tiles, 387, 388
 - time of departure, 196
 - time scale, 18
 - time series, 48
 - time series analysis task view, 512
 - time variables, 129, 340
 - time zone, 351

- times and dates, 336
timing commands, 141, 479
title, 56
tokenizer, 434
Tolstoy, Leo, 121
tools, software, 531
top billing, 457
Toronto, 350
tortured shapes, 397
tournament, 467
Toyota, 264, 265, 267
track changes, 535
tradeoff
 database vs. disk, 329
 variance and bias, 216, 255, 551
Tragedy of Macbeth, 425
trail crossings, 542
training, 216
training data, 210, 217, 230
transforming data, 105, 106
transition, 466
 phase, 453
transition matrix, 466
translating
 integer codes, 127
 to SQL, 327
translation, 480
transparency, 172, 176
transpose, 113, 114
transposing data, 112, 113
Transverse Mercator, 387
trap, spider, 473
travel policy, 184, 193
tree
 aggregated, 237
 branches, 235, 263
 conditional inference, 258
 decision, 230
 evolutionary, 207, 263
 pruning, 235
 regression, 209
trend line, *see* smoother
triadic closure, 452, 454
trials
 randomized, 199, 204
triple, 96
trucks, 264
true coefficients, 552
true positive, 219
Trump, Donald, 162
truncate output, 343
truth, 222, 248, 531
truthful falsehoods, 160
Tufte, Edward, 24, 25, 28, 31, 316
Tukey, John, 31
Tumblr, 135
tuning parameters, 236
tuple relational calculus, 331
turnout, 106
Turocy, Ted, 365
Twilight, 53
Twitter, 160, 331, 451, 452
type coercion, 503
Type I error, 202, 212
Type II error, 202, 212
U.S. Bureau of Labor Statistics, 288
Ubuntu, 563
UGESP, 164, 168
UMass, 469, 470, 473
uncertainty, 203
undergraduate guidelines, 529
underscores, 120
undirected, 451
uniform distribution, 285, 473
Uniform Guidelines on Employee Selection Procedures (UGESP), 164, 168
uniform random number generator, 285
uniform random variable, 285
UNION statement, *see* SQL
Unionist and Conservative Party, 275
unique, 191
unique cases, 109
unique key, 365
UNIQUE KEY definition, *see* SQL
unit tests, 529
United Airlines, 194, 356
United Kingdom Ordnance Survey, 387
United Nations, 176
United States Bureau of Transportation Statistics, 7, 332
United States Census, 209
United States Census of 1890, 3
United States Constitution, 392
United States Department of Energy, 264
United States Food and Drug Administration, 166
United States House of Representatives, 10
United States Office of Federal Contract Compliance Programs, 164

- United States Senate, 10
- United States Social Security Administration, 7, 105
- United States Supreme Court, 399
- univariate analysis, 25
- univariate displays, 43
- universal resource locator, 123
- University of Auckland, 499
- University of California, Berkeley, 177
- University of Massachusetts, 166, 467
- University of Michigan, 176
- University of North Carolina, 174
- University of Washington, 483
- UNIX epoch, 130
- UNIX shell, 563
- UNIX-like, 563
- unmeasured features, 207
- unnamed arguments, 521
- unstructured data, 425
- unsupervised learning, 207, 263, 278
- update database, 369
- UPDATE statement, *see* SQL
- URL, 123
- user accounts, 564
- user interface, 308
- Vaidyanathan, Ramnath, 301
- validation, cross, 216, 240
- valuations, houses, 173
- VALUES clause, *see* SQL
- varchar fields, 364
- variability, 186
- variable, 105, 108
 - categorical, 108
 - dependent, 541
 - factor, 546
 - independent, 541
 - indicator, 545
 - names, special characters, 133
 - pathological names, 445
 - quantitative, 108
 - renaming, 218
- variance, 216
 - bias tradeoff, 216, 255, 551
 - residuals, 554
- variety, 477
- vector
 - character, 425, 525
 - extract elements, 503
 - indexing, 501
 - numeric, 520
- operations, 139
- recycling, 501
- vectorized, 141
- velocity, 477
- Venn diagrams, 50
- verbs, 67
- version control, *see* GitHub
- version number, 510
- Vertica, 331
- vertices, 52, 293, 451, 459, 467
- vetting, 174
- victims, 163
- victory, margin of, 468
- view task, *see* task views
- violations, health, 290, 310
- Vioxx, 166, 170
- Virgin America, 194
- Virgin Islands, 392
- Virginia Tech, 473
- visual analytics, 29
- visual cues, 17, 35, 60
- Visual Display of Quantitative Information*, 31
- visualization, 4, 9, 183, 377
 - graphs, 459
 - interactive, 301, 322
 - multivariate, 46
- vocalist, 445
- volume, 477
- voting, 106, 109, 276, 399, 466
- Wagaman, Amy, xv
- walk, 451
- Wang, Susan, xv
- war chest, 10
- ward, 106
- Warren, Elizabeth, 16
- Washington National Airport, 348
- Washington, D.C., 109, 370, 389
- Watergate, 69
- Watts, Duncan, 454
- weather data, 7, 493
- weave, *see* markdown
- web application, 6, 301, 308, 329, 499
- Web comic, 316
- Web Crawler, 465
- Web scraping, 121
- Web search, 465
- Web technologies, 135
- Web technologies and services task view, 512

- website for book, [xii](#)
- weekday, [549](#)
- weighted, [451](#)
- Weka machine learning library, [258](#)
- Weld, William, [172](#)
- Well-Known Text, [387](#)
- West Coast, [91](#)
- West, Jevin, [176](#)
- Whately, MA, [410](#)
- WHERE clause, *see* [SQL](#)
- while statement, [500](#)
- White House, [389](#)
- Wickham, Hadley, [xv](#), [17](#), [31](#), [35](#), [62](#), [67](#),
[84](#), [105](#), [120](#), [121](#), [489](#), [514](#)
- wide data, [112](#), [113](#)
- widgets, [309](#)
- Wikipedia, [124](#), [445](#)
- Wilkinson, Leland, [31](#), [62](#)
- Williams College, [126](#)
- Williams, Robin, [118](#)
- Windows, [563](#)
- winning percentage, [77](#), [147](#)
- Winston-Salem, [399](#)
- with replacement, [237](#)
- witness, expert, [171](#)
- WKT, [385](#)
- women's names, [60](#)
- Word
 - format, [533](#), [535](#)
 - Microsoft, [532](#)
- word cloud, [436](#)
- word frequency, [435](#), [440](#), [483](#)
- word pairs, [439](#)
- workflow, [xi](#), [xiii](#), [183](#), [477](#)
 - reproducible, [121](#), [532](#), [536](#)
- workspace, [509](#)
 - browser, [499](#)
 - conflicts, [505](#), [509](#)
- world records, [20](#), [124](#)
- World Series, [80](#)
- World Wide Web, [466](#)
- wrangling, [4](#), [84](#)
 - cheat sheets, [84](#)
 - data, [xi](#), [67](#), [106](#), [526](#)
 - tidy data, [103](#), [108](#)
 - times and dates, [336](#)
- xkcd, [316](#)
- XML, [122](#), [402](#)
- XML files, [402](#)
- Yahoo, [465](#), [478](#)
- Yankees, [149](#)
- Yau, Nathan, [16](#), [58](#), [317](#)
- year, month, day format, [129](#)
- Yu, Jessica, [xv](#)
- Zillow.com, [173](#)
- Zimmerman, George, [162](#)
- ZIP file, [264](#)
- zodiac sign, [165](#)

R index

- ... syntax, 522
- across(), 142, 143, 155, 526, 527
- add_busy_spinner(), 312
- addMarkers(), 389
- addPolygons(), 399
- addPolylines(), 409
- addPopups(), 390
- addTiles(), 389, 399, 409
- adjust option, 43
- aes(), 36, 41, 46, 48, 134, 147, 150, 287, 471, 524
- aes_string(), 155
- airlines package, *see library(airlines)*
- all.equal(), 502
- alpha option, 147
- alr3 package, *see library(alr3)*
- and operator, 502
- annotate(), 150, 251, 292
- annotation_map_tile(), 382, 383
- anti_join(), 435, 447
- aov object, 505
- ape package, *see library(ape)*
- apropos(), 500
- args(), 521
- arithmetic operator, 502
- arrange(), 67, 73, 80, 106, 112, 144, 325, 333, 359, 458, 469, 486
- aRxiv package, *see library(aRxiv)*
- arxiv_search(), 431
- as.character(), 525, 526
- as.data.frame(), 454, 505
- as.factor(), 288
- as.formula(), 254
- as.matrix(), 265, 444, 505
- as.numeric(), 430, 557
- as.party(), 233
- as.vector(), 474
- as_adjacency_matrix(), 471, 473
- as_tibble(), 444, 479, 504
- assert_that(), 528
- assertthat package, *see library(assertthat)*
- assign(), 501
- assignment operator, 501
- attach(), 505
- attributes(), 505, 506
- augment(), 544, 556
- autoplot(), 219
- available package, *see library(available)*
- babynames package, *see library(babynames)*
- base package, *see library(base)*
- base::mean(), 510
- base::units(), 510
- bbox(), 388
- bench package, *see library(bench)*
- biglm package, *see library(biglm)*
- biglm(), 479
- bigmemory package, *see library(bigmemory)*
- bigrquery package, *see library(bigrquery)*
- bind_rows(), 245, 248, 318, 350, 484
- bind_tf_idf(), 441
- binom.test(), 286
- binwidth option, 43, 290
- bookdown package, *see library(bookdown)*
- bootstrapPage(), 309
- bq_project_query(), 488
- breaks option, 47, 290
- broom package, *see library(broom)*
- browser(), 527, 528
- browseURL(), 535
- by option, 96, 127
- c(), 325, 501, 520
- c_across(), 243
- caret package, *see library(caret)*
- case_when(), 81, 82
- cast_dtm(), 444
- cat(), 484
- cbind(), 504
- cdist(), 520
- centrality_pageRank(), 468, 471
- cforest(), 258
- checkboxGroupInput(), 309
- ci_calc(), 520, 521, 528
- class(), 140, 326, 486, 505, 570
- clean_names(), 132, 445
- coef(), 148, 546
- collect(), 328, 458, 570
- color option, 147
- colorNumeric(), 399
- colors(), 315

colSums(), 471
comparison operators, 502
compress option, 121
conf_mat(), 212
conflicts(), 505, 510
coord_flip(), 45, 61
coord_quickmap(), 382
coord_trans(), 39
copy_to(), 486
corlimit option, 444
count(), 211, 434, 445, 447
ctree(), 258
cummean(), 523
cut(), 47

data(), 493
data.frame(), 148, 326, 328, 504, 505, 522
data_grid(), 224, 253, 558
datatable(), 302
dataTableOutput(), 310
dbConnect(), 374, 569, 572
dbConnect_scidb(), 326, 457, 493
dbGetQuery(), 358, 457, 487, 570, 571
DBI package, *see library(DBI)*
dbname option, 569
dbplyr package, *see library(dbplyr)*
debug(), 527, 528
debugonce(), 527, 528
decreasing option, 445
desc(), 82, 106, 325, 458
detach(), 505
detectCores(), 481, 482
diameter(), 464
diff(), 546
dim(), 76, 330, 467, 507
dir_ls(), 265
discrim package, *see library(discrim)*
dist(), 265
distinct(), 265, 290
dmy(), 130, 133
download.file(), 264, 316, 395, 467
dplyr package, *see library(dplyr)*
DT package, *see library(DT)*
dyears(), 70
dygraph(), 304
dygraphs package, *see library(dygraphs)*
dyRangeSelector(), 304

eccentricity(), 464
edge_attr(), 463
element_line(), 320

element_rect(), 320
else statement, 500
enframe(), 291
environment tab, 509
equality operator, 501
error, standard, *see standard error*
etl package, *see library(etl)*
etl_NCI60(), 282, 283
etude package, *see library(etude)*
example(), 500
exists(), 501
exp(), 141
extract operator, 504
extrafont package, *see library(extrafont)*

facet_wrap(), 48
facet_grid(), 40, 58, 60, 189
facet_wrap(), 40, 47, 58, 289, 316, 544
factors, 525
FALSE, 502
family option, 221, 246, 557
fct_relevel(), 48
fec package, *see library(fec)*
fec12 package, *see library(fec12)*
fec16 package, *see library(fec16)*
file option, 121
fill option, 119
filter(), 41, 54, 67–70, 76, 82, 91, 94, 95, 103, 104, 118, 144, 271, 309, 325, 327, 333, 336, 342, 371, 374, 396, 430, 433, 447, 458, 460, 467, 468, 481, 486, 527, 544, 550

find(), 509
findAssocs(), 444
findFreqTerms(), 444
first(), 96
fit_k(), 148, 149
flexdashboard package, *see library(flexdashboard)*
FlickrAPI package, *see library(FlickrAPI)*
font_import(), 316
for operator, 140
for statement, 500
for(), 139–141, 143
forcats package, *see library(forcats)*
formula(), 505
fromJSON(), 483
fs package, *see library(fs)*
full_join(), 469
function(), 145, 148, 149, 240, 288, 315,

453, 454, 460, 481, 519, 520, 522, 528
furrr package, *see library(furrr)*
future package, *see library(future)*
future_map(), 482

gen_samp(), 289
gendist(), 522
geocode(), 407
geom_*() , 389
geom_abline(), 287
geom_bar(), 45, 50
geom_boxplot(), 50
geom_col(), 41, 45, 50, 55, 289, 319
geom_count(), 559
geom_curve(), 56
geom_density(), 43, 50
geom_edge_*() , 461
geom_edges(), 461, 462, 471
geom_histogram(), 43, 50, 189, 290
geom_hline(), 147, 320, 524
geom_jitter(), 557
geom_label(), 40
geom_line(), 48, 55, 150, 524
geom_linerange(), 61
geom_mosaic(), 50
geom_node_*() , 461
geom_nodes(), 461, 462, 471
geom_nodetext(), 471
geom_path(), 320
geom_point(), 36, 38, 46, 48, 50, 61, 134, 150, 287, 382, 524
geom_polygon(), 52, 397
geom_segment(), 251, 320
geom_sf(), 382
geom_smooth(), 46, 48, 134, 150, 542, 544, 557
geom_text(), 36, 40, 56, 61, 320
geom_tile(), 559
geom_vline(), 147, 150, 231, 290, 292
geomnet package, *see library(geomnet)*
get_sentiments(), 436
get_stopwords(), 435, 447
getURL(), 425
GGally package, *see library(GGally)*
ganimate package, *see library(ganimate)*
ggmosaic package, *see library(ggmosaic)*
gnetwork package, *see library(gnetwork)*
gnetwork(), 461, 462, 471

ggplot(), 36, 41, 46, 48, 55, 134, 147, 150, 287, 319, 524, 544, 557
ggplot2 package, *see library(ggplot2)*
ggplotly(), 302
ggraph package, *see library(ggraph)*
ggrepel package, *see library(ggrepel)*
ggspatial package, *see library(ggspatial)*
ggthemes package, *see library(ggthemes)*
ggtile(), 56, 320
glance(), 545, 552
glimpse(), 89, 90, 131, 140, 372, 430, 506, 525, 542
glm(), 211, 221, 242, 246, 557
glmnet package, *see library(glmnet)*
googlesheets4 package, *see library(googlesheets4)*
graph_from_data_frame(), 459, 468
gray.colors(), 396
greater than operator, 502
grep(), 427
grepl(), 427
group_by(), 74, 82, 93–95, 106, 118, 144, 147, 149, 152, 272, 325, 333, 359, 392, 416, 458, 469, 486
group_modify(), 146–149, 152, 201
group_split(), 483
gs_key(), 103
gs_read(), 103
gsub(), 317, 359
guide_legend(), 471
guides(), 320, 471
gutenbergr package, *see library(gutenbergr)*

h3(), 309
haven package, *see library(haven)*
hclust(), 267
head(), 45, 76, 119, 333, 343, 426, 459
help option, 510
help(), 110, 500
help.search(), 500
help.start(), 500
here package, *see library(here)*
Hmisc package, *see library(Hmisc)*
Hmisc(), 509
hour(), 130
hr_leader(), 149
html_nodes(), 124, 132, 445
html_table(), 125, 132
htmlwidgets package, *see library(htmlwidgets)*

if statement, 500
if(), 141
ifelse(), 56, 71, 72, 81, 82, 198, 272, 393, 546
igraph package, *see library(igraph)*
importance(), 238
in statement, 500
index operator, 504
initial_split(), 210
inner_join(), 90–92, 95, 98, 108, 110, 174, 325, 333, 344, 397, 418, 437
install.packages(), 493, 508, 509
install_github(), 493, 509
instaR package, *see library(instR)*
interval(), 70, 130
is.connected(), 454
is.data.frame(), 504
is.factor(), 526
is.matrix(), 503, 504
is.na(), 92, 184
is.numeric(), 142
is.vector(), 140, 503
is_tibble(), 504

janitor package, *see library(janitor)*
jsonlite package, *see library(jsonlite)*

kableExtra package, *see library(kableExtra)*
key option, 119
kknn package, *see library(kknn)*
kknn(), 239, 254
kmeans(), 273
kml(), 402
knitr package, *see library(knitr)*

label option, 150
Lahman package, *see library(Lahman)*
lars package, *see library(lars)*
lattice package, *see library(lattice)*
layer option, 381
lazyeval package, *see library(lazyeval)*
leaflet package, *see library(leaflet)*
leaflet(), 389, 399, 409
left_join(), 91, 92, 98, 110, 127, 333, 344, 358, 438, 469
legend.position option, 52
length(), 124, 140, 425, 507, 520, 522, 528
less than operator, 502
level(), 507
library(), 509, 510

library(airlines), 325, 332, 377, 493
library(alr3), 494
library(ape), 494
library(aRxiv), 135, 431, 494
library(assertthat), 494, 528
library(available), 494
library(babynames), 7, 53, 54, 56, 302, 371, 486, 494, 571
library(base), 35, 123, 510
library(bench), 141, 494
library(biglm), 479, 480, 494
library(bigmemory), 480
library(bigrquery), 488, 494
library(bookdown), xiii, 494
library(broom), 197, 198, 200, 201, 257, 494, 542, 544, 556
library(caret), 258, 494
library(class), 239
library(data.table), 479
library(DBI), 122, 354, 457, 487, 494, 568–570
library(dbplyr), 122, 494, 568
library(discrim), 242, 494
library(dplyr), xv, 4, 35, 67, 75–77, 84, 122, 136, 146, 174, 201, 325–329, 333, 334, 336, 342, 344, 350, 360, 371, 374, 396, 397, 457, 479–481, 486, 487, 493, 494, 526, 529, 566, 568–572
library(DT), 302, 304, 308, 310, 494
library(dygraphs), 304, 305, 494
library(etl), 8, 493, 494, 498
library(etude), xv, 497
library(extrafont), 322, 494
library(fec), 7
library(fec12), 8, 392, 493, 498
library(fec16), 8, 493, 494
library(flexdashboard), 306–308, 322, 494
library(FlickrAPI), 135
library(forcats), 48, 131, 494, 514
library(foreign), 122
library(fs), 494
library(furrr), 482, 494
library(future), 482, 494
library(geomnet), 461
library(GGally), 461, 494
library(gganimate), 306, 494
library(ggmosaic), 50, 494
library(ggnetwork), 461, 471, 474
library(ggplot2), xiii, xv, 4, 17, 20, 31, 35, 36, 39, 43, 46, 50, 53, 54, 56, 62,

67, 155, 219, 301–303, 306, 313–
 317, 382, 383, 389, 391, 397, 407,
 453, 461, 483, 493, 495
 library(ggraph), 453, 461, 495
 library(ggrepel), 495
 library(ggspatial), 382, 383, 495
 library(ggthemes), 31, 316, 495
 library(glmnet), 258, 495
 library(googlesheets4), 103, 122, 495
 library(gutenbergr), 448, 495
 library(haven), 122, 495
 library(here), 495
 library(Hmisc), 56, 495, 509, 510
 library(htmlwidgets), 301, 302, 389, 495
 library(igraph), 453, 459, 462, 468, 473,
 474, 495
 library(instaR), 135
 library(janitor), 121, 132, 445, 495
 library(jsonlite), 483, 495
 library(kableExtra), 495
 library(kknn), 239, 495
 library(knitr), xiv, 174, 354, 495, 499, 532,
 533, 535
 library(Lahman), 7, 76, 92, 99, 139, 325,
 478, 495
 library(lars), 258, 495
 library(lattice), 35, 495
 library(lazyeval), 495, 529
 library(leaflet), 302, 389, 391, 392, 399,
 400, 407, 495
 library(libgdal-dev), 380
 library(libproj-dev), 380
 library(lubridate), 50, 70, 129, 130, 132,
 133, 198, 432, 495
 library(macleish), 8, 48, 410, 420, 487,
 493, 495
 library(magick), 495
 library(map), 482
 library(mapproj), 384, 495
 library(maps), 384, 495
 library(maptools), 402
 library(mclust), 495
 library(mdsr), 7, 8, 35, 54, 81, 103, 210,
 229, 265, 282, 283, 313, 316, 317,
 326, 378, 425, 431, 453, 457, 467,
 479, 493, 495, 498, 508, 536
 library(methods), 124, 445
 library(modelr), 495
 library(MonetDBLite), 331
 library(mosaic), 286, 493, 495, 520, 533
 library(mosaicData), 7, 110, 111, 136, 495,
 542
 library(network), 474, 496
 library(NeuralNetTools), 243, 496
 library(NHANES), 49, 154, 156, 208, 496,
 525, 550
 library(nycflights13), 7, 89, 100, 184, 196,
 493, 496
 library(OpenCL), 482
 library(openrouteservice), 408, 498
 library(packrat), 496, 537
 library(palmerpenguins), 307, 496
 library(parallel), 481, 482
 library(parsnip), 210, 496
 library(partykit), 233, 258, 496
 library(patchwork), 155, 496
 library(plan), 482
 library(plotKML), 402
 library(plotly), 135, 302, 496
 library(print), 330
 library(purrr), xiii, 125, 132, 143, 146, 155,
 187, 222, 248, 289, 482, 496
 library(randomForest), 237, 238, 496
 library(RColorBrewer), 19–21, 32, 398,
 401, 496
 library(Rcpp), 480, 496
 library(RCurl), 425, 496
 library(readr), 122, 123, 126, 128, 133,
 317, 329, 467, 496, 503
 library(readxl), 122, 264, 265, 496
 library(remotes), 493, 496, 509
 library(renv), 496, 537
 library(reproducible), 537
 library(reticulate), 484, 496
 library(Rfacebook), 135
 library(rgdal), 380, 403, 496
 library(rgeos), 380, 403
 library(rlang), 155, 496, 526
 library(Rlinkedin), 135
 library(rmarkdown), xiv, 496, 532, 535
 library(RMySQL), 496, 568
 library(rpart), 230, 496
 library(RPostgreSQL), 568
 library(rsconnect), 496
 library(RSocrata), 135
 library(RSQLite), 496, 563, 568, 572
 library(rvest), 122, 124, 445, 496
 library(RWeka), 258, 496
 library(scales), 496
 library(sessioninfo), 496, 510

library(sf), [xiii](#), [380](#), [382](#), [396](#), [403](#), [407](#),
[408](#), [410](#), [413](#), [416](#), [418](#), [423](#), [496](#)
library(shiny), [310](#), [496](#)
library(shinybusy), [312](#), [497](#)
library(simstudy), [296](#)
library(sna), [474](#), [497](#)
library(sp), [380](#), [403](#), [497](#)
library(sparklyr), [485](#)–[487](#), [497](#)
library(st_area), [412](#)
library(stopwords), [435](#), [497](#)
library(streamgraph), [304](#), [305](#), [498](#)
library(stringr), [393](#), [425](#), [427](#), [484](#), [497](#)
library(styler), [121](#), [497](#)
library(testthat), [497](#), [522](#), [529](#)
library(textdata), [448](#), [497](#)
library(tibble), [238](#)
library(tidycensus), [497](#)
library(tidygeocoder), [407](#), [497](#)
library(tidygraph), [453](#), [455](#), [497](#)
library(tidymodels), [xiii](#), [210](#), [211](#), [227](#),
[230](#), [497](#)
library(tidyr), [84](#), [113](#), [135](#), [283](#), [445](#), [497](#)
library(tidytext), [434](#), [435](#), [444](#), [447](#), [448](#),
[497](#)
library(tidyverse), [xiii](#), [xiv](#), [35](#), [67](#), [84](#), [99](#),
[210](#), [380](#), [479](#), [497](#), [504](#), [508](#), [514](#),
[526](#), [527](#)
library(tigris), [403](#), [497](#)
library(tm), [444](#), [497](#)
library(transformr), [497](#)
library(tumblR), [135](#)
library(twitteR), [135](#), [497](#)
library(units), [408](#), [497](#)
library(usethis), [497](#)
library(viridis), [20](#), [32](#), [497](#)
library(viridisLite), [32](#), [497](#)
library(webshot), [497](#)
library(wordcloud), [436](#), [497](#)
library(wru), [165](#), [169](#), [497](#)
library(xaringantherem), [497](#), [509](#)
library(xfun), [497](#)
library(xkcd), [316](#), [317](#), [497](#)
library(yardstick), [210](#), [211](#), [218](#), [497](#)
linetype option, [147](#)
list(), [221](#), [246](#), [502](#), [520](#), [557](#)
list.files(), [371](#), [380](#)
lists, [124](#)
lm object, [505](#)
lm(), [197](#)–[199](#), [242](#), [257](#), [479](#)–[481](#), [541](#), [542](#),
[544](#), [546](#), [547](#)
loadfonts(), [316](#)
location package, *see library(location)*
log(), [453](#)
logical operator, [502](#)
ls(), [509](#), [525](#), [528](#)
lubridate package, *see library(lubridate)*
Macbeth_raw, [425](#)
macleish package, *see library(macleish)*
magick package, *see library(magick)*
make_babynames_dist(), [54](#)
map package, *see library(map)*
Map(), [482](#), [483](#)
map(), [116](#), [117](#), [143](#)–[145](#), [154](#), [155](#), [187](#),
[222](#), [240](#), [247](#), [249](#), [384](#), [481](#), [482](#),
[484](#)
map2(), [225](#), [454](#)
map2_dbl(), [222](#), [248](#)
map_chr(), [144](#)
map_dbl(), [141](#), [143](#), [144](#), [152](#), [240](#), [291](#),
[444](#)
map_dfr(), [146](#), [147](#), [187](#), [192](#), [193](#), [245](#),
[294](#)
map_int(), [144](#), [145](#)
mapper(), [484](#)
mapproj package, *see library(mapproj)*
maps package, *see library(maps)*
mark(), [141](#)
matrix(), [330](#), [479](#), [503](#)
max(), [74](#), [93](#), [96](#), [287](#), [458](#)
mclust package, *see library(mclust)*
mdsr package, *see library(mdsr)*
mdsr2exercises package, *see library(mdsr2exercises)*
mdy_hms(), [129](#)
mean(), [74](#), [140](#), [141](#), [143](#), [150](#), [152](#), [327](#),
[510](#), [522](#), [546](#)
methods package, *see library(methods)*
methods(), [505](#)
min(), [74](#), [287](#)
missing values, [99](#)
ml_linear_regression(), [487](#)
mlp(), [243](#)
mode(), [506](#)
modelr package, *see library(modelr)*
MonetDBLite package, *see library(MonetDBLite)*
month(), [50](#)
mosaic package, *see library(mosaic)*
mosaicData package, *see library(mosaicData)*
mplot(), [554](#)

mutate(), 47, 67, 70, 71, 78, 81, 107, 112, 128, 133, 142, 149, 195, 221, 222, 240, 247, 249, 283, 286, 328, 333, 336, 430, 445, 457, 468, 469, 523, 525, 526
 my_paste(), 327, 328
 n(), 60, 74, 75, 81, 327, 458
 n_distinct(), 93, 95, 458
 NA, 99
 na.omit(), 223, 250
 na.rm option, 140, 143
 naive_Bayes(), 242, 254
 names(), 103, 132, 139, 155, 210, 229, 317, 507, 509
 nchar(), 144, 145
 ncol(), 507
 nearest_neighbor(), 240
 nest(), 115, 118
 network package, *see library(network)*
 NeuralNetTools package, *see library(NeuralNetTools)*
 new_function(), 519
 next statement, 500
 nflip(), 291
 NHANES package, *see library(NHANES)*
 nls(), 147
 nnet(), 243, 254
 not operator, 502
 nrow(), 77, 193, 210, 392, 507, 545
 ntree option, 237
 NULL, 140
 numeric operator, 502
 numericInput(), 309
 nycflights13 package, *see library(nycflights13)*
 object.size(), 330
 objects(), 509
 ogrListLayers(), 381
 OpenCL package, *see library(OpenCL)*
 openrouteservice, 408
 openrouteservice package, *see library(openrouteservice)*
 options(), 507
 or operator, 502
 ors_api_key(), 408
 ors_directions(), 408, 409
 packrat package, *see library(packrat)*
 page_rank(), 468, 473, 474
 palette option, 399
 palmerpenguins package, *see library(palmerpenguins)*
 parallel package, *see library(parallel)*
 parse_character(), 128, 131
 parse_number(), 128, 133, 457
 parsnip package, *see library(parsnip)*
 partykit package, *see library(partykit)*
 paste(), 93, 115, 467, 479, 508
 paste0(), 327, 380, 395
 patchwork package, *see library(patchwork)*
 path(), 395
 pipe operator, 4, 77, 522
 pivot_longer(), 104, 113, 114, 243, 283, 355
 pivot_wider(), 113, 119, 127, 195, 196, 271, 272, 283, 304
 plan package, *see library(plan)*
 plan(), 482
 play_barabasi_albert(), 455
 play_erdos_renyi(), 453
 play_smallworld(), 455
 plot(), 382
 plotcp(), 235
 plotly package, *see library(plotly)*
 plotnet(), 243
 plotOutput(), 309
 pluck(), 116, 125, 132
 pmap_dfr(), 289
 pmin(), 119, 120
 predict(), 211, 221, 222, 234, 236, 238, 240, 242, 247, 249, 558
 print package, *see library(print)*
 print(), 83, 330, 506, 509
 print.data.frame(), 506
 print.default(), 506
 print.tbl(), 506
 print.tbl_df(), 506
 printcp(), 235
 projection option, 384
 pull(), 56, 116, 221, 222, 247, 249, 438, 525
 purrr package, *see library(purrr)*
 qdata(), 185, 193
 qt(), 520, 521
 quantile(), 152, 508
 randomForest package, *see library(randomForest)*

randomForest(), 237, 254
 rbind(), 189, 350
 rcauchy(), 523
 RColorBrewer package, *see* library(RColorBrewer)
 Rcpp package, *see* library(Rcpp)
 RCurl package, *see* library(RCurl)
 read.csv(), 123, 131
 read_csv(), 123, 124, 126, 210, 229, 317, 329, 467, 503
 read_excel(), 265
 read_html(), 124, 445
 readOGR(), 389
 readr package, *see* library(readr)
 readr::read_csv(), 131
 readRDS(), 121, 122
 readxl package, *see* library(readxl)
 Reduce(), 482, 483
 reducer(), 484
 relevel(), 507
 relist(), 503
 remotes package, *see* library(remotes)
 rename(), 70, 72, 78, 82, 133, 336, 445, 458, 469
 render(), 535
 renderPlot(), 309, 310, 312
 renv package, *see* library(renv)
 reorder(), 44, 45
 rep(), 288, 454, 473, 504, 523
 repeat statement, 500
 reproducible package, *see* library(reproducible)
 req(), 313
 require(), 509
 resample(), 286
 reticulate package, *see* library(reticulate)
 return(), 288, 520–522
 rexp(), 286
 Rfacebook package, *see* library(Rfacebook)
 rgdal package, *see* library(rgdal)
 rgeos package, *see* library(rgeos)
 rlang package, *see* library(rlang)
 Rlinkedin package, *see* library(Rlinkedin)
 rm(), 501
 rmarkdown package, *see* library(rmarkdown)
 RMySQL package, *see* library(RMySQL)
 rnorm(), 286, 288, 479, 481, 521
 roc_curve(), 218, 219
 round(), 265, 471
 row.names(), 507
 row_number(), 283
 rownames(), 265
 rownames_to_column(), 238
 rowwise(), 243
 rpar package, *see* library(rpar)
 rpart package, *see* library(rpart)
 rpart(), 230, 254, 258
 rpart.control(), 236, 254
 rpois(), 286
 RPostgreSQL package, *see* library(RPostgreSQL)
 rsconnect package, *see* library(rsconnect)
 RSiteSearch(), 500
 RSocrata package, *see* library(RSocrata)
 RSQLite package, *see* library(RSQLite)
 rsquared(), 545, 547
 rt(), 523, 524
 runApp(), 310
 runave(), 523
 runif(), 285, 286, 330
 rvest package, *see* library(rvest)
 RWeka package, *see* library(RWeka)
 sample(), 152, 286, 550
 saveRDS(), 121, 122
 scale(), 471
 scale_color(), 39
 scale_color_manual(), 559
 scale_color_viridis(), 20
 scale_fill_brewer(), 52
 scale_fill_distiller(), 398
 scale_fill_gradient(), 559
 scale_fill_manual(), 320, 397
 scale_size(), 559
 scale_size_continuous(), 461, 462
 scale_x_continuous(), 39, 290, 320, 455
 scale_x_discrete(), 39
 scale_x_log10(), 231
 scale_y_continuous(), 39, 320
 scale_y_discrete(), 39
 scales package, *see* library(scales)
 sd(), 283, 520
 se option, 46, 150
 search(), 505
 select(), 67–69, 76, 82, 103, 143, 225, 254, 273, 333, 336, 458, 468, 483, 525–527
 select_one(), 285
 semi_join(), 344
 sep option, 508

seq(), 318, 454
 seq_range(), 224, 253, 558
 server(), 310
 session_info(), 510
 sessioninfo package, *see* **library(sessioninfo)**
 set.seed(), 184, 210, 294, 295, 523
 set_names(), 484
 set_units(), 408
 set_vertex_attr(), 459
 setView(), 399
 sf package, *see* **library(sf)**
 sg_fill_brewer(), 304
 shiny package, *see* **library(shiny)**
 shinyApp(), 313
 shinybusy package, *see* **library(shinybusy)**
 shinyServer(), 309
 shinyUI(), 309
 shortest_paths(), 463
 show_query(), 326
 shuffle(), 283, 286
 simstudy package, *see* **library(simstudy)**
 size option, 147
 skim(), 50, 81, 184, 188, 394
 slice(), 333, 343
 slice_sample(), 48, 187, 193, 285, 436
 sna package, *see* **library(sna)**
 sort(), 73, 434, 444
 sp package, *see* **library(sp)**
 spark_connect(), 486
 spark_install(), 485
 sparklyr package, *see* **library(sparklyr)**
 spTransform-methods(), 388
 sql(), 569
 src_tbls(), 486
 st_as_sf(), 407
 st_bbox(), 382
 st_cast(), 414
 st_contains(), 414
 st_coordinates(), 408
 st_crs(), 385, 387
 st_distance(), 408, 421
 st_intersection(), 413, 414
 st_intersects(), 413, 418
 st_join(), 418
 st_layers(), 381, 395
 st_length(), 408, 417
 st_read(), 381, 395
 st_set_crs(), 407
 st_transform(), 388
 st_union(), 416
 st_within(), 414, 418
 st_write(), 402
 stat_function(), 147
 stop(), 528
 stopwords package, *see* **library(stopwords)**
 str(), 131, 140, 372, 506
 str_detect(), 427
 str_subset(), 427
 str_which(), 427
 str_detect(), 132, 133, 427, 429, 447, 484
 str_extract(), 427, 430, 434
 str_remove_all(), 445
 str_split(), 425
 str_sub(), 393
 str_subset(), 426, 427, 446
 str_which(), 426
 str_wrap(), 484
 streamgraph package, *see* **library(streamgraph)**
 streamgraph(), 304
 stringr package, *see* **library(stringr)**
 styler package, *see* **library(styler)**
 sum(), 74, 81, 444, 502
 summarize(), 56, 67, 74, 75, 81, 82, 94, 95,
 106, 118, 141, 142, 144, 325, 327,
 359, 392, 416, 458, 469, 486
 summary(), 55, 131, 372, 459, 487, 505,
 509, 525
 summary.aov(), 505
 summary.lm(), 505
 system_time(), 479, 481, 482
 tables(), 132
 tail(), 277
 tally(), 286, 433, 434
 tbl(), 326, 374, 569–571
 tbl_df(), 328
 testing(), 210
 testthat package, *see* **library(testthat)**
 textdata package, *see* **library(textdata)**
 theme(), 41, 314, 320
 theme_blank(), 461
 theme_bw(), 314
 theme_classic(), 314
 theme_excel(), 316
 theme_fivethirtyeight(), 316
 theme_grey(), 313–315
 theme_mdsr(), 315, 316
 theme_minimal(), 314
 theme_solarized(), 316

theme_tufte(), 316
theme_void(), 397
theme_xkcd(), 316
tibble(), 240, 288, 318, 454, 504
tidy(), 197, 198, 200, 201, 257, 401, 542, 547, 552
tidycensus package, *see* [library\(tidycensus\)](#)
tidygeocoder package, *see* [library\(tidygeocoder\)](#)
tidygraph package, *see* [library\(tidygraph\)](#)
tidymodels package, *see* [library\(tidymodels\)](#)
tidyrr package, *see* [library\(tidyrr\)](#)
tidytext package, *see* [library\(tidytext\)](#)
tidyverse package, *see* [library\(tidyverse\)](#)
tigris package, *see* [library\(tigris\)](#)
tm package, *see* [library\(tm\)](#)
toJSON(), 483
tolower(), 317
training(), 210
transformr package, *see* [library\(transformr\)](#)
translate_sql(), 327
tribble(), 56, 126, 319, 320, 429
TRUE, 502
try(), 528
tumblR package, *see* [library\(tumblR\)](#)
twitteR package, *see* [library\(twitteR\)](#)
typeof(), 506

uiOutput(), 310
undebug(), 527, 528
unique(), 458, 468
unit(), 320
units package, *see* [library\(units\)](#)
units(), 510
unlist(), 503
unnest(), 103, 117, 118, 225, 254, 320, 430
unnest_tokens(), 434, 435, 440, 447
unzip(), 264, 395
update.packages(), 509, 512
usethis package, *see* [library\(usethis\)](#)

value option, 119
var(), 545
vertex_attr(), 464
viridis package, *see* [library\(viridis\)](#)
viridisLite package, *see* [library\(viridisLite\)](#)

webshot package, *see* [library\(webshot\)](#)